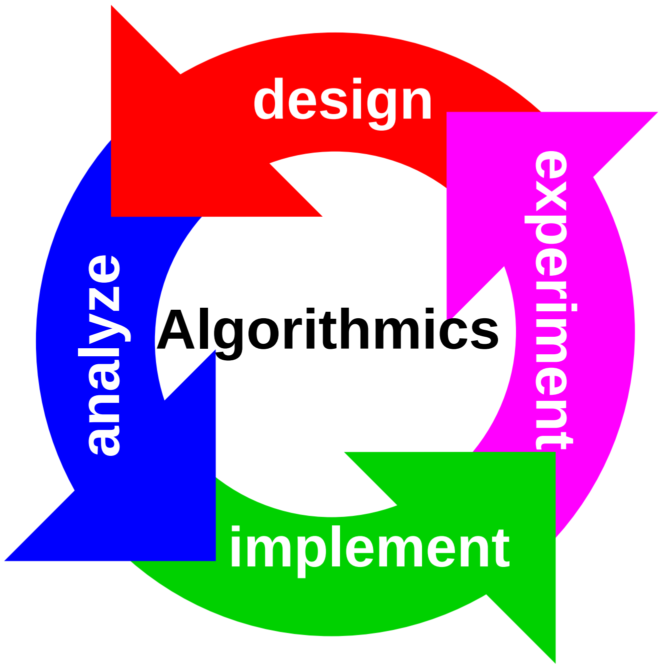
Felix Putze and Peter Sanders



Course Notes
Algorithm Engineering
TU Karlsruhe
October 19, 2009

Preface

These course notes cover a lecture on algorithm engineering for the basic toolbox that Peter Sanders is reading at Universität Karlsruhe since 2004. The text is compiled from slides, scientific papers and other manuscripts. Most of this material is in English so that this language was adopted as the main language. However, some parts are in German. The primal sources of our material are given at the beginning of each chapter. Please refer to the original publications for further references.
This document is still work in progress. Please report bugs of any type (content, language, layout, ...) to `putze@ira.uka.de`. Thank you!

Contents

Chapter 1

Was ist Algorithm Engineering?

1.1 Einführung

Algorithmen (einschließlich Datenstrukturen) sind das Herz jeder Computeranwendung und damit von entscheidender Bedeutung für große Bereiche von Technik, Wirtschaft, Wissenschaft und täglichem Leben. Die *Algorithmik* befasst sich mit der systematischen Entwicklung effizienter Algorithmen und hat damit entscheidenden Anteil an der effektiven Entwicklung verläßlicher und ressourcenschonender Technik. Wir nennen hier nur einige besonders spektakuläre Beispiele:

Das schnelle Durchsuchen der gewaltigen Datenmengen des Internet (z.B. mit Google) hat die Art verändert, wie wir mit Wissen und Information umgehen. Möglich wurde dies durch Algorithmen zur Volltextsuche, die in Sekundenbruchteilen alle Treffer aus Terabytes von Daten herausfischen können und durch Ranking-Algorithmen, die Graphen mit Milliarden von Knoten verarbeiten, um aus der Flut von Treffern relevante Antworten zu filtern. Weniger sichtbar aber ähnlich wichtig sind Algorithmen für die effiziente Verteilung von sehr häufig zugegriffenen Daten unter massiven Lastschwankungen oder gar Überlastungsangriffen (distributed denial of service attacks). Der Marktführer auf diesem Gebiet, Akamai, wurde von Algorithmikern gegründet. Eines der wichtigsten wissenschaftlichen Ereignisse der letzten Jahre war die Veröffentlichung des menschlichen Genoms. Mitentscheidend für die frühe Veröffentlichung war die von der Firma Celera verwendete und durch algorithmische Überlegungen begründete Ausgestaltung des Sequenzierprozesses (whole genome shotgun sequencing). Die Algorithmik hat sich hier nicht auf die Verarbeitung der von Naturwissenschaftlern produzierten Daten beschränkt, sondern gestaltenden Einfluss auf den gesamten Prozess ausgeübt.

Die Liste der Bereiche, in denen ausgefeilte Algorithmen eine Schlüsselrolle spielen, ließe sich fast beliebig fortsetzen: Computergrafik, Bildverarbeitung, geografische Informationssysteme, Kryptografie, Produktions-, Logistik- und Verkehrsplanung . . .

Wie funktioniert nun der Transfer algorithmischer Innovation in Anwendungsbere-

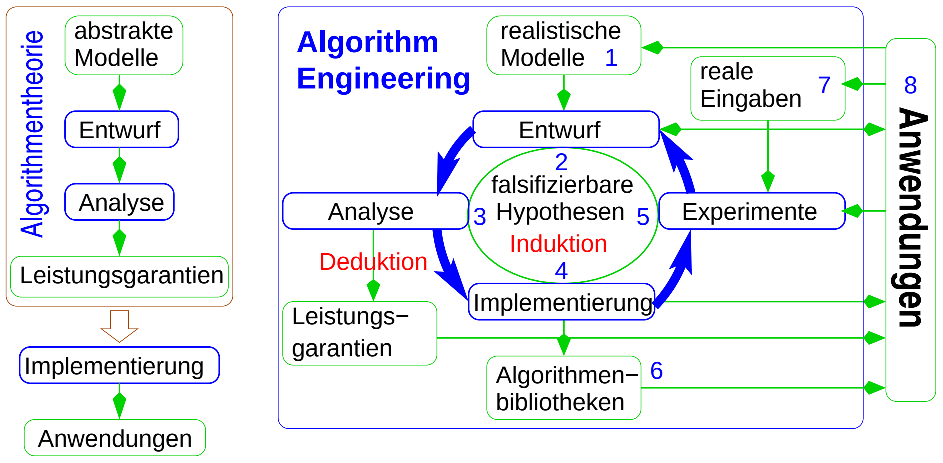


Figure 1.1: Zwei Sichtweisen der Algorithmik: Links: traditionell. Rechts: AE = Algorithmik als von falsifizierbaren Hypothesen getriebener Kreislauf aus Entwurf, Analyse, Implementierung, und experimenteller Bewertung von Algorithmen.

iche? Traditionell hat sich die Algorithmik der Methodik der *Algorithmentheorie* bedient, die aus der Mathematik stammt: Algorithmen werden für einfache und abstrakte Problem- und Maschinenmodelle entworfen. Hauptergebnis sind dann beweisbare Leistungsgarantien für alle möglichen Eingaben. Dieser Ansatz führt in vielen Fällen zu eleganten, zeitlosen Lösungen, die sich an viele Anwendungen anpassen lassen. Die harten Leistungsgarantien ergeben zuverlässig hohe Effizienz auch für zur Implementierungszeit unbekannte Typen von Eingaben. Aufgreifen und Implementieren eines Algorithmus ist aus Sicht der Algorithmentheorie Teil der Anwendungsentwicklung. Nach allgemeiner Beobachtung ist diese Art des Ergebnistransfers aber ein sehr langsamer Prozess. Bei wachsenden Anforderungen an innovative Algorithmen ergeben sich daraus wachsende Lücken zwischen Theorie und Praxis: Reale Hardware entwickelt sich durch Parallelismus, Pipelining, Speicherhierarchien u.s.w. immer weiter weg von einfachen Maschinenmodellen. Anwendungen werden immer komplexer. Gleichzeitig entwickelt die Algorithmentheorie immer ausgeklügeltere Algorithmen, die zwar wichtige Ideen enthalten aber manchmal kaum implementierbar sind. Außerdem haben reale Eingaben oft wenig mit den worst-case Szenarien der theoretischen Analyse zu tun. Im Extremfall werden viel versprechende algorithmische Ansätze vernachlässigt, weil eine vollständige Analyse mathematisch zu schwierig wäre.

Seit Beginn der 1990er Jahre wird deshalb eine breitere Sichtweise der Algorithmik immer wichtiger, die als *algorithm engineering* (AE) bezeichnet wird und bei der Entwurf, Analyse, Implementierung und experimentelle Bewertung von Algorithmen gleichberechtigt nebeneinander stehen. Der gegenüber der Algorithmentheorie größere Methodenapparat, die Einbeziehung realer Software und der engere Bezug zu Anwendungen verspricht realistischere Algorithmen, die Überbrückung entstandener Lücken zwischen Theorie und Praxis, und einen schnelleren Transfer von algorithmischem Know-how in Anwendungen. Abbildung 1.1 zeigt die Sichtweise der Algorithmik als AE und eine Aufteilung in acht eng interagierende Aktivitäten. Ziele und Arbeitsprogramm des Schwerpunktprogramms ergeben sich daraus in natürlicher Weise: Einsatz der vollen Schlagkraft der AE Methodologie mit dem Ziel, Lücken zwischen Theorie und Praxis zu überbrücken.

1. Studium realistischer Modelle für Maschinen und algorithmische Probleme.
2. Entwurf von einfachen und auch in der Realität effizienten Algorithmen.
3. Analyse praktikabler Algorithmen zwecks Etablierung von Leistungsgarantien, die Theorie und Praxis einander näher bringen.
4. Sorgfältige Implementierungen, die die Lücken zwischen dem besten theoretischen Algorithmus und dem besten implementierten Algorithmus verkleinern.
5. Systematische, reproduzierbare Experimente, die der Widerlegung oder Stärkung aussagekräftiger, falsifizierbarer Hypothesen dienen, die sich aus Entwurf, Analyse oder früheren Experimenten ergeben. Oft wird es z.B. um den Vergleich von Algorithmen gehen, deren theoretische Analyse zu viele Fragen offen lässt.
6. Entwicklung und Ausbau von Algorithmenbibliotheken, die Anwendungsentwicklungen beschleunigen und algorithmisches Know-how breit verfügbar machen.
7. Sammeln von großen und realistischen Probleminstanzen sowie Entwicklung von Benchmarks.
8. Einsatz von algorithmischem Know-how in konkreten Anwendungen.

1.2 Stand der Forschung und offene Probleme

Im Folgenden beschreiben wir die Methodik des AE anhand von Beispielen.

Fallbeispiel: Routenplanung in Straßennetzen

Jeder kennt diese zunehmend wichtige Anwendung: Man gibt Start- und Zielpunkt in ein Navigationssystem ein und wartet auf die Ausgabe der schnellsten Route. Hier hat das AE in letzter Zeit Lösungen entwickelt, die in Sekundenbruchteilen optimale Routen berechnen wo kommerzielle Lösungen trotz erheblich längerer Rechenzeiten bisher keine Qualitätsgarantien geben können und gelegentlich deutlich daneben liegen. Auf den ersten Blick ist das Anwendungsmodell ein klassisches und wohl studiertes Problem aus der Graphentheorie: kürzeste Wege in Graphen. Die altbekannte Lehrbuchlösung — Dijkstra's Algorithmus — hätte allerdings auf einem Hochleistungs-Server Antwortzeiten im Minutenbereich und wäre auf leistungsschwächerer mobiler Hardware mit begrenztem Hauptspeicher hoffnungslos langsam. Kommerzielle Routenplaner greifen deshalb zu Heuristiken, die annehmbare Antwortzeiten haben, aber nicht immer die beste Route finden.

Auf den zweiten Blick bietet sich ein verfeinertes Problemmodell an, das die Vorberechnung von Informationen zulässt, die dann für viele Anfragen verwendet werden. Die Theorie winkt ab und beweist, dass nur eine unpraktikabel große vorberechnete Datenstruktur die Berechnung von schnellsten Routen in beliebigen Graphen beschleunigt. Reale Straßengraphen haben jedoch Eigenschaften, welche die Vorberechnungsidee praktikabel machen. Die Wirksamkeit dieser Ansätze hängt von Hypothesen über die Eigenschaft der Straßengraphen ab, wie "'weit weg von Start und Ziel kann man die Suche auf überregionale Straßen beschränken"' oder "'Straßen, die vom Ziel wegführen, darf man ignorieren"'. Solche intuitiven Formulierungen gilt es dann so zu formalisieren, dass sich daraus Algorithmen mit Leistungsgarantien entwickeln lassen. Letztlich lassen diese Hypothesen sich aber nur durch Experimente mit Implementierungen überprüfen, die realistische Straßengraphen verwenden. Letzteres ist in der Praxis schwierig, da viele Firmen nur ungern Daten an Forscher herausgeben. Besonders wertvoll ist deshalb ein frei verfügbarer Graph der USA, der aus Daten im Web konstruiert wurde und jetzt für eine DIMACS Implementation Challenge zur Routenplanung Verwendung finden soll. Die Experimente decken Schwachstellen auf, die wiederum zum Entwurf verbesserter Algorithmen führen. Zum Beispiel stellte sich heraus, dass schon wenige Langstreckenfährverbindungen den Vorberechnungsaufwand der ersten Version des Algorithmus enorm in die Höhe treiben.

Trotz der Erfolge gibt es viele offene Fragen. Kann man die Heuristiken auch theoretisch analysieren um zu allgemeineren Leistungsgarantien zu kommen? Wie verträgt sich die Idee der Vorberechnung mit Anwendungsanforderungen wie Änderung des Straßennetzes, Baustellen, Staus, oder verschiedenen Zielfunktionen der Benutzer? Wie lassen sich die komplexen Speicherhierarchien von Mobilgeräten berücksichtigen?

Modelle

Ein wichtiger Aspekt des AE sind Maschinenmodelle. Sie betreffen im Prinzip alle Anwendungen und sind die Schnittstelle zwischen der Algorithmik und der rasanten technologische Entwicklung mit immer komplexerer Hardware. Wegen seiner großen Einfachheit ist das streng sequentielle, mit uniformem Speicher ausgestattete von Neumann Maschinenmodell immer noch Grundlage der meisten algorithmischen Arbeiten. Dies ist vor allem bei der Verarbeitung großer Datenmengen ein Problem, da die Zugriffszeit auf den Speicher sich um viele Größenordnungen ändert, je nachdem, ob auf den schnellsten Cache eines Prozessors, auf den Hauptspeicher oder auf die Festplatte zugegriffen wird. Speicherhierarchien werden in der Algorithmik bisher meist auf zwei Schichten beschränkt (I/O Modell). Dieses Modell ist sehr erfolgreich und eine Vielzahl von Ergebnissen dazu ist bekannt. Allerdings klaffen oft große Lücken zwischen den besten bekannten Algorithmen und den implementierten Verfahren. Bibliotheken für Sekundärspeicheralgorithmen wie STXXL versprechen diese Situation zu verbessern. In letzter Zeit gibt es aber verstärktes Interesse an weiteren immer noch einfachen Modellen zur Verarbeitung großer Datenmengen, z.B. einfache Modelle für mehrschichtige Speicherhierarchien, Datenstrommodelle, bei denen die Daten über ein Netzwerk hereinkommen, oder Sublinearzeitalgorithmen, bei denen gar nicht alle Daten berührt werden müssen.

Nur punktuelle Ergebnisse gibt es bisher zu anderen komplexen Eigenschaften moderner Prozessoren, wie den Ersetzungsmechanismen von Hardwarecaches oder Sprungvorhersage.

Wir erwarten, dass die Erforschung *paralleler* Algorithmen in nächster Zeit eine Renaissance erfahren wird, denn durch die Verbreitung von Multithreading, Multi-Core-CPUs und Clustern hält die Parallelverarbeitung nun Einzug in den Mainstream der Datenverarbeitung. Die traditionellen "‘flachen"’ Modelle für Parallelverarbeitung sind hier allerdings nur von begrenztem Nutzen, da es parallel zur Speicherhierarchie eine Hierarchie mehr oder weniger eng gekoppelter Verarbeitungseinheiten gibt.

Entwurf

Eine entscheidende Komponente des AE ist die Entwicklung implementierbarer Algorithmen, die effiziente Ausführung in realistischen Situationen erwarten lassen. Leichte Implementierbarkeit bedeutet vor allem Einfachheit aber auch Möglichkeiten zur Codewiederverwendung. Effiziente Ausführung bedeutet in der Algorithmentheorie gute asymptotische Ausführungszeit und damit gute Skalierungseigenschaften für sehr große Eingaben. Im AE sind aber auch konstante Faktoren und die Ausnutzung einfacher Probleminstanzen wichtig.

Ein Beispiel hierzu:

Der Sekundärspeicheralgorithmus zur Berechnung minimaler Spannbäume war der erste Algorithmus, der ein nichttriviales Graphenproblem mit Milliarden von Knoten auf einem PC löst. Theoretisch ist er suboptimal, weil er einen Faktor $O(\log \frac{m}{M})$ mehr Plattenzugriffe benötigt als der theoretisch beste Algorithmus (dabei ist m die Anzahl der Kanten des Eingabegraphen und M der Hauptspeicher der Maschine). Auf sinnvoll konfigurierten Maschinen benötigt er aber jetzt und in absehbarer Zukunft höchstens ein Drittel der Plattenzugriffe der asymptotisch besten bekannten Algorithmen. Hat man eine Prioritätsliste für Sekundärspeicher zur Verfügung wie in STXXL, ist der Pseudocode des Algorithmus ein Zwölfzeiler und die Analyse der erwarteten Ausführungszeit ein Siebenzeiler.

Analyse

Selbst einfache, in der Praxis bewährte Algorithmen sind oft schwer zu analysieren und dies ist ein Hauptgrund für Lücken zur Algorithmentheorie. Die Analyse solcher Algorithmen ist damit ein wichtiger Aspekt des AE. Zum Beispiel sind randomisierte (zufallsgesteuerte) Algorithmen oft viel einfacher und schneller als die besten bekannten deterministischen Algorithmen. Allerdings sind selbst einfache randomisierte Algorithmen oft schwer zu analysieren.

Viele komplexe Optimierungsprobleme werden mittels *Metaheuristiken* wie (randomisierter) lokaler Suche oder genetischer Programmierung gelöst. So entworfene Algorithmen sind einfach und flexibel an das jeweils vorliegende Problem anpassbar. Nur ganz wenige dieser Algorithmen sind aber bisher analysiert worden obwohl Leistungsgarantien von großem theoretischen und praktischen Interesse wären.

Ein berühmtes Beispiel für lokale Suche ist der Simplexalgorithmus zur linearen Optimierung — vielleicht der praktisch wichtigste Algorithmus in der mathematischen Optimierung. Einfache Varianten des Simplexalgorithmus benötigen für spezielle, konstruierte Eingaben exponentielle Zeit. Es wird aber vermutet, dass es Varianten gibt, die in polynomieller Zeit laufen. In der Praxis jedenfalls genügt eine lineare Zahl Iterationen. Bisher kennt man aber lediglich subexponentielle erwartete Laufzeitschranken für inpraktikable Varianten. Spielmann und Teng konnten jedoch zeigen, dass selbst kleine zufällige Veränderungen der Koeffizienten eines beliebigen linearen Programms genügen, um die erwartete Laufzeit des Simplexalgorithmus polynomiell zu machen. Dieses Konzept der *geglätteten Analyse (smoothed analysis)* ist eine Verallgemeinerung der *average case analysis* und auch jenseits des Simplexalgorithmus ein interessantes Werkzeug des AE. Zum Beispiel konnten Beier und Vöcking für eine wichtige Familie NP-harter Probleme zeigen, dass ihre geglättete Komplexität polynomiell ist. Dieses Ergebnis erklärt u.a., warum das NP-harte Rucksackproblem sich in der Praxis effizient lösen lässt und hat auch zur Verbesserung der besten Codes für Rucksackprobleme geführt. Es gibt auch enge Beziehungen zwischen geglätteter Komplexität, Näherungsalgorithmen und sogenannten pseudopolynomiellen Algorithmen, die ebenfalls ein interessanter Ansatz zur praktischen Lösungen NP-harter Probleme sind.

Implementierung

Die Implementierung ist nur scheinbar der am klarsten vorgezeichnete und langweiligste Schritt im Kreislauf des AE. Ein Grund dafür sind die großen semantischen Lücken zwischen abstrakt formulierten Algorithmen, imperativen Programmiersprachen und realer Hardware.

Ein extremes Beispiel für die semantische Lücke sind viele geometrische Algorithmen, die unter der Annahme exakter Arithmetik mit reellen Zahlen und ohne explizite

Berücksichtigung degenerierter Fälle entworfen sind. Die Robustheit geometrischer Algorithmen kann deshalb als eigener Zweig des AE betrachtet werden.

Selbst Implementierungen relativ einfacher grundlegender Algorithmen können sehr anspruchsvoll sein. Dort gilt es nämlich oft mehrere Kandidaten auf Grund kleiner konstanter Faktoren in ihrer Ausführungszeit zu vergleichen. Der einzige verlässliche Weg besteht dann darin, alle Kontrahenten voll auszureizen, denn schon kleine Implementierungsdetails können sich zu einem Faktor zwei in der Ausführungszeit auswachsen. Selbst ein Vergleich des erzeugten Maschinencodes kann angezeigt sein, um Zweifelsfälle aufzuklären.

Oft liefern erst Implementierungen von Algorithmen einen letzten Beleg für deren Korrektheit bzw. die Qualität der Ergebnisse. In der Geometrie und bei Graphenproblemen wird natürlicherweise meist eine graphische Ausgabe der Ergebnisse erzeugt, wodurch Nachteile des Algorithmus oder sogar Fehler sofort sichtbar werden.

Zum Beispiel wurde zur Einbettung eines planaren Graphen 20 Jahre lang auf eine Arbeit von Hopcroft und Tarjan[1] verwiesen. Dort findet sich aber nur eine vage Beschreibung wie sich ein Planaritätstestalgorithmus erweitern lässt. Einige Versuche einer detaillierteren Beschreibung waren fehlerhaft. Dies wurde erst bemerkt, als die ersten korrekten Implementierungen erstellt wurden. Lange Zeit gelang es niemandem, einen berühmten Algorithmus[2] zur Berechnung von 3-Zusammenhangskomponenten (ein wichtiges Werkzeug beim Graphenzeichnen und in der Signalverarbeitung) zu implementieren. Erst während einer Implementierung im Jahr 2000 wurden die Fehler im Algorithmus identifiziert und korrigiert.

Es gibt sehr viele interessante Algorithmen für wichtige Probleme, die noch nie implementiert wurden. Zum Beispiel, die asymptotisch besten Algorithmen für viele Fluss- und Matchingprobleme, die meisten Algorithmen für mehrschichtige Speicherhierarchien (cache oblivious Algorithmen) oder geometrische Algorithmen, die Cuttings oder ϵ-Netze benutzen.

Experimente

Aussagekräftige Experimente sind der Schlüssel zum Schließen des Kreises im AE-Prozess. Zum Beispiel brachten Experimente[3] zur Kreuzungsminimierung beim Graphenzeichnen eine neue Qualität in diesen Bereich. Alle vorhergehenden Studien arbeiteten mit relativ dichten Graphen und wiesen jeweils nach, dass die Kreuzungszahl recht nahe an die jeweiligen oberen theoretischen Schranken herankam. In den ange-

[1]J. Hopcroft and R. E. Tarjan: Efficient planarity testing. J. of the ACM, 21(4):549–568, 1974.

[2]R. E. Tarjan and J. E. Hopcroft: Dividing a graph into triconnected components. SIAM J. Comput., 2(3):135–158, 1973.

[3]M. Jünger and P. Mutzel: 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. Journal of Graph Algorithms and Applications (JGAA), 1(1):1–25, 1997.

sprochenen Experimenten wird dagegen auch mit optimalen Algorithmen und den in der Praxis wichtigen dünnen Graphen gearbeitet. Es stellte sich heraus, dass die Ergebnisse mancher Heuristiken um ein Vielfaches über der optimalen Kreuzungszahl liegen. Dieses Papier gehört inzwischen zu den am meisten zitierten Arbeiten im Bereich des Graphenzeichnens.

Experimente können auch entscheidenden Einfluss auf die Algorithmenanalyse haben: Die Rekonstruktion einer Kurve aus einer Menge von Messpunkten ist die grundlegendste Variante einer wichtigen Familie von Bildverarbeitungsproblemen. In einer Arbeit von Althaus und Mehlhorn[4] wird ein scheinbar recht aufwendiges Verfahren untersucht, das auf dem Handlungsreisendenproblem beruht. Bei Experimenten stellte sich heraus, dass "'vernünftige"' Eingaben zu leicht lösbaren Instanzen des Handlungsreisendenproblems führen. Diese Beobachtung wurde dann formalisiert und bewiesen.

Gegenüber den Naturwissenschaften ist das AE in der privilegierten Situation, eine Vielzahl von Experimenten schnell und vergleichsweise kostengünstig durchführen zu können. Die Rückseite dieser Medaille ist aber eine hochgradig nichttriviale Planung, Auswertung, Archivierung, Aufbereitung und Interpretation dieser Ergebnisse. Ausgangspunkt sollten dabei falsifizierbare Hypothesen über das Verhalten der untersuchten Algorithmen sein, die aus Entwurf, Analyse, Implementierung oder früheren Experimenten stammen. Ergebnis ist eine Widerlegung, Bestätigung oder Verfeinerung dieser Hypothesen. Diese führen als Ergänzung beweisbarer Leistungsgarantien nicht nur zu besserem Verständnis der Algorithmen, sondern liefern auch Ideen für bessere Algorithmen, genauere Analyse oder effizientere Implementierung.

Erfolgreiches Experimentieren hat viel mit Software Engineering zu tun. Ein modularer Aufbau der Implementierungen ermöglicht flexible Experimente. Geschickter Einsatz von Werkzeugen erleichtert die Auswertung. Sorgfältige Dokumentation und Versionsverwaltung erleichtert Reproduzierbarkeit — eine zentrale Anforderung wissenschaftlicher Experimente, die bei den schnellen Modellwechseln von Soft- und Hardware eine große Herausforderung darstellt.

Probleminstanzen

Sammlungen von realistischen Probleminstanzen zwecks Benchmarking haben sich als entscheidend für die Weiterentwicklung von Algorithmen erwiesen. Zum Beispiel gibt es interessante Sammlungen für einige NP-harte Probleme wie das Handlungsreisendenproblem, das Steinerbaumproblem, Satisfiability, Set Covering oder Graphpartitionierung. Besonders bei den beiden ersten Problemen hat das zu erstaunlichen Durchbrüchen geführt. Mit Hilfe tiefer mathematischer Einsichten in die Struktur der

[4]E. Althaus and K. Mehlhorn: Traveling salesman-based curve reconstruction in polynomial time. SIAM Journal on Computing, 31(1):27–66, 2002.

Probleme kann man selbst große, realistische Instanzen des Handlungsreisendenproblems und des Steinerbaumproblems exakt lösen.

Merkwürdigerweise sind realistische Probleminstanzen für polynomiell lösbare Probleme viel schwerer zu bekommen. Zum Beispiel gibt es dutzende praktischer Anwendungen der Berechnung maximaler Flüsse aber die Algorithmenentwicklung muss bislang mit synthetischen Instanzen vorlieb nehmen.

Anwendungen

Die Algorithmik spielt eine Schlüsselrolle bei der Entwicklung innovativer IT-Anwendungen und entsprechend sind anwendungsorientierte AE-Projekte aller Art eine sehr wichtiger Teil des Schwerpunktprogramms. Hier nennen wir nur einige *grand challenge* Anwendungen, bei denen Algorithmik eine wichtige Rolle spielen könnte und die ein besonderes Potential haben, einen wichtigen Einfluss auf Wissenschaft, Technik, Wirtschaft oder tägliches Leben zu haben.[5]

Bioinformatik Neben dem bereits genannten Problem der Genomsequenzierung hält die Mikrobiologie viele weitere algorithmische Herausforderungen bereit: die Berechnung der *Tertiärstrukturen* von Proteinen; Algorithmen zur Berechnung von *Stammbäumen* von Arten; data mining in den Daten zur Genaktivierung, die in großem Umfang mit *DNA chips* gewonnen werden... Diese Probleme können nur in enger Kooperation mit Molekularbiologen oder Chemikern gelöst werden.

Information Retrieval Die zu Beginn erwähnten Indizierungs- und Rankingalgorithmen von Internetsuchmaschinen sind zwar sehr erfolgreich, lassen aber noch viel zu wünschen übrig. Viele Heuristiken sind kaum publiziert, geschweige denn mit Leistungsgarantien ausgestattet. Nur in kleineren Systemen wird bisher ernsthaft versucht, Ähnlichkeitssuche zu unterstützen und es zeichnet sich ein Rüstungswettlauf zwischen Rankingalgorithmen und Spammern ab, die diese zu täuschen versuchen.

Verkehrsplanung Der Einsatz von Algorithmen in der Verkehrsplanung hat gerade erst begonnen. Neben einzelnen Anwendungen im Flugverkehr, wo Probleminstanzen relativ klein und das Einsparpotenzial groß ist, beschränken sich diese Anwendungen auf verhältnismäßig einfache, isolierte Bereiche: Datenerfassung (Netze, Straßenkategorien, Fahrzeiten), Monitoring und partielle Lenkung (Toll Collect, Bahnleitstände),

[5]Dessen ungeachtet wird natürlich nicht erwartet, dass ein einzelnes Teilprojekt den Durchbruch bei einer *grand challenge* bringt, und viele Teilprojekte werden sich mit weniger spektakulären, aber ebenso interessanten Anwendungen beschäftigen.

Prognose (Simulation, Vorhersagemodelle) und einfache Nutzerunterstützung (Routenplanung, Fahrplanabfrage). Das AE kann wesentlich zur Weiterentwicklung und Integration dieser verschiedenen Aspekte hin zu leistungsfähigen Algorithmen für eine bessere Planung und Lenkung unserer Verkehrssysteme (Lenkung durch Maut, Fahrplanoptimierung, Linienplanung, Fahrzeug- und Personaleinsatzplanung) beitragen. Besondere Herausforderungen sind dabei die sehr komplexen Anwendungsmodelle und die daraus entstehenden riesigen Problemgrößen.

Geografische Informationssysteme Moderne Erdbeobachtungssatelliten und andere Datenquellen erzeugen inzwischen täglich viele Terabyte an Informationen, die wichtige Anwendungen in Landwirtschaft, Umweltschutz, Katastrophenschutz, Tourismus etc. versprechen. Die effektive Verarbeitung solch gewaltiger Datenmengen ist aber eine echte Herausforderung bei der Know-how aus geometrischen Algorithmen, Parallelverarbeitung und Speicherhierarchien sowie AE mit realen Eingabedaten eine wichtige Rolle spielen wird.

Kommunikationsnetzwerke Im selben Maße wie Netzwerke immer vielseitiger und größer werden, wächst der Bedarf an effizienten Verfahren zu ihrer Organisation. Besonderes Interesse gilt hier mobilen, ad-hoc und Sensornetzen, sowie Peer-to-peer Netzen und der Koordination konkurrierender Agenten mit spieltheoretischen Techniken. All diesen neuartigen Anwendungen ist gemeinsam, dass sie ohne eine zentrale Planung und Organisation auskommen müssen.

Viele der hier untersuchten Fragestellungen kann man als noch-nicht-Anwendungen bezeichnen. Aus der Perspektive des AE ist daran besonders interessant, dass auch praktische Arbeiten hier keine verlässlichen Daten über Größe und Eigenarten der späteren Anwendungssituation haben. Einerseits ergibt sich daraus ein noch größerer Bedarf an beweisbaren Leistungsgarantien. Andererseits sind die Modelle vieler theoretischer Arbeiten auf diesem Gebiet noch weiter von der Realität entfernt als sonst.

Planungsprobleme Zeitliche Abläufe in Produktion und Logistik werden stets enger, und der Bedarf an algorithmischer Unterstützung und Optimierung wächst. Erste Ansätze hierzu aus der Algorithmik werden durch Onlinealgorithmen (*dial a ride*, Scheduling) und *flows over time* (Routing mit Zeitfenstern, dynamische Flüsse) gegeben. Die Entwicklung steht jedoch erst in den Anfängen. Für aussagekräftige Qualitätsaussagen zu Onlinealgorithmen muss insbesondere die kompetitive Analyse überdacht werden, die sich zu sehr am groben worst-case Verhalten orientiert. Flows over Time verlangen nach besseren Techniken, um algorithmisch möglichst effizient mit der Dimension "‘Zeit"’ umzugehen.

Chapter 2

Data Structures

Most material in this chapter was taken from a yet unpublished book manuscript by Peter Sanders and Ulrich Mehlhorn. Some parts on external data structures were presented in [7]. Notice that during the lecture, the latter topics were covered in the talk on external algorithms, not in the introduction on data structures. If you are unfamiliar with external memory models, please read the introduction in 5.2 or the short overview in the appendix 12.1.

2.1 Arrays & Lists

For starters, we will study how algorithm engineering can be applied to the (apparently?) easy field of sequence data structures.

Bounded Arrays: Usually the most basic, built-in sequence data structure in programming languages. They have constant running time for $[\cdot]$-, popBack- and pushBack-operations which remove or add an element behind the currently last entry. Their major drawback is that their size has to be known in advance to reserve enough memory.

Unbounded Arrays: To bypass this often unconvenient restriction, unbounded arrays are introduced (`std::vector` from the C++ STL is an example). They are implemented on a bounded array. If this array runs out of space for new elements, a new array of double size is allocated and the old content copied. If the filling degree is reduced to a quarter by pop-operations, the array is replaced with new one, using only the half space. We can show amortized costs of O(1) for pushBack and popBack implemented that way. A proof is given in the appendix in 12.2. Note that is not possible to already shrink the array when it is half full since repeated insertion and deletion at that point would lead to costs of $\mathcal{O}(n)$ for a single operation.

Double Linked Lists[1]: Figure 2.1 shows the basic building block of a linked list. A

[1]Sometimes singly linked lists (maintaining only a successor pointer) are sufficient and more space

```
Class Item of Element
    e : Element
    next : Handle
    prev : Handle
    invariant next→prev=prev→next=this
```

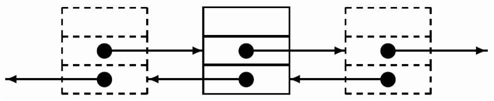


Figure 2.1: Prototype of a segment in doubly linked list

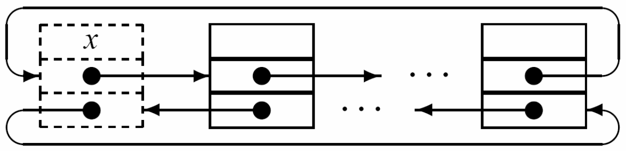


Figure 2.2: Structure of a double linked list

list item (a link of a chain) stores one element and pointers to successor and predecessor. This sounds simple enough, but pointers are so powerful that we can make a big mess if we are not careful. What makes a consistent list data structure? We make a simple and innocent looking decision and the basic design of our list data structure will follow from that: The successor of the predecessor of an item must be the original item, and the same holds for the predecessor of a successor. If all items fulfill this invariant, they will form a collection of cyclic chains. This may look strange, since we want to represent sequences rather than loops. Sequences have a start and an end, wheras loops have neither. Most implementations of linked lists therefore go a different way, and treat the first and last item of a list differently. Unfortunately, this makes the implementation of lists more complicated, more errorprone and somewhat slower. Therefore, we stick to the simple cyclic internal representation.

For conciseness, we implement all basic list operations in terms of the single operation splice depicted in Figure 2.3. splice cuts out a sublist from one list and inserts it after some target item. The target can be either in the same list or in a different list but it must not be inside the sublist. splice can easily be specialized to common methods like `insert`, `delete`,...

Since splice never changes the number of items in the system, we assume that there is one special list freeList that keeps a supply of unused elements. When inserting new elements into a list, we take the necessary items from freeList and when deleting elements

efficient. As they have non-intuitive semantics on some operations and are less versatile, we focus on doubly linked lists.

we return the corresponding items to freeList. The function checkFreeList allocates memory for new items when necessary. A freeList is not only useful for the splice operation but it also simplifies our memory management which can otherwise easily take 90% of the work since a `malloc` would be necessary for every element inserted[2]. It remains to decide how to simulate the start and end of a list. The class List in Figure 2.2 introduces a dummy item h that does not store any element but seperates the first element from the last element in the cycle formed by the list. By definition of Item, h points to the first "proper" item as a successor and to the last item as a predecessor. In addition, a handle head pointing to h can be used to encode a position before the first element or after the last element. Note that there are n+1 possible positions for inserting an element into an list with n elements so that an additional item is hard to circumvent if we want to code handles as pointers to items. With these conventions in place, a large number of useful operations can be implemented as one line functions that all run in constant time. Thanks to the power of splice, we can even manipulate arbitrarily long sublists in constant time. The dummy header can also be useful for other operations. For example consider the following code for finding the next occurrence of x starting at item from. If x is not present, head should be returned. We use the header as a sentinel. A sentinel is a dummy element in a data structure that makes sure that some loop will terminate. By storing the key we are looking for in the header, we make sure that the search terminates even if x is originally not present in the list. This trick saves us an additional test in each iteration whether the end of the list is reached. A drawback of dummy headers is that it requires additional space. This seems negligible for most applications but may be costly for many, nearly empty lists. This is a typical scenario for hash tables using chaining on collisions.

2.2 External Lists

The direct implementation of a linked list in an external memory model will have costs of 1 I/O when following a link, which leads to $\Theta(N)$ I/Os for traversing N elements. This is caused by the high degree of freedom in the allocation of list elements within memory[3]. A first idea to improve this is to introduce locality by requiring to store B consecutive elements together. Traversal is now only $N/B = \mathcal{O}(\text{scan}(N))$ I/Os, but an insertion or deletion can cost $\Theta(N/B)$ I/Os for moving all following elements. We relax the invariant to $\geq \frac{2}{3}B$ elements in every pair of consecutive blocks. Traversal is still available for $\leq 3N/B = \mathcal{O}(\text{scan}(N))$ I/Os. For inserting in block i, we have to distinguish to cases: If block i has space we pay 1 I/O and are done. If it is full but a neighbor has space, we push

[2]Another countermeasure to allocation overhead is scheduling many insertions at the same time, resulting in only one `malloc` and possibly less cache faults as many items reside in the same memory block

[3]A faster traversal is possible if we use list ranking (see 5.9) as preprocessing, which can be done in $\mathcal{O}(\text{sort}(N))$. Sorting with respect to each element's rank (distance from last node) will then give a scannable representation of the list

```
//Remove $\langle a,\ldots,b\rangle$ from its current list and insert it after $t$
//$\ldots,a',a,\ldots,b,b',\ldots,t,t',\ldots) \mapsto (\ldots,a',b',\ldots,t,a,\ldots,b,t',\ldots)$
**Procedure** *splice*$(a,b,t : Handle)$
    **assert** $b$ is not before $a \wedge t \notin \langle a,\ldots,b\rangle$
    $a' := a \rightarrow prev$
    $b' := b \rightarrow next$
    $a' \rightarrow next := b'$
    $b' \rightarrow prev := a'$


    $b \rightarrow next := t'$
    $a \rightarrow prev := t$

    $t \rightarrow next := a$
    $t' \rightarrow prev := b$
```

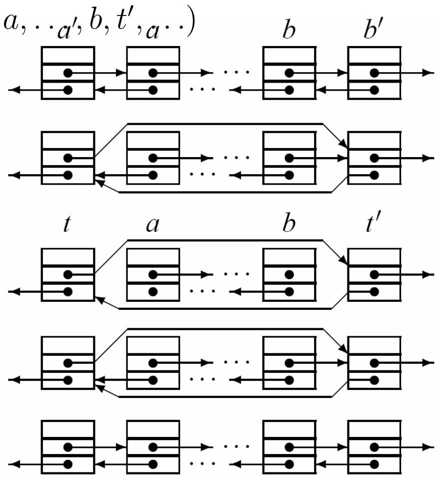


Figure 2.3: The splice method

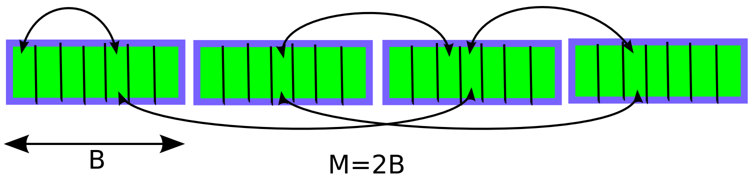


Figure 2.4: The direct implementation of linked lists is not suited for external memory.

an element to it for $\mathcal{O}(1)$ I/Os. If both neighbors are full, we split block i into 2 blocks of $\approx B/2$ elements, for (amortized) costs of $\mathcal{O}(1)$ I/Os ($\geq B/6$ deletions needed to violate the invariant). For deletion from block i: if blocks i and $i+1$ or blocks i and $i-1$ have $\leq 2B/3$ elements $\Rightarrow$ merge the two blocks, again for (amortized) $\mathcal{O}(1)$ I/Os.

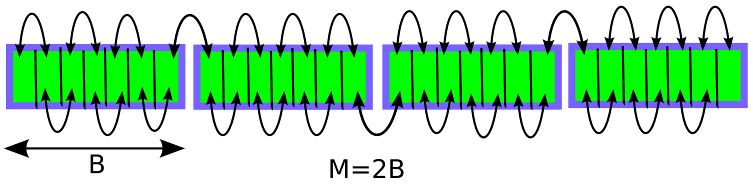


Figure 2.5: First approach: block B consecutive list elements together

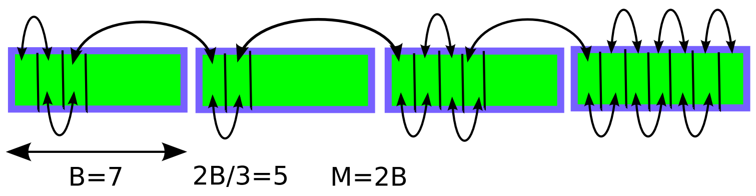


Figure 2.6: Second approach: block $\geq \frac{2}{3}B$ consecutive list elements together

	S-List	B-Array	U-Array
dynamic	$+$	$-$	$+$
space wasting	pointer set free?	too large?	too large?
time wasting	cache miss	$+$	resizing
worst case time	$(+)$	$+$	$-$

Table 2.1: Pros and cons for implementation variants of a stack

2.3 Stacks, Queues & Variants

We now want to use these general sequence types to implement another important data structure: A stack with operations push (insert at the end of the sequence) and pop (return and remove the last element) which we both want to implement with constant costs. Let us examine the alternatives:

A bounded array is only feasible if you can give a tight limit for the number of inserted elements; otherwise, you have to allocate much memory in advance to avoid running out of space. A linked list comes with nontrivial memory management and a lot of cache faults (when every successor is in a different memory block). An unbounded array has no constant cost guaranty for a single operation and can consume up to twice the actually required space. So none of the basic data structures comes without major drawbacks. For an optimal solution, we need to take a hybrid approach:

A hybrid stack is a linked list containing bounded arrays of size B. When the current array is full, another one is allocated and linked. We now have a dynamic data structure

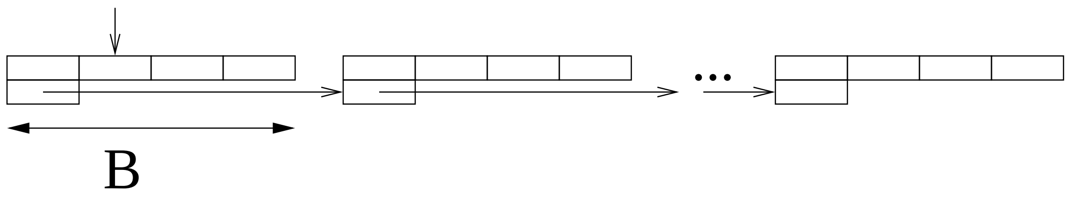


Figure 2.7: A hybrid stack

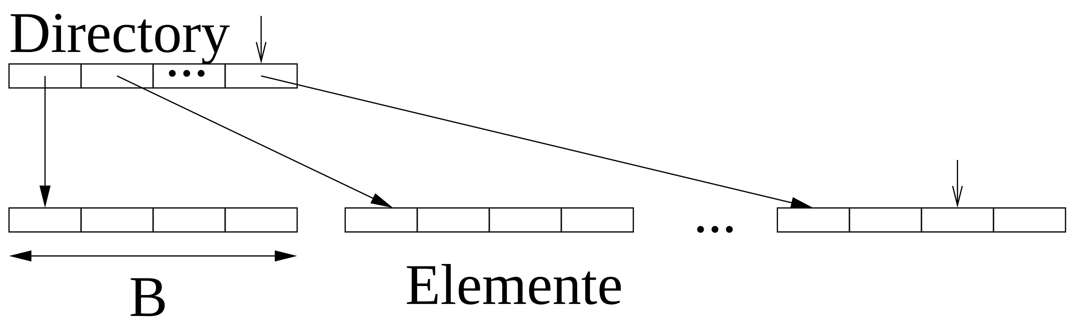


Figure 2.8: A variant of the hybrid stack

with (small) constant worst case access time[4] at the back pointer. We give up a maximum of $n/B + B$ wasted space (for pointers and one empty block). This is minimized for $B = \Theta(\sqrt{n})$.

A variant of this stack works as follows: Instead of having each block maintain a pointer to its successor, we use a directory (implemented as an unbounded array) containing these. Together with two additional references to the current dictionary entry and the current position in the last block, we gain the functionality of a stack. Additionally, it is now easy to implement $[\cdot]$ in constant time using integer division and modulo arithmetic. The drawback of this approach is non-constant worst case insertion time (although we still have constant *amortized* costs).

There are further specialized data structures that can be useful for certain algorithms: a FIFO queue allows insertion at one end and extraction at the other. FIFO queues are easy to implement with singly linked lists with a pointer to the last element. For bounded queues, we can also use cyclic arrays where entry zero is the successor of the last entry. Now it suffices to maintain two indices h and t delimiting the range of valid queue entries. These indices travel around the cycle as elements are queued and dequeued. The cyclic semantics of the indices can be implemented using arithmetics modulo the array size.[5] Our implementation always leaves one entry of the array empty because otherwise it would be difficult to distinguish a full queue from an empty queue. Bounded queues can be made unbounded using similar techniques as for unbounded arrays.

Finally, deques – allowing read/write access at both ends – cannot be implemented efficiently using singly linked lists. But the array based FIFO from Figure 2.3 is easy to generalize. Circular array can also support access using $[\cdot]$ (interpreting $[i]$ as $[i + h \mathbf{mod}\, n]$.

With techniques from both the hybrid stack variant and the cyclic FIFO queue, we can derive a data structure with constant costs for random accesses and costs $\mathcal{O}(\sqrt{n})$ for

[4]although inserting at the end of the current array is still costlier

[5]On some machines one might get significant speedups by choosing the array size as a power of two and replacing $\mathbf{mod}$ by bit operations.

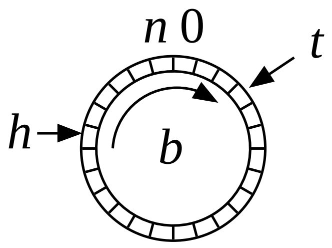


Figure 2.9: A variant of the hybrid stack

insertion/deletion on arbitrary positions: Instead of bounded arrays, we have our directory point to *cyclic arrays*. Random access works as above. For insertion at a random location, shift the elements in the corresponding cyclic array that follow the new element's position. If the array was full, we have no room for the last element so it is propagated to the next cyclic array. Here, it replaces the last element (which can travel further) and the indices are rotated by one, giving the new element index 0. In the worst case, we have B elements to move in the first array and constant time operations for the other n/B subarrays. This is again minized for $B = \Theta(\sqrt{n})$.

Another specialized variant we can develop is an I/O efficient stack[6]: We use 2 buffers of size B in main memory and a pointer to the end of the stack. When both buffers are full, we write the one containing the older elements to disk and use the freed room for new insertions. When both buffers run empty, we refill one with a block from disk. This leads to amortized I/O costs of $\mathcal{O}(1/B)$. Mind that only one buffer is not sufficient: A sequence of B insertions followed by alternating insertions and deletions will incur 1 I/O per operation.

[image] $\Longleftarrow$

Table 2.2 summarizes some of the results found in this chapter by comparing running times for common operations of the presented data structures. Predictably, arrays are better at indexed access whereas linked lists have their strenghts at sequence manipulation at arbitrary positions. However, both basic approaches can implement the special operations needed for stacks and queues roughly equally well. Where both approaches work, arrays are more cache efficient whereas linked lists provide worst case performance guarantees. This is particularly true for all kinds of operations that scan through the sequence; *findNext* is only one example.

[6] 12.1 gives an introduction on our external memory model

Operation	$List$	$UArray$	$hybr.\ Stack$	$hybr.\ Array$	$cycl.\ Array$	explanation of '*'
$[\cdot]$	n	1	$\sqrt{n}$	1	1	
$\lvert\cdot\rvert$	1^*	1	1	1	1	not with inter-list *splice*
first	1	1	1	1	1	
last	1	1	1	1	1	
insert	1	n	n	$\sqrt{n}$	n	
remove	1	n	n	$\sqrt{n}$	n	
pushBack	1	1^*	1	1	1^*	amortized
pushFront	1	n	n	$\sqrt{n}$	1^*	amortized
popBack	1	1^*	1	1	1^*	amortized
popFront	1	n	n	$\sqrt{n}$	1^*	amortized
concat	1	n	n	n	n	
splice	1	n	n	n	n	
findNext,...	n	n^*	n^*	n^*	n^*	cache efficient

Table 2.2: Running times of operations on sequences with n elements. Entries have an implicit $\mathcal{O}(\cdot)$ around them.

Chapter 3

Sorting

The findings on how branch mispredictions affect quicksort are taken from [1]. Super Scalar Sample Sort is described in [2], Multiway Merge Sort is covered in [3], the analysis of duality between prefetching and buffered writing is from [4].

3.1 Quicksort Basics

Sorting is one of the most important algorithmic problems both practically and theoretically. Quicksort is perhaps the most frequently used sorting algorithm since it is very fast in practice, needs almost no additional memory, and makes no assumptions on the distribution of the input.

```
Function quickSort(s : Sequence of Element) : Sequence of Element
    if $|s| \leq 1$ then return s                              // base case
    pick $p \in s$ uniformly at random                         // pivot key
    a := $\langle e \in s : e < p \rangle$                            // (A)
    b := $\langle e \in s : e = p \rangle$                            // (B)
    c := $\langle e \in s : e > p \rangle$                            // (C)
    return concatenation of $quickSort(a)$, b, and $quickSort(c)$
```

Figure 3.1: Quicksort (high-level implementation)

Analysis shows that Quicksort picking pivots randomly will perform an expected number of $\approx 1.4n\log(n)$ comparisons[1]. A proof for this bound is given in the appendix

[1]With other other strategies for selecting a pivot, better constant factors can be achieved: e.g. "median of three" reduces the expected number of comparisons to $\approx 1.2n\log(n)$

```
quickSort           qsort               i->  partition  <-j
3 6 8 1 0 7 2 4 5 9 3 6 8 1 0 7 2 4 5 9 3 6 8 1 0 7 2 4 5 9
1 0 2|3|6 8 7 4 5 9 2 0 1|8 6 7 3 4 5 9 2 6 8 1 0 7 3 4 5 9
0|1|2   4 5|6|8 7 9 1 0|2|5 6 7 3 4|8 9 2 0 8 1 6 7 3 4 5 9
        4|5   7|8|9 0 1| |4 3|7 6 5|8 9 2 0 1|8 6 7 3 4 5 9
0 1 2 3 4 5 6 7 8 9    | |3 4|5 6|7|         j i
                       | |  |5 6| |
                    0 1 2 3 4 5 6 7 8 9
```

Figure 3.2: Execution of both high-level and refined version of $quickSort$. (Figure 3.1 and Figure 3.3) on $\langle 2,7,1,8,2,8,1\rangle$ using the first character of a subsequence as the pivot. The right block shows the first execution of the repeat loop for partitioning the input in $qSort$.

in 12.3. The worst case occurs if all elements are different and we are always so unlucky to pick the largest or smallest element as a pivot and results in $\Theta(n^2)$ comparisons. As the number of executed instructions and cache faults is proportional to the number of comparisons, this is (at least in theory) a good measure for the total runtime of Quicksort.

3.2 Refined Quicksort

Figure 3.3 gives pseudocode for an array based quicksort that works in-place and uses several implementation tricks that make it faster and very space efficient.

To make a recursive algorithm compatible to the requirement of in-place sorting of an array, quicksort is called with a reference to the array and the range of array indices to be sorted. Very small subproblems with size up to n_0 are sorted faster using a simple algorithm like insertion sort[2]. The best choice for the constant n_0 depends on many details of the machine and the compiler. Usually one should expect values around 10–40. An efficient implementation of Insertion Sort is given in the appendix in 12.4.

The pivot element is chosen by a function $pickPivotPos$ that we have not specified here. The idea is to find a pivot that splits the input more accurately than just choosing a random element. A method frequently used in practice chooses the median ('middle') of three elements. An even better method would choose the exact median of a random sample of elements.

The repeat-until loop partitions the subarray into two smaller subarrays. Elements

[2]Some books propose to leave small pieces unsorted and clean up at the end using a single insertion sort that will be fast as the sequence is already almost sorted. Although this nice trick reduces the number of instructions executed by the processor, our solution is faster on modern machines because the subarray to be sorted will already be in cache.

```
//Sort the subarray $a[\ell..r]$
**Procedure** $qSort(a$ : **Array of** $Element;\ \ell, r : \mathbb{N})$
    **while** $r - \ell \geq n_0$ **do**                                   // Use divide-and-conquer
        $j := pickPivotPos(a, l, r)$
        $swap(a[\ell], a[j])$                                  // Helps to establish the invariant
        $p := a[\ell]$
        $i := \ell;\ j := r$
        **repeat**                                             // $a$:  $\ell$      $i\rightarrow$  $\leftarrow j$      $r$
            **invariant** 1: $\forall i' \in \ell..i-1$: $a[i'] \leq p$          // $a$: |$\forall \leq p$|                    |
            **invariant** 2: $\forall j' \in j+1..r$ $a[j'] \geq p$           // $a$: |                    |$\forall \geq p$|
            **invariant** 3: $\exists i' \in i..r$     : $a[i'] \geq p$          // $a$: |        |     $\exists \geq p$      |
            **invariant** 4: $\exists j' \in \ell..j$     : $a[j'] \leq p$          // $a$: |     $\exists \leq p$      |        |
            **while** $a[i] < p$ **do** $i$++                          // Scan over elements (A)
            **while** $a[j] > p$ **do** $j$−−                           // on the correct side (B)
            **if** $i \leq j$ **then** $swap(a[i], a[j])$; $i$++ ; $j$−−
        **until** $i > j$                                          // Done partitioning
        **if** $i < \frac{l+r}{2}$ **then** qSort(a,$\ell$,j); $\ell := j$
        **else**    qSort(a,i,r) ; $r := i$
    insertionSort($a[l..r]$)                                    // faster for small $r - l$
```

Figure 3.3: Refined quicksort

equal to the pivot can end up on either side or between the two subarrays. Since quicksort spends most of its time in this partitioning loop, its implementation details are important. Index variable i scans the input from left to right and j scans from right to left. The key invariant is that elements left of i are no larger than the pivot whereas elements right of j are no smaller than the pivot. Loops (A) and (B) scan over elements that already satisfy this invariant. When $a[i] \geq p$ and $a[j] \leq p$, scanning can be continued after swapping these two elements. Once indices i and j meet, the partitioning is completed. Now, $a[\ell..j]$ represents the left partition and $a[i..r]$ represents the right partition. This sounds simple enough but for a correct and fast implementation, some subtleties come into play.

To ensure termination, we verify that no single piece represents all of $a[\ell..r]$ even if p is the smallest or largest array element. So, suppose p is the smallest element. Then loop A first stops at $i = \ell$; loop B stops at the last occurrence of p. Then $a[i]$ and $a[j]$ are swapped (even if $i = j$) and i is incremented. Since i is never decremented, the right partition $a[i..r]$ will not represent the entire subarray $a[\ell..r]$. The case that p is the largest element can be handled using a symmetric argument.

The scanning loops A and B are very fast because they make only a single test. On the first glance, that looks dangerous. For example, index i could run beyond the right boundary r if all elements in $a[i..r]$ were smaller than the pivot. But this cannot happen. Initially, the pivot is in $a[i..r]$ and serves as a sentinel that can stop Scanning Loop A. Later, the elements swapped to the right are large enough to play the role of a sentinel. Invariant 3 expresses this requirement that ensures termination of Scanning Loop A. Symmetric arguments apply for Invariant 4 and Scanning Loop B.

Our array quicksort handles recursion in a seemingly strange way. It is something like "semi-recursive". The smaller partition is sorted recursively, while the larger partition is sorted iteratively by adjusting ℓ and r. This measure ensures that recursion can never go deeper than $\lceil \log \frac{n}{n_0} \rceil$ levels. Hence, the space needed for the recursion stack is $\mathcal{O}(\log n)$. Note that a completely recursive algorithm could reach a recursion depth of $n - 1$ so the space needed for the recursion stack could be considerably larger than for the input array itself.

3.3 Lessons from experiments

We now run Quicksort on real machines to check if it behaves differently than our analysis on the RAM model predicted. We will see that modern hardware architecture can have influence on the runtime and try to find algorithmic solutions to these problems.

In the analysis, we saw that the number of comparisons determines the runtime of Quicksort. On a real machine a comparison and the corresponding if-clause are mapped to a branch instruction. In modern processors with long execution pipelines and superscalar execution, dozens of subsequent instructions are executed in parallel to achieve a high

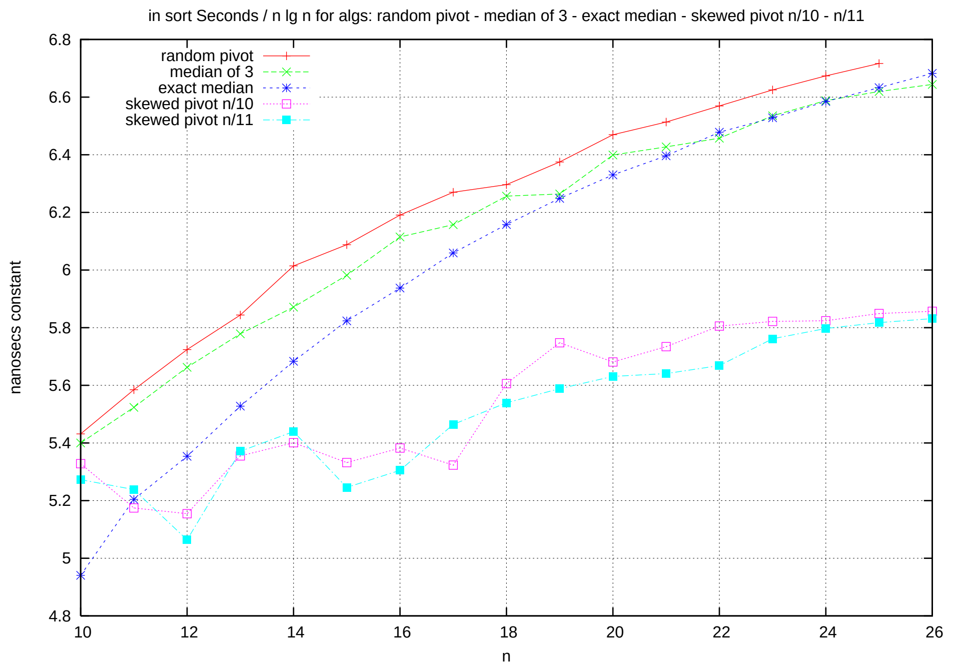


Figure 3.4: Runtime for Quicksort using different strategies for pivot selection

peak throughput. To keep the pipeline filled, the outcome of each branch is predicted by the hardware (based on several possible heuristics). When a branch is mispredicted, much of the work already done on the instructions following the predicted branch direction turns out to be wasted. Therefore, ingenious and very successful schemes have been devised to accurately predict the direction a branch takes . Unfortunately, we are facing a dilemma here. Information theory tells us that the optimal number of $n \log n$ element comparisons for sorting can only be achieved if each element comparison yields one bit of information, i.e., there is a 50% chance for the branch to take either direction. In this situation, even the most clever branch prediction algorithm is helpless. A painfully large number of branch mispredictions seems to be unavoidable.

Figure 3.4 compares the runtime of Quicksort implementations using different strategies of selecting a pivot. Together with standard techniques (random, median of three, ...) α-skewed pivots are used, i.e., pivots which have a rank of αn. Theory suggests large constant factors in execution time for these strategies with $\alpha \ll \frac{1}{2}$ compared to a perfect median. In practice, Figure 3.4 shows that these implementations are actually *faster* than those that use an (approximated) median as pivot.

An explanation for this can be found in Figure 3.5: A pivot with rank close to $n/2$ produces many more branch mispredictions than a pivot that separates the sequence in two parts of very different sizes. The costs to flush the entire instruction pipeline outweigh

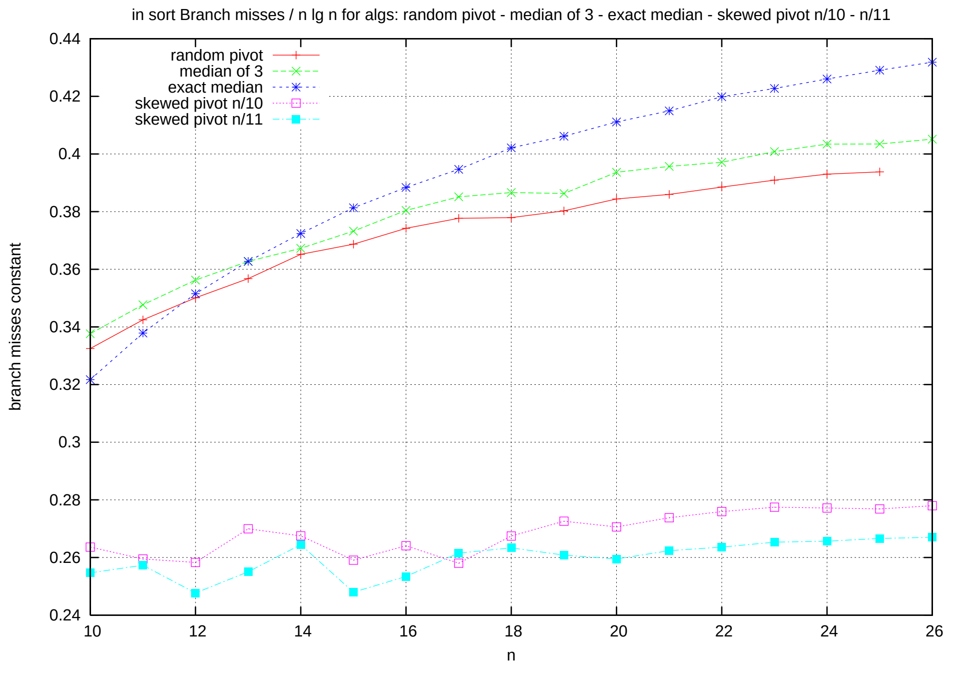


Figure 3.5: Number of branch mispredictions for Quicksort using different strategies for pivot selection

the fewer partition steps of these variants.

3.4 Super Scalar Sample Sort

We now study a sorting algorithm which is aware of hardware phenomenons like branch mispredictions or superscalar execution. This algorithm is called Super Scalar Sample Sort (SSSS) which is an engineered version of Sample Sort which in turn is a generalization of Quicksort.

```
Function sampleSort($e = \langle e_1, \ldots, e_n \rangle$ : Sequence of Element, $k : \mathbb{Z}$) : Sequence of Element
    if $n/k$ is "small" then return smallSort($e$)                // base case, e.g. quicksort
    let $\langle S_1, \ldots, S_{ak-1} \rangle$ denote a random sample of $e$
    sort $S$                // or at least locate the elements whose rank is a multiple of $a$
    $\langle s_0, s_1, s_2, \ldots, s_{k-1}, s_k \rangle := \langle -\infty, S_a, S_{2a}, \ldots, S_{(k-1)a}, \infty \rangle$        // determine splitters
    for $i := 1$ to $n$ do
        find $j \in \{1, \ldots, k\}$ such that $s_{j-1} < e_i \leq s_j$
        place $e_i$ in bucket $b_j$
    return concatenate($sampleSort(b_1), \ldots, sampleSort(b_k)$)
```

Figure 3.6: Standard Sample Sort

Our starting point is ordinary sample sort. Fig. 3.6 gives high level pseudocode. Small inputs are sorted using some other algorithm like quicksort. For larger inputs, we first take a sample of $s = ak$ randomly chosen elements. The *oversampling factor* a allows a flexible tradeoff between the overhead for handling the sample and the accuracy of splitting. Our splitters are those elements whose rank in the sample is a multiple of a. Now each input element is located in the splitters and placed into the corresponding bucket. The buckets are sorted recursively and their concatenation is the sorted output. A first advantage of Sample Sort over Quicksort is the number of $\log_k n$ recursion levels which is by a factor $\log_2 k$ smaller than the recursion depth of Quicksort $\log_2 n$. Every element is moved once during each level, resulting in less cache faults for Sample Sort. However, this alone does not resolve the central issue of branch mispredictions and only comes to bear for very large inputs.

SSSS is an implementation strategy for the basic sample sort algorithm. All sequences are represented as arrays. More precisely, we need two arrays of size n. One for the original input and one for temporary storage. The flow of data between these two arrays alternates in different levels of recursion. If the number of recursion levels is odd, a final copy operation makes sure that the output is in the same place as the input. Using an array

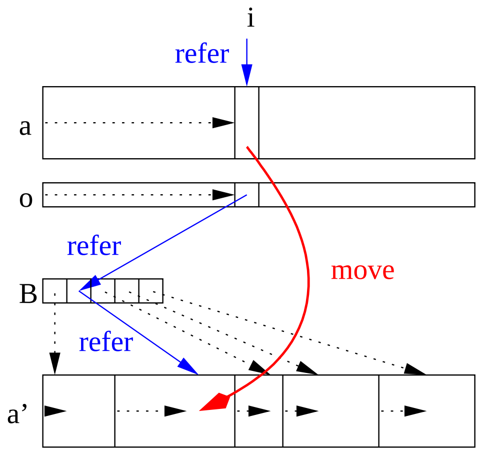
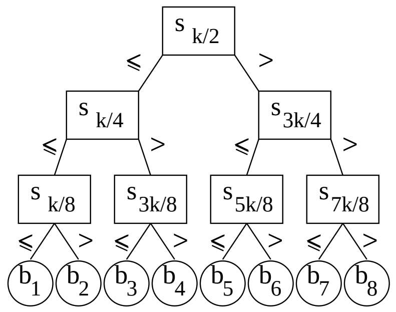


Figure 3.7: Two-pass element distribution in Super Scalar Sample Sort

of size n to accommodate all buckets means that we need to know exactly how big each bucket is. In radix sort implementations this is done by locating each element twice. But this would be prohibitive in a comparison based algorithm. Therefore we use an additional auxiliary array, o, of n *oracles* – $o(i)$ stores the bucket index for element e_i. A first pass computes the oracles and the bucket sizes. A second pass reads the elements again and places element e_i into bucket $b_{o(i)}$. This two pass approach incurs costs in space and time. However these costs are rather small since bytes suffice for the oracles and the additional memory accesses are sequential and thus can almost completely be hidden via software or hardware prefetching[3]. In exchange we get simplified memory management, no need to test for bucket overflows. Perhaps more importantly, decoupling the expensive tasks of finding buckets and distributing elements to buckets facilitates software pipelining by the compiler and prevents cache interferences of the two parts. This optimization is also known as *loop distribution*.

Theoretically the most expensive and algorithmically the most interesting part is how to locate elements with respect to the splitters. Fig. 3.8 gives pseudocode and a picture for this part. Assume k is a power of two. The splitters are placed into an array t such that they form a complete binary search tree with root $t_1 = s_{k/2}$. The left successor of t_j is stored at t_{2j} and the right successor is stored at t_{2j+1}. This is the arrangement well known from binary heaps but used for representing a search tree here. To locate an element a_i, it suffices to travel down this tree, multiplying the index j by two in each level and adding one if the element is larger than the current splitter. This increment is the only instruction that depends on the outcome of the comparison. Some architectures

[3]This is true as long as we can accommodate one buffer per bucket in the cache, limiting the parameter k. Other limiting factors are the size of the TLB (translation lookaside buffer, storing mappings of virtual to physical memory addresses) and $k \leq 256$ if we want to store the bucket indices in one byte



```
$t$:= $\langle s_{k/2}, s_{k/4}, s_{3k/4}, s_{k/8}, s_{3k/8}, s_{5k/8}, s_{7k/8}, \ldots\rangle$
for $i$ := 1 to $n$ do                    // locate each element
    $j$:= 1                   // current tree node := root
    repeat $\log k$ times              // will be unrolled
        $j$:= $2j + (a_i > t_j)$              // left or right?
    $j$:= $j - k + 1$                          // bucket index
    $|b_j|$++                            // count bucket size
    $o_i$:= $j$                              // remember oracle
```

Figure 3.8: Finding buckets using implicit search trees. The picture is for $k = 8$. We adopt the convention from C that "$x > y$" is one if $x > y$ holds, and zero else.

```
     cmp.gt p7=r1,r2            cmp.gt p6=r1,r2
(p7) br.cond .label        (p6) add r3=4,r3
     add r3=4,r3
     .label:
```

Table 3.1: Translation of `if(r1 > r2) r3 := r3 + 4` with branches (left) and predicated instructions (right)

like IA-64 have predicated arithmetic instructions that are only executed if the previously computed condition code in the instruction's predicate register is set. Others at least have a conditional move so that we can compute j:= $2j$ and then, speculatively, j':= $j+1$. Then we conditionally move j' to j. The difference between such predicated instructions and ordinary branches is that they do not affect the instruction flow and hence cannot suffer from branch mispredictions.

Experiments (conducted on an Intel Itanium processor with Intel's compiler to have support for predicated instructions and software pipelining) show that our implementation of SSSS outperforms two well known library implementations for sorting. In the Experiment 32 bit random integers in the range $[0, 10^9]$ were sorted[4].

For this first version of SSSS, several improvements are possible. For example, the current implementation suffers from many identical keys. This could be fixed without much overhead: If $s_{i-1} < s_i = s_{i+1} = \cdots = s_j$ (identical splitters are an indicator for many identical keys), $j > i$, change s_i to $s_i - 1$. Do not recurse on buckets b_{i+1},…,b_j – they all contain identical keys. Now SSSS can even profit from an input like this.¡

Another disadvantage compared to quicksort is that SSSS is not inplace. One could make it almost inplace however. This is most easy to explain for the case that both input

[4]note that the algorithm's runtime is not influenced by the distribution of elements, so a random distribution of elements is no unfair advantage for SSSS

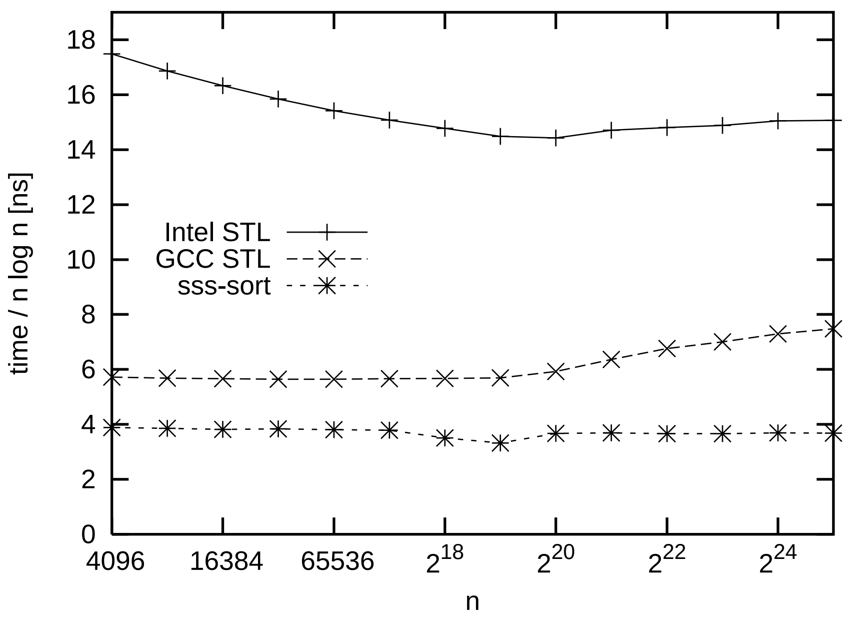


Figure 3.9: Runtime for sorting using SSSS and other algorithms

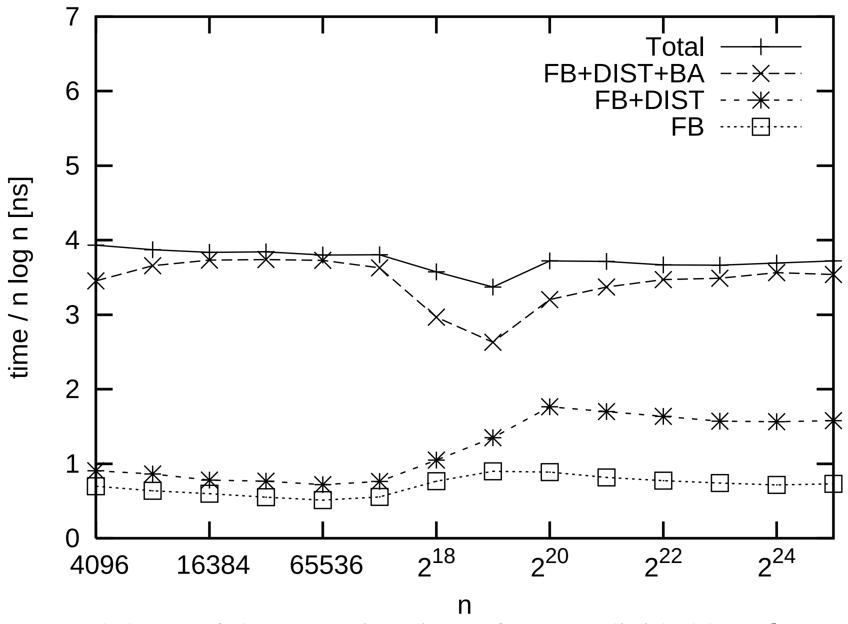


Figure 3.10: Breakdown of the execution time of SSSS (divided by $n \log n$) into phases. "FB" denotes the finding of buckets for the elements, "DIST" the distribution of the elements to the buckets, "BA" the base sorting routines. The remaining time is spent in finding the splitters etc.

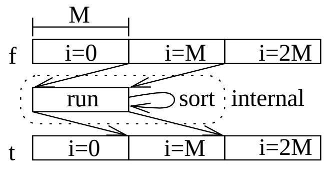


Figure 3.11: Run formation

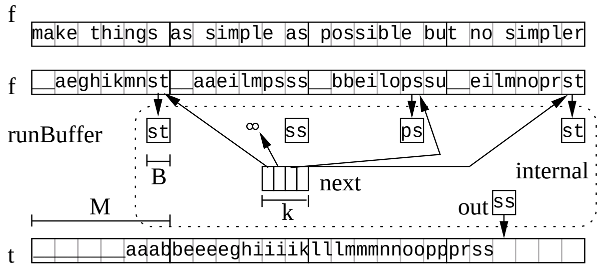

f
make things as simple as possible but no simpler
f
__aeghikmnst__aaeilmpsss__bbeilopssu__eilmnoprst
runBuffer
st
∞
ss
ps
st
B
next
internal
k
M
out ss
t
aaabbeeeeghiiiiklllmmmnnooppprss

Figure 3.12: Example of 4-way merging with $M = 12$, $B = 2$

and output are a sequence of blocks (compare chapter 2). Sampling takes sublinear space and time. Distribution needs at most $2k$ additional blocks and can otherwise recycle freed blocks of the input sequence. Although software pipelining may be more difficult for this distribution loop, the block representation facilitates a single pass implementation without the time and space overhead for oracles so that good performance may be possible. Since it is possible to convert inplace between block list representation and an array representation in linear time, one could actually attempt an almost inplace implementation of SSSS.

3.5 Multiway Merge Sort

We will now study another algorithm based on the concept of Merge Sort which is especially well suited for external sorting. For external algorithms, an efficient sorting subroutine is even more important than for main memory algorithms because one often tries to avoid random disk accesses by ordering the data, allowing a sequential scan.

Multiway Merge Sort first splits the data into $\lceil n/M \rceil$ *runs* which fit into main memory where they are sorted. We merge these runs until only one is left. Instead of ordinary 2-way-merging, we merge $k := M/B$ runs in a single pass resulting in a smaller number of merge phases. We only have to keep one block (containing the currently smallest elements) per run in main memory. We maintain a priority queue containing the smallest elements of each run in the current merging step to efficiently keep track of the overall smallest element.

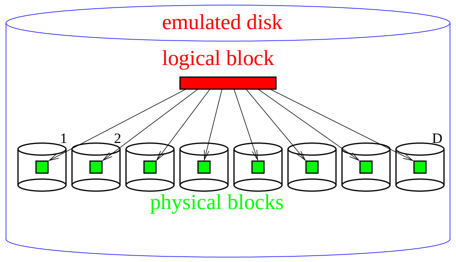


Figure 3.13: Striping: one logical block consists of D physical blocks.

Every element is read/written twice for forming the runs (in blocks of size B) and twice for every merging phase. Access granularity is blocks. This leads to the following (asymptotically optimal) total number of I/Os:

$$\frac{2n}{B}\left(1+\lceil\log_k \#runs\rceil\right) = \frac{2n}{B}\left(1+\left\lceil\log_{M/B}\frac{n}{M}\right\rceil\right) := \mathrm{sort}(n) \tag{3.1}$$

Let us consider the following realistic parameters: $B = 2\mathrm{MB}$, $M = 1\mathrm{GB}$. For inputs up to a size of $n = 512\mathrm{GB}$, we get only one merging phase! In general, this is the case if we can store $\lceil n/M\rceil$ buffers (one for each run) of size B in internal memory (i.e., $n \leq M^2/B$). Therefore, only one additional level can increase the I/O volume by 50%.

3.6 Sorting with parallel disks

We now consider a system with D disks. There are different ways to model this situation (see Figure 3.15) but all have in common that in one I/O step we can fetch up to D blocks so we can hope to reduce the number of I/Os by this factor:

$$\frac{2n}{BD}\left(1+\left\lceil\log_{M/B}\frac{n}{M}\right\rceil\right) \tag{3.2}$$

An obvious idea to handle multiple disks is the concept of striping: An emulated disk contains logical blocks of size DB consisting of one physical block per disk. The algorithms for run formation and writing the output can be used unchanged on this emulated disk. For the merging step however, we have to be careful: With larger (logical) blocks the number of I/Os becomes:

$$\frac{2n}{BD}\left(1+\left\lceil\log_{M/BD}\frac{n}{M}\right\rceil\right) \tag{3.3}$$

The algorithm will move more data in one I/O step (compared to the setup with one disk) but requires a possibly deeper recursion level. In practice, this can make the difference between one or two merge phases. We therefore have to work on the level of physical

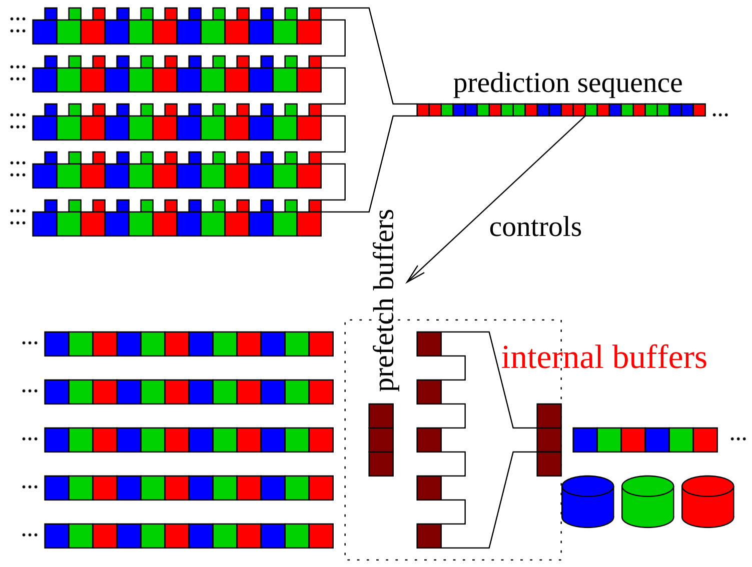


Figure 3.14: The smallest element of each block triggers fetch.

blocks to achieve optimal constant factors. This comes with the necessity to distribute the runs in an intelligent way among the disks and to find a schedule for fetching blocks into the merger.

For starters, it is necessary to find out which block on which disk will be required next when one of the merging buffers runs out of elements. This can be computed offline when all runs are formed: A block is required the moment its smallest element is required. We can therefore sort the set of all smallest elements to get a *prediction sequence*.

To be able to refill the merge buffers in time we maintain prefetch buffers which we fill (if necessary) while the merging of the current elements takes place. This allows parallel access of next due blocks and helps for an efficiency near 1 (i.e. fetching D blocks in one I/O step). How many prefetch buffers should we use?

We first approach this question by using a simplified model ((a) in figure 3.15) where we have D read-/write-heads on one large disk. Here, D prefetch buffers suffice: In one I/O-step we can refill all buffers, transferring D blocks of size B which leads to a total (optimal) number of I/Os as in equation 3.2.

If we replace the multihead model with D independent disks (each with its own right/write-head) we get a more realistic model. But now D prefetch buffers seem too few as it is possible that all next k blocks reside on the same disk which would need that many I/O steps for filling the buffers while the other disks lie idle, leading to a non-optimal efficiency.

A first solution is to increase the number of prefetch buffers to kD. But that would leave us with less space for merge buffers, write buffers and other data that we have to

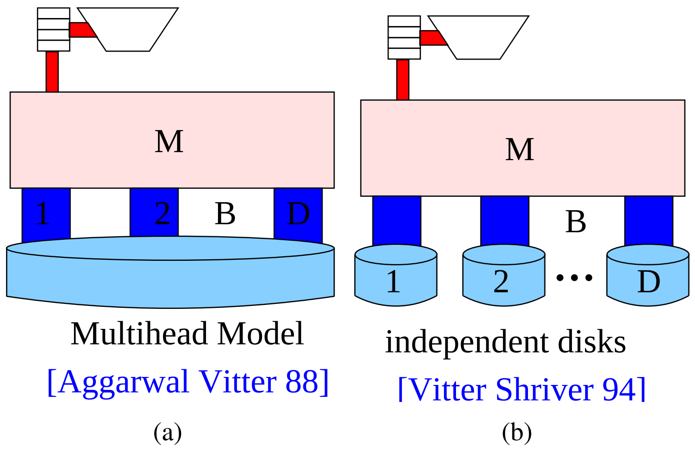


Figure 3.15: Different models for systems with several disks

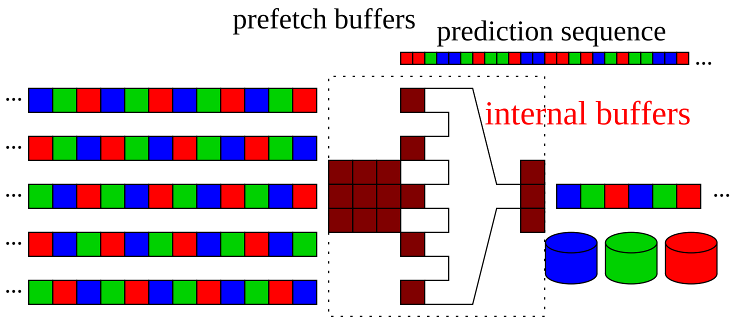


Figure 3.16: Distribution of runs using randomized cycling.

keep in main memory.

Instead, we use the *randomized cycling* pattern while forming runs: For every run j, we map block i to $\pi_j(i \bmod D)$ for a random permutation π_j. This makes the event of getting a “difficult“ distribution highly unlikely.

With a naive prefetching strategy and random cycling, we can achieve a good performance with only $\mathcal{O}(D \log D)$ buffers. Is it possible to even reduce this to $\mathcal{O}(D)$?

The prefetching strategy leaves more room for optimization. The naive approach fetches in one I/O-step the next blocks from the prediction sequence until all free buffers are filled or a disk would be accessed twice.

The problem is now to find an optimal offline prefetching schedule (offline, because the prediction sequence yields the order in which the blocks on each disk are needed). For the solution, we make a digression to online buffered writing and use the principle of *duality* to transform our result here into a schedule for offline prefetching.

In online buffered writing, we have a sequence Σ of blocks to be written to one of D

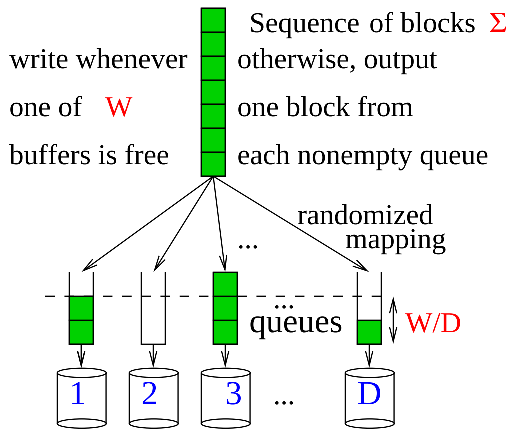


Figure 3.17: The online buffered writing problem and its optimal solution.

disks. We also have W buffers, W/D for each disk. It can be shown that randomized, equally distributed writing to one of the free buffers and outputting one block of each queue if no capacity is left is an optimal strategy and achieves an expected efficiency of $1 - \mathrm{o}(D/W)$.

We can now *reverse* this process to obtain an optimal offline prefetching algorithm called *lazy prefetching*: Given the prediction sequence Σ, we calculate the optimal online writing sequence T for Σ^R and use T^R as prefetching schedule. Note that we do not use the distribution between the disks the writing algorithm produces and that the random distribution during the writing process corresponds to random cycling.

Figure 3.19 gives an example in which our optimal strategy yields a better result than a naive prefetching approach: The upper half shows the result of the example schedule from 3.18 created by inverting a writing schedule. The bottom half shows the result of naive prefetching, always fetching the next block from every disk in one step (as long as there are free buffers).

3.7 Internal work of Multiway Mergesort

Until now we have only regarded the number of I/Os. In fact, when running with several disks our sorting algorithm can very well be compute bound, i.e. prefetching D new blocks requires less time than merging them. We use the technique of *overlapping* to minimize wait time for whichever task is bounding our algorithm in a given environment. Take the following example on run formation (i denotes a run):

Thread A: Loop { wait-*read* i; *sort* i; post-*write* i};

Thread B: Loop { wait-*write* i; post-*read* i+2};

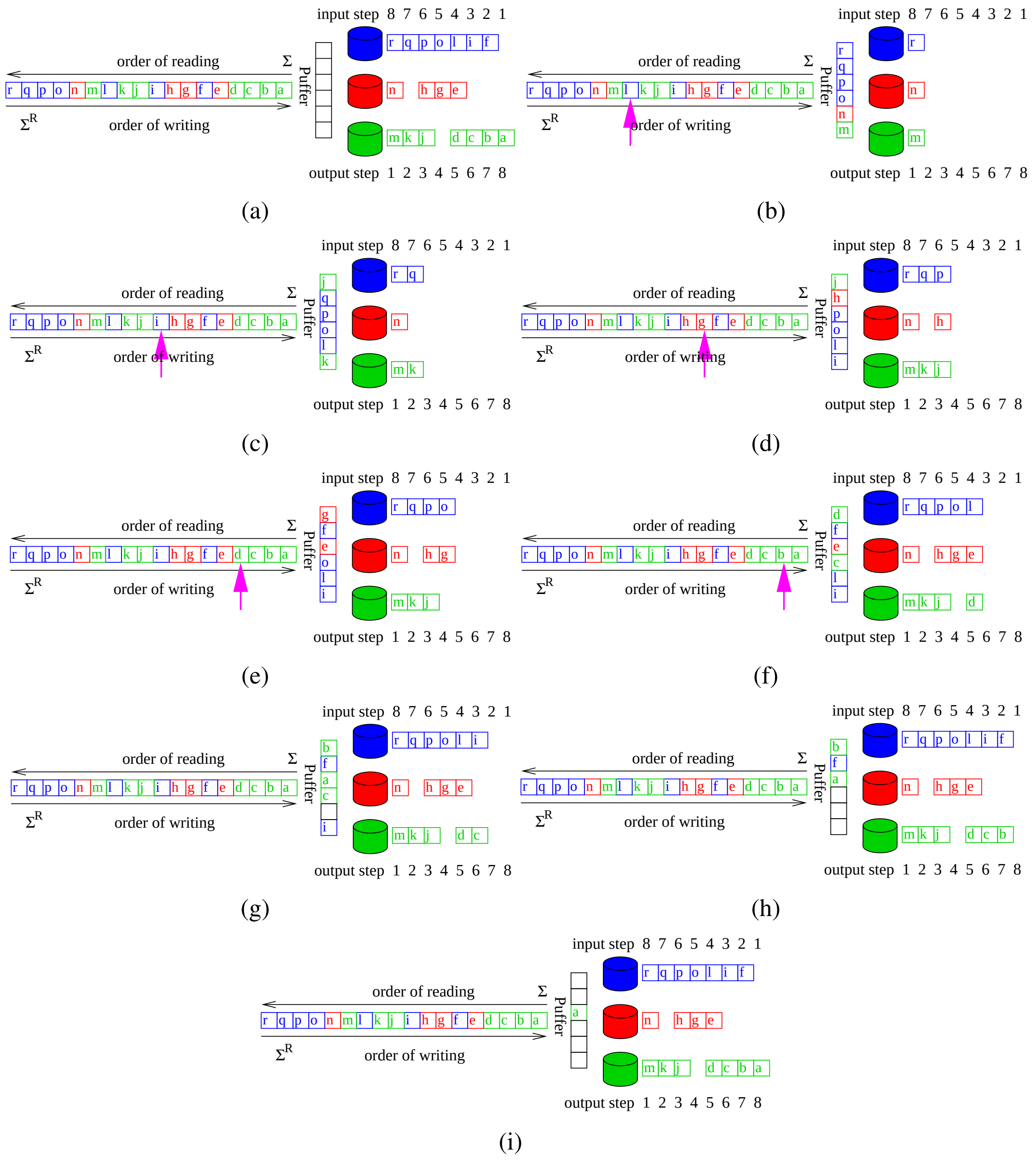

input step 8 7 6 5 4 3 2 1
order of reading
Σ
Puffer
Σ^R
order of writing
output step 1 2 3 4 5 6 7 8
r q p o n m l k j i h g f e d c b a
(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)

Figure 3.18: Example: Optimal randomized online writing

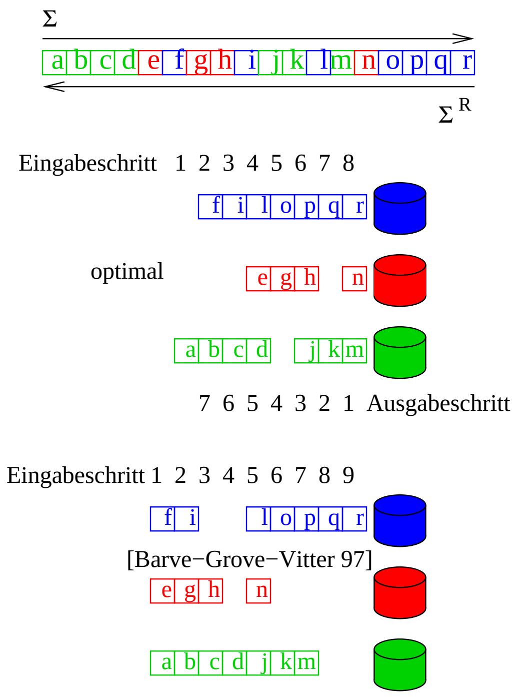

Σ
a b c d e f g h i j k l m n o p q r
Σ R
Eingabeschritt 1 2 3 4 5 6 7 8
f i l o p q r
optimal
e g h n
a b c d j k m
7 6 5 4 3 2 1 Ausgabeschritt
Eingabeschritt 1 2 3 4 5 6 7 8 9
f i l o p q r
[Barve−Grove−Vitter 97]
e g h n
a b c d j k m

Figure 3.19: Example: resulting offline reading schedule

During initialization, runs 1 and 2 are read, i is set to 1. Thread A sorts runs in memory and writes them to disk. Thread B will wait until run i is finished (and thread A works on $i+1$) and reads the next run $i+2$ into the freed space. The thread doing the more intense work will never wait for the other one.

A similar result can be achieved during the merging step but this is considerably more complicated and beyond the scope of this course.

As internal work influences running time, we need a fast solution for the most compute intense step during merging: A *Tournament Tree* (or Loser Tree) is a specialized data structure for finding the smallest element of all runs. For $k = 2^K$, it is a complete binary tree with K levels, where each leaf contains the currently smallest element of one run. Each internal node contains the 'loser' (i.e., the greater) of the 'competition' between its two child nodes. Above the root node, we store the global winner along with a pointer to the corresponding run. After writing this element to the output buffer, we simply have to move the next element of its run up until there is a new global winner. Compared to general purpose data structures like binary heaps, we get exactly $\log k$ comparisons (no hidden constant factors). Similar to the implicit search trees we used for Sample Sort, Tournament Trees can be implemented as arrays where finding the parent node simply maps to an index shift to the right. The inner loop for moving from leaf to root can be unrolled and contains predictable load instructions and index computations allowing

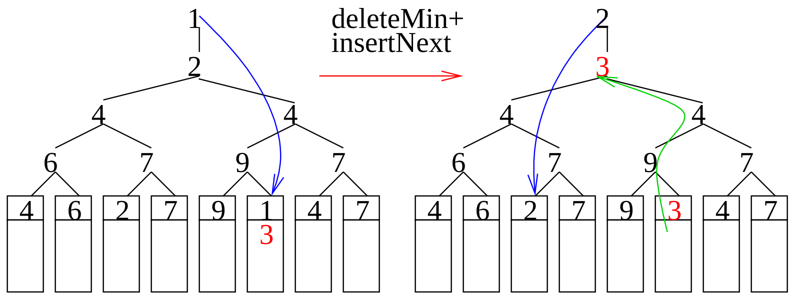


Figure 3.20: A tournament tree

```
for (int i=(winnerIndex+kReg)>>1; i>0; i>>=1){
  currentPos = entry + i;
  currentKey = currentPos->key;
  if (currentKey < winnerKey) {
    currentIndex       = currentPos->index;
    currentPos->key    = winnerKey;
    currentPos->index = winnerIndex;
    winnerKey          = currentKey;
    winnerIndex        = currentIndex;}}
```

Figure 3.21: Inner loop of Tournament Tree computation

exploitation of instruction parallelism.

3.8 Experiments

Experiments on Multiway Merge Sort were performed in 2001 on a $2 \times$ 2GHz Xeon $\times$ 2 threads machine (Intel IV with Netburst) with several 66 MHz PCI-buses, 4 fast IDE controllers (Promise Ultra100 TX2) and 8 fast IDE disks (IBM IC35L080AVVA07). This inexpensive (mid 2002) setup gave a high I/O-bandwidth of 360 MB/s. The keys consisted of 16 GByte random 32 bit integers, run size was 256 MByte, block size B was 2MB (if not otherwise mentioned).

Figure 3.22 shows the running time for different element sizes (for a constant total data volume of 16 GByte). The smaller the elements, the costlier becomes internal work, especially during run formation (there are more elements to sort). With a high I/O throughput and intelligent prefetching algorithms, I/O wait time never makes up for more than half of the total running time. This proves the point that overlapping and tuning of internal work are important.

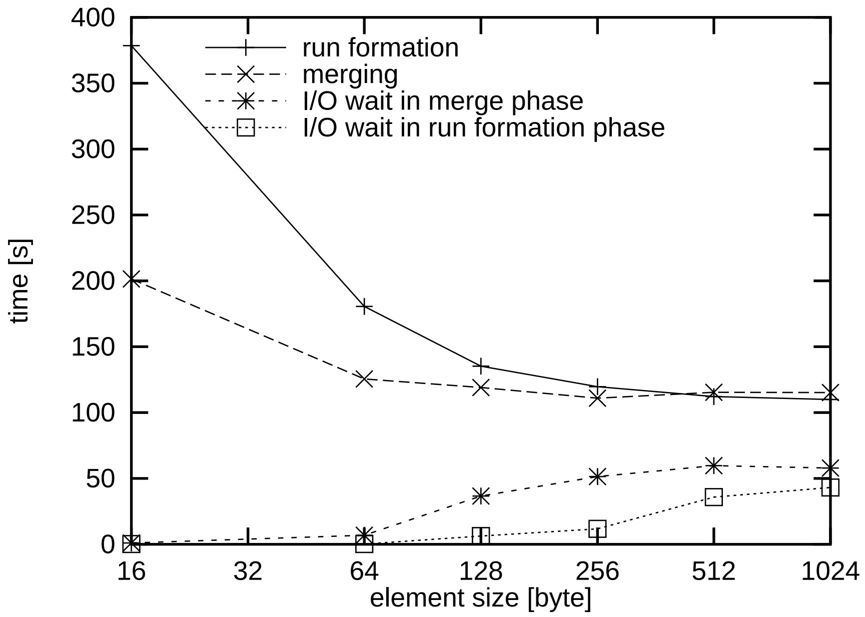


Figure 3.22: Multiway Merge Sort with different element sizes

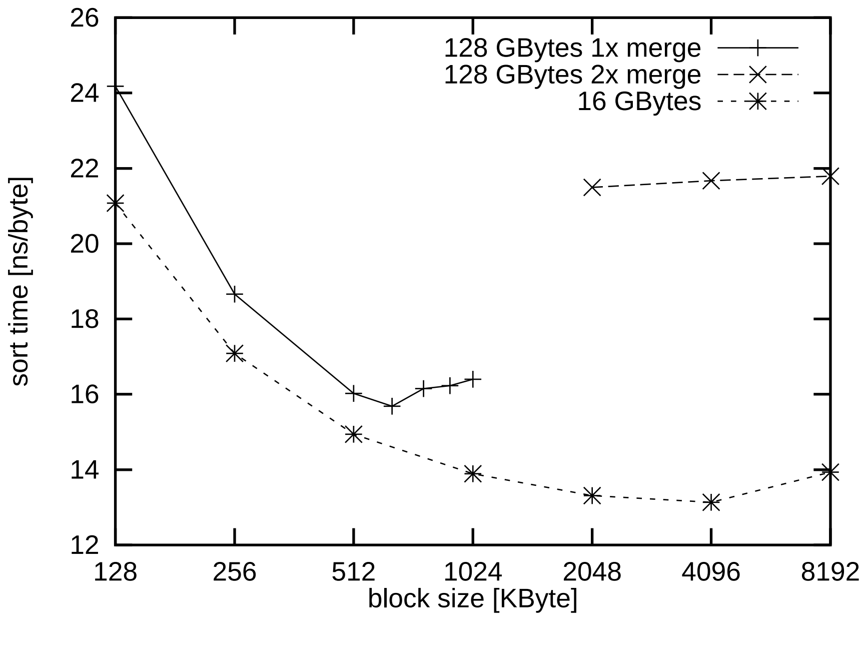


Figure 3.23: Performance using different block sizes

What is a good block size B? An intuitive approach would link B to the size of a physical disk block. However, figure 3.23 shows that B is no technology constant but a tuning parameter: A larger B is better (as it reduces the amortized costs of $\mathcal{O}(1/B)$ I/Os per element), as long as the resulting smaller k still allows for a single merge phase (see curve for 128GB).

Chapter 4

Priority Queues

The material on external priority queues was first published in [5].

4.1 Introduction

Priority queues are an important data structure for many applications, including: shortest path search (Dijkstra's Algorithm), sorting, construction of mimimum spanning trees, branch and bound search, discrete event simulaton and many more. While the first examples are widely known and also covered in other chapters, we give a short explanation of the latter two applications: The *best first branch-and-bound* approach to optimization elements are partial solutions of an optimization problem and the keys are optimistic estimates of the obtainable solution quality. The algorithm repeatedly removes the best looking partial solution, refines it, and inserts zero or more new partial solutions. In a *discrete event simulation* one has to maintain a set of pending events. Each event happens at some scheduled point in time and creates zero or more new events scheduled to happen at some time in the future. Pending events are kept in a priority queue. The main loop of the simulation deletes the next event from the queue, executes it, and inserts newly generated events into the priority queue.
Our (non-addressable) priority queue M needs to support the following operations:

Procedure $\mathit{build}(\{e_1, \ldots, e_n\})$ $\quad M := \{e_1, \ldots, e_n\}$
Procedure $\mathit{insert}(e)$ $\quad M := M \cup \{e\}$
Function *deleteMin* $\quad e := \min M$; $\quad M := M \setminus \{e\}$; $\quad$ **return** e

There are different approaches to implementing priority queues but most of them resort to an implicit or explicit tree representation which is heap-ordered[1]: If w is a successor of v, the key stored in w is not greater than the key stored in v. This way, the overall smallest key is stored in the root.

[1]In 4.4 we will see implementations using a whole forest of heap-ordered trees

4.2 Binary Heaps

Priority queues are often implemented as binary heaps, stored in an array h where the successors for an element at position i are stored at positions $2i$ and $2i+1$. This is an implicit representation of a near-perfect binary tree which only might lack some leafs in the bottom level. We require that this array is *heap-ordered*, i.e.,

$$\text{if } 2 \leq j \leq n \text{ then } h[\lfloor j/2 \rfloor] \leq h[j].$$

Binary Heaps with arrays are bounded in space, but they can be made unbounded in the same way as bounded arrays are made unbounded. Asuming non-hierarchical memory, we can implement all desired operations in an efficient manner:

An $insert$ puts a new element e tentatively at the end of the heap h, i.e., e is put at a leaf of the tree represented by h.[reformulated:more rendundancy, less ambiguity] ⟸ Then e is moved to an appropriate position on the path from the leaf $h[n]$ to the root.

```
**Procedure** *insert*$(e : Element)$
    **assert** $n < w$
    $n{+}{+}$ ; *h[n]:= e*
    $siftUp(n)$
```

where $siftUp(s)$ moves the contents of node s towards the root until the heap property[was heap condition] holds. ⟸

```
**Procedure** $siftUp(i : \mathbb{N})$
    **assert** the heap property holds except maybe for $j = i$
    **if** $i = 1 \vee h[\lfloor i/2 \rfloor] \leq h[i]$ **then return**
    **assert** the heap property holds except for $j = i$
    *swap*$(h[i], h[\lfloor i/2 \rfloor])$
    **assert** the heap property holds except maybe for $j = \lfloor i/2 \rfloor$
    $siftUp(\lfloor i/2 \rfloor)$
```

Since $siftUp$ will potentially move the element up to the root and perform a comparison on every level, $insert$ takes $\mathcal{O}(\log n)$ time. On avergage, a constant number of comparisons will suffice.

$deleteMin$ in its basic form replaces the root with the leftmost leaf which is then sifted down (analogously to $siftUp$), resulting in $2 \log n$ key comparisons (on every level, we have to find the minimum of three elements). The bottom-up heuristic suggests an improvement for that operation: The hole left by the removed minimum is “sifted down“ to a leaf (requiring only one comparison per level between the two successors of the hole), is only now replaced by the rightmost leaf which is then sifted up again (costing constant time on average, like insertion).

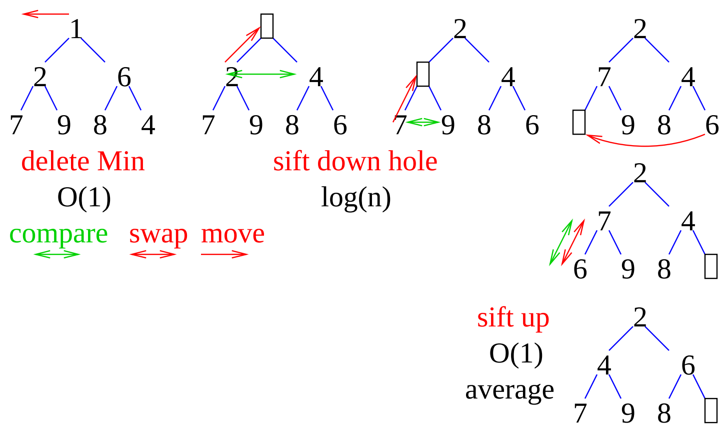


Figure 4.1: The bottom-up heuristic

```
int i=1, m=2, t = a[1];
m += (m != n  &&  a[m] > a[m + 1]);
if (t > a[m]) {
  do { a[i] = a[m];
       i    = m;
       m    = 2*i;
       if (m > n) break;
       m += (m != n  &&  a[m] > a[m + 1]);
  } while (t > a[m]);
  a[i] = t;}
```

Figure 4.2: An efficient version of standard `deleteMin`

This approach should be a factor two faster then the naive implementation. However, if the latter is programmed properly (see figure 4.2), there are no measureable differences in runtime: The given implementation has $\log n$ comparisons more than bottom-up but these are stop criterions for the loop and thus easy to handle for branch prediction. Notice how the increment of m avoids branches within the loop.

[siftDown mit logn + loglogn Vergleichen] $\Longleftarrow$

For the initial construction of a heap there are also two competing approaches: *buildHeapBackwards* moves from the leaves to the root, ensuring the heap property on every level. *buildHeapRecursive* first fixes this properties recursively on the two subtrees of the root and then sifts the remaining node down. Here, we have the reverse situation compared to *deleteMin*: Both algorithms asymptotically cost $\mathcal{O}(n)$ time but on a real machine, the recursive variant is faster by a factor two: It is more cache efficient. Note, that a subtree with B leaves and therefore $\log B$ levels can be stored in $B \log B$ blocks of

```
**Procedure** *buildHeapBackwards*
    **for** $i := \lfloor n/2 \rfloor$ **downto** $1$ **do** $siftDown(i)$

**Procedure** $buildHeapRecursive(i : \mathbb{N})$
    **if** $4i \leq n$ **then**
        $buildHeapRecursive(2i)$
        $buildHeapRecursive(2i + 1)$
    $siftDown(i)$
```

Figure 4.3: Two implementations for $buildHeap$

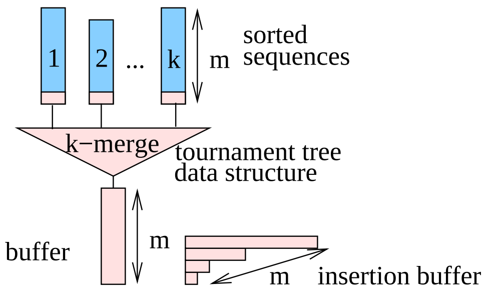


Figure 4.4: A simple external PQ for $n < km$

size B. If these blocks fit into the cache, we only require $\mathcal{O}(n/B)$ I/O operations.

4.3 External Priority Queues

We now study a variant of external priority queues[2] which are called *sequence heaps*.

Merging k sorted sequences into one sorted sequence (k-way merging) is an I/O efficient subroutine used for sorting – we saw this in chapter 3.5. The basic idea of sequence heaps is to adapt k-way merging to the related but more dynamical problem of priority queues.

Let us start with the simple case, that at most km insertions take place where m is the size of a buffer that fits into fast memory. Then the data structure could consist of k sorted sequences of length up to m. We can use k-way merging for deleting a batch of the m smallest elements from k sorted sequences. The next m deletions can then be served from a buffer in constant time.

A separate binary heap with capacity m allows an arbitrary mix of insertions and

[2]if “I/O“ is replaced by “cache fault“, we can use this approach also one level higher in the memory hierarchy

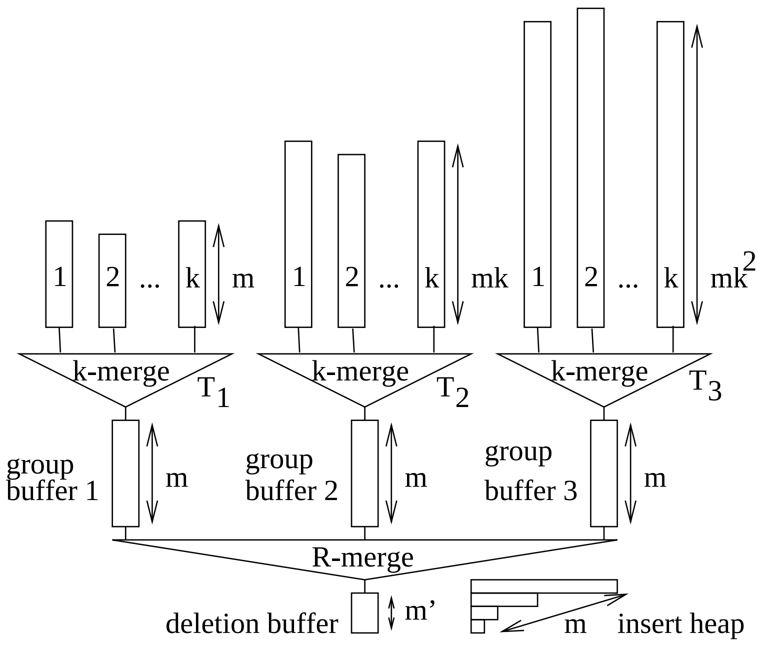


Figure 4.5: Overview of the complete data structure for $R = 3$ merge groups

deletions by holding the recently inserted elements. Deletions have to check whether the smallest element has to come from this *insertion buffer*. When this buffer is full, it is sorted, and the resulting sequence becomes one of the sequences for the k-way merge.

How can we generalize this approach to handle more than km elements? We cannot increase m beyond M, since the insertion heap would not fit into fast memory. We cannot arbitrarily increase k, since eventually k-way merging would start to incur cache faults. Sequence heaps make room by merging all the k sequences producing a larger sequence of size up to km.

Now the question arises how to handle the larger sequences. Sequence heaps employ R *merge groups* $G_1, \ldots, G_R$ where G_i holds up to k sequences of size up to mk^{i-1}. When group G_i overflows, all its sequences are merged, and the resulting sequence is put into group G_{i+1}.

Each group is equipped with a *group buffer* of size m to allow batched deletion from the sequences. The smallest elements of these buffers are deleted in batches of size $m' \ll m$. They are stored in the *deletion buffer*. Fig. 4.5 summarizes the data structure. We now have enough information to understand how deletion works:

DeleteMin: The smallest elements of the deletion buffer and insertion buffer are compared, and the smaller one is deleted and returned. If this empties the deletion buffer, it is refilled from the group buffers using an R-way merge. Before the refill, group buffers with less than m' elements are refilled from the sequences in their group (if the group is nonempty).

DeleteMin works correctly provided the data structure fulfills the heap property, i.e., elements in the group buffers are not smaller than elements in the deletion buffer, and in turn, elements in a sorted sequence are not smaller than the elements in the respective group buffer. Maintaining this invariant is the main difficulty for implementing insertion.

Insert: New elements are inserted into the insert heap. When its size reaches m, its elements are sorted (e.g. using merge sort or heap sort). The result is then merged with the concatenation of the deletion buffer and the group buffer 1. The smallest resulting elements replace the deletion buffer and group buffer 1. The remaining elements form a new sequence of length at most m. The new sequence is finally inserted into a free slot of group G_1. If there is no free slot initially, G_1 is emptied by merging all its sequences into a single sequence of size at most km, which is then put into G_2. The same strategy is used recursively to free higher level groups when necessary. When group G_R overflows, R is incremented and a new group is created. When a sequence is moved from one group to the other, the heap property may be violated. Therefore, when G_1 through G_i have been emptied, the group buffers 1 through $i+1$ are merged, and put into G_1.

For cached memory, where the speed of internal computation matters, it is also crucial how to implement the operation of k-way merging. How is can be done in an efficient way is described in the chapter about Sorting (3.7).

Analysis

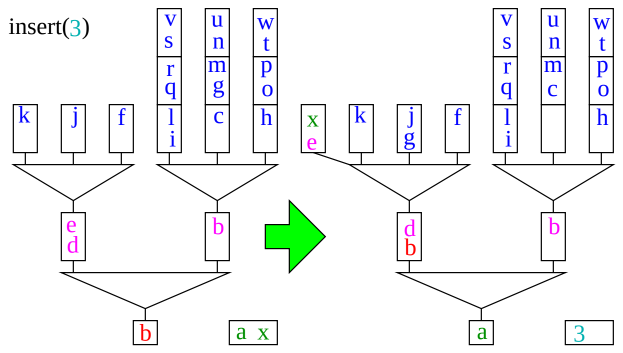
We will now give a sketch for the I/O analysis of our priority queues. Let i denote the number of insertions and an upper bound to the number of deleteMin operations.

First note that Group G_i can overflow at most every $m(k^i - 1)$ insertions: The only complication is the slot in group G_1 used for invalid group buffers. Nevertheless, when groups G_1 through G_i contain k sequences each, this can only happen if

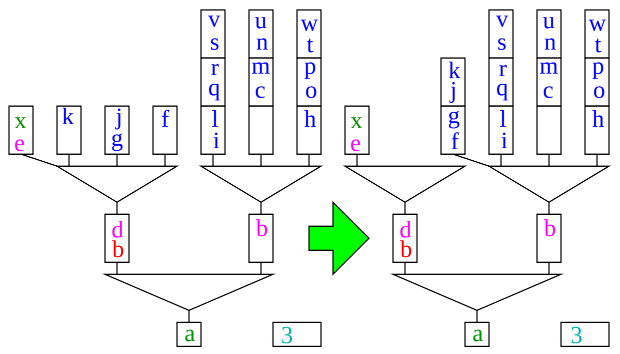
$$\sum_{j=1}^{R} m(k-1)k^{j-1} = m(k^i - 1)$$

insertions have taken place. Therefore, $R = \left\lceil \log_k \frac{I}{m} \right\rceil$ groups suffice.

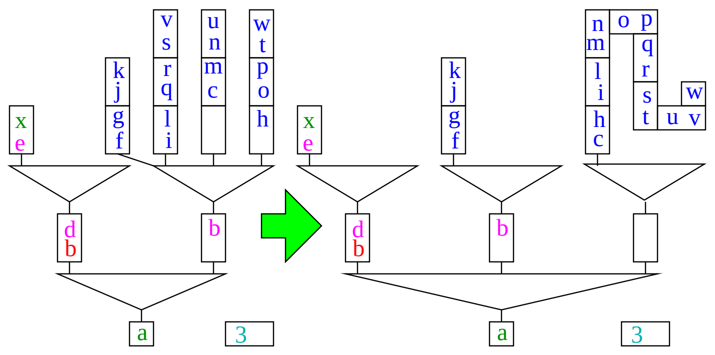
Now consider the I/Os performed for an element moving on the following *canonical* data path: It is first inserted into the insert buffer and then written to a sequence in group G_1 in a batched manner, i.e., $1/B$ I/Os are charged to the insertion of this element. Then it is involved in emptying groups until it arrives in group G_R. For each emptying operation, the element is involved into one batched read and one batched write, i.e., it is



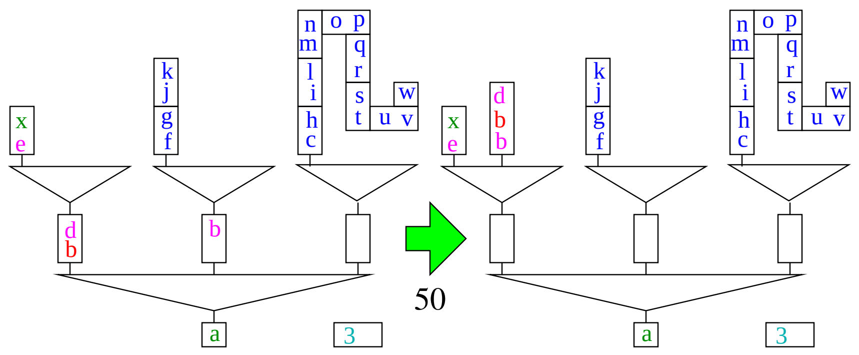
(a) Inserting element 3 leads to overflow of insert heap: it is merged with deletion buffer and group buffer 1 and then inserted into group 1



(b) Overflow in group 1: all old elements are merged and inserted in next group

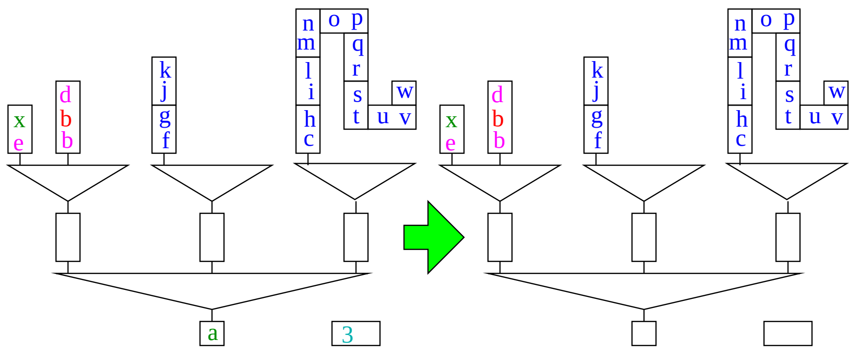


(c) Overflow in group 2: all old elements are merged and inserted in next group

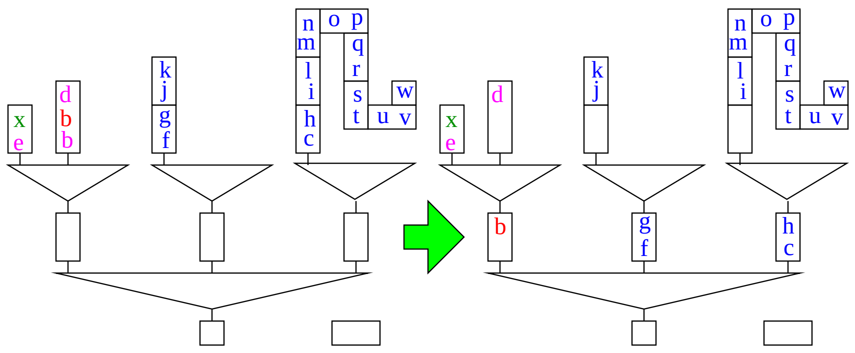


(d) Group buffers are invalid now: merge and inserted them to group 1

Figure 4.6: Example of an insertion on the sequence heap



(a) Deletion of two elements empties insert heap and deletion buffer



(b) Every Group fills its buffer via k-way-merging, the deletion buffer is filled from group buffers via M-way-merging

Figure 4.7: Example of a deletion on the sequence

charged with $2(R-1)/B$ I/Os for tree emptying operations. Eventually, the element is read into group buffer R yielding a charge of $1/B$ I/Os for. All in all, we get a charge of $2R/B$ I/Os for each insertion.

What remains to be shown (and is ommited here) is that the remaining I/Os only contribute lower order terms or replace I/Os done on the canonical path. For example, we save I/Os when an element is extracted before it reaches the last group. We use the costs charged for this to pay for swapping the group buffers in and out. Eventually, we have $\mathcal{O}(sort(I))$ I/Os.

In a similar fashion, we can show that I operations inflict $I \log I$ key comparisons on average. As for sorting, this is a good measure for the internal work, since in efficient implementations of priority queues for the comparison model, this number is close to the number of unpredictable branch instructions (whereas loop control branches are usually well predictable by the hardware or the compiler), and the number of key comparisons is also proportional to the number of memory accesses. These two types of operations often have the largest impact on the execution time, since they are the most severe limit to instruction parallelism in a super-scalar processor.

Experiments

We now present the results of some experiments conducted to compare our *sequence heap* with other priority queue implementations. Random 32 bit integers were used as keys for another 32 bits of associated information. The operation sequence used was $(\texttt{Insert} - \texttt{deleteMin} - \texttt{Insert})^N(\texttt{deleteMin} - \texttt{Insert} - \texttt{deleteMin})^N$. The choice of this sequence is nontrivial as it can have measurable influence (factor two and more) on the performance. Figure 4.9 show this: Here we have the sequence $(\texttt{Insert}\ (\texttt{deleteMin}\ \texttt{Insert})^s)^N\ (\texttt{deleteMin}\ (\texttt{Insert}\ \texttt{deleteMin})^s)^N$ for several values of s. For larger s, the performance gets better when N is large enough. This can be explained with a “locality effect“: New elements tend to be smaller than most old elements (the smallest of the old elements have long been removed before). Therefore, many elements never make it into group G_1 let alone the groups for larger sequences. Since most work is performed while emptying groups, this work is saved. So that these instances should come close to the worst case. To make clear that sequence heaps are nevertheless still much better than binary or 4-ary heaps, Figure 4.9 additionally contains their timing for $s = 0$.

The parameters chosen for the experiments where $m' = 32$, $m = 256$ and $k = 128$ on all machines tried. While there were better settings for individual machines, these global values gave near optimal performance in all cases.

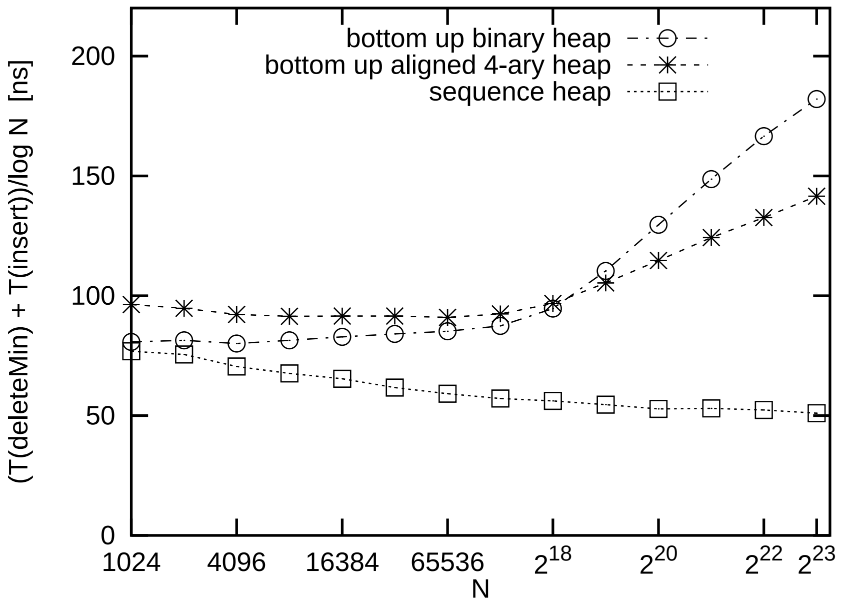


Figure 4.8: Runtime comparison for several PQ implementations (on a 180MHz MIPS R10000)

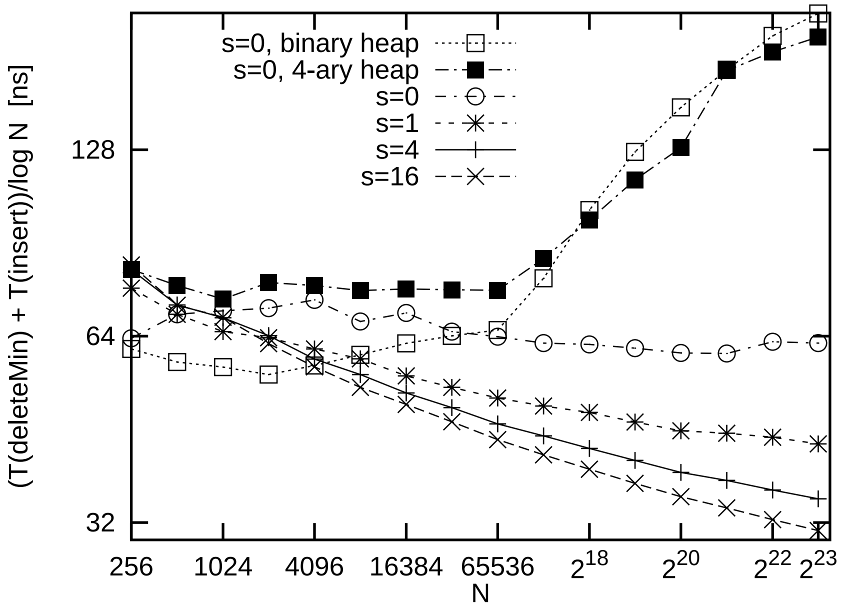


Figure 4.9: Runtime comparison for different operation sequences

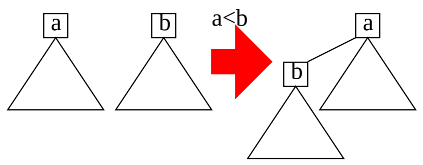


Figure 4.10: Link: Merge to trees preserving the heap property

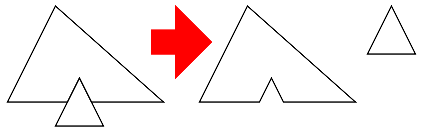


Figure 4.11: Cut: remove subtree and add it to the forest

4.4 Adressable Priority Queues

For adressable Priority Queues, we want to add the following functionality to the interface of our basic data structure:

Function remove$(h : Handle)$ $\quad e := h; \quad M := M \setminus \{e\};$ **return** e
Procedure decreaseKey$(h : Handle, k : Key)$ **assert** $key(h) \geq k; \quad key(h) := k$
Procedure merge(M') $\quad M := M \cup M'$

This extended interface is required to efficiently implement Dijkstra's Algorithm for shortest paths or the Jarnik-Prim Algorithm for calculating Minimum Spanning Trees (both make use of the `decreaseKey` operation).

It is not possible to extend our hitherto approach to become adressable as keys are constantly swapped in our array for deleteMin and other operations. For this domain, we implement priority queues as a set of heap-ordered trees and a pointer for finding the tree containing the globally minimal element. The elementary form of these priority queues is called *Pairing Heap*.

With just two basic operations, we can implement adressable priority queues:

Now we can already give a high-level implementation of all necessary operations:

```
**Procedure** *insertItem*$(h : Handle)$
    $newTree(h)$

**Procedure** *newTree*$(h : Handle)$
    $forest := forest \cup \{h\}$
    **if** $e < \min$ **then** $minPtr := h$

**Procedure** *decreaseKey*$(h : Handle, k : Key)$
    $key(h) := k$
    **if** $h$ is not a root **then** *cut*$(h)$
**Function** $deleteMin : Handle$
    $m$:= $minPtr$
    $forest$:= $forest \setminus \{m\}$
    **foreach** *child* $h$ *of* $m$ **do** *newTree*$(h)$
    Perform a pairwise link of the tree roots in $forest$
    **return** $m$
**Procedure** *merge*$(o : AdressablePQ)$
    **if** $minPtr > o.minPtr$ **then** $minPtr$:= $o.minPtr$
    $forest$:= $forest \cup o.forest$
    $o.forest$:= $\emptyset$
```

An `insert` adds a new single node tree to the forest. So a sequence of n inserts into an initially empty heap will simply create n single node trees. The cost of an insert is clearly $\mathcal{O}(1)$.

A `deleteMin` operation removes the node indicated by `minPtr`. This turns all children of the removed node into roots. We then scan the set of roots (old and new) to find the new minimum. To find the new minimum we need to inspect all roots (old and new), a potentially very costly process. We make the process even more expensive (by a constant factor) by doing some useful work on the side, namely combining some trees into larger trees. Pairing heaps do this by just doing one step of pairwise linking of arbitrary trees. There are variants doing more complicated operations to prove better theoretical bounds.

We turn to the `decreaseKey` operation next. It is given a handle h and a new key k and decreases the key value of h to k. In order to maintain the heap property, we cut the subtree rooted at h and turn h into a root. Cutting out subtrees causes the more subtle problem that it may leave trees that have an awkward shape. While Pairing heaps do nothing to prevent thiss, some variants of addressable priority queues perform additional operations to keep the trees in shape.

The remaining operations are easy. We can `remove` an item from the queue by first decreasing its key so that it becomes the minimum item in the queue and then perform a `deleteMin`. To merge a queue o into another queue we compute the union of $roots$ and $o.roots$. To update $minPtr$ it suffices to compare the minima of the merged queues. If the root sets are represented by linked lists, and no additional balancing is done, a merge needs only constant time.

Pairing heaps are the simplest form of forest-based adressable priority queues. A more elaborated and (in theory, at least) faster variant are *Fibonacci Heaps*. They maintain a *rank* (initially zero, denoting the number of its children) for every element, which is increased for root nodes when another tree is linked to them and a *mark* flag that is set when the node lost a child due to a `decreaseKey`. Root nodes of the same rank are

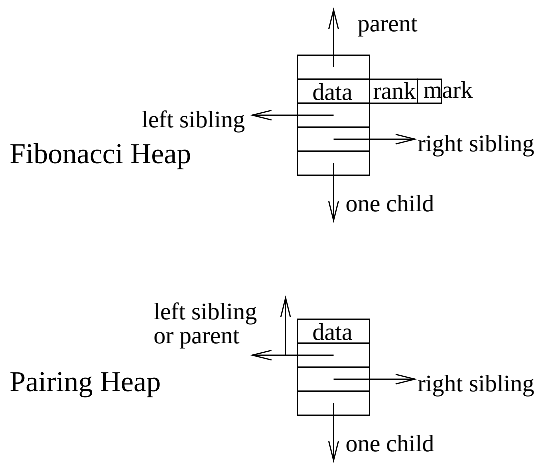


Figure 4.12: Structure of one item in a Pairing Heap or a Fibonacci Heap.

linked after a `deleteMin` to limit the number of trees. If a cut is executed on a node with an already marked parent, the parent is cut as well. These rules lead to an amortized complexity of $\mathcal{O}(\log n)$ for deleteMin an $\mathcal{O}(1)$ for all other operations. However, both the constant factors and the worst case performance for a single operation are high, making Fibonacci Heaps a mainly theoretical tool. In addition, more metainformation per node increases the memory overhead of Fibonacci Heaps.

Chapter 5

External Memory Algorithms

The introduction of this chapter is based on [6]. The sections on time-forward processing, graph algorithms and cache oblivious algorithms use material from the book chapters [10], [8] and [9]. The cache oblivious model was first presented in [11]. The section on Funnelsort is based on [19]. The external BFS section is from [12] for the presentation of the algorithm and from [13] for tuning and experiments. Additional material in multiple sections is from [7].

5.1 Introduction

Massive data sets arise naturally in many domains. Spatial data bases of geographic information systems like GoogleEarth and NASA's World Wind store terabytes of geographically-referenced information that includes the whole Earth. In computer graphics one has to visualize huge scenes using only a conventional workstation with limited memory. Billing systems of telecommunication companies evaluate terabytes of phone call log files. One is interested in analyzing huge network instances like a web graph or a phone call graph. Search engines like Google and Yahoo provide fast text search in their data bases indexing billions of web pages. A precise simulation of the Earth's climate needs to manipulate with petabytes of data. These examples are only a sample of numerous applications which have to process huge amount of data.

For economical reasons, it is not feasible to build all of the computer's memory of the fastest type or to extend the fast memory to dimensions that could hold all relevant data. Instead, modern computer architectures contain a memory hierarchy of increasing size, decreasing speed and costs from top to bottom: On top, we have the registers integrated in the CPU, a number of caches, main memory and finally disks, which are often referenced as *external memory* as opposed to *internal memory*.

The internal memories of computers can keep only a small fraction of these large data sets. During the processing the applications need to access the external memory (e. g.

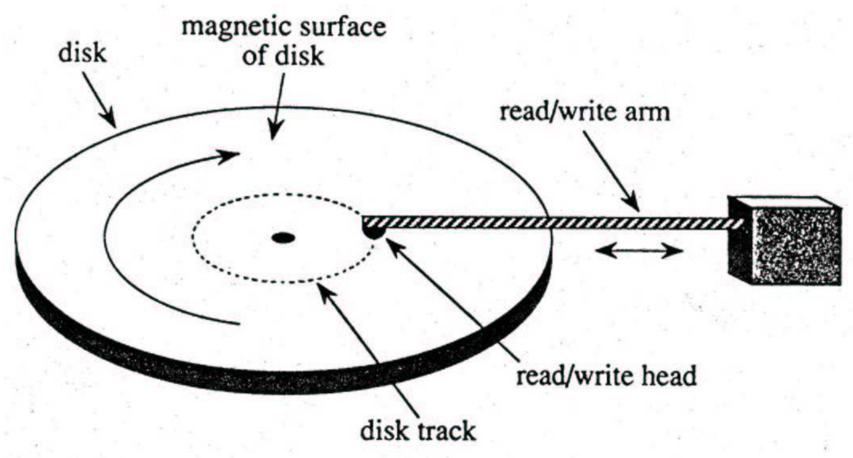


Figure 5.1: schematic construction of a hard disk

hard disks) very frequently. One such access can be about 106 times slower than a main memory access. Therefore, the disk accesses (I/Os) become the main bottleneck.

The reason for this high latency is the mechanical nature of the disk access. Figure 5.1 shows the schematic construction of a hard disk. The time needed for finding the data position on the disk is called seek time or (seek) latency and averages to about 3–10 ms for modern disks. The seek time depends on the surface data density and the rotational speed and can hardly be reduced because of the mechanical nature of hard disk technology, which still remains the best way to store massive amounts of data. Note that after finding the required position on the surface, the data can be transferred at a higher speed which is limited only by the surface data density and the bandwidth of the interface connecting CPU and hard disk. This speed is called sustained throughput and achieves up to 80 MByte/s nowadays. In order to amortize the high seek latency one reads or writes the data in chunks (blocks). The block size is balanced when the seek latency is a fraction of the sustained transfer time for the block. Good results show blocks containing a full track. For older low density disks of the early 90's the track capacities were about 16-64 KB. Nowadays, disk tracks have a capacity of several megabytes.

Operating systems implement the so called virtual memory mechanism that provides an additional working space for an application, mapping an external memory file (page file) to virtual main memory addresses. This idea supports the Random Access Machine model in which a program has an infinitely large main memory with uniform random access cost. Since the memory view is unified in operating systems supporting virtual memory, the application does not know where its working space and program code are located: in the main memory or (partially) swapped out to the page file. For many applications and algorithms with non-linear access pattern, these remedies are not useful and even counterproductive: the swap file is accessed very frequently; the data code can be swapped out in favor of data blocks; the swap file is highly fragmented and thus many random input/output operations (I/Os) are needed even for scanning.

5.2 The external memory model and things we already saw

If we bypass the virtual memory mechanism, we cannot apply the RAM model for analysis anymore since we now have to explicitly handle different levels of memory hierarchy, while the RAM model uses one large, uniform memory.

Several simple models have been introduced for designing I/O-efficient algorithms and data structures (also called external memory algorithms and data structures). The most popular and realistic model is the Parallel Disk Model (PDM) of Vitter and Shriver. In this model, I/Os are handled explicitly by the application. An I/O operation transfers a block of B consecutive bytes from/to a disk to amortize the latency. The application tries to transfer D blocks between the main memory of size M bytes and D independent disks in one I/O step to improve bandwidth. The input size is N bytes which is (much) larger than M. The main complexity metrics of an I/O-efficient algorithm in this model are:

- I/O complexity: the number of I/O steps should be minimized (the main metric),
- CPU work complexity: the number of operations executed by the CPU should be minimized as well.

The PDM model has become the standard theoretical model for designing and analyzing I/O-efficient algorithms.

There are some "golden rules" that can guide the process of designing I/O efficient algorithms: Unstructured memory access is often very expensive as it comes with 1 I/O per operation whereas we want $1/B$ I/Os for an efficient algorithm. Instead, we want to *scan* the external memory, always loading the next due block of size B in one step and processing it internally. An optimal scan will only cost $\mathrm{scan}(N) := \Theta(\frac{N}{D \cdot B})$ I/Os. If the data is not stored in a way that allows linear scanning, we can often use *sorting* to reorder and than scan it. As we saw in chapter 3, external sorting can be implemented with $\mathrm{sort}(N) := \Theta(\frac{N}{D \cdot B} \cdot \log_{M/B} \frac{N}{B})$ I/Os.

A simple example of this technique is the following task: We want to reorder the elements in an array A into an array B using a given "rank" stored in array C. This should be done in an I/O efficient way.

```
int[1..N] A,B,C;
for i=1 to N do A[i]:=B[C[i]];
```

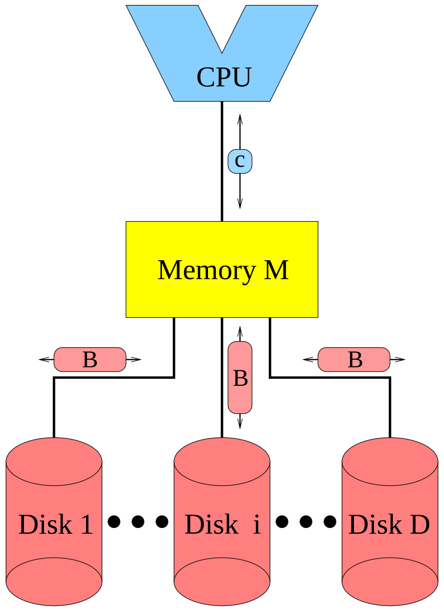


Figure 5.2: Vitter's I/O model with several independent disks

The literal implementation would have worst case costs of $\Omega(N)$ I/Os. For $N = 10^6$, this would take $\approx T = 10000$ seconds $\approx$ 3 hours. Using the sort-and-scan technique, we can lower this to $\text{sort}(N)$ and the algorithm would finish in less than a second:

```
SCAN C:     (C[1]=17,1),  (C[2]=5,2),   ...
SORT(1st):  (C[73]=1,73), (C[12]=2,12), ...
par SCAN :  (B[1],73),    (B[2],12),    ...
SORT(2nd):  (B[C[1]],1),  (B[C[2]],2),  ...
```

We already saw some I/O efficient algorithms using this model in previous chapters: Chapter 2 presented an external stack, a large section of chapter 3 dealt with external sorting and in chapter 4 we saw external priority queues. Chapter 8 will present an external approach to minimal spanning trees. In this chapter, we will see some more algorithms, study how these algorithms and data structures can be implemented in a convenient way using an algorithm library and learn about other models of external computation.

5.3 The Stxxl library

The Stxxl library is an algorithm library aimed to speed up the process of implementing I/O-efficient algorithms, abstracting away the details of how I/O is performed. Many high-performance features are supported: disk parallelism, explicit overlapping of I/O and

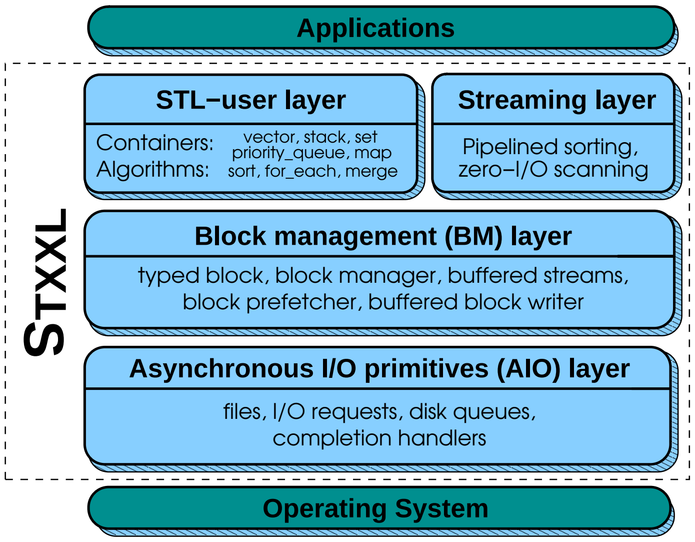


Figure 5.3: three layer structure of the Stxxl library

computation, external memory algorithm pipelining to save I/Os, improved utilization of CPU resources (many of these techniques are introduced in Chapter 3 on external sorting). The high-level algorithms and data structures of our library implement interfaces of the well known C++ Standard Template Library (STL). This allows to elegantly reuse the STL code such that it works I/O-efficiently without any change, and to shorten the development times for the people who already know STL. Our STL-compatible I/O-efficient implementations include the following data structures and algorithms: unbounded array (vector), stacks, queue, deque, priority queue, search tree, sorting, etc. They are all well-engineered and have robust interfaces allowing a high degree of flexibility. Stxxl is a layered library consisting of three layers (see Figure 5.3):

The lowest layer, the Asynchronous I/O primitives layer (AIO layer), abstracts away the details of how asynchronous I/O is performed on a particular operating system. Other existing external memory algorithm libraries only rely on synchronous I/O APIs or allow reading ahead sequences stored in a file using the POSIX asynchronous I/O API. These libraries also rely on uncontrolled operating system I/O caching and buffering in order to overlap I/O and computation in some way. However, this approach has significant performance penalties for accesses without locality. Unfortunately, the asynchronous I/O APIs are very different for different operating systems (e.g. POSIX AIO and Win32 Overlapped I/O). Therefore, we have introduced the AIO layer to make porting Stxxl easy. Porting the whole library to a different platform requires only reimplementing the AIO layer using native file access methods and/or native multithreading mechanisms.

The Block Management layer (BM layer) provides a programming interface emulating the parallel disk model. The BM layer provides an abstraction for a fundamental concept in the external memory algorithm design — a block of elements. The block manager implements block allocation/deallocation, allowing several block-to-disk assignment strategies: striping, randomized striping, randomized cycling, etc. The block management layer provides an implementation of parallel disk buffered writing, optimal prefetching HSV01], and block caching. The implementations are fully asynchronous and designed to explicitly support overlapping between I/O and computation.

The top of Stxxl consists of two modules. The STL-user layer provides external memory sorting, external memory stack, external memory priority queue, etc. which have (almost) the same interfaces (including syntax and semantics) as their STL counterparts. The Streaming layer provides efficient support for pipelining external memory algorithms. Many external memory algorithms, implemented using this layer, can save a factor of 2–3 in I/Os. For example, the algorithms for external memory suffix array construction implemented with this module require only 1/3 of the number of I/Os which must be performed by implementations that use conventional data structures and algorithms (either from Stxxl STL-user layer, LEDASM, or TPIE). The win is due to an efficient interface that couples the input and the output of the algorithm–components (scans, sorts, etc.). The output from an algorithm is directly fed into another algorithm as input, without needing to store it on the disk in-between. This generic pipelining interface is the first of this kind for external memory algorithms.

5.4 Time-Forward Processing

This section is based on material from [10].

Time-Forward Processing is an elegant technique for solving problems that can be expressed as a traversal of a directed acyclic graph (DAG) from its sources to its sinks. Problems of this type arise mostly in I/O-efficient graph algorithms, even though applications of this technique for the construction of I/O-efficient data structures are also known. Formally, the problem that can be solved using time-forward processing is that of evaluating a DAG G: Let ϕ be an assignment of labels $\phi(v)$ to the vertices of G. Then the goal is to compute another labelling ψ of the vertices of G so that for every vertex $v \in G$, label $\psi(v)$ can be computed from labels $\phi(v)$ and $\psi(u_1), \ldots, \psi(u_k)$, where $u_1, \ldots, u_k$ are the in-neighbors of v.

As an illustration, consider the problem of expression-tree evaluation. For this problem, the input is a binary tree T whose leaves store real numbers and whose internal vertices are labelled with one of the four elementary binary operations $+, -, *, /$. The *value* of a vertex is defined recursively. For a leaf v, its value $val(v)$ is the real number stored at v. For an internal vertex v with label $\circ \in \{+, -, *, /\}$, left child x, and right child y, $val(v) = val(x) \circ val(y)$. The goal is to compute the value of the root of T. Cast in terms of the general DAG evaluation problem defined above, tree T is a DAG whose

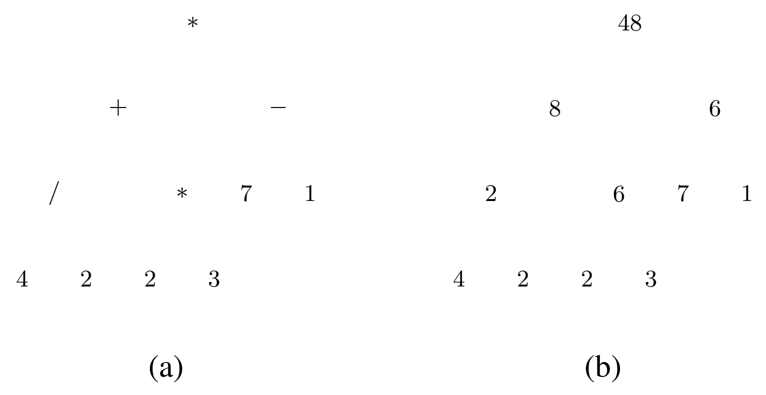


Figure 5.4: (a) The expression tree for the expression $((4\,/\,2) + (2 * 3)) * (7 - 1)$. (b) The same tree with its vertices labelled with their values.

edges are directed from children to parents, labelling ϕ is the initial assignment of real numbers to the leaves of T and of operations to the internal vertices of T, and labelling ψ is the assignment of the values $val(v)$ to all vertices $v \in T$. For every vertex $v \in T$, its label $\psi(v) = val(v)$ is computed from the labels $\psi(x) = val(x)$ and $\psi(y) = val(y)$ of its in-neighbors (children) and its own label $\phi(v) \in \{+, -, *, /\}$.

In order to be able to evaluate a DAG G I/O-efficiently, two assumptions have to be satisfied: (1) The vertices of G have to be stored in topologically sorted order. That is, for every edge $(v, w) \in G$, vertex v precedes vertex w. (2) Label $\psi(v)$ has to be computable from labels $\phi(v)$ and $\psi(u_1), \dots, \psi(u_k)$ in $\mathcal{O}(\mathrm{sort}(k))$ I/Os. The second condition is trivially satisfied if every vertex of G has in-degree no more than M.

Given these two assumptions, time-forward processing visits the vertices of G in topologically sorted order to compute labelling ψ. Visiting the vertices of G in this order guarantees that for every vertex $v \in G$, its in-neighbors are evaluated before v is evaluated. Thus, if these in-neighbors "send" their labels $\psi(u_1), \dots, \psi(u_k)$ to v, v has these labels and its own label $\phi(v)$ at its disposal to compute $\psi(v)$. After computing $\psi(v)$, v sends its own label $\psi(v)$ "forward in time" to its out-neighbors, which guarantees that these out-neighbors have $\psi(v)$ at their disposal when it is their turn to be evaluated.

The implementation of this technique due to Arge is simple and elegant. The "sending" of information is realized using a priority queue Q. When a vertex v wants to send its label $\psi(v)$ to another vertex w, it inserts $\psi(v)$ into priority queue Q and gives it priority w. When vertex w is evaluated, it removes all entries with priority w from Q. Since every in-neighbor of w sends its label to w by queuing it with priority w, this provides w with the required inputs. Moreover, every vertex removes its inputs from the priority queue before it is evaluated, and all vertices with smaller numbers are evaluated before w. Thus, at the time when w is evaluated, the entries in Q with priority w are those with lowest priority, so that they can be removed using a sequence of DELETEMIN operations.

Using the buffer tree of Arge to implement priority queue Q, INSERT and DELETEMIN operations on Q can be performed in $\mathcal{O}\big((1/B) \cdot \log_{M/B}(|E|/B)\big)$ I/Os amortized because priority queue Q never holds more than $|E|$ entries. The total number of priority queue operations performed by the algorithm is $\mathcal{O}(|E|)$, one INSERT and one DELETEMIN operation per edge. Hence, all updates of priority queue Q can be processed in $\mathcal{O}(()\,\mathrm{sort}(|E|)$ I/Os. The computation of labels $\psi(v)$ from labels $\phi(v)$ and $\psi(u_1), \ldots, \psi(u_k)$, for all vertices $v \in G$, can also be carried out in $\mathcal{O}(\mathrm{sort}(|E|))$ I/Os, using the above assumption that this computation takes $\mathcal{O}(()\,\mathrm{sort}(k))$ I/Os for a single vertex v. Hence, we obtain the following result.

Theorem 1 *Given a DAG $G = (V, E)$ whose vertices are stored in topologically sorted order, graph G can be evaluated in $\mathcal{O}(\mathrm{sort}(|V| + |E|))$ I/Os, provided that the computation of the label of every vertex $v \in G$ can be carried out in $\mathcal{O}\big(\mathrm{sort}(\deg^-(v))\big)$ I/Os, where $\deg^-(v)$ is the in-degree of vertex v.*

5.5 Cache-oblivious Algorithms

Have a look at table 5.1, whiches gives size and other attributes of different levels in the memory hierarchy for various systems. How can we write portable code that runs efficiently on different multilevel caching architectures? Not only is the external memory model restricted to two levels of memory, for most algorithms we have to explicitly give values for M and B which are different on every level and system. The cache oblivious model suggests a solution: We want to design algorithms that are not given M and B and that nevertheless perform well on every memory hierarchy.

The ideal cache oblivious memory model is a two level memory model. We will assume that the faster level has size M and the slower level always transfers B consecutive words of data to the faster level. These two levels could represent the memory and the disk, memory and the cache, or any two consecutive levels of the memory hierarchy. In this chapter, M and B can be assumed to be the sizes of any two consecutive levels of the memory hierarchy subject to some assumptions about them (For instance the inclusion property which we will see soon). We will assume that the processor can access the faster level of memory which has size M. If the processor references something from the second level of memory, an I/O fault occurs and B words are fetched into the faster level of the memory. We will refer to a *block* as the minimum unit that can be present or absent from a level in the two level memory hierarchy. We will use B to denote the size of a *block* as in the external memory model. If the faster level of the memory is full (i.e. M is full), a block gets evicted to make space.

	Pentium 4	Pentium III	MIPS 10000	AMD Athlon	Itanium 2
Clock rate	2400 MHz	800 MHz	175 MHz	1333 MHz	1137 MHz
L1 data cache size	8 KB	16 KB	32 KB	128 KB	32 KB
L1 line size	128 B	32 B	32 B	64 B	64 B
L1 associativity	4-way	4-way	2-way	2-way	4-way
L2 cache size	512 KB	256 KB	1024 KB	256 KB	256 KB
L2 line size	128 B	32 B	32 B	64 B	128 B
L2 associativity	8-way	4-way	2-way	8-way	8-way
TLB entries	128	64	64	40	128
TLB associativity	full	4-way	64-way	4-way	full
RAM size	512 MB	256 MB	128 MB	512 MB	3072 MB

Table 5.1: some exemplary cache and memory configurations

The ideal cache oblivious memory model enables us to reason about a two level memory model like the external memory model but prove results about a multi-level memory model. Compared with the external memory model it seems surprising that without any memory specific parametrization, or in other words, without specifying the parameters M, B, an algorithm can be efficient for the whole memory hierarchy, nevertheless it is possible. The model is built upon some basic assumptions which we enumerate next.
Optimal replacement: The *replacement policy* refers to the policy chosen to replace a block when a cache miss occurs and the cache is full. In most hardware, this is implemented as FIFO, LRU or Random. The model assumes that the cache line chosen for replacement is the one that is accessed furthest in the future. This is known as the *optimal off-line replacement* strategy.
Two levels of memory: There are certain assumptions in the model regarding the two levels of memory chosen. They should follow the *inclusion property* which says that data cannot be present at level i unless it is present at level $i + 1$. Another assumption is that the size of level i of the memory hierarchy is strictly smaller than level $i + 1$.
Full associativity: When a block of data is fetched from the slower level of the memory, it can reside in any part of the faster level.
Automatic replacement: When a block is to be brought in the faster level of the memory, it is automatically done by the OS/hardware and the algorithm designer does not have to care about it while designing the algorithm. Note that we could access single blocks for reading and writing in the external memory model, which is not allowed in the cache oblivious model.

We will now examine each of the assumptions individually. First we consider the optimal replacement policy. The most commonly used replacement policy is LRU (*least recently used*). We have the following lemma, whose proof is omitted here:

Lemma 2 *An algorithm that causes $Q^*(n, M, B)$ cache misses on a problem of size n using a (M, B)-ideal cache incurs $Q(n, M, B) \leq 2Q^*(n, \frac{M}{2}, B)$ cache misses on a (M, B) cache that uses LRU or FIFO replacement. This is only true for algorithms which follow a regularity condition.*

An algorithm whose cache complexity satisfies the condition $Q(n, M, B) \leq O(Q(n, 2M, B))$ is called *regular* (All algorithms presented in this chapter are regular). Intuitively, algorithms that slow down by a constant factor when memory (M) is reduced to half, are called regular. It immediately follows from the above lemma that if an algorithm whose number of cache misses satisfies the regularity condition does $Q(n, M, B)$ cache misses with optimal replacement then this algorithm would make $\Theta(Q(n, M, B))$ cache misses on a cache with LRU or FIFO replacement.

The automatic replacement and full associativity assumption can be implemented in software by using LRU implementation based on hashing. It was shown that a fully associative LRU replacement policy can be implemented in $O(1)$ expected time using $O(\frac{M}{B})$ records of size $O(B)$ in ordinary memory. Note that the above description about the cache oblivious model proves that any optimal cache oblivious algorithm can also be optimally implemented in the external memory model.

We now turn our attention to multi-level ideal caches. We assume that all the levels of this cache hierarchy follow the inclusion property and are managed by an optimal replacement strategy. Thus on each level, an optimal cache oblivious algorithm will incur an asymptotically optimal number of cache misses. From Lemma 2, this becomes true for cache hierarchies maintained by LRU and FIFO replacement strategies.

Apart from not knowing the values of M, B explicitly, some cache oblivious algorithms (for example optimal sorting algorithms) require a *tall cache* assumption. The tall cache assumption states that $M = \Omega(B^2)$ which is usually true in practice. Recently, compiler support for cache oblivious type algorithms have also been looked into.

In cache oblivious algorithm design some algorithm design techniques are used ubiquitously. One of them is a *scan* of an array which is laid out in contiguous memory. Irrespective of B, a scan takes at most $1 + \lceil \frac{N}{B} \rceil$ I/Os. The argument is trivial and very similar to the external memory scan algorithm. The difference is that in the cache oblivious setting the buffer of size B is not explicitly maintained in memory. In the assumptions of the model, B is the size of the data that is always fetched from level 2 memory to level 1 memory. The scan does not touch the level 2 memory until its ready to evict the last loaded buffer of size B already in level 1. Hence, the total number of times the scan algorithm will force the CPU to bring buffers from the level 2 memory to level 1 memory is upper bounded by $1 + \lceil \frac{N}{B} \rceil$.

Another common approach in the cache oblivious model are divide and conquer algorithms. Why does divide and conquer help in general for cache oblivious algorithms? Divide and conquer algorithms split the instance of the problem to be solved into several subproblems such that each of the subproblems can be solved independentally. Since the

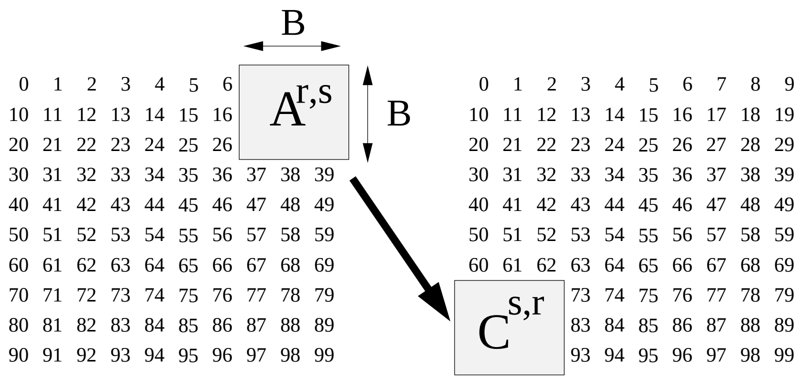


Figure 5.5: cache aware matrix transposition using block access

algorithm recurses on the subproblems, at some point of time, the subproblems fit inside M and subsequent recursion, fits the subproblems into B.

5.5.1 Matrix Transposition

We will see the recursive approach in our first example, dealing with matrix transposition. We will first give an algorithm using the external memory model that requires the knowledge of M and B. We than have the opportunity to compare both implementations (cache aware and oblivious) experimentically.

The naive matrix transposition algorithm:

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    C[j][i] = A[i][j];
```

accessing the source matrix A in a row major fashion will have I/O costs of $\Theta(N)$ for writing the target matrix C, leading to total of $\Theta(N^2)$ I/Os. But if M and B are known, we can switch from row major access to block access: Partition A and C into blocks of size $r \times s$ with $r, s = \Theta(\sqrt{M})$. Apply the naive algorithm to $N/r \times N/s$ matrices (their elements are $s \times s$ sub-matrices). This requires $\mathcal{O}((N/s)^3 \cdot s^2/B) = \mathcal{O}(N^3/(s \cdot B)) = \mathcal{O}(N^3/(B \cdot \sqrt{M}))$ I/Os, which is optimal.

The oblivious approach works by partitioning the matrix in two blocks $A1$ and $A2$, transposing them recursively and writing the results in the appropriate blocks in the result matrix C.

Here is the C code for cache oblivious matrix transposition. The following code takes as input a submatrix given by $(x, y) - (x + delx, y + dely)$ in the input matrix I and

$A = \begin{pmatrix} A1 & A2 \end{pmatrix} \; C = \begin{pmatrix} C1 \\ C2 \end{pmatrix}$

```
CO_Transpose(A,C)
{
    CO_Transpose(A1,C1);
    CO_Transpose(A2,C2);
}
```

Figure 5.6: pseudo code for cache oblivious matrix transposition

```
void transpose(int x, int delx, int y, int dely,
 ElementType I[N][P], ElementType O[P][N]) {
        if((delx == 1) \&\& (dely == 1)) {
                O[y][x] = I[x][y];
                return;
        }
        if(delx >= dely){
                int xmid = delx / 2;
                transpose(x,xmid,y,dely,I,O);
                transpose(x+xmid,delx-xmid,y,dely,I,O);
                return;
        }
        // Similarly cut from ymid into two subproblems
        ...
}
```

Figure 5.7: C code for cache oblivious matrix transposition

transposes it to the output matrix O. `ElementType`[1] can be any element type , for instance `long`.

The code works by divide and conquer, dividing the bigger side of the matrix in the middle and recursing. It is short, easy to understand and contains no tuning parameters that have to be tweaked for every new setup. The algorithm is always I/O efficient:

Let the input be a matrix of $N \times P$ size. There are three cases:

[1]In all our experiments, `ElementType` was set to `long`.

$\log_2 N$	Naive	CA	CO
10	0.21	0.10	0.08
11	0.86	0.49	0.45
12	3.37	1.63	2.16
13	*13.56*	**6.38**	6.69

$\log_2 N$	Naive	CA	CO
10	0.14	0.12	0.09
11	0.87	0.42	0.47
12	3.36	1.46	2.03
13	*13.46*	**5.74**	6.86

Table 5.2: Running time of naive, cache aware (CA) and cache oblivious (CO) matrix transposition for $B = 32$ and $B = 128$

Case I: $\max\{N, P\} \leq \alpha B$ In this case,

$$Q(N,P) \leq \frac{NP}{B} + O(1)$$

Case II: $N \leq \alpha B < P$ In this case,

$$Q(N,P) \leq \begin{cases} O(1+N) & \text{if } \frac{\alpha B}{2} \leq P \leq \alpha B \\ 2Q(N,P/2) + O(1) & N \leq \alpha B < P \end{cases}$$

Case III: $P \leq \alpha B < N$ Analogous to Case II.
Case IV: $\min\{N, P\} \geq \alpha B$

$$Q(N,P) \leq \begin{cases} O(N + P + \frac{NP}{B}) & \text{if } \frac{\alpha B}{2} \leq N, P \leq \alpha B \\ 2Q(N,P/2) + O(1) & P \geq N \\ 2Q(N/2,P) + O(1) & N \geq P \end{cases}$$

The above recurrence solves to $Q(N,P) = O(1 + \frac{NP}{B})$.

There is a simpler way to visualize the above mess. Once the recursion makes the matrix small enough such that $\max(N,P) \leq \alpha B \leq \beta\sqrt{M}$ (here β is a suitable constant), or such that the submatrix (or the block) we need to transpose fits in memory, the number of I/O faults is equal to the scan of the elements in the submatrix. A packing argument of these not so small submatrices (blocks) in the large input matrix shows that we do not do too many I/O faults compared to a linear scan of all the elements.

Table 5.2 shows the result of an experiment performed on a 300 MHz UltraSPARC-II with 2 MB L2 cache, 16 KB L1 cache, page size 8 KB and 64 TLB entries. The (tuned) cache aware implementation is slightly slower than the cache oblivious one, but both outperform the naive implementation.

5.5.2 Searching Using Van Emde Boas Layout

In this section we report a method to speed up simple binary searches on a balanced binary tree. This method could be used to optimize or speed up any kind of search on

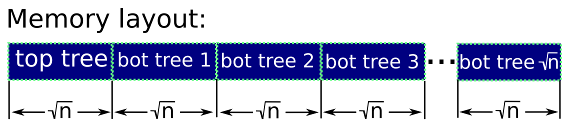


Figure 5.8: memory layout of cache oblivious search trees

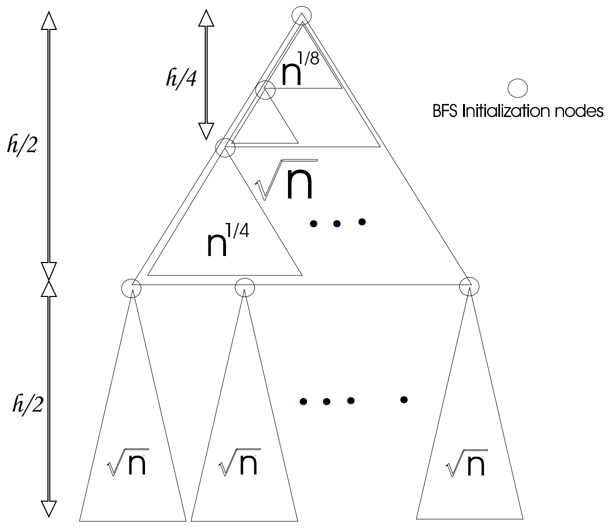


Figure 5.9: BFS structre of cache oblivious search trees

a tree as long as the tree is static and balanced. It is easy to code, uses the fact that the memory consists of a cache hierarchy, and could be exploited to speed up tree based search structures on most current machines. Experimental results show that this method could speed up searches by a factor of 5 or more in certain cases!

It turns out that a balanced binary tree has a very simple layout that is cache-oblivious. By layout here, we mean the mapping of the nodes of a binary tree to the indices of an array where the nodes are actually stored. The nodes should be stored in the bottom array in the order shown for searches to be fast and use the cache hierarchy.

Given a complete binary tree, we describe a mapping from the nodes of the tree to positions of an array in memory. Suppose the tree has N items and has height $h = \log N + 1$. Split the tree in the middle, at height $h/2$. This breaks the tree into a top recursive subtree of height $\lfloor h/2 \rfloor$ and several bottom subtrees $B_1, B_2, ..., B_k$ of height $\lceil h/2 \rceil$. There are $\sqrt{N}$ bottom recursive subtrees, each of size $\sqrt{N}$. The top subtree occupies the top part in the array of allocated nodes, and then the B_i's are laid out. Every subtree is recursively laid out.

Another way to see the algorithm is to run a breadth first search on the top node of the tree and run it till $\sqrt{N}$ nodes are in the BFS, see Fig. 5.9. The figure shows the run of the algorithm for the first BFS when the tree size is $\sqrt{N}$. Then the tree consists of the part that is covered by the BFS and trees hanging out. BFS can now be recursively run on each tree, including the covered part. Note that in the second level of recursion, the tree size is $\sqrt{N}$ and the BFS will cover only $N^{\frac{1}{4}}$ nodes since the same algorithm is run on each subtree of $\sqrt{N}$. The main idea behind the algorithm is to store recursive sub-trees in contiguous blocks of memory.

Lets now try to analyze the number of cache misses when a search is performed. We can conceptually stop the recursion at the level of detail where the size of the subtrees has size $\leq B$. Since these subtrees are stored contiguously, they at most fit in two blocks. (A block can not span three blocks of memory when stored). The height of these subtrees is $\log B$. A search path from root to leaf crosses $O\left(\frac{\log N}{\log B}\right) = O(\log_B N)$ subtrees. So the total number of cache misses is bounded by $O(\log_B N)$.

We did a very simple experiment to see how in real life, this kind of layout would help. A vector was sorted and a binary search tree was built on it. A query vector was generated with random numbers and searched on this BST which was laid out in pre-order. Why we chose pre-order compared to random layout was because most people code a BST in either pre/post/in-order compared to randomly laying it (Which incidentally is very bad for cache health). Once this query was done, we laid the BST using Van Emde Boas Layout and gave it the same query vector. The experiment reported in Fig. 5.10 were done on a Itanium dual processor system with 2GB RAM. (Only one processor was being used)

5.5.3 Funnel sorting

Funnel sorting is a cache oblivious sorting strategy. We will describe a simplifcation called lazy funnelsort, which was introduced by Brodal and Fagerberg [18]. Funnelsort, in turn, is a sort of lazy mergesort. This algorithm will be our first application of the tall-cache assumption (see 5.5. For simplicity, we assume that $M = \Omega(B^2)$. The same results can be obtained when $M = \Omega(B^{1+\gamma})$ by increasing the constant 3; refer to [18] for details. Interestingly, optimal cache-oblivious sorting is not achievable without the tall-cache assumption. The heart of the funnelsort algorithm is a static data structure which we call a funnel. For now, we treat a K-Funnel as a black box that merges K sorted lists of total size K^3 using $\mathcal{O}\left(\frac{K^3}{B} \log_{M/B} \frac{K^3}{B} + K\right)$ memory transfers. The space occupied by a K-Funnel is $\Theta(K^2)$.

Once we have such a fast merging procedure, we can sort using a K-way mergesort. How should we choose K? The larger the K, the faster the algorithm, because we cannot predict the optimal (M/B) multiplicity of the merge. This property suggests choosing $K = N$, in which case the entire sorting algorithm is in the merge. In fact, however, a

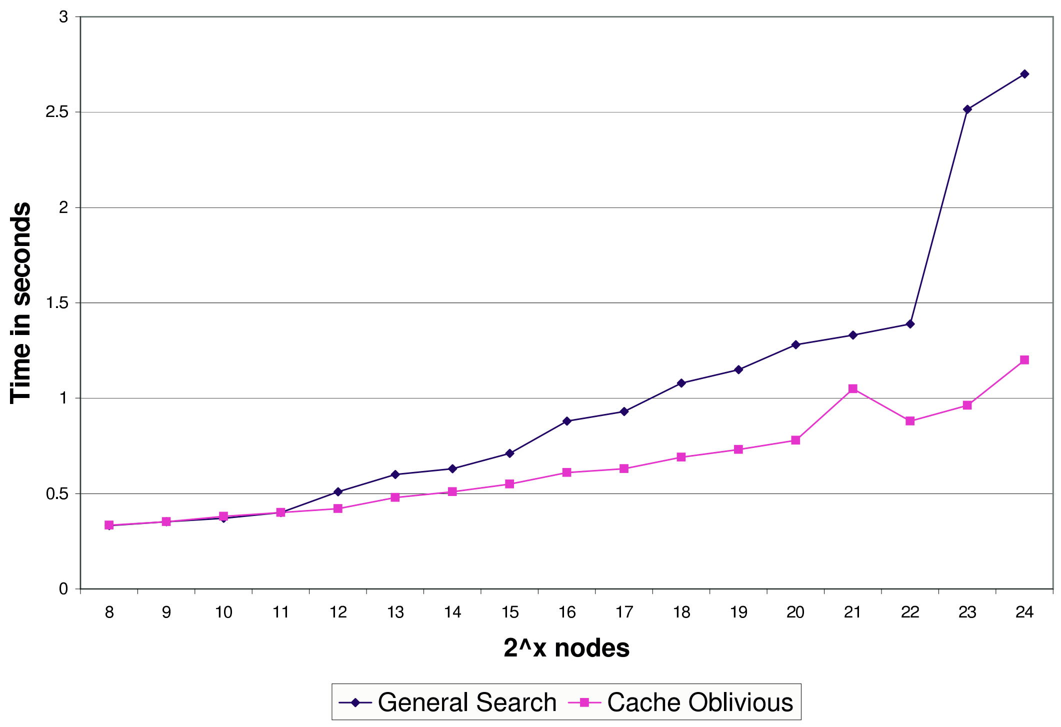


Figure 5.10: Comparison of Van Emde Boas searches with pre-order searches on a balanced binary tree. Similar to the last experiment, this experiment was performed on a Itanium with 48 byte node size.

K-Funnel is fast only if it is fed at least K^3 elements. Also, a K-Funnel occupies $\Theta(K^2)$ space, and we want a linear-space algorithm. Thus, we choose $K = N^{1/3}$.
Now the sorting algorithm proceeds as follows:

1. Split the array into $K = N^{1/3}$ contiguous segments each of size $N/K = N^{2/3}$.

2. Recursively sort each segment.

3. Apply the K-Funnel to merge the sorted segments.

Memory transfers are made just in Steps 2 and 3, leading to the recurrence:
$T(N) = N^{1/3}T(N^{2/3}) + \mathcal{O}\left(\frac{N}{B}\log_{M/B} N/B + N^{1/3}\right)$

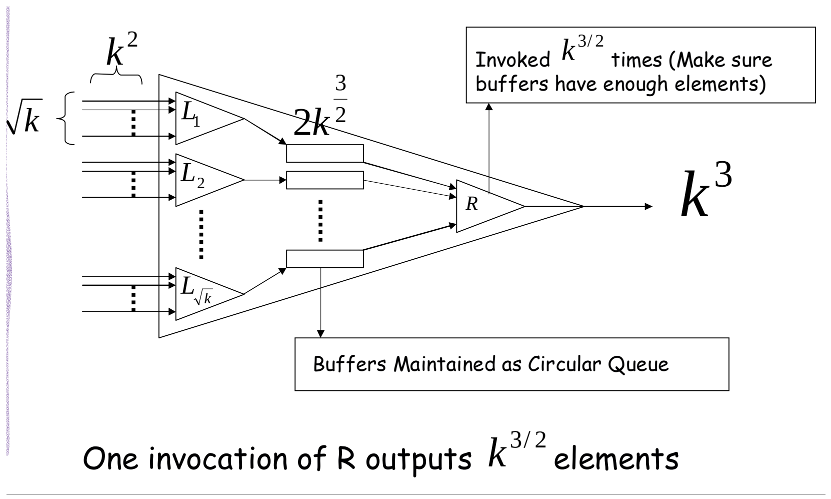

k^2
$\sqrt{k}$
L_1
L_2
$L_{\sqrt{k}}$
$2k^{\frac{3}{2}}$
R
k^3
Invoked $k^{3/2}$ times (Make sure buffers have enough elements)
Buffers Maintained as Circular Queue
One invocation of R outputs $k^{3/2}$ elements

Figure 5.11: Example of a funnel merger

The base case is $T(\mathcal{O}(B^2)) = \mathcal{O}(B)$ because the tall-cache assumption says that $M \geq B^2$. Above the base case, $N = \Omega(B^2)$, so $B = \sqrt{N}$, and the $N/B \log \ldots$ cost dominates the $N^{1/3}$ cost.
The recursion tree has N/B^2 leaves, each costing $\mathcal{O}\big(B \log_{M/B} B + B^{1/3}\big) = \mathcal{O}(B)$ memory transfers, for a total leaf cost of $\mathcal{O}(N/B)$. The root divide-and-merge cost is $\mathcal{O}\big(\frac{N}{B} \log_{M/B} N/B\big)$, which dominates the recurrence. Thus, modulo the details of the funnel, we have proved the following theorem:

Theorem 3 *Assuming $M = \Omega(B^2)$, funnelsort sorts N comparable elements in $\mathcal{O}\big(\frac{N}{B} \log_{M/B} N/B\big)$ memory transfers.*

It can also be shown that the number of comparisons is $\mathcal{O}(N \log N)$; see [18] for details.

Now, how do K-Funnels look like? Our goal is to develop a K-Funnel which merges K sorted lists of total size K3 using $\mathcal{O}\left(\frac{K^3}{B} \log_{M/B} \frac{K^3}{B} + K\right)$ memory transfers and $\Omega(K^2)$ space.

A K-Funnel is a complete binary tree with K leaves, stored according to the van Emde Boas layout we saw in 5.5.2. Thus, each of the recursive subtrees of a K-Funnel is a $\sqrt{K}$-funnel. In addition to the nodes, edges in a K-Funnel store buffers; see figure 5.11. The edges at the middle level of a K-Funnel, partitioning the funnel into two recursive $\sqrt{K}$-subfunnels, have size $K^{3/2}$ each, for a total buffer size of K^2 at that level. Buffers within the subfunnels are recursively smaller. We store these buffers of size $K^{3/2}$ in the recursive layout alongside the recursive pK-subfunnels within the K-Funnel. The buffers can be stored in an arbitrary order along with the recursive subtrees.

For consistency in describing the algorithms, we view a K-Funnel as having an additional buffer of size K^3 along the edge connecting the root of the tree to its imaginary parent. To maintain the lemma above that the storage is $\mathcal{O}(K^2)$, this buffer is not actually stored; rather, it can be viewed as the output mechanism. The algorithm to fill this buffer above the root node, thereby merging the entire input, is a simple recursion. We merge the elements in the buffers along the left and right children edges of the node, as long as those two buffers remain nonempty. (Initially, all buffers are empty.) Whenever either of the buffers be- comes empty, we recursively fill it. At the bottom of the tree, a leaf buffer (a buffer immediately below a leaf) corresponds to one of the input lists.

For the analysis on K-Funnels, we refer to [18].

5.5.4 Is the Model an Oversimplification?

In theory, both the cache oblivious and the external memory models are nice to work with, because of their simplicity. A lot of the work done in the external memory model has been turned into practical results as well. Before one makes his hand "dirty" with implementing an algorithm in the cache oblivious or the external memory model, one should be aware of practical things that might become detrimental to the speed of the code but are not caught in the theoretical setup.

Here we list a few practical glitches that are shared by both the cache oblivious and the external memory model. The ones that are not shared are marked[2] accordingly. A reader that wants to use these models to design practical algorithms and especially one who wants to write code, should keep these issues in mind. Code written and algorithms designed keeping the following things in mind, could be a lot faster than just directly coding an algorithm that is optimal in either the cache oblivious or the external memory model.

TLB^o: TLBs are caches on page tables, are usually small with 128-256 entries and are like just any other cache. They can be implemented as fully associative. The model does not take into account the fact that TLBs are not tall.

Concurrency: The model does not talk about I/O and CPU concurrency, which automatically looses it a 2x factor in terms of constants. The need for speed might drive future *uniprocessor* systems to diversify and look for alternative solutions in terms of concurrency on a single chip, for instance the hyper-threading[3] introduced by Intel in its latest Xeons is a glaring example. On these kind of systems and other multiprocessor systems,

[2]A superscript 'o' means this issue only applies to the cache oblivious model.

[3]One physical processor Intel Xeon MP forms two logical processors which share CPU computational resources The software sees two CPUs and can distribute work load between them as a normal dual processor system.

coherence misses might become an issue. This is hard to capture in the cache oblivious model and for most algorithms that have been devised in this model already, concurrency is still an open problem. A parallel cache oblivious model would be really welcome for practitioners who would like to apply cache oblivious algorithms to multiprocessor systems.
Associativity^o: The assumption of the fully associative cache is not so nice. In reality caches are either direct mapped or k-way associative (typically $k = 2, 4, 8$). If two objects map to the same location in the cache and are referenced in temporal proximity, the accesses will become costlier than they are assumed in the model (also known as cache interference problem). Also, $k-$way set associative caches are implemented by using more comparators.
Instruction/Unified Caches: Rarely executed, special case code disrupts locality. Loops with few iterations that call other routines make loop locality hard to exploit and plenty of loopless code hampers temporal locality. Issues related to instruction caches are not modeled in the cache oblivious model. *Unified caches* (e.g. the latest Intel Itanium chips L2 and L3 caches) are used in some machines where instruction and data caches are merged(e.g. Intel PIII, Itaniums). These are another challenge to handle in the model.
Replacement Policy^o: Current operating systems do not page more than 4GB of memory because of address space limitations. That means one would have to use legacy code on these systems for paging. This problem makes portability of cache oblivious code for big problems a myth! In the experiments reported in this chapter, we could not do external memory experimentation because the OS did not allow us to allocate array sizes of more than a GB or so. One can overcome this problem by writing one's own paging system over the OS to do experimentation of cache oblivious algorithms on huge data sizes. But then its not so clear if writing a paging system is easier or handling disks explicitly in an application. This problem does not exist on 64-bit operating systems and should go away with time.
Multiple Disks^o: For "most" applications where data is huge and external memory algorithms are required, using Multiple disks is an option to increase I/O efficiency. As of now, the cache oblivious model does not take into account the existence of multiple disks in a system.
Write-through caches^o: L1 caches in many new CPUs is write through, i.e. it transmits a written value to L2 cache immediately. Write through caches are simpler to manage and can always discard cache data without any bookkeeping (Read misses can not result in writes). With write through caches (e.g. DECStation 3100, Intel Itanium), one can no longer argue that there are no misses once the problem size fits into cache! *Victim Caches* implemented in HP and Alpha machines are caches that are implemented as small buffers to reduce the effect of conflicts in set-associative caches. These also should be kept in mind when designing code for these machines.
Complicated Algorithms^o and Asymptotics: For non-trivial problems the algorithms

can become quite complicated and impractical, a glaring instance of which is sorting. The speed by which different levels of memory differ in data transfer are constants! For instance the speed difference between L1 and L2 caches on a typical Intel pentium can be around 10. Using an $O()$ notation for an algorithm that is trying to beat a constant of 10, and sometimes not even talking about those constants while designing algorithms can show up in practice). For instance there are "constants" involved in simulating a fully associative cache on a k-way associative cache. Not using I/O concurrently with CPU can make an algorithm loose another constant. Can one really afford to hide these constants in the design of a cache oblivious algorithm in real code?

Despite these limitations the model does perform very well for some applications, but might be outperformed by more coding effort combined with cache aware algorithms. Here's an intercept from an experimental paper by Chatterjee and Sen:

> Our major conclusion are as follows: Limited associativity in the mapping from main memory addresses to cache sets can significantly degrade running time; the limited number of TLB entries can easily lead to thrashing; the fanciest optimal algorithms are not competitive on real machines even at fairly large problem sizes unless cache miss penalties are quite high; low level performance tuning "hacks", such as register tiling and array alignment, can significantly distort the effect of improved algorithms, ...

5.6 External BFS

The material of this section was taken from [12].

5.6.1 Introduction

Large graphs arise naturally in many applications and very often we need to traverse these graphs for solving optimization problems. Breadth first search (BFS) is a fundamental graph traversal strategy. It decomposes the input graph $G = (V, E)$ of n nodes and m edges into at most n levels where level i comprises all nodes that can be reached from a designated source s via a path of i edges, but cannot be reached using less than i edges. Typical real-world applications of BFS on large graphs (and some of its generalizations like shortest paths or A^*) include crawling and analyzing the WWW, route planning using small navigation devices with flash memory cards and state space exploration.

BFS is well-understood in the RAM model. There exists a simple linear time algorithm (hereafter refered as IM_BFS) for the BFS traversal in a graph. IM_BFS keeps a set of appropriate candidate nodes for the next vertex to be visited in a FIFO queue Q. Furthermore, in order to find out the unvisited neighbours of a node from its adjacency list, it marks the nodes as either visited or unvisited. Unfortunately, even when half of the

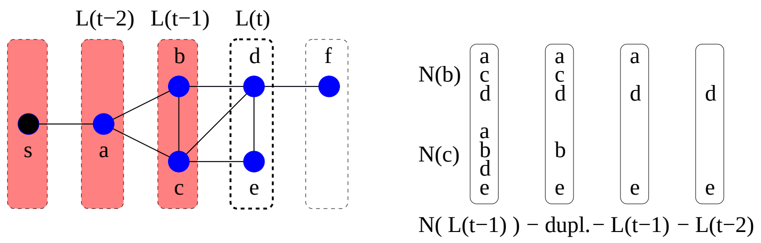


Figure 5.12: A phase in the BFS algorithm of Munagala and Ranade. Level $L(t)$ is composed out of the disjoint neighbor vertices of level $L(t-1)$ excluding those vertices already existing in either $L(t-2)$ or $L(t-1)$.

graph fits in the main memory, the running time of this algorithm deviates significantly from the predicted RAM performance (*hours* as compared to *minutes*) and for massive graphs, such approaches are simply non-viable. As discussed before, the main cause for such a poor performance of this algorithm on massive graphs is the number of I/Os it incurs. Remembering visited nodes needs $\Theta(m)$ I/Os in the worst case and the unstructured indexed access to adjacency lists may result in $\Theta(n)$ I/Os.

5.6.2 Algorithm of Munagala and Ranade

We turn to the basic BFS algorithm of Munagala and Ranade [14], MR_BFS for short.

Let $L(t)$ denote the set of nodes in BFS level t, and let $|L(t)|$ be the number of nodes in $L(t)$. MR_BFS builds $L(t)$ as follows: let $A(t) := N(L(t-1))$ be the multi-set of neighbor vertices of nodes in $L(t-1)$; $N(L(t-1))$ is created by $|L(t-1)|$ accesses to the adjacency lists, one for each node in $L(t-1)$. Since the graph is stored in adjacency-list representation, this takes $\mathcal{O}\left(|L(t-1)| + |N(L(t-1))|/B\right)$ I/Os. Then the algorithm removes duplicates from the multi-set A. This can be done by sorting $A(t)$ according to the node indices, followed by a scan and compaction phase; hence, the duplicate elimination takes $\mathcal{O}(\mathrm{sort}(|A(t)|)$ I/Os. The resulting set $A'(t)$ is still sorted.

Now the algorithm computes $L(t) := A'(t) \setminus \{L(t-1) \cup L(t-2)\}$. Fig. 5.12 provides an example. Filtering out the nodes already contained in the sorted lists $L(t-1)$ or $L(t-2)$ is possible by parallel scanning. Therefore, this step can be done using

$$\mathcal{O}\Big(\mathrm{sort}\big(\,|N(L(t-1))|\,\big) + \mathrm{scan}\big(\,|L(t-1)| + |L(t-2)|\,\big)\Big) \text{ I/Os.}$$

Since $\sum_t |N(L(t))| = \mathcal{O}(|E|)$ and $\sum_t |L(t)| = \mathcal{O}(|V|)$, the whole execution of MR_BFS requires $\mathcal{O}(|V| + \mathrm{sort}(|E|))$ I/Os.

The correctness of this BFS algorithm crucially depends on the fact that the input graph is undirected. Assume that the levels $L(0), \ldots, L(t-1)$ have already been computed correctly. We consider a neighbor v of a node $u \in L(t-1)$: the distance from s

to v is at least $t-2$ because otherwise the distance of u would be less than $t-1$. Thus $v \in L(t-2) \cup L(t-1) \cup L(t)$ and hence it is correct to assign precisely the nodes in $A'(t) \setminus \{L(t-1) \cup L(t-2)\}$ to $L(t)$.

Theorem 4 *BFS on arbitrary undirected graphs can be solved using* $\mathcal{O}(|V| + \mathrm{sort}(|V| + |E|))$ *I/Os.*

5.6.3 An Improved BFS Algorithm with sublinear I/O

The MM_BFS algorithm of Mehlhorn and Meyer [15] refines the approach of Munagala and Ranade [14]. It trades-off unstructured I/Os with increasing the number of iterations in which an edge may be involved. MM_BFS operates in two phases: in a first phase it preprocesses the graph and in a second phase it performs BFS using the information gathered in the first phase. We first sketch a variant with a randomized preprocessing. Then we outline a deterministic version.

The Randomized Partitioning Phase

The preprocessing step partitions the graph into disjoint connected subgraphs $\mathcal{S}_i$, $0 \leq i \leq K$, with small expected diameter. It also partitions the adjacency lists accordingly, i.e., it constructs an external file $\mathcal{F} = \mathcal{F}_0\mathcal{F}_1 \ldots \mathcal{F}_i \ldots \mathcal{F}_{K-1}$ where $\mathcal{F}_i$ contains the adjacency lists of all nodes in $\mathcal{S}_i$. The partition is built by choosing *master nodes* independently and uniformly at random with probability $\mu = \min\{1, \sqrt{(|V| + |E|)/(B \cdot |V|)}\}$ and running a local BFS from all master nodes "in parallel" (for technical reasons, the source node s is made the master node of $\mathcal{S}_0$): in each round, each master node s_i tries to capture all unvisited neighbors of its current sub-graph $\mathcal{S}_i$; this is done by first sorting the nodes of the active fringes of all $\mathcal{S}_i$ (the nodes that have been captured in the previous round) and then scanning the dynamically shrinking adjacency-lists representation of the yet unexplored graph. If several master nodes want to include a certain node v into their partitions then an arbitrary master node among them succeeds. The selection can be done by sorting and scanning the created set of neighbor nodes.

The expected number of master nodes is $K := \mathcal{O}(1 + \mu \cdot n)$ and the expected shortest-path distance (number of edges) between any two nodes of a subgraph is at most $2/\mu$. Hence, the expected total amount of data being scanned from the adjacency-lists representation during the "parallel partition growing" is bounded by

$$X := \mathcal{O}(\sum_{v \in V} 1/\mu \cdot (1 + \mathrm{degree}(v))) = \mathcal{O}((|V| + |E|)/\mu).$$

The total number of fringe nodes and neighbor nodes sorted and scanned during the partitioning is at most $Y := \mathcal{O}(|V| + |E|)$. Therefore, the partitioning requires

$$\mathcal{O}(\mathrm{scan}(X) + \mathrm{sort}(Y)) = \mathcal{O}(\mathrm{scan}(|V| + |E|)/\mu + \mathrm{sort}(|V| + |E|))$$

expected I/Os.

After the partitioning phase each node knows the (index of the) subgraph to which it belongs. With a constant number of sort and scan operations MM_BFS can reorganize the adjacency lists into the format $\mathcal{F}_0\mathcal{F}_1 \ldots \mathcal{F}_i \ldots \mathcal{F}_{|\mathcal{S}|-1}$, where $\mathcal{F}_i$ contains the adjacency lists of the nodes in partition $\mathcal{S}_i$; an entry $(v, w, \mathcal{S}(w), f_{\mathcal{S}(w)})$ from the adjacency list of $v \in \mathcal{F}_i$ stands for the edge (v, w) and provides the additional information that w belongs to subgraph $\mathcal{S}(w)$ whose subfile $\mathcal{F}_{\mathcal{S}(w)}$ starts at position $f_{\mathcal{S}(w)}$ within $\mathcal{F}$. The edge entries of each $\mathcal{F}_i$ are lexicographically sorted. In total, $\mathcal{F}$ occupies $\mathcal{O}((|V| + |E|)/B)$ blocks of external storage.

The BFS Phase

In the second phase the algorithm performs BFS as described by Munagala and Ranade (Section 5.6.2) with one crucial difference: MM_BFS maintains an external file $\mathcal{H}$ (= hot adjacency lists); it comprises unused parts of subfiles $\mathcal{F}_i$ that contain a node in the current level $L(t-1)$. MM_BFS initializes $\mathcal{H}$ with $\mathcal{F}_0$. Thus, initially, $\mathcal{H}$ contains the adjacency list of the root node s of level $L(0)$. The nodes of each created BFS level will also carry identifiers for the subfiles $\mathcal{F}_i$ of their respective subgraphs $\mathcal{S}_i$.

When creating level $L(t)$ based on $L(t-1)$ and $L(t-2)$, MM_BFS does not access single adjacency lists like MR_BFS does. Instead, it performs a parallel scan of the sorted lists $L(t-1)$ and $\mathcal{H}$ and extracts $N(L(t-1))$; In order to maintain the invariant that $\mathcal{H}$ contains the adjacency lists of all vertices on the current level, the subfiles $\mathcal{F}_i$ of nodes whose adjacency lists are not yet included in $\mathcal{H}$ will be merged with $\mathcal{H}$. This can be done by first sorting the respective subfiles and then merging the sorted set with $\mathcal{H}$ using one scan. Each subfile $\mathcal{F}_i$ is added to $\mathcal{H}$ at most once. After an adjacency list was copied to $\mathcal{H}$, it will be used only for $\mathcal{O}(1/\mu)$ expected steps; afterwards it can be discarded from $\mathcal{H}$. Thus, the expected total data volume for scanning $\mathcal{H}$ is $\mathcal{O}(1/\mu \cdot (|V| + |E|))$, and the expected total number of I/Os to handle $\mathcal{H}$ and $\mathcal{F}_i$ is $\mathcal{O}\left(\mu \cdot |V| + \mathrm{sort}(|V| + |E|) + 1/\mu \cdot \mathrm{scan}(|V| + |E|)\right)$. The final result follows with $\mu = \min\{1, \sqrt{\mathrm{scan}(|V| + |E|)/|V|}\}$.

Theorem 5 ([15]) *External memory BFS on undirected graphs can be solved using* $\mathcal{O}\left(\sqrt{|V| \cdot \mathrm{scan}(|V| + |E|)} + \mathrm{sort}(|V| + |E|)\right)$ *expected I/Os.*

The Deterministic Variant

In order to obtain the result of Theorem 5 in the worst case, too, it is sufficient to modify the preprocessing phase as follows: instead of growing subgraphs around randomly selected master nodes, the deterministic variant extracts the subfiles $\mathcal{F}_i$ from an Euler Tour around a spanning tree for the connected component C_s that contains the source node s.

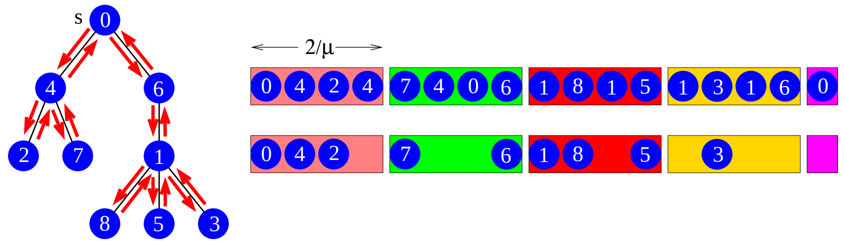


Figure 5.13: Using an Euler tour around a spanning tree of the input graph in order to obtain a partition for the deterministic BFS algorithm.

Observe that C_s can be obtained with the deterministic connected-components algorithm of [14] using
$\mathcal{O}((1 + \log\log(B \cdot |V|/|E|)) \cdot \mathrm{sort}(|V| + |E|))$ =
$\mathcal{O}(\sqrt{|V| \cdot \mathrm{scan}(|V| + |E|)} + \mathrm{sort}(|V| + |E|))$ I/Os. The same number of I/Os suffices to compute a (minimum) spanning tree T_s for C_s [20].

After T_s has been built, the preprocessing constructs an Euler tour around T_s using a constant number of sort- and scan-steps [16]. Then the tour is broken at the root node s; the elements of the resulting list can be stored in consecutive order using the deterministic list ranking algorithm of [16]. This takes $\mathcal{O}(\mathrm{sort}(|V|))$ I/Os. Subsequently, the Euler tour can be cut into pieces of size $2/\mu$ in a single scan. These Euler tour pieces account for subgraphs $\mathcal{S}_i$ with the property that the distance between any two nodes of $\mathcal{S}_i$ in G is at most $2/\mu - 1$. See Fig. 5.13 for an example. Observe that a node v of degree d may be part of $\Theta(d)$ different subgraphs $\mathcal{S}_i$. However, with a constant number of sorting steps it is possible to remove multiple node appearances and make sure that each node of C_s is part of exactly one subgraph $\mathcal{S}_i$. Eventually, the reduced subgraphs $\mathcal{S}_i$ are used to create the reordered adjacency-list files $\mathcal{F}_i$; this is done as in the randomized preprocessing and takes another $\mathcal{O}(\mathrm{sort}(|V| + |E|))$ I/Os. Note that the reduced subgraphs $\mathcal{S}_i$ may not be connected any more; however, this does not matter as our approach only requires that any two nodes in a subgraph are relatively close in the original input graph.

The BFS-phase of the algorithm remains unchanged; the modified preprocessing, however, guarantees that each adjacency-list will be part of the external set $\mathcal{H}$ for at most $2/\mu$ BFS levels: if a subfile $\mathcal{F}_i$ is merged with $\mathcal{H}$ for BFS level $L(t)$, then the BFS level of any node v in $\mathcal{S}_i$ is at most $L(t) + 2/\mu - 1$. Therefore, the adjacency list of v in $\mathcal{F}_i$ will be kept in $\mathcal{H}$ for at most $2/\mu$ BFS levels.

Theorem 6 ([15]) *External memory BFS on undirected graphs can be solved using* $\mathcal{O}\left(\sqrt{|V| \cdot \mathrm{scan}(|V| + |E|)} + \mathrm{sort}(|V| + |E|)\right)$ *I/Os in the worst case.*

5.6.4 Improvements in the previous implementations of MR_BFS and MM_BFS_R

The computation of each level of MR_BFS involves sorting and scanning of neighbours of the nodes in the previous level. Even if there are very few elements to be sorted, there is a certain overhead associated with initializing the external sorters. In particular, while the Stxxl stream sorter (with the flag DStxxl_SMALL_INPUT_PSORT_OPT) does not incur an I/O for sorting less than B elements, it still requires to allocate some memory and does some computation for initialization. This overhead accumulates over all levels and for large diameter graphs, it dominates the running time. This problem is also inherited by the BFS phase of MM_BFS[4]. Since in the pipelined implementation of [17], we do not know in advance the exact number of elements to be sorted, we can't switch between the external and the internal sorter so easily. In order to get around this problem, we first buffer the first B elements and initialize the external sorter only when the buffer is full. Otherwise, we sort it internally.

In addition to this, we make the graph representation for MR_BFS more compact. Except the source and the destination node pair, no other information is stored with the edges.

Designing MM_BFS_D

Graph class	n	m	Long clusters	Random clusters
Grid($2^{14} \times 2^{14}$)	2^{28}	2^{29}	51	28

Table 5.3: Time taken (in hours) by the BFS phase of MM_BFS_D with long and random clustering

As for list ranking, we found Sibeyn's algorithm (we talk about in 5.9) promising as it has low constant factors in its I/O complexity. Sibeyn's implementation relies on the operating system for I/Os and does not guarantee that the top blocks of all the stacks remain in the internal memory, which is a necessary assumption for the asymptotic analysis of the algorithm. Besides, its reliance on internal arrays and swap space puts a restriction on the size of the lists it can rank. The deeper integration of the algorithm in the Stxxl framework, using the Stxxl stacks and vectors in particular, made it possible to obtain a scalable solution, which could handle graph instances of the size we require while keeping the theoretical worst case bounds.

[4]We use MM_BFS_R to refer to the randomized variant and MM_BFS_D for the deterministic variant of MM_BFS

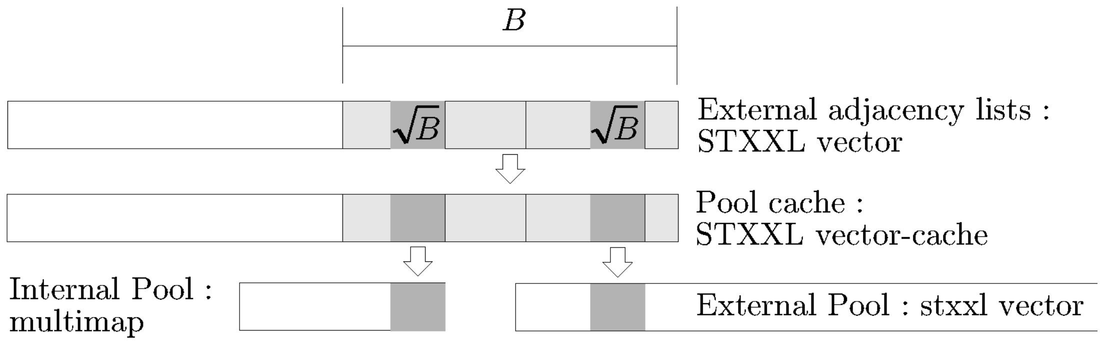

B
$\sqrt{B}$
$\sqrt{B}$
External adjacency lists : STXXL vector
Pool cache : STXXL vector-cache
Internal Pool : multimap
External Pool : stxxl vector

Figure 5.14: Schema depicting the implementation of our heuristic

To summarize, our Stxxl based implementation of MM_BFS_D uses our adaptation of Sibeyn's algorithm for list ranking the Euler tour around the minimum spanning tree computed by EM_MST. The Euler tour is then chopped into sets of $\sqrt{B}$ consecutive nodes which after duplicate removal gives the requisite graph partitioning. The BFS phase remains similar to MM_BFS_R.

Quality of the spanning tree The quality of the spanning tree computed can have a significant impact on the clustering and the disk layout of the adjacency list after the deterministic preprocessing, and consequently on the BFS phase. For instance, in the case of grid graph, a spanning tree containing a list with elements in a snake-like row major order produces long and narrow clusters, while a "random" spanning tree is likely to result in clusters with low diameters. Such a "random" spanning tree can be attained by assigning random weights to the edges of the graph and then computing a minimum spanning tree or by randomly permuting the indices of the nodes. The nodes in the long and narrow clusters tend to stay longer in the pool and therefore, their adjacency lists are scanned more often. This causes the pool to grow external and results in larger I/O volume. On the other hand, low diameter clusters are evicted from the pool sooner and are scanned less often reducing the I/O volume of the BFS phase. Consequently as Table 5.3 shows, the BFS phase of MM_BFS_D takes only 28 hours with clusters produced by "random" spanning tree, while it takes 51 hours with long and narrow clusters.

5.6.5 A Heuristic for maintaining the pool

As noted above, the asymptotic improvement and the performance gain in MM_BFS over MR_BFS is obtained by decomposing the graph into low diameter clusters and maintaining an efficiently accessible pool of adjacency lists which will be required in the next few levels. Whenever the first node of a cluster is visited during the BFS, the remaining nodes of this cluster will be reached soon after and hence, this cluster is loaded into the pool. For computing the neighbours of the nodes in the current level, we just need to scan the pool and not the entire graph. Efficient management of this pool is thus, crucial for the performance of MM_BFS. In this section, we propose a heuristic for efficient management of the pool, while keeping the worst case I/O bounds of MM_BFS.

For many large diameter graphs, the pool fits into the internal memory most of the time. However, even if the number of edges in the pool is not so large, scanning all the edges in the pool for each level can be computationally quite expensive. Hence, we keep a portion of the pool that fits in the internal memory as a multi-map hash table. Given a node as a key, it returns all the nodes adjacent to the current node. Thus, to get the neighbours of a set of nodes we just query the hash function for those nodes and delete them from the hash table. For loading the cluster, we just insert all the adjacency lists of the cluster in the hash table, unless the hash table has already $O(M)$ elements.

Recall that after the deterministic preprocessing, the elements are stored on the disk in the order in which they appear on the Euler tour around a spanning tree of the input graph. The Euler tour is then chopped into clusters with $\sqrt{B}$ elements (before the duplicate removal) ensuring that the maximum distance between any two nodes in the cluster is at most $\sqrt{B} - 1$. However, the fact that the contiguous elements on the disk are also closer in terms of BFS levels is not restricted to intra-cluster adjacency lists. The adjacency lists that come alongside the requisite cluster will also be required soon and by caching these other adjacency lists, we can save the I/Os in the future. This caching is particularly beneficial when the pool fits in the internal memory. Note that we still load the $\sqrt{B}$ node clusters in the pool, but keep the remaining elements of the block in the pool-cache. For the line graphs, this means that we load the $\sqrt{B}$ nodes in the internal pool, while keeping the remaining $O(B)$ adjacency lists which we get in the same block, in the pool-cache, thereby reducing the I/O complexity for the BFS traversal on line graphs to the computation of a spanning tree.

We represent the adjacency lists of nodes in the graph as a Stxxl vector. Stxxl already provides a fully associative vector-cache with every vector. Before doing an I/O for loading a block of elements from the vector, it first checks if the block is already there in the vector-cache. If so, it avoids the I/O loading the elements from the cache instead. Increasing the vector-cache size of the adjacency list vector with a layout computed by the deterministic preprocessing and choosing the replacement policy to be LRU provides us with an implementation of the pool-cache. Figure 5.14 depicts the implementation of our heuristic.

5.7 Maximal Independent Set

In this section we describe a simple technique proposed in [21] that can be used to make internal memory graph algorithms of a sufficiently simple structure I/O-efficient. For this technique to be applicable, the algorithm has to compute a labelling of the vertices of the graph, and it has to do so in a particular way. We call a vertex labelling algorithm $\mathcal{A}$ *single-pass* if it computes the desired labelling λ of the vertices of the graph by visiting every vertex exactly once and assigns label $\lambda(v)$ to v during this visit. We call $\mathcal{A}$ *local* if label $\lambda(v)$ can be computed in $\mathcal{O}(\mathrm{sort}(k))$ I/Os from labels $\lambda(u_1), \ldots, \lambda(u_k)$, where $u_1, \ldots, u_k$ are the neighbors of v whose labels are computed before $\lambda(v)$. Finally, algorithm $\mathcal{A}$ is *presortable* if there is an algorithm that takes $\mathcal{O}(\mathrm{sort}(|V| + |E|))$ I/Os to compute an order of the vertices of the graph so that $\mathcal{A}$ produces a correct result if it visits the vertices of the graph in this order. The technique we describe here is applicable if algorithm $\mathcal{A}$ is presortable, local, and single-pass.

So let $\mathcal{A}$ be a presortable local single-pass vertex-labelling algorithm computing some labelling λ of the vertices of a graph $G = (V, E)$. In order to make algorithm $\mathcal{A}$ I/O-efficient, the two main problems are to determine an order in which algorithm $\mathcal{A}$ should visit the vertices of G and devise a mechanism that provides every vertex v with the labels of its previously visited neighbors $u_1, \ldots, u_k$. Since algorithm $\mathcal{A}$ is presortable, there exists an algorithm $\mathcal{A}'$ that takes $\mathcal{O}(\mathrm{sort}(|V| + |E|))$ I/Os to compute an order of the vertices of G so that algorithm $\mathcal{A}$ produces the correct result if it visits the vertices of G in this order. Assume w.l.o.g. that this ordering of the vertices of G is expressed as a numbering. We use algorithm $\mathcal{A}'$ to number the vertices of G and then derive a DAG G' from G by directing every edge of G from the vertex with smaller number to the vertex with larger number. DAG G' has the property that for every vertex v, the in-neighbors of v in G' are exactly those neighbors of v that are labelled before v. Hence, labelling λ can be computed using time-forward processing. In particular, by the locality of $\mathcal{A}$, the label $\lambda(v)$ of every vertex can be computed in $\mathcal{O}(\mathrm{sort}(k))$ I/Os from the labels $\lambda(u_1), \ldots, \lambda(u_k)$ of its in-neighbors $u_1, \ldots, u_k$ in G', which is a simplified version of the condition for the applicability of time-forward processing. This leads to the following result.

Theorem 7 *[21] Every graph problem $\mathcal{P}$ that can be solved by a presortable local single-pass vertex labelling algorithm can be solved in $\mathcal{O}(\mathrm{sort}(|V| + |E|))$ I/Os.*

An important observation to be made is that in this application of time-forward processing, the restriction that the vertices of the DAG to be evaluated have to be given in topologically sorted order does not pose a problem because the directions of the edges are chosen only after fixing an order of the vertices that is to be the topological order.

In order to compute a *maximal independent set* S of a graph $G = (V, E)$ in internal memory, the following simple algorithm can be used: *Process the vertices in an arbitrary order. When a vertex $v \in V$ is visited, add it to S if none of its neighbors is in S.*

Translated into a labelling problem, the goal is to compute the characteristic function $\chi_S : V \to \{0, 1\}$ of S, where $\chi_S(v) = 1$ if $v \in S$, and $\chi_S(v) = 0$ if $v \notin S$. Also note that if S is initially empty, then any neighbor w of v that is visited after v cannot be in S at the time when v is visited, so that it is sufficient for v to inspect all its neighbors that are visited before v to decide whether or not v should be added to S. The result of these modifications is a vertex-labelling algorithm that is presortable (since the order in which the vertices are visited is unimportant), local (since only previously visited neighbors of v are inspected to decide whether v should be added to S, and a single scan of labels $\chi_S(u_1), \ldots, \chi_S(u_k)$ suffices to do so), and single-pass. This leads to the following result.

Theorem 8 *Given an undirected graph $G = (V, E)$, a maximal independent set of G can be found in $\mathcal{O}(\text{sort}(|V| + |E|))$ I/Os and linear space.*

5.8 Euler Tours

An *Euler tour* of a tree $T = (V, E)$ is a traversal of T that traverses every edge exactly twice, once in each direction. Such a traversal is useful, as it produces a linear list of vertices or edges that captures the structure of the tree. Hence, it allows standard parallel or external memory algorithms to be applied to this list, in order to solve problems on tree T that can be expressed as some function to be evaluated over the Euler tour.

Formally, the tour is represented as a linked list L whose elements are the edges in the set $\{(v, w), (w, v) : \{v, w\} \in E\}$ and so that for any two consecutive edges e_1 and e_2, the target of e_1 is the source of e_2. In order to define an Euler tour, choose a circular order of the edges incident to each vertex of T. Let $\{v, w_1\}, \ldots, \{v, w_k\}$ be the edges incident to vertex v. Then let $\text{succ}((w_i, v)) = (v, w_{i+1})$, for $1 \leq i < k$, and $\text{succ}((w_k, v)) = (v, w_1)$. The result is a circular linked list of the edges in T. Now an Euler tour of T starting at some vertex r and returning to that vertex can be obtained by choosing an edge (v, r) with $\text{succ}((v, r)) = (r, w)$, setting $\text{succ}((v, r)) = \textbf{null}$, and choosing (r, w) as the first edge of the traversal.

List L can be computed from the edge set of T in $\mathcal{O}(\text{sort}(N))$ I/Os: First scan set E to replace every edge $\{v, w\}$ with two directed edges (v, w) and (w, v). Then sort the resulting set of directed edges by their target vertices. This stores the incoming edges of every vertex consecutively. Hence, a scan of the sorted edge list now suffices to compute the successor of every edge in L.

Theorem 9 *An Euler tour L of a tree with N vertices can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os.*

Given an unrooted (and undirected) tree T, choosing one vertex of T as the root defines a direction on the edges of T by requiring that every edge be directed from the parent to the child. The process of *rooting* tree T is that of computing these directions explicitly for all edges of T. To do this, we construct an Euler tour starting at an edge (r, v) and compute the rank of every edge in the list. For every pair of opposite edges (u, v) and (v, u), we call the edge with the lower rank a *forward edge*, and the other a *back edge*. Now it suffices to observe that for any vertex $x \neq r$ in T, edge $(parent(x), x)$ is traversed before edge $(x, parent(x))$ by any Euler tour starting at r. Hence, for every pair of adjacent vertices x and $parent(x)$, edge $(parent(x), x)$ is a forward edge, and edge $(x, parent(x))$ is a back edge. That is, the set of forward edges is the desired set of edges directed from parents to children. Constructing and ranking an Euler tour starting at the root r takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os. Given the ranks of all edges, the set of forward edges can be extracted by sorting all edges in L so that for any two adjacent vertices v and w, edges (v, w) and (w, v) are stored consecutively and then scanning this sorted edge list to discard the edge with higher rank from each of these edge pairs. Hence, a tree T can be rooted in $\mathcal{O}(\mathrm{sort}(N))$ I/Os.

Instead of discarding back edges, it may be useful to keep them, but tag every edge of the Euler tour L as either a forward or back edge. Using this information, well-known labellings of the vertices of T can be computed by ranking list L after assigning appropriate weights to the edges of L. For example, consider the weighted ranks of the edges in L after assigning weight one to every forward edge and weight zero to every back edge. Then the *preorder* number of every vertex $v \neq r$ in T is one more than the weighted rank of the forward edge with target v; the preorder number of the root r is always one. The size of the subtree rooted at v is one more than the difference between the weighted ranks of the back edge with source v and the forward edge with target v. To compute a *postorder* numbering, we assign weight zero to every forward edge and weight one to every back edge. Then the postorder number of every vertex $v \neq r$ is the weighted rank of the back edge with source v. The postorder number of the root r is always N.

After labelling every edge in L as a forward or back edge, the appropriate weights for computing the above labellings can be assigned in a single scan of list L. The weighted ranks can then be computed in $\mathcal{O}(\mathrm{sort}(N))$ I/Os, by 10. Extracting preorder and postorder numbers from these ranks takes a single scan of list L again. To extract the sizes of the subtrees rooted at the vertices of T, we sort the edges in L so that opposite edges with the same endpoints are stored consecutively. Then a single scan of this sorted edge list suffices to compute the size of the subtree rooted at every vertex v. Hence, all these labels can be computed in $\mathcal{O}(\mathrm{sort}(N))$ I/Os for a tree with N vertices.

5.9 List Ranking

List ranking and the Euler tour technique (5.8) are two techniques that have been applied successfully in the design of PRAM algorithms for labelling problems on lists and rooted

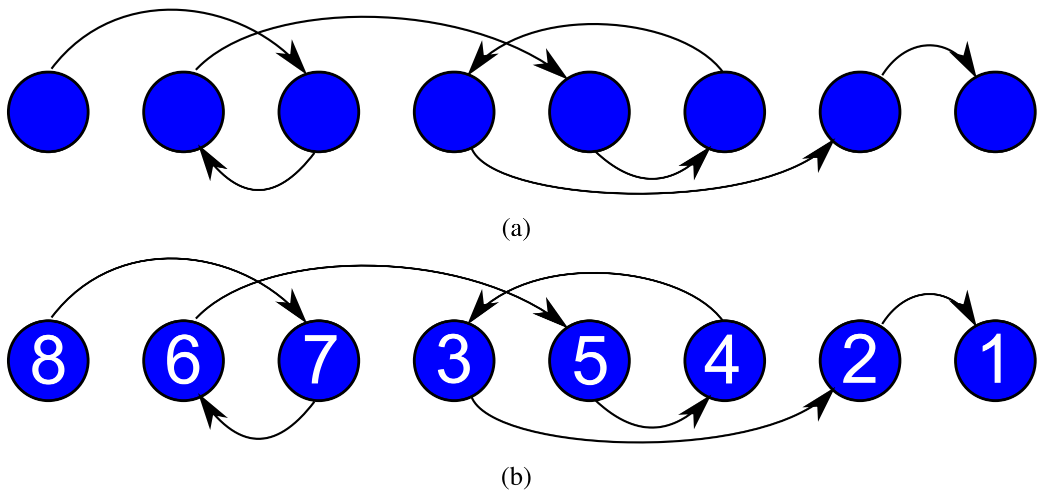


Figure 5.15: Example input and output for the list ranking task

trees and problems that can be reduced efficiently to one of these problems. Given the similarity of the issues to be addressed in parallel and external memory algorithms, it is not surprising that the same two techniques can be applied in I/O-efficient algorithms as well.

Let L be a linked list, i.e., a collection of vertices $x_1, \dots, x_N$ such that each vertex x_i, except the tail of the list, stores a pointer $\mathrm{succ}(x_i)$ to its successor in L, no two vertices have the same successor, and every vertex can reach the tail of L by following successor pointers. Given a pointer to the head of the list (i.e., the vertex that no other vertex in the list points to), the *list ranking* problem is that of computing for every vertex x_i of list L, its distance from the head of L, i.e., the number of edges on the path from the head of L to x_i.

In internal memory this problem can easily be solved in linear time using the following algorithm: *Starting at the head of the list, follow successor pointers and number the vertices of the list from* 0 *to* $N - 1$ *in the order they are visited.* Often we use the term "list ranking" to denote the following generalization of the list ranking problem, which is solvable in linear time using a straightforward generalization of the above algorithm: Given a function $\lambda : \{x_1, \dots, x_N\} \to X$ assigning labels to the vertices of list L and a multiplication $\otimes : X \times X \to X$ defined over X, compute a label $\phi(x_i)$ for each vertex x_i of L such that $\phi\big(x_{\sigma(1)}\big) = \lambda\big(x_{\sigma(1)}\big)$ and $\phi\big(x_{\sigma(i)}\big) = \phi\big(x_{\sigma(i-1)}\big) \otimes \lambda\big(x_{\sigma(i)}\big)$, for $1 < i \leq N$, where $\sigma : [1, N] \to [1, N]$ is a permutation so that $x_{\sigma(1)}$ is the head of L and $\mathrm{succ}\big(x_{\sigma(i)}\big) = x_{\sigma(i+1)}$, for $1 \leq i < N$.

Unfortunately the simple internal memory algorithm is not I/O-efficient: Since we have no control over the physical order of the vertices of L on disk, an adversary can easily arrange the vertices of L in a manner that forces the internal memory algorithm to perform one I/O per visited vertex, so that the algorithm performs $\Omega(N)$ I/Os in total. On the other hand, the lower bound for list ranking shown in [16] is only $\Omega(\mathrm{perm}(N))$. Next we sketch a list ranking algorithm proposed in [16] that takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os and thereby closes the gap between the lower and the upper bound.

We make the simplifying assumption that multiplication over X is associative. If this is not the case, we determine the distance of every vertex from the head of L, sort the vertices of L by increasing distances, and then compute the prefix product using the internal memory algorithm. After arranging the vertices by increasing distances from the head of L, the internal memory algorithm takes $\mathcal{O}(\mathrm{scan}(N))$ I/Os. Hence, the whole procedure still takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os, and the associativity assumption is not a restriction.

Given that multiplication over X is associative, the algorithm of [16] uses graph contraction to rank list L as follows: First an independent set I of L is found so that $|I| = \Omega(N)$. Then the elements in I are removed from L. That is, for every element $x \in I$ with predecessor y and successor z in L, the successor pointer of y is updated to succ$(y) = z$. The label of x is multiplied with the label of z, and the result is assigned to z as its new label in the compressed list. It is not hard to see that the weighted ranks of the elements in $L - I$ remain the same after adjusting the labels in this manner. Hence, their ranks can be computed by applying the list ranking algorithm recursively to the compressed list. Once the ranks of all elements in $L - I$ are known, the ranks of the elements in I are computed by multiplying their labels with the ranks of their predecessors in L.

If the algorithm excluding the recursive invocation on the compressed list takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os, the total I/O-complexity of the algorithm is given by the recurrence $\mathcal{I}(N) = \mathcal{I}(cN) + \mathcal{O}(\mathrm{sort}(N))$, for some constant $0 < c < 1$. The solution of this recurrence is $\mathcal{O}(\mathrm{sort}(N))$. Hence, we have to argue that every step, except the recursive invocation, can be carried out in $\mathcal{O}(\mathrm{sort}(N))$ I/Os.

Given independent set I, it suffices to sort the vertices in I by their successors and the vertices in $L - I$ by their own IDs, and then scan the resulting two sorted lists to update the weights of the successors of all elements in I. The successor pointers of the predecessors of all elements in I can be updated in the same manner. In particular, it suffices to sort the vertices in $L - I$ by their successors and the vertices in I by their own IDs, and then scan the two sorted lists to copy the successor pointer from each vertex in I to its predecessor. Thus, the construction of the compressed list takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os, once set I is given.

Theorem 10 *A list of length N can be ranked in $\mathcal{O}(\mathrm{sort}(N))$ I/Os.*

List ranking alone is of very limited use. However, combined with the Euler tour technique, it becomes a very powerful tool for solving problems on trees that can be expressed as functions over a traversal of the tree or problems on general graphs that can be expressed in terms of a traversal of a spanning tree of the graph. An important application is the *rooting* of an undirected tree T, which is the process of directing all edges of T from parents to children after choosing one vertex of T as the root. Given a rooted tree T (i.e., one where all edges are directed from parents to children), the Euler tour technique and list ranking can be used to compute a preorder or postorder numbering of the vertices of T, or the sizes of the subtrees rooted at the vertices of T. Such labellings are used in many classical graph algorithms, so that the ability to compute them is a first step towards solving more complicated graph problems.

Chapter 6

van Emde Boas Trees

The original description of this search tree was published in [22], the implementation study can be found in [23].

6.1 From theory to practice

Sorted lists with an auxiliary data structure that support the following operations on a sorted sequence s:

build: Build the data structure from a set of elements

insert: Insert an element

remove: Delete an element specified by a key or by a reference to that element.

locate: Given a key k, find the smallest element e in s such that $e \geq k$. If such an element does not exist, return an indication of this fact, i.e., a handle to a dummy element with key ∞.

range query: Return alle elements in s with key in a specified range $[k, k']$.

Sorted sequences are one of the most versatile data structures. In current algorithm libraries, they are implemented using comparison based data structures such as ab-trees, red-black trees, splay trees, or skip lists. These implementations support insertion, deletion, and search in time $\mathcal{O}(\log n)$ and range queries in time $\mathcal{O}(k + \log n)$ where n is the number of elements and k is the size of the output. For w bit integer keys, a theoretically attractive alternative are *van Emde Boas stratified trees* (vEB-trees) that replace the $\log n$ by a $\log w$: A vEB tree T for storing subsets M of $w = 2^{k+1}$ bit integers stores the set directly if $|M| = 1$. Otherwise it contains a root (hash) table r such that $r[i]$ points to a vEB tree T_i for 2^k bit integers. T_i represents the set $M_i = \{x \bmod 2^{2^k} : x \in M \wedge x \gg$

$2^k = i\}$.[1] Furthermore, T stores $\min M$, $\max M$, and a *top data structure* t consisting of a 2^k bit vEB tree storing the set $M_t = \left\{x \gg 2^k : x \in M\right\}$. This data structure takes space $\mathcal{O}(|M| \log w)$ and can be modified to consume only linear space. It can also be combined with a doubly linked sorted list to support fast successor and predecessor queries.

However, for a long time there was no known implementation of vEB-trees that could compete with comparison based data structures used in algorithm libraries. The following describes a specialized and highly tuned version of vEB-trees for storing 32-bit integers that can often outperform the classic data structures in terms of runtime.

Figure 6.1 outlines the transformation from a general vEB-tree to our specialized version. The starting point were vEB search trees as described above but we arrive at a nonrecursive data structure: We get a three level search tree. The root is represented by an *array* of size 2^{16} and the lower levels use hash tables of size up to 256. Due to this small size, hash functions can be implemented by table lookup. Locating entries in these tables is achieved using hierarchies of bit patterns.

The main operation we are interested in is `locate`(y). `locate` returns $\min(x \in M : y \leq x)$. Note that for plain lookup, a hash table would be faster than every data structure discussed here.

[todo: example figure with detailed explanation] $\Longleftarrow$

6.2 Implementation

Root Table

The root-table r contains a plain array with one entry for each possible value of the 16 most significant bits of the keys. $r[i] = \textbf{null}$ if there is no $x \in M$ with $x[16..31] = i$. If $|M_i| = 1$, it contains a pointer to the element list item corresponding to the unique element of M_i. Otherwise, $r[i]$ points to an L2-table containing $M_i = \{x \in M : x[16..31] = i\}$. The two latter cases can be distinguished using a flag stored in the least significant bit of the pointer.[2] Note that the root-table only uses 256kB memory and therefore easily fits into the cache.

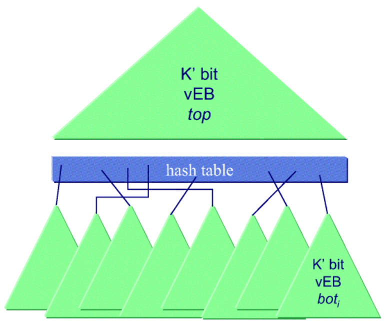
L2-table

An L2-table r_i stores the elements in M_i. If $|M_i| \geq 2$ it uses a hash table storing an entry with key j if $\exists x \in M_i : x[8..15] = j$.

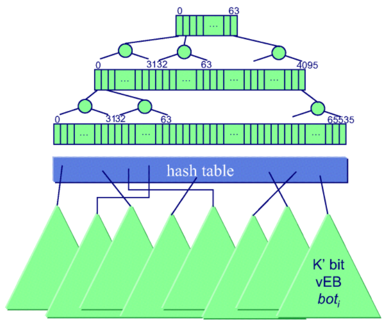
Let $M_{ij} = \{x \in M : x[8..15] = j, x[16..31] = i\}$. If $|M_{ij}| = 1$ the hash table entry points to the element list and if $|M_{ij}| \geq 2$ it points to an L3-table representing M_{ij} using

[1]We use the C-like shift operator '$\gg$', i.e., $x \gg i = \lfloor x/2^i \rfloor$.

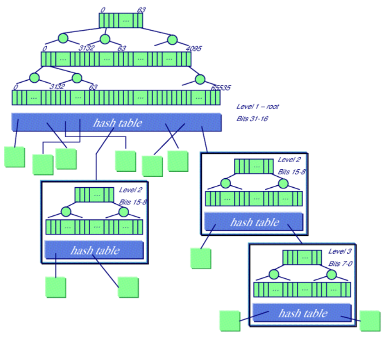
[2]This is portable without further measures because all modern systems use addresses that are multiples of four (except for strings).



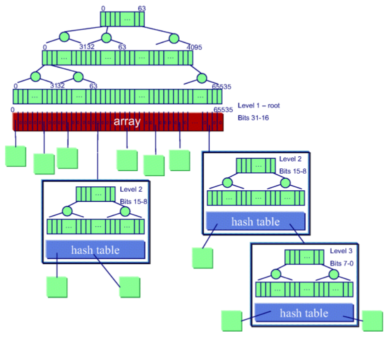
(a) The abstract definition



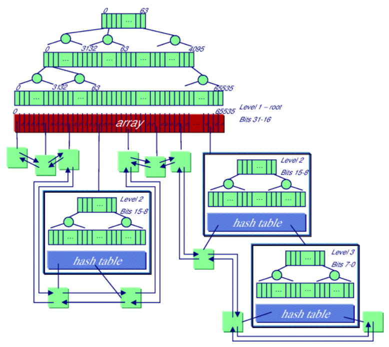
(b) Efficient inner-level lookup structures



(c) Removed recursion



(d) Replace large hash table with an array



(e) Allow range queries

Figure 6.1: Evolution of the vEB data structure

a similar trick as in the root-table.

L3-table

An L3-table r_{ij} stores the elements in M_{ij}. If $|M_{ij}| \geq 2$, it uses a hash table storing an entry with key k if $\exists x \in M_{ij} : x[0..7] = k$. This entry points to an item in the element list storing the element with $x[0..7] = k, x[8..15] = j, x[16..31] = i$.

Auxiliary data structures

To locate an element x in the data structure we first lookup $i = x[16 \dots 31]$ in the root-table. If $r[i] \neq \textbf{null}$ and $y \leq \max M_i$, we can proceed to the next level of the tree[3]. Otherwise, we have to find the subtree M_r with $r = \min \{k : k \geq i \wedge M_k \neq \textbf{null}\}$ (If no such j exists, we return ∞). To do this efficiently, we need some additional data structures. For every level L of the tree, we have some top data structures: t^1 and t^2 for every level, t^3 only for root level. We explain that concept for the root level. To find r, we first use t^1, which is a bit table containing a flag for every possible subtree of the root table, indicating if $M_i \neq \textbf{null}$. Via i `div` n we find the machine word a (of length n) in which i is located and check if it contains r by setting bits $\leq i$ to zero and checking for the most significant bit[4]. Only if $a = 0$ we have to inspect another word. To do that, we jump to t^2 in which every entry is an logical or over 32 bits in t^1. Analogously to t^1 we try to find the first nonnull word right of a. Again, we check only the word containing i and switch to t^3 (every entry is logical or over 32 bits in t^2) if unsuccessful. t^3 is 64 bit small and can be searched efficiently.

If we need to access L2 (when the located subtree contains more than one element), we calculate $j = x[8..15]$. j is our key for an hash table to locate the entry corresponding to x in L2. Repeat the procedure described above and possibly inspect L3 in the same manner. Figure 6.2 gives pseudo code for `locate`.

Our hash tables use open addressing with linear probing. The table size is always a power of two between 4 and 256[5]. The size is doubled when a table of size k contains more than $3k/4$ entries and $k < 256$. The table shrinks when it contains less than $k/4$ entries. Since all keys are between 0 and 255, we can afford to implement the hash function as a full lookup table h that is shared between all tables. This lookup table is

[3]We have to store the maximum of every subtree to do this efficiently

[4]Finding the position of the most significant bit can be implemented in constant time by converting the number to floating point and then inspecting the exponent. In our implementation, two 16-bit table lookups turn out to be somewhat faster.

[5]Note that this is much smaller than for the original definition as we removed the large hash table from the top layer

```
//return handle of $\min x \in M : y \leq x$
Function locate($y : \mathbb{N}$) : ElementHandle
    if $y > \max M$ then return $\infty$                                  // no larger element
    $i$ := $y[16..31]$                                                  // index into root table $r$
    if $r[i] = $ null $\vee y > \max M_i$ then return $\min M_{t^1.\texttt{locate}(i)}$
    if $M_i = \{x\}$ then return $x$                                     // single element case
    $j$ := $y[8..15]$                                                   // key for L2 hash table at $M_i$
    if $r_i[j] = $ null $\vee y > \max M_{ij}$ then return $\min M_{i,t_i^1.\texttt{locate}(j)}$
    if $M_{ij} = \{x\}$ then return $x$                                  // single element case
    return $r_{ij}[t_{ij}^1.\texttt{locate}(y[0..7])]$                   // L3 Hash table access

//find the smallest $j \geq i$ such that $t^k[j] = 1$
Method locate($i$) for a bit array $t^k$ consisting of $n$ bit words
    //$n = 32$ for $t^1$, $t^2$, $t_i^1$, $t_{ij}^1$; $n = 64$ for $t^3$; $n = 8$ for $t_i^2$, $t_{ij}^2$
    //Assertion: some bit in $t^k$ to the right of $i$ is nonzero
    $j$ := $i$ div $n$                                  // which $n$ bit word in $b$ contains bit $i$?
    $a$ := $t^k[nj..nj + n - 1]$                                            // get this word
    set $a[(i \bmod n) + 1..n - 1]$ to zero              // erase the bits to the left of bit $i$
    if $a = 0$ then                               // nothing here → look in higher level bit array
        $j$ := $t^{k+1}.\texttt{locate}(j)$                   // $t^{k+1}$ stores the or of $n$-bit groups of $t^k$
        $a$ := $t^k[nj..nj + n - 1]$                      // get the corresponding word in $t^k$
    return $nj + \texttt{msbPos}(a)$
```

Figure 6.2: Pseudo code for locating the smallest $x \in M$ with $y \leq x$.

initialized to a random permutation $h : 0..255 \rightarrow 0..255$. Hash function values for a table of size $256/2^i$ are obtained by shifting $h[x]$ i bits to the right. Note that for tables of size 256 we obtain a perfect hash function, i.e., there are no collisions between different table entries.

The worst case for all input sizes is if there are pairs of elements that only differ in the 8 least significant bits and differ from all other elements in the 16 most significant bits. In this case, hash tables and top data structures at levels two and three are allocated for each such pair of elements. This example shows that the faster `locate` comes at the price of potentially larger memory overhead.

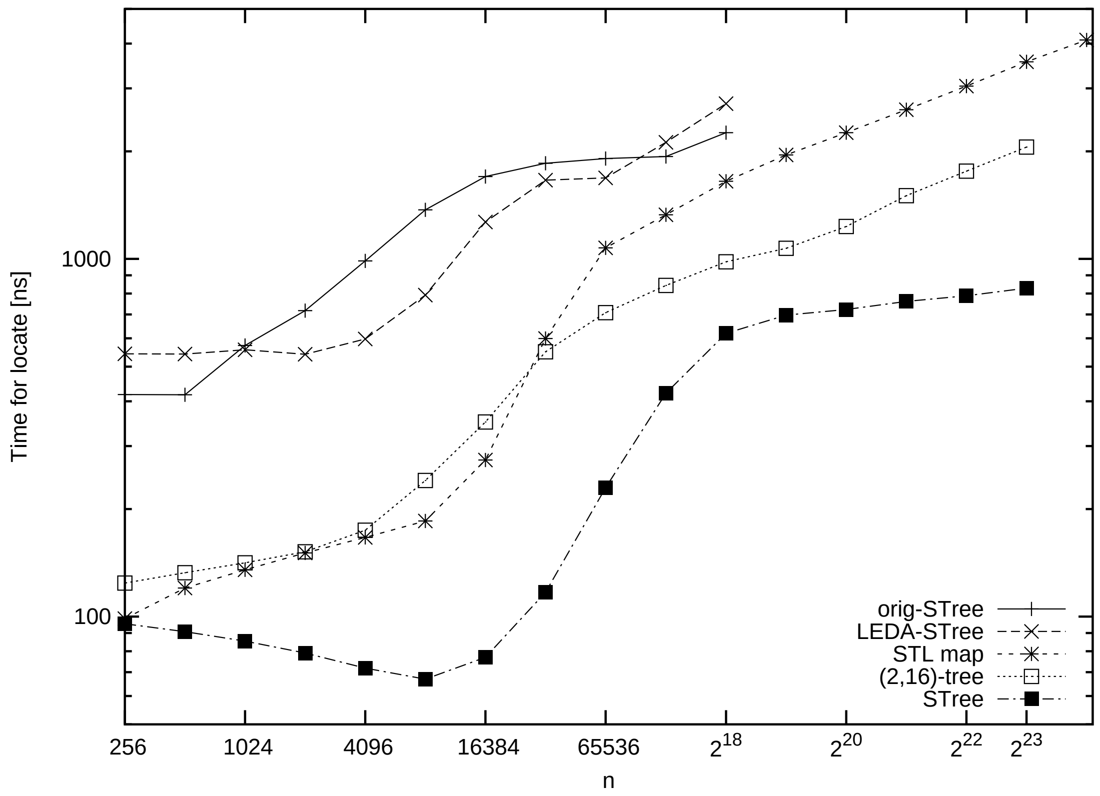


Figure 6.3: Locating randomly distributed keys

6.3 Experiments

We now compare several implementations of search tree like data structures. As comparison based data structures we use the STL `std::map` which is based on red-black trees and `ab_tree` from LEDA which is based on (a, b)-trees with $a = 2$, $b = 16$ which fared best in a previous comparison of search tree data structures in LEDA.

The implementations run under Linux on a 2GHz Intel Xeon processor with 512 KByte of L2-cache using an Intel E7500 Chip set. The machine has 1GByte of RAM and no swap space to exclude swapping effects. We use the `g++ 2.95.4` compiler with optimization level `-O6`. We report the average execution time per operation in nanoseconds on an otherwise unloaded machine. The average is taken over at least 100 000 executions of the operation. Elements are 32 bit unsigned integers plus a 32 bit integer as associated information.

Figure 6.3 shows the time for the `locate` operation for random 32 bit integers and independently drawn random 32 bit queries for locate. Already the comparison based

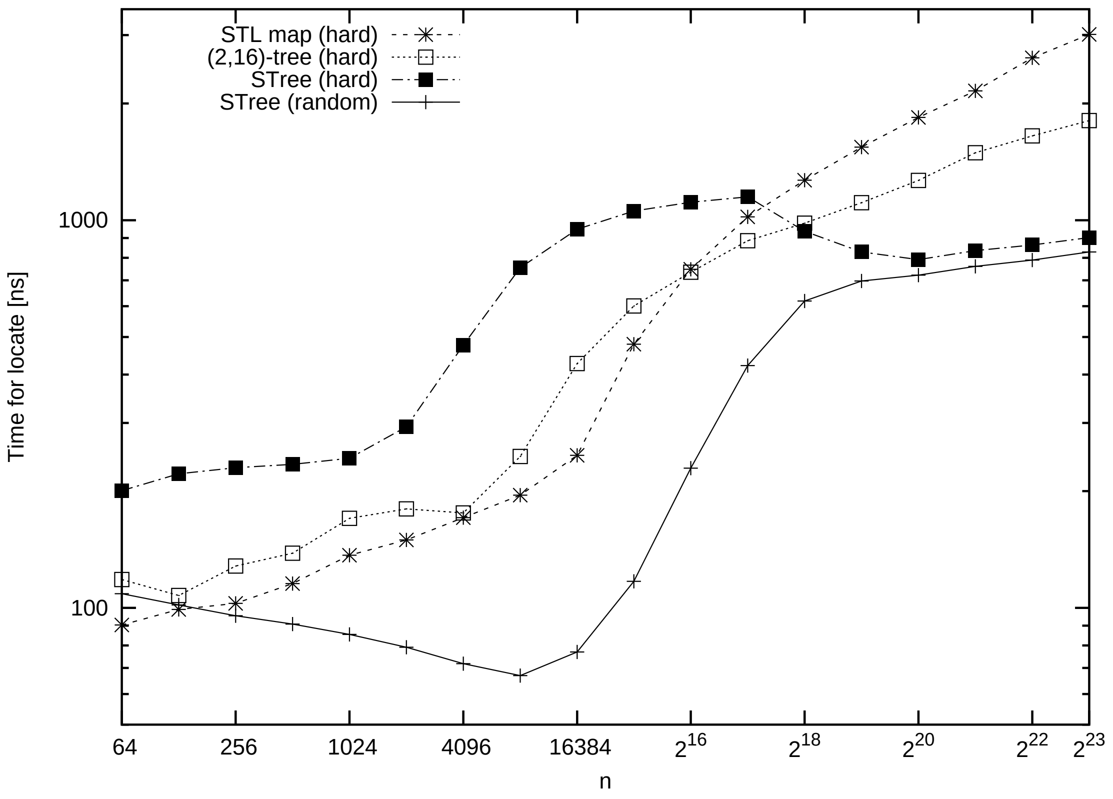


Figure 6.4: Locating on a hard instance

data structures show some interesting effects. For small n, when the data structures fit in cache, red-black trees outperform $(2, 16)$-trees indicating that red-black trees execute less instructions. For larger n, this picture changes dramatically, presumably because $(2, 16)$-trees are more cache efficient.

Our vEB tree (called STree here) is fastest over the entire range of inputs. For small n, it is much faster than comparison based structures up to a factor of 4.1. For random inputs of this size, `locate` mostly accesses the root-top data structure which fits in cache and hence is very fast. It even gets *faster* with increasing n because then `locate` rarely has to go to the second or even third level t^2 and t^3 of the root-top data structure. For medium size inputs there is a range of steep increase of execution time because the L2 and L3 data structures get used more heavily and the memory consumption quickly exceeds the cache size. But the speedup over $(2, 16)$-trees is always at least 1.5. For large n the advantage over comparison based data structures is growing again reaching a factor of 2.9 for the largest inputs.

Figures 6.3 shows the result for an attempt to obtain close to worst case inputs for our vEB tree. For a given set size $|M| = n$, we store $M_{hard} = \{2^8 i\Delta, 2^8 i\Delta + 255 : i = 0..n/2 - 1\}$ where $\Delta = \lfloor 2^{25}/n \rfloor$. M_{hard} maximizes space consumption of our implementation. Furthermore, locate queries of the form $2^8 j\Delta + 128$ for random $j \in 0..n/2 - 1$ force the vEB tree to go through the root table, the L2-table, both levels of the L3-top data structure, and the L3-table. As to be expected, the comparison based implementations are not affected by this change of input. For $n \leq 2^{18}$, the vEB tree is now slower than its comparison based competitors. However, for large n we still have a similar speedup as for random inputs.

Chapter 7

Shortest Path Search

The overview of classical algorithms was taken from [30]. The section on highway hierarchies is mainly based on material from [31] and [32]. The material on transit node routing was taken from [33] and the section on dynamic highway routing is from [34]. More and newer material can be found on Dominik Schultes' website: `http://algo2.iti.uni-karlsruhe.de/schultes/hwy/`.

Some material from this chapter (especially on Highway Hierarchies) was not covered during the lecture in 2007. For self-containment, we include these paragraphs for further studies. In the following chapter, these supplemental sections are marked with an asterisk: $*$.

7.1 Introduction

Computing shortest paths in graphs (networks) with nonnegative edge weights is a classical problem of computer science. From a worst case perspective, the problem has largely been solved by Dijkstra in 1959 who gave an algorithm that finds all shortest paths from a starting node s using at most $m + n$ priority queue operations for a graph $G = (V, E)$ with n nodes and m edges.

However, motivated by important applications (e.g., in transportation networks), there has recently been considerable interest in the problem of accelerating *shortest path queries*, i.e., the problem to find a shortest path between a source node s and a target node t. In this case, Dijkstra's algorithm can stop as soon as the shortest path to t is found.

A classical technique that gives a constant factor speedup is *bidirectional search* which simultaneously searches forward from s and backwards from t until the search frontiers meet. All further speedup techniques either need additional information (e.g., geometry information for *goal directed search*) or *precomputation*. There is a trade-off between the time needed for *precomputation*, the *space* needed for storing the precomputed information, and the resulting *query time*.

In particular, from now on we focus on shortest paths in large *road networks* where we use 'shortest' as a synomym for 'fastest'. The graphs used for North America or Western Europe already have around 20 000 000 nodes so that significantly superlinear preprocessing time or even slightly superlinear space is prohibitive. To our best knowledge, all commercial applications currently only compute paths heuristically that are not always shortest possible. The basic idea of these heuristics is the observation that shortest paths "usually" use small roads only locally, i.e., at the beginning and at the end of a path. Hence the heuristic algorithm only performs some kind of *local search* from s and t and then switches to search in a *highway network* that is much smaller than the complete graph. Typically, an edge is put into the highway network if the information supplied on its road type indicates that it represents an important road.

7.2 "Classical" and other Results

The following section gives a short review of older speedup techniques.

Dijkstra's Algorithm

The classical algorithm for route planning—maintains an array of *tentative distances* $D[u] \geq d(s,u)$ for each node. The algorithm *visits* (or *settles*) the nodes of the road network in the order of their distance to the source node and maintains the invariant that $D[u] = d(s,u)$ for visited nodes. We call the rank of node u in this order its *Dijkstra rank* $r_s(u) = r$. When a node u is visited, its outgoing edges (u,v) are *relaxed*, i.e., $D[v]$ is set to $\min(D[v], d(s,u) + w(u,v))$. Dijkstra's algorithm terminates when the target node is visited. The size of the search space is $O(n)$ and $n/2$ (nodes) on the average. We will assess the quality of route planning algorithms by looking at their *speedup* compared to Dijkstra's algorithm, i.e., how many times faster they can compute shortest-path distances.

Priority Queues.

Dijkstra's algorithm can be implemented using $\mathcal{O}(n)$ priority queue operations. In the comparison based model this leads to $\mathcal{O}(n \log n)$ execution time. In other models of computation and on the average, better bounds exist. However, in practice the impact of priority queues on performance for large road networks is rather limited since cache faults for accessing the graph are usually the main bottleneck. In addition, our experiments indicate that the impact of priority queue implementations diminishes with advanced speedup techniques since these techniques at the same time introduce additional overheads and dramatically reduce the queue sizes.

Bidirectional Search

Bidirectional Search executes Dijkstra's algorithm simultaneously forward from the source and backwards from the target. Once some node has been visited from both directions, the shortest path can be derived from the information already gathered. In a road network, where search spaces will take a roughly circular shape, we can expect a speedup around two —one disk with radius $d(s, t)$ has twice the area of two disks with half the radius. Bidirectional search is important since it can be combined with most other speedup techniques and, more importantly, because it is a necessary ingredient of the most efficient advanced techniques.

Geometric Goal Directed Search (A^*)

The intuition behind goal directed search is that shortest paths 'should' lead in the general direction of the target. A^* search achieves this by modifying the weight of edge (u, v) to $w(u, v) - \pi(u) + \pi(v)$ where $\pi(v)$ is a lower bound on $d(v, t)$. Note that this manipulation shortens edges that lead towards the target. Since the added and subtracted *vertex potentials* $\pi(v)$ cancel along any path, this modification of edge weights preserves shortest paths. Moreover, as long as all edge weights remain nonnegative, Dijkstra's algorithm can still be used. The classical way to use A^* for route planning in road maps estimates $d(v, t)$ based on the Euclidean distance between v and t and the average speed of the fastest road anywhere in the network. Since this is a very conservative estimation, the speedup for finding quickest routes is rather small.

Heuristics

In the last decades, commercial navigation systems were developed which had to handle ever more detailed descriptions of road networks on rather low-powered processors. Vendors resolved to heuristics still used today that do not give any performance guarantees: use A^* search with estimates on $d(u, t)$ rather than lower bounds; do not look at 'unimportant' streets, unless you are close to the source or target. The latter heuristic needs careful hand tuning of road classifications to produce reasonable results but yields considerable speedups.

Small Separators

Road networks are almost planar, i.e., most edges intersect only at nodes. Hence, techniques developed for planar graphs will often also work for road networks. Using $O(n \log^2 n)$ space and preprocessing time, query time $O(\sqrt{n} \log n)$ can be achieved for directed planar graphs without negative cycles. Queries accurate within a factor $(1 + \epsilon)$ can be answered in near constant time using $O((n \log n)/\epsilon)$ space and preprocessing time. Most of these theoretical approaches look difficult to use in practice since they are complicated and need superlinear space.

The first published practical approach to fast route planning uses a set of nodes V_1 whose removal partitions the graph $G = G_0$ into small components. Now consider the *overlay graph* $G_1 = (V_1, E_1)$ where edges in E_1 are *shortcuts* corresponding to shortest paths in G that do not contain nodes from V_1 in their interior. Routing can now be restricted to G_1 and the components containing s and t respectively. This process can be iterated yielding a multi-level method. A limitation of this approach is that the graphs at higher levels become much more dense than the input graphs thus limiting the benefits gained from the hierarchy. Also, computing small separators and shortcuts can become quite costly for large graphs.

Reach-Based Routing

Let $R(v) := \max_{s,t \in V} R_{st}(v)$ denote the *reach* of node v where $R_{st}(v) := \min(d(s, v), d(v, t))$. Gutman [35] observed that a shortest-path search can be stopped at nodes with a reach too small to get to source or target from there. Variants of reach-based routing work with the reach of edges or characterize reach in terms of geometric distance rather than shortest-path distance. The first implementation had disappointing speedups and preprocessing times that would be prohibitive for large networks.

Edge Labels

The idea behind edge labels is to precompute information for an edge e that specifies a set of nodes $M(e)$ with the property that $M(e)$ is a superset of all nodes that lie on a shortest path starting with e. In an s–t query, an edge e need not be relaxed if $t \notin M(e)$. In [26], $M(e)$ is specified by an *angular range*. More effective is information that can distinguish between long range and short range edges. In [27] many *geometric containers* are evaluated. Very good performance is observed for axis parallel rectangles. A disadvantage of geometric containers is that they require a complete all-pairs shortest-path computation. Faster precomputation is possible by partitioning the graph into k regions that have similar size and only a small number of boundary nodes. Now $M(e)$ is represented as a k-vector of *edge flags* [29, 28] where flag i indicates whether there is a shortest path containing e that leads to a node in region i. Edge flags can be computed using a single-source shortest-path computation from all boundary nodes of the regions.

Landmark A^*

Using the triangle inequality, quite strong bounds on shortest-path distances can be obtained by precomputing distances to a set of around 20 *landmark* nodes that are well distributed over the far ends of the network [24]. Using reasonable space and much less preprocessing time than for edge labels, these lower bounds yield considerable speedup for route planning.

Precomputed Cluster Distances (PCD)

In [25], we give a different way to use precomputed distances for goal-directed search. We partition the network into clusters and then precompute the shortest connection between any pair of clusters U and V, i.e., $\min_{u \in U, v \in V} d(u, v)$. PCDs cannot be used together with A^* search since reduced edge weights can become negative. However, PCDs yield upper and lower bounds for distances that can be used to prune search. This gives speedup comparable to landmark-A^* using less space.

7.3 Highway Hierarchy

7.3.1 Introduction

Our first approach is based on the idea to compute *exact* shortest paths by defining the notion of *local search* and *highway network* appropriately. This is very simple. We define local search to be a search that visits the H closest nodes from s (or t) where H is a tuning parameter. This definition already fixes the highway network. An edge $(u, v) \in E$ should be a highway edge if there are nodes s and t such that (u, v) is on the shortest path from s to t, v is not within the H closest nodes from s, and u is not within the H closest nodes from t.

So far, the highway network still contains all the nodes of the original network. However, we can prune it significantly: Isolated nodes are not needed. Trees attached to a biconnected component can only be traversed at the beginning and end of a path. Similarly, paths consisting of nodes with degree two can be replaced by a single edge[1]. The result is a *contracted highway network* that only contains nodes of degree at least three. We can iterate the above approach, define local search on the highway network, find a

[1]note that this list of possible contractions was only used in an early version of the algorithm but still gives a good idea where contraction might be useful.

"superhighway network", contract it, ... We arrive at a multi-level highway network — a *highway hierarchy*.

The next section formalizes some of these ideas.

7.3.2 Hierarchies and Contraction

Graphs and Paths. We expect a *directed* graph $G = (V, E)$ with n nodes and m edges (u, v) with *nonnegative* weights $w(u, v)$ as input. We assume w.l.o.g. that there are no self-loops, parallel edges, and zero weight edges in the input—they could be dealt with easily in a preprocessing step. The *length* $w(P)$ of a path P is the sum of the weights of the edges that belong to P. $P^* = \langle s, \ldots, t\rangle$ is a *shortest path* if there is no path P' from s to t such that $w(P') < w(P^*)$. The *distance* $d(s,t)$ between s and t is the length of a shortest path from s to t. If $P = \langle s, \ldots, s', u_1, u_2, \ldots, u_k, t', \ldots, t\rangle$ is a path from s to t, then $P|_{s' \to t'} = \langle s', u_1, u_2, \ldots, u_k, t'\rangle$ denotes the *subpath* of P from s' to t'.

Dijkstra's Algorithm. Dijkstra's algorithm can be used to solve the *single source shortest path (SSSP) problem*, i.e., to compute the shortest paths from a single source node s to all other nodes in a given graph. It is covered by virtually any textbook on algorithms, so that we confine ourselves to introducing our terminology: Starting with the source node s as root, Dijkstra's algorithm grows a *shortest path tree* that contains shortest paths from s to all other nodes. During this process, each node of the graph is either *unreached*, *reached*, or *settled*. A node that already belongs to the tree is *settled*. If a node u is settled, a shortest path P^* from s to u has been found and the distance $d(s,u) = w(P^*)$ is known. A node that is adjacent to a settled node is *reached*. Note that a settled node is also reached. If a node u is reached, a path P from s to u, which might not be the shortest one, has been found and a *tentative distance* $\delta(u) = w(P)$ is known. Nodes that are not reached are *unreached*.

A *bidirectional* version of Dijkstra's algorithm can be used to find a shortest path from a given node s to a given node t. Two Dijkstra searches are executed in parallel: one searches from the source node s in the original graph $G = (V, E)$, also called *forward graph* and denoted as $\overrightarrow{G} = (V, \overrightarrow{E})$; another searches from the target node t backwards, i.e., it searches in the *reverse graph* $\overleftarrow{G} = (V, \overleftarrow{E})$, $\overleftarrow{E} := \{(v, u) \mid (u, v) \in E\}$. The reverse graph $\overleftarrow{G}$ is also called *backward graph*. When both search scopes meet, a shortest path from s to t has been found.

A *highway hierarchy* of a graph G consists of several levels $G_0, G_1, G_2, \ldots, G_L$, where the number of levels $L + 1$ is given. We provide an inductive definition:

- Base case (G'_0, G_0): level 0 ($G_0 = (V_0, E_0)$) corresponds to the original graph G; furthermore, we define $G'_0 := G_0$.

- First step ($G'_\ell \to G_{\ell+1}, 0 \le \ell < L$): for given *neighbourhood radii*, we will define the *highway network* $G_{\ell+1}$ of a graph G'_ℓ.
- Second step ($G_\ell \to G'_\ell, 1 \le \ell \le L$): for a given set $B_\ell \subseteq V_\ell$ of *bypassable* nodes, we will define the *core* G'_ℓ of level ℓ (This is the contraction step).

First step (*highway network*). For each node u, we choose a nonnegative *neighbourhood radius* $r_\ell^{\rightarrow}(u)$ for the forward graph and a radius $r_\ell^{\leftarrow}(u) \ge 0$ for the backward graph. To avoid some case distinctions, for any direction $\leftrightharpoons \in \{\rightarrow, \leftarrow\}$, we set the neighbourhood radius $r_\ell^{\leftrightharpoons}(u)$ to infinity for $u \notin V'_\ell$ and for $\ell = L$.

The level-ℓ *neighbourhood* of a node $u \in V'_\ell$ is $\mathcal{N}_\ell^{\rightarrow}(u) := \{v \in V'_\ell \mid d_\ell(u,v) \le r_\ell^{\rightarrow}(u)\}$ with respect to the forward graph and, analogously, $\mathcal{N}_\ell^{\leftarrow}(u) := \{v \in V'_\ell \mid d_\ell^{\leftarrow}(u,v) \le r_\ell^{\leftarrow}(u)\}$ with respect to the backward graph, where $d_\ell(u,v)$ denotes the distance from u to v in the forward graph G_ℓ and $d_\ell^{\leftarrow}(u,v) := d_\ell(v,u)$ in the backward graph $\overleftarrow{G_\ell}$.

The *highway network* $G_{\ell+1} = (V_{\ell+1}, E_{\ell+1})$ of a graph G'_ℓ is the subgraph of G'_ℓ induced by the edge set $E_{\ell+1}$: an edge $(u,v) \in E'_\ell$ belongs to $E_{\ell+1}$ iff there are nodes $s,t \in V'_\ell$ such that the edge (u,v) appears in some shortest path $\langle s,\dots,u,v,\dots,t\rangle$ from s to t in G'_ℓ with the property that $v \notin \mathcal{N}_\ell^{\rightarrow}(s)$ and $u \notin \mathcal{N}_\ell^{\leftarrow}(t)$.

The definition of the highway network suggests that we need an all pairs shortest path search to find all its edges, which would be very time-consuming. Fortunately, it is possible to design an efficient algorithm that performs only 'local search' from each node. The main idea is that it is not necessary to look at node pairs s,t that are very far apart: Suppose that $(u,v) \in E_1$ is witnessed by source and target nodes s and t. If $d(s,u) \gg r_\ell^{\rightarrow}(s)$ and $d(v,t) \gg r_\ell^{\leftarrow}(t)$, then we may expect that there are other witnesses s' and t' that are much closer to the edge (u,v).

For each node $s_0 \in V$, we compute and store the values $r_\ell^{\rightarrow}(s_0)$ and $r_\ell^{\leftarrow}(s_0)$. This can be easily done by a Dijkstra search from each node s_0 that is aborted as soon as H nodes have been settled. Then, we start with an empty set of highway edges E_1. For each node s_0, two phases are performed: the forward construction of a partial shortest path tree B and the backward evaluation of B. The construction is done by a single source shortest path (SSSP) search from s_0; during the evaluation phase, paths from the leaves of B to the root s_0 are traversed and for each edge on these paths, it is decided whether to add it to E_1 or not. The crucial part is the specification of an abort criterion for the SSSP search in order to restrict it to a 'local search'.

*Phase 1: Construction of a Partial Shortest Path Tree** A Dijkstra search from s_0 is executed. During the search, a reached node is either in the state *active* or *passive*. The source node s_0 is active; each node that is reached for the first time (*insert*) and each reached node that is updated (*decreaseKey*) adopts the activation state from its (tentative) parent in the shortest path tree B. When a node p is settled using the path $\langle s_0, s_1, \dots, p\rangle$,

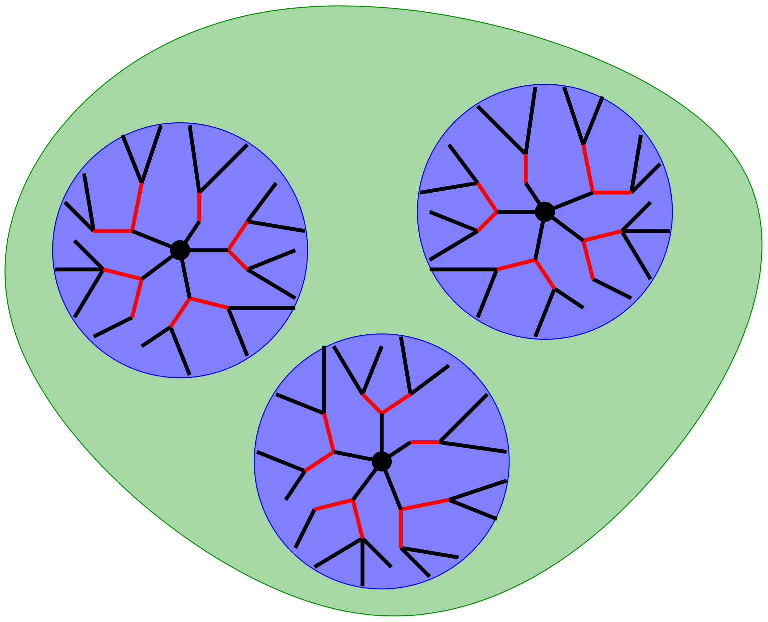


Figure 7.1: Instead of a complete all-to-all shortest path search, we can identify all highway edges by a local search for each node, visiting only its close neighbors.

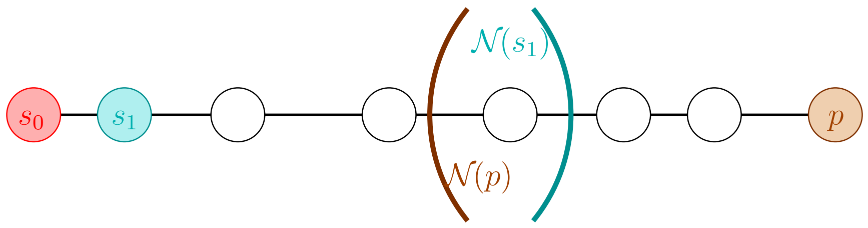


Figure 7.2: The abort criterion for finding highway edges ensures local search.

then p's state is set to passive if $|\mathcal{N}(s_1) \cap \mathcal{N}(p)| \leq 1$. When no active unsettled node is left, the search is aborted and the growth of B stops.

*Phase 2: Selection of the Highway Edges** During Phase 2, all edges (u, v) are added to E_1 that lie on paths $\langle s_0, \ldots, u, v, \ldots, t_0 \rangle$ in B with the property that $v \notin \mathcal{N}(s_0)$ and $u \notin \mathcal{N}(t_0)$, where t_0 is a leaf of B. This can be done in time $O(|B|)$.

Speeding Up Construction* An active node v is declared to be a *maverick* if $d(s_0, v) > f \cdot d_H(s_0)$, where f is a parameter. Normally, the search cannot be aborted before the search radius reaches $d(s_0, v)$ because we have to proof that we found the shortest path. Now, when all active nodes are mavericks, the search from passive nodes is no longer continued. This way, the construction process is accelerated and E_1 becomes a superset of the highway network. Hence, queries will be slower, but still compute exact shortest paths. The *maverick factor* f enables us to adjust the trade-off between construction and query time. Long-distance ferries are a typical example of mavericks.

Second step (*core*)* For a given set $B_\ell \subseteq V_\ell$ of *bypassable* nodes, we define the set S_ℓ of *shortcut edges* that bypass the nodes in B_ℓ: for each path $P = \langle u, b_1, b_2, \ldots, b_k, v \rangle$ with $u, v \in V_\ell \setminus B_\ell$ and $b_i \in B_\ell, 1 \leq i \leq k$, the set S_ℓ contains an edge (u, v) with $w(u, v) = w(P)$. The *core* $G'_\ell = (V'_\ell, E'_\ell)$ of level ℓ is defined in the following way: $V'_\ell := V_\ell \setminus B_\ell$ and $E'_\ell := (E_\ell \cap (V'_\ell \times V'_\ell)) \cup S_\ell$.

Contraction of a Graph In order to obtain the core of a highway network, we contract it, which yields several advantages. The search space during the queries gets smaller since bypassed nodes are not touched and the construction process gets faster since the next iteration only deals with the nodes that have not been bypassed. Furthermore, a more effective contraction allows us to use smaller neighbourhood sizes without compromising the shrinking of the highway networks. This improves both construction and query times. However, bypassing nodes involves the creation of shortcuts, i.e., edges that represent the bypasses. Due to these shortcuts, the average degree of the graph is increased and the memory consumption grows. In particular, more edges have to be relaxed during the queries. Therefore, we have to carefully select nodes so that the benefits of bypassing them outweigh the drawbacks.

An intuive justification for contraction is the following consideration, which was in fact the basis of contraction in an earlier version of Highway Hierarchies: Imagine a long path where the inner nodes have no other edges. It is possible to contract this path to a single edge between the starting and the end node and still receive all shortest paths. Another example of contractable structures are attached trees where every shortest path to a node outside has to go through the root.

We give an iterative algorithm that combines the selection of the bypassable nodes B_ℓ with the creation of the corresponding shortcuts. We manage a stack that contains

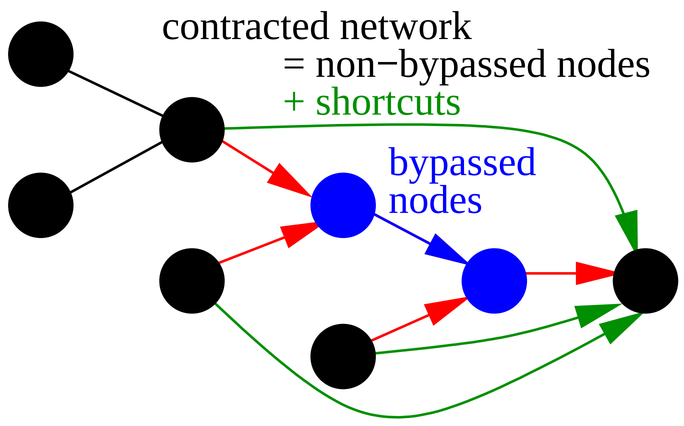


Figure 7.3: Contraction a graph seperates bypassable components from the core.

all nodes that have to be considered, initially all nodes from V_ℓ. As long as the stack is not empty, we deal with the topmost node u. We check the *bypassability criterion* $\#\text{shortcuts} \leq c \cdot (\deg_{\text{in}}(u) + \deg_{\text{out}}(u))$, which compares the number of shortcuts that would be created when u was bypassed with the sum of the in- and outdegree of u. The magnitude of the contraction is determined by the parameter c. If the criterion is fulfilled, the node is bypassed, i.e., it is added to B_ℓ and the appropriate shortcuts are created. Note that the creation of the shortcuts alters the degree of the corresponding endpoints so that bypassing one node can influence the bypassability criterion of another node. Therefore, all adjacent nodes that have been removed from the stack earlier, have not been bypassed, yet, and are bypassable now are pushed on the stack once again. It happens that shortcuts that were created once are discarded later when one of its endpoints is bypassed.

7.3.3 Query

Our *highway query algorithm* is a modification of the bidirectional version of Dijkstra's algorithm. For now, we assume that the search is *not* aborted when both search scopes meet. We only describe the modifications of the forward search since forward and backward search are symmetric. In addition to the *distance* from the source, the key of each node includes the search *level* and the *gap* to the next applicable neighbourhood border. The search starts at the source node s in level 0. First, a local search in the neighbourhood of s is performed, i.e., the gap to the next border is set to the neighbourhood radius of s in level 0. When a node v is settled, it adopts the gap of its parent u minus the length of the edge (u, v). As long as we stay inside the current neighbourhood, everything works as usual. However, if an edge (u, v) crosses the neighbourhood border (i.e., the length of the

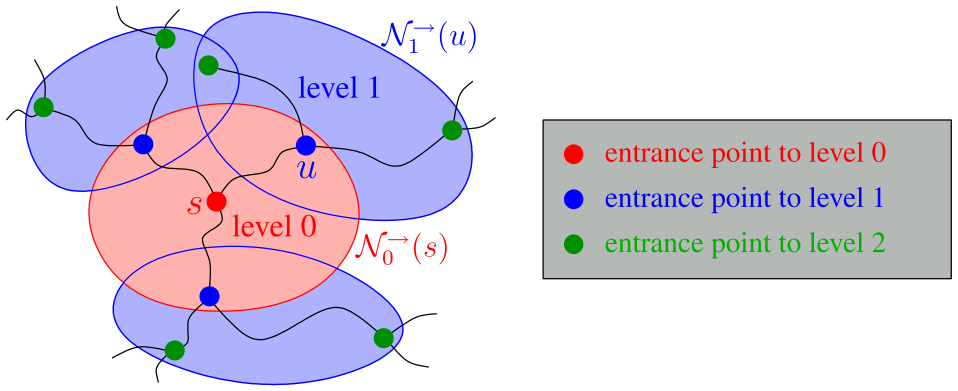

$\mathcal{N}_1^{\rightarrow}(u)$
level 1
u
s
level 0
$\mathcal{N}_0^{\rightarrow}(s)$
● entrance point to level 0
● entrance point to level 1
● entrance point to level 2

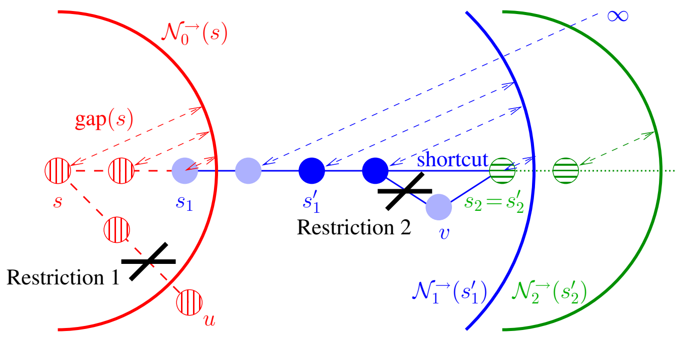
Figure 7.4: A schematic diagram of a highway query. Only the forward search is depicted.

edge is greater than the gap), we switch to a higher search level ℓ. The node u becomes an *entrance point* to the higher level. If the level of the edge (u, v) is less than the new search level ℓ, the edge is *not* relaxed—this is one of the two restrictions that cause the speedup in comparison to Dijkstra's algorithm (Restriction 1). Otherwise, the edge is relaxed. If the relaxation is successful, v adopts the new search level ℓ and the gap to the border of the neighbourhood of u in level ℓ since u is the corresponding entrance point to level ℓ. Figure 7.4 illustrates this process.

To increase the speedup and make use of the contracted graph, we introduce another restriction (Restriction 2): when a node $u \in V'_\ell$ is settled, all edges (u, v) that lead to a bypassed node $v \in B_\ell$ in search level ℓ are *not* relaxed.

A detailed example* Figure 7.5 gives a detailed example of the forward search of a highway query. The search starts at node s. The gap of s is initialised to the distance from s to the border of the neighbourhood of s in level 0. Within the neighbourhood of s, the search process corresponds to a standard Dijkstra search. The edge that leads to u leaves the neighbourhood. It is not relaxed due to Restriction 1 since the edge belongs only to level 0. In contrast, the edge that leaves s_1 is relaxed since its level allows to switch to level 1 in the search process. s_1 and its direct successor are bypassed nodes in level 1. Their neighbourhoods are unbounded, i.e., their neighbourhood radii are infinity so that the gap is set to infinity as well. At s'_1, we leave the component of bypassed nodes and enter the core of level 1. Now, the search is continued in the core of level 1 within the neighbourhood of s'_1. The gap is set appropriately. Note that the edge to v is not relaxed due to Restriction 2 since v is a bypassed node. Instead, the direct shortcut to $s_2 = s'_2$ is used. Here, we switch to level 2. In this case, we do not enter the next level through a component of bypassed nodes, but we get directly into the core. The search is continued in the core of level 2 within the neighbourhood of s'_2. And so on.

Despite of Restriction 1, we always find the optimal path since the construction of the



$\mathcal{N}_0^{\rightarrow}(s)$
gap(s)
s
s_1
s'_1
shortcut
∞
Restriction 2
$s_2 = s'_2$
v
Restriction 1
u
$\mathcal{N}_1^{\rightarrow}(s'_1)$
$\mathcal{N}_2^{\rightarrow}(s'_2)$

Figure 7.5: A detailed example of a highway query. Only the forward search is depicted. Nodes in level 0, 1, and 2 are vertically striped, solid, and horizontally striped, respectively. In level 1, dark shades represent core nodes, light shades bypassed nodes. Edges in level 0, 1, and 2 are dashed, solid, and dotted, respectively.

highway hierarchy guarantees that the levels of the edges that belong to the optimal path are sufficiently high so that these edges are not skipped. Restriction 2 does not invalidate the correctness of the algorithm since we have introduced shortcuts that bypass the nodes that do not belong to the core. Hence, we can use these shortcuts instead of the original paths.

The Algorithm.* We use two priority queues $\overrightarrow{Q}$ and $\overleftarrow{Q}$, one for the forward search and one for the backward search. The key of a node u is a triple $(\delta(u), \ell(u), \mathrm{gap}(u))$, the (tentative) distance $\delta(u)$ from s (or t) to u, the search level $\ell(u)$, and the gap $\mathrm{gap}(u)$ to the next applicable neighbourhood border. A key $(\delta, \ell, \mathrm{gap})$ is less than another key $(\delta', \ell', \mathrm{gap}')$ iff $\delta < \delta'$ or $\delta = \delta' \wedge \ell > \ell'$ or $\delta = \delta' \wedge \ell = \ell' \wedge \mathrm{gap} < \mathrm{gap}'$.

Figure 7.6 contains the pseudo-code of the highway query algorithm.

Speeding Up the Search in the Topmost Level. Let us assume that we have a distance table that contains for any node pair $s, t \in V'_L$ the optimal distance $d_L(s,t)$. Such a table can be precomputed during the preprocessing phase by $|V'_L|$ SSSP searches in V'_L. Using the distance table, we do not have to search in level L. Instead, when we arrive at a node $u \in V'_L$ that 'leads' to level L, we add u to a set $\overrightarrow{I}$ or $\overleftarrow{I}$ depending on the search direction; we do not relax the edge that leads to level L. After the sets $\overrightarrow{I}$ and $\overleftarrow{I}$ have been determined, we consider all pairs $(u,v), u \in \overrightarrow{I}, v \in \overleftarrow{I}$, and compute the minimum path length $D := d_0(s,u) + d_L(u,v) + d_0(v,t)$. Then, the length of the shortest s-t-path is the minimum of D and the length of the tentative shortest path found so far (in case that the search scopes have already met in a level $< L$).

```
input: source node $s$ and target node $t$

$\boxed{1}$ $\overrightarrow{Q}$.insert($s$, $(0,\underline{0, r_0^{\rightarrow}(s)})$); $\overleftarrow{Q}$.insert($t$, $(0,\underline{0, r_0^{\leftarrow}(t)})$);
 2 while ($\overrightarrow{Q} \cup \overleftarrow{Q} \neq \emptyset$) do {
 3     $\leftrightarrows \in \{\rightarrow, \leftarrow\}$;                              // select direction
 4     $u :=$ $\overset{\leftrightarrows}{Q}$.deleteMin();
$\boxed{5}$     if gap$(u) \neq \infty$ then gap$'$ := gap$(u)$ else gap$'$ := $r^{\leftrightarrows}_{\ell(u)}(u)$;
 6     foreach $e = (u, v) \in \overset{\leftrightarrows}{E}$ do {
$\boxed{7}$         for ($\ell := \ell(u)$, gap := gap$'$; $w(e) >$ gap; $\ell$++ )
              gap := $r^{\leftrightarrows}_{\ell+1}(u)$;                    // go “upwards”
$\boxed{8}$         if $\ell(e) < \ell$ then continue;                   // Restriction 1
$\boxed{9}$         if $u \in V'_\ell \wedge v \in B_\ell$ then continue;          // Restriction 2
$\boxed{10}$        $k := (\delta(u) + w(e)\underline{, \ell, \text{gap} - w(e)})$;
11          if $v \in \overset{\leftrightarrows}{Q}$ then $\overset{\leftrightarrows}{Q}$.decreaseKey($v$, $k$); else $\overset{\leftrightarrows}{Q}$.insert($v$, $k$);
12      }
13 }
```

Figure 7.6: The highway query algorithm. Differences to the bidirectional version of Dijkstra’s algorithm are marked: additional and modified lines have a framed line number; in modified lines, the modifications are underlined.

7.3.4 Experiments

Environment and Instances. The experiments were done on one core of an AMD Opteron Processor 270 clocked at 2.0 GHz with 4 GB main memory and 2 $\times$ 1 MB L2 cache, running SuSE Linux 10.0 (kernel 2.6.13). The program was compiled by the GNU C++ compiler 4.0.2 using optimisation level 3. We use 32 bits to store edge weights and path lengths.

We deal with the road networks of Western Europe[2] and of the USA (without Hawaii) and Canada. Both networks have been made available for scientific use by the company PTV AG. The original graphs contain for each edge a length and a road category, e.g., motorway, national road, regional road, urban street. We assign average speeds to the road categories, compute for each edge the average travel time, and use it as weight.

We report only the times needed to compute the shortest path distance between two nodes without outputting the actual route. In order to obtain the corresponding subpaths in the original graph, we are able to extract the used shortcuts without using any extra data. However, if a fast output routine is required, we might want to spend some additional space to accelerate the unpacking process. For details, we refer to the full paper. Table 7.3.4 summarises the properties of the used road networks and key results of the experiments.

Parameters. Unless otherwise stated, the following default settings apply. We use the contraction rate $c = 1.5$ and the neighbourhood sizes H as stated in Tab. 7.3.4—the same neighbourhood size is used for all levels of a hierarchy. First, we contract the original graph. Then, we perform four iterations of our construction procedure, which determines a highway network and its core. Finally, we compute the distance table between all level-4 core nodes.

In one test series (Fig. 7.7), we used all the default settings except for the neighbourhood size H, which we varied from 40 to 90. On the one hand, if H is too small, the shrinking of the highway networks is less effective so that the level-4 core is still quite big. Hence, we need much time and space to precompute and store the distance table. On the other hand, if H gets bigger, the time needed to preprocess the lower levels increases because the area covered by the local searches depends on the neighbourhood size. Furthermore, during a query, it takes longer to leave the lower levels in order to get to the topmost level where the distance table can be used. Thus, the query time increases as well. We observe that we get good space-time tradeoffs for neighbourhood sizes around 60. In particular, we find that a good choice of the parameter H does not vary very much from graph to graph.

In another test series (Tab. 7.3.4a), we did not use a distance table, but repeated the construction process until the topmost level was empty or the hierarchy consisted of 15

[2]14 countries: at, be, ch, de, dk, es, fr, it, lu, nl, no, pt, se, uk

		Europe	USA/CAN	USA (Tiger)
INPUT	#nodes	18 029 721	18 741 705	24 278 285
	#directed edges	42 199 587	47 244 849	58 213 192
	#road categories	13	13	4
PARAM.	average speeds [km/h]	10–130	16–112	40–100
	H	50	60	60
PREPROC.	CPU time [min]	15	20	18
	∅overhead/node [byte]	68	69	50
QUERY	CPU time [ms]	0.76	0.90	0.88
	#settled nodes	884	951	1 076
	#relaxed edges	3 182	3 630	4 638
	speedup (CPU time)	8 320	7 232	7 642
	speedup (#settled nodes)	10 196	9 840	11 080
	worst case (#settled nodes)	8 543	3 561	5 141

Table 7.1: Overview of the used road networks and key results. '∅overhead/node' accounts for the additional memory that is needed by our highway hierarchy approach (divided by the number of nodes). The amount of memory needed to store the original graph is not included. Query times are average values based on 10 000 random s-t-queries. 'Speedup' refers to a comparison with Dijkstra's algorithm (unidirectional). Worst case is an upper bound for *any* possible query in the respective graph.

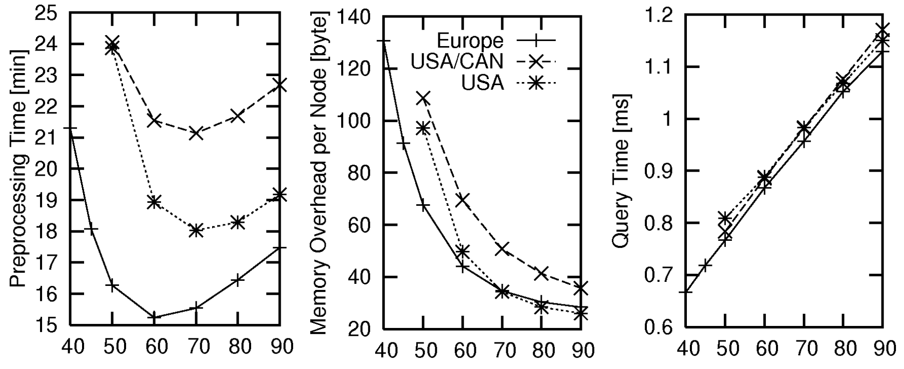


Figure 7.7: Preprocessing and query performance depending on the neighbourhood size H.

levels. We varied the contraction rate c from 0.5 to 2. In case of $c = 0.5$ (and $H = 50$), the shrinking of the highway networks does not work properly so that the topmost level is still very big. This yields huge query times. In earlier implementations we used a larger neighbourhood size to cope with this problem. Choosing larger contraction rates

contr. rate c	PREPROCESSING time [min]	over-head	∅deg.	QUERY time [ms]	#settled nodes	#relaxed edges
0.5	89	27	3.2	176.05	242 156	505 086
1	16	27	3.7	1.97	2 321	8 931
1.5	13	27	3.8	1.58	1 704	7 935
2	13	28	3.9	1.70	1 681	8 607

(a)

#levels	PREPROC. time [min]	over-head	QUERY time [ms]	#settled nodes
5	16	68	0.77	884
7	13	28	1.19	1 290
9	13	27	1.51	1 574
11	13	27	1.62	1 694

(b)

Table 7.2: Preprocessing and query performance for the European road network depending on the contraction rate c (a) and the number of levels (b). 'overhead' denotes the average memory overhead per node in bytes.

reduces the preprocessing and query times since the cores and search spaces get smaller. However, the memory usage and the average degree are increased since more shortcuts are introduced. Adding too many shortcuts ($c = 2$) further reduces the search space, but the number of relaxed edges increases so that the query times get worse.

In a third test series (Tab. 7.3.4b), we used the default settings except for the number of levels, which we varied from 5 to 11. In each test case, a distance table was used in the topmost level. The construction of the higher levels of the hierarchy is very fast and has no significant effect on the preprocessing times. In contrast, using only five levels yields a rather large distance table, which somewhat slows down the preprocessing and increases the memory usage. However, in terms of query times, '5 levels' is the optimal choice since using the distance table is faster than continuing the search in higher levels.

Fast vs. Precise Construction. During various experiments, we came to the conclusion that it is a good idea *not* to take a fixed maverick factor f for all levels of the construction process, but to start with a low value (i.e. fast construction) and increase it level by level (i.e. more precise construction). For the following experiments, we used the sequence $0, 2, 4, 6, \ldots$.

Best Neighbourhood Sizes. For two levels ℓ and $\ell + 1$ of a highway hierarchy, the *shrinking factor* is the ratio between $|E'_\ell|$ and $|E'_{\ell+1}|$. In our experiments, we observed that the highway hierarchies of the USA and Europe were almost *self-similar* in the sense that the shrinking factor remained nearly unchanged from level to level when we used the same neighbourhood size H for all levels. We kept this approach and applied the same H iteratively until the construction led to an empty highway network. Figure 7.8 demonstrates the shrinking process for Europe. For most levels, we observe an almost constant shrinking factor (which appears as a straight line due to the logarithmic scale of the y-axis). The greater the neighbourhood size, the greater the shrinking factor. The

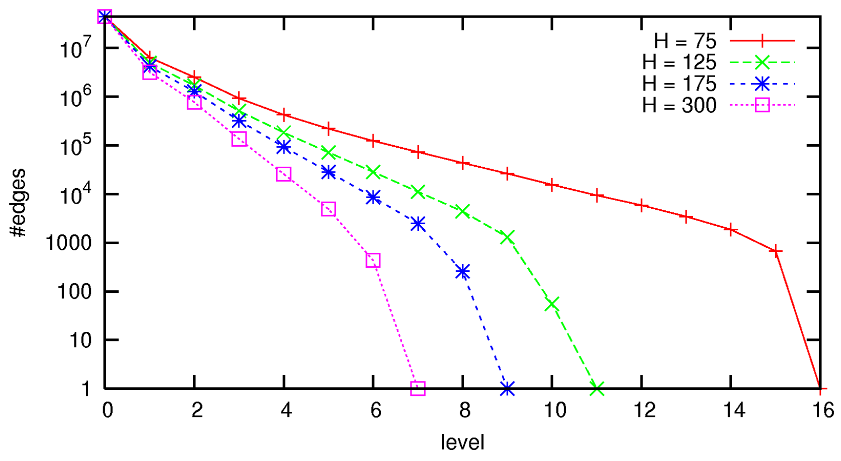


Figure 7.8: Shrinking of the highway networks of Europe. For different neighbourhood sizes H and for each level ℓ, we plot $|E'_\ell|$, i.e., the number of edges that belong to the core of level ℓ.

first iteration (Level 0→1) and the last few iterations are exceptions: at the first iteration, the construction works very well due to the characteristics of the real world road network (there are many trees and lines that can be contracted); at the last iterations, the highway network collapses, i.e., it shrinks very fast, because nodes that are close to the border of the network usually do not belong to the next level of the highway hierarchy, and when the network gets small, almost all nodes are close to the border.

Multilevel Queries. Table 7.3.4 contains average values for queries, where the source and target nodes are chosen randomly. For the two large graphs we get a speedup of more than 2 000 compared to Dijkstra's algorithm both with respect to (query) time[3] and with respect to the number of settled nodes.

For our largest road network (USA), the number of nodes that are settled during the search is *less* than the number of nodes that belong to the shortest paths that are found. Thus, we get an efficiency that is greater than 100%. The reason is that edges at high levels will often represent long paths containing many nodes.[4]

For use in applications it is unrealistic to assume a uniform distribution of queries in large graphs such as Europe or the USA. On the other hand, it would be hardly more realistic to arbitrarily cut the graph into smaller pieces. Therefore, we decided to measure

[3]It is likely that Dijkstra would profit more from a faster priority queue than our algorithm. Therefore, the time-speedup could decrease by a small constant factor.

[4]The reported query times do not include the time for expanding these paths. We have made measurements with a naive recursive expansion routine which never take more than 50% of the query time. Also note that this process could be radically sped up by precomputing unpacked representations of edges.

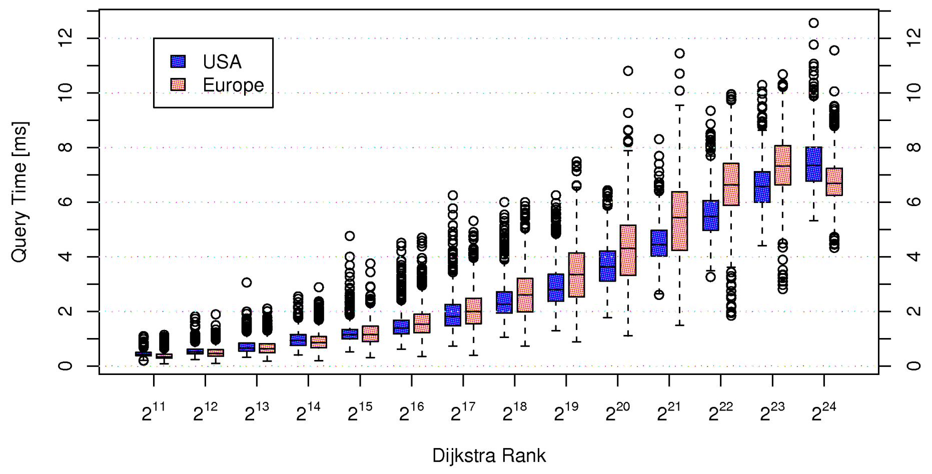


Figure 7.9: Multilevel Queries. For each road network and each Dijkstra rank on the x-axis, 1 000 queries from random source nodes were performed. The results are represented as box-and-whisker plot: each box spreads from the lower to the upper quartile and contains the median, the whiskers extend to the minimum and maximum value omitting outliers, which are plotted individually. It is important to note that a logarithmic scale is used for the x-axis.

local queries within the big graphs: For each power of two $r = 2^k$, we choose random sample points s and then use Dijkstra's algorithm to find the node t with Dijkstra rank $r_s(t) = r$. We then use our algorithm to make an s-t query. By plotting the resulting statistics for each value $r = 2^k$, we can see how the performance scales with a natural measure of difficulty of the query. Figure 7.9 shows the query times. Note that the median query times are scaling quite smoothly and the growth is much slower than the exponential increase we would expect in a plot with logarithmic x axis, linear y axis, and any growth rate of the form r^ρ for Dijkstra rank r and some constant power ρ. The curve is also not the straight line one would expect from a query time logarithmic in r. Note that for the largest rank, query times are actually *decreasing*. A possible explanation is that these queries will have at least one of source or destination node at the border area of the map where the road network is often not very dense (e.g. northern Norway). This plot was done without using distance tables which would also cut the costs at some point where every query will move to the highest level and then resort to the table.

Worst Case Upper Bounds. By executing a query from each node of a given graph to an added isolated dummy node and a query from the dummy node to each actual node in the backward graph, we obtain a distribution of the search spaces of the forward and backward search, respectively. We can combine both distributions to get an upper bound for the distribution of the search spaces of bidirectional queries: when $\mathcal{F}_{\rightarrow}(x)$ ($\mathcal{F}_{\leftarrow}(x)$)

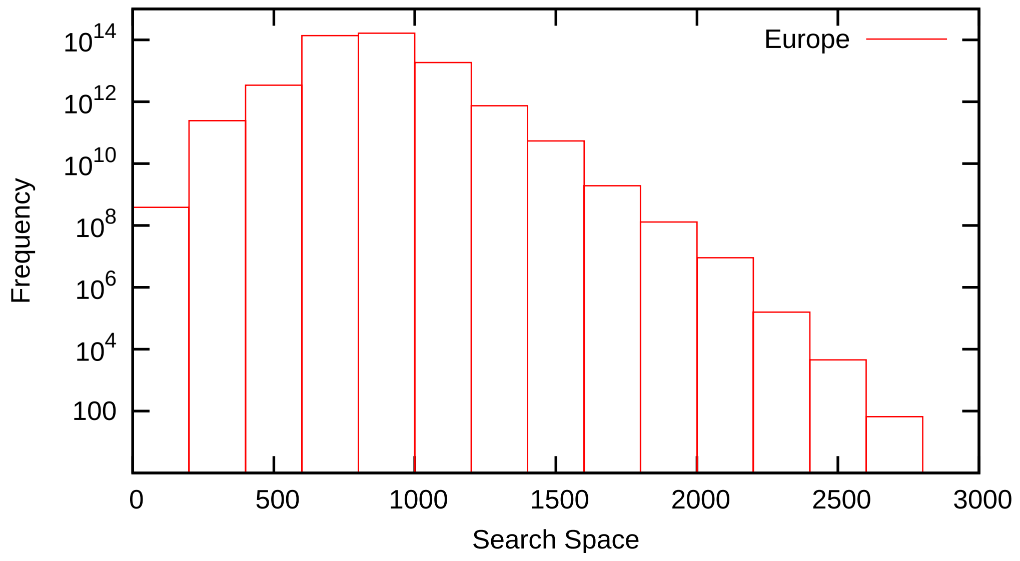


Figure 7.10: Histogram of upper bounds for the search spaces of s-t-queries.

denotes the number of source (target) nodes whose search space consists of x nodes in a forward (backward) search, we define $\mathcal{F}_{\leftrightarrow}(z) := \sum_{x+y=z} \mathcal{F}_{\rightarrow}(x) \cdot \mathcal{F}_{\leftarrow}(y)$, i.e., $\mathcal{F}_{\leftrightarrow}(z)$ is the number of s-t-pairs such that the upper bound of the search space size of a query from s to t is z. In particular, we obtain the upper bound $\max\{z \mid \mathcal{F}_{\leftrightarrow}(z) > 0\}$ for the worst case without performing all n^2 possible queries. Figure 7.10 visualises the distribution $\mathcal{F}_{\leftrightarrow}(z)$ as a histogram.

7.4 Transit Node Routing

When you drive to somewhere 'far away', you will leave your current location via one of only a few 'important' traffic junctions. For graphs representing road networks, this means: First, there is a relatively small set of *transit nodes*, about 10 000 for the US road network, with the property that for every pair of nodes that are 'not too close' to each other, the shortest path between them passes through *at least one* of these transit nodes. Second, for every node, the set of transit nodes encountered first when going far—we call these *access nodes*—is small (about 10). We will now try to exploit this property.

To simplify notation we will present the approach for undirected graphs. However, the method is easily generalised to directed graphs and our highway hierarchy implementation already handles directed graphs. Consider any set $\mathcal{T} \subseteq V$ of *transit nodes*, an *access mapping* $A : V \to 2^{\mathcal{T}}$, and a *locality filter* $L : V \times V \to \{\mathsf{true}, \mathsf{false}\}$. We require that

$\neg L(s,t)$ implies that the shortest path distance is

$$d(s,t) = \min\{d(s,u) + d(u,v) + d(v,t) : u \in A(s), v \in A(t)\} \quad . \tag{7.1}$$

Equation 7.1 implies that the shortest path between nodes that are not near to each other goes through transit nodes at both ends. In principle, we can pick any set of transit nodes, any access mapping, and any locality filter fulfilling Equation (7.1) to obtain a transit node query algorithm:
Assume we have precomputed all distances between nodes in $\mathcal{T}$. If $\neg L(s,t)$ then compute $d(s,t)$ using Equation (7.1). Else, use any other routing algorithm.

Of course, we want a good choice of $(\mathcal{T}, A, L)$. $\mathcal{T}$ should be small but allow many global queries, L should efficiently identify as many of these global query pairs as possible, and we should be able to store and evaluate A efficiently.

We can apply a *second layer* of *generalised transit node routing* to the remaining local queries (that may dominate some real world applications). We have a node set $\mathcal{T}_2 \supset \mathcal{T}$, an access mapping $A_2 : V \to 2^{\mathcal{T}_2}$, and a locality filter L_2 such that $\neg L_2(s,t)$ implies that the shortest path distance is defined by Equation 7.1 or by

$$d(s,t) = \min\{d(s,u) + d(u,v) + d(v,t) : u \in A_2(s), v \in A_2(t)\} \quad . \tag{7.2}$$

In order to be able to evaluate Equation 7.2 efficiently we need to precompute the local connections from $\{d(u,v) : u, v \in \mathcal{T}_2 \wedge L(u,v)\}$ which cannot be obtained using Equation 7.1.

In an analogous way we can add further layers.

7.4.1 Computing Transit Nodes

Computing Access Nodes: Backward Approach.

Start a Dijkstra search from each transit node $v \in \mathcal{T}$. Run it until all paths leading to nodes in the priority queue pass over another node $w \in \mathcal{T}$. Record v as an access node for any node u on a shortest path from v that does not lead over another node in $\mathcal{T}$. Record an edge (v,w) with weight $d(v,w)$ for a *transit graph* $G[\mathcal{T}] = (\mathcal{T}, E_{\mathcal{T}})$. When this local search has been performed from all transit nodes, we have found all access nodes and the distance table can be computed using an all-pairs shortest path computation in $G[\mathcal{T}]$.

Layer 2 Information

is computed similarly to the top level information except that a search on the transit graph $G[\mathcal{T}_2]$ can be stopped when all paths in the priority queue pass over a top level transit node $w \in \mathcal{T}$. Level 2 distances from each node $v \in \mathcal{T}_2$ can be stored space efficiently in a static hash table. We only need to store distances that actually improve on the distances obtained going via the top level $\mathcal{T}$.

Computing Access Nodes: Forward Approach.

Start a Dijkstra search from each node u. Stop when all paths in the shortest path tree are 'covered' by transit nodes. Take these transit nodes as access points of u. Applied naively, this approach is rather inefficient. However, we can use two tricks to make it efficient. First, during the search we do not relax the edges leaving transit nodes. This leads to the computation of a superset of the access points. Fortunately, this set can be easily reduced if the distances between all transit nodes are already known: if an access point v' can be reached from u via another access point v on a shortest path, we can discard v'. Second, we can only determine the access point sets $A(v)$ for all nodes $v \in \mathcal{T}_2$ and the sets $A_2(u)$ for all nodes $u \in V$. Then, for any node u, $A(u)$ can be computed as $\bigcup_{v \in A_2(u)} A(v)$. Again, we can use the reduction technique to remove unnecessary elements from the set union.

Locality Filters.

There seem to be two basic approaches to transit node routing. One that starts with a locality filter L and then has to find a good set of transit nodes $\mathcal{T}$ for which L works. The other approach starts with $\mathcal{T}$ and then has to find a locality filter that can be efficiently evaluated and detects as accurately as possible whether local search is needed. One approach that we found very effective is to use the information gained when computing the distance table for layer $i+1$ to define a locality filter for layer i. For example, we can compute the radius $r^i(u)$ of a circle around every node $u \in \mathcal{T}_{i+1}$ that contains for each entry $d(u,v)$ in the layer-$(i+1)$ table the meeting point of a bidirectional search between u and v. We can use this information in several ways. We can (pre)compute conservative circle radii for arbitrary nodes v as $r^i(v) := \max\{||v-u||_2 + r^i(u) : u \in A_{i+1}(v)\}$. Note that even if we are not able to store the information gathered during a precomputation at layer $i+1$, it might still make sense to run it in order to gather the more compact locality information.

Combining with Highway Hierachies

Nodes on high levels of a highway hierarchy have the property that they are used on shortest paths far away from starting and target nodes. 'Far away' is defined with respect to the Dijkstra rank. Hence, it is natural to use (the core of) some level K of the highway hierarchy for the transit node set $\mathcal{T}$. Note that we have quite good (though indirect) control over the resulting size of $\mathcal{T}$ by choosing the appropriate neighbourhood sizes and the appropriate value for $K =: K_1$. In our current implementation this is level 4, 5, or 6. In addition, the highway hierarchy helps us to efficiently compute the required information. Note that there is a difference between the *level* of the highway hierarchy and the *layer* of transit node search.

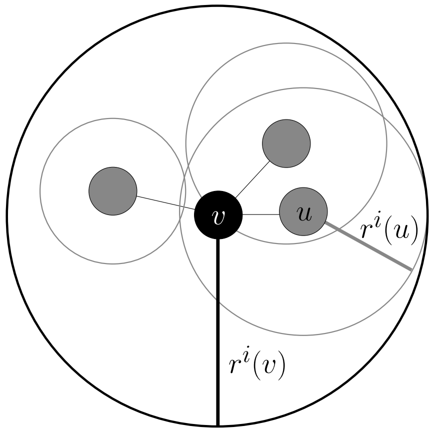


Figure 7.11: Example for the extension of the geometric locality filter. The grey nodes constitute the set $A_{i+1}(v)$.

We can also combine the techniques of distance tables (many-to-many queries) with transit nodes. Roughly, this algorithm first performs independent backward searches from all transit nodes and stores the gathered distance information in *buckets* associated with each node. Then, a forward search from each transit node scans all buckets it encounters and uses the resulting path length information to update a table of tentative distances. This approach can be generalised for computing distances at layer $i > 1$. We use the forward approach to compute the access point sets (In our case, we do not perform Dijkstra searches, but highway searches).

7.4.2 Experiments

Environment, Instances, and Parameters

The experiments were done on one core of an AMD Opteron Processor 270 clocked at 2.0 GHz with 8 GB main memory and 2×1 MB L2 cache, running SuSE Linux 10.0 (kernel 2.6.13). The program was compiled by the GNU C++ compiler 4.0.2 using optimisation level 3. We deal with the same networks we already used for experiments on Highway Hierarchies. We assign average speeds to the road categories, compute for each edge the average travel time, and use it as weight. In addition to this *travel time metric*, we perform experiments on variants of the European graph with a *distance metric* and the *unit metric*.

We use two variants of the transit node approach: Variant *economical* aims at a good compromise between space consumption, preprocessing time and query time. Economical uses two layers and reconstructs the access node set and the locality filter needed for the layer-1 query using information only stored with nodes in $\mathcal{T}_2$, i.e., for a layer-1 query with source node s, we build the union $\bigcup_{u \in A_2(s)} A(u)$ of all layer-1 access nodes of all layer-2 access nodes of s to determine on-the-fly a layer-1 access node set for s. Similarly, a layer-1 locality filter for s is built using the locality filters of the layer-2 access nodes. Variant *generous* accepts larger distance tables by choosing $K = 4$ (however using somewhat larger neighbourhoods for constructing the hierarchy). Generous stores all information required for a query with every node. To obtain a high quality layer-2 filter L_2, the generous variant performs a complete layer-3 preprocessing based on the core of level 1 and also stores a distance table for layer 3.

Since it has turned out that a better performance is obtained when the preprocessing starts with a contraction phase, we practically skip the first construction step (by choosing neighbourhood sets that contain only the node itself) so that the first highway network virtually corresponds to the original graph. Then, the first real step is the contraction of level 1 to get its core. Note that compared to numbers presented on Highway Hierarchies, we use a slightly improved contraction heuristic, which sorts the nodes according to degree and then tries to bypass the node with the smallest degree first.

Main Results

Table 7.3 gives the preprocessing times for both road networks and both the travel time and the distance metric; in case of the travel time metric, we distinguish between the economical and the generous variant. In addition, some key facts on the results of the preprocessing, e.g., the sizes of the transit node sets, are presented. It is interesting to observe that for the travel time metric in layer 2 the actual distance table size is only about 0.1% of the size a naive $|\mathcal{T}_2| \times |\mathcal{T}_2|$ table would have. As expected, the distance metric yields more access points than the travel time metric (a factor 2–3) since not only junctions on very fast roads (which are rare) qualify as access point. The fact that we have to increase the neighbourhood size from level to level in order to achieve an effective shrinking of the highway networks leads to comparatively high preprocessing times for the distance metric.

Table 7.4 summarises the average case performance of transit node routing. For the travel time metric, the generous variant achieves average query times more than two orders of magnitude lower than highway hierarchies alone. At the cost of a factor 2.4 in query time, the economical variant saves around a factor of two in space and a factor of 3.5 in preprocessing time.

Finding a good locality filter is one of the biggest challenges of a highway hierarchy based implementation of transit node routing. The values in Tab. 7.4 indicate that our filter is suboptimal: for instance, only 0.0064% of the queries performed by the economical variant in the US network with the travel time metric would require a local search to answer them correctly. However, the locality filter L_2 forces us to perform local searches in 0.278% of all cases. The high-quality layer-2 filter employed by the generous variant

metric	variant	layer 1 $\lvert\mathcal{T}\rvert$	layer 1 $\lvert$table$\rvert$ [$\times 10^6$]	layer 1 $\lvert A\rvert$	layer 2 $\lvert\mathcal{T}_2\rvert$	layer 2 $\lvert$table$_2\rvert$ [$\times 10^6$]	layer 2 $\lvert A_2\rvert$	layer 3 $\lvert\mathcal{T}_3\rvert$	layer 3 $\lvert$table$_3\rvert$ [$\times 10^6$]	space [B/node]	time [h]
	eco	12 111	147	6.1	184 379	30	4.9	–	–	111	0:59
USA time	gen	10 674	114	5.7	485 410	204	4.2	3 855 407	173	244	3:25
dist	eco	15 399	237	17.0	102 352	41	10.9	–	–	171	8:58
	eco	8 964	80	10.1	118 356	20	5.5	–	–	110	0:46
EUR time	gen	11 293	128	9.9	323 356	130	4.1	2 954 721	119	251	2:44
dist	eco	11 610	135	20.3	69 775	31	13.1	–	–	193	7:05

Table 7.3: Statistics on preprocessing for the highway hierarchy approach. For each layer, we give the size (in terms of number of transit nodes), the number of entries in the distance table, and the average number of access points to the layer. 'Space' is the total *overhead* of our approach.

	metric	variant	layer 1 [%] correct	layer 1 [%] stopped	layer 2 [%] correct	layer 2 [%] stopped	layer 3 [%] correct	layer 3 [%] stopped	query time
		eco	99.86	98.87	99.9936	99.7220	–	–	11.5 μs
USA	time	gen	99.89	99.20	99.9986	99.9862	99.99986	99.99984	4.9 μs
	dist	eco	98.43	91.90	99.9511	97.7648	–	–	87.5 μs
		eco	99.46	97.13	99.9908	99.4157	–	–	13.4 μs
EUR	time	gen	99.74	98.65	99.9985	99.9810	99.99981	99.99972	5.6 μs
	dist	eco	95.32	81.68	99.8239	95.7236	–	–	107.4 μs

Table 7.4: Performance of transit node routing with respect to 10 000 000 randomly chosen (s,t)-pairs. Each query is performed in a top-down fashion. For each layer i, we report the percentage of the queries that is answered correctly in some layer $\leq i$ and the percentage of the queries that is stopped after layer i (i.e., $\neg L_i(s,t)$).

is considerably more effective, still the percentage of false positives is about 90%.

For the distance metric, the situation is worse. Only 92% and 82% of the queries are stopped after the top layer has been searched (for the US and the European network, respectively). This is due to the fact that we had to choose the cores of levels 6 and 4 as layers 1 and 2 since the shrinking of the highway networks is less effective so that lower levels would be too big. It is important to note that we concentrated on the travel time metric—since we consider the travel time metric more important for practical applications—, and we spent comparatively little time to tune our approach for the distance metric. For example, a variant using a third layer (namely levels 6, 4, and 2 as layers 1, 2, and 3), which is not yet supported by our implementation, seems to be promising. Nevertheless, the current version shows feasibility and still achieves an improvement of a factor of 71 and 56 (for the US and the European network, respectively) over highway hierarchies alone. We use again a box-and-whisker plot to account for variance in

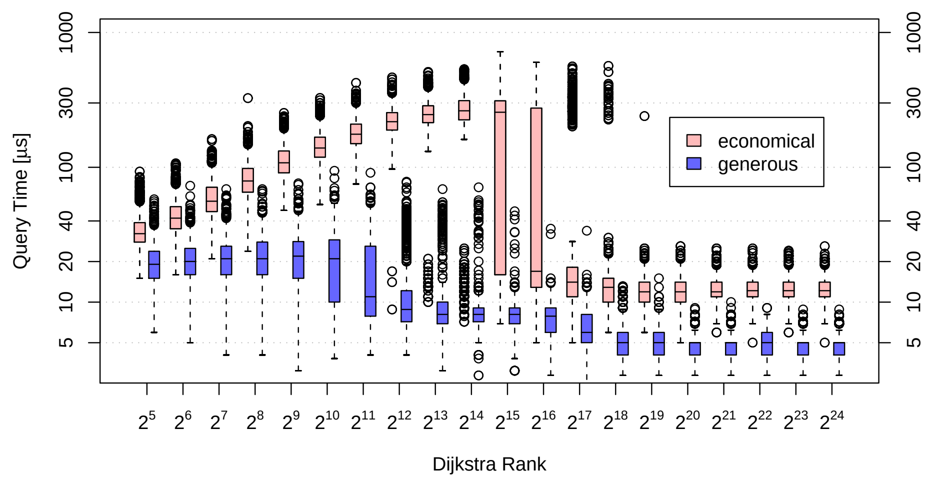


Figure 7.12: Query times for the USA with the travel time metric as a function of Dijkstra rank.

query times. For the generous approach, we can easily recognise the three layers of transit node routing with small transition zones in between: For ranks 2^{18}–2^{24} we usually have $\neg L(s, t)$ and thus only require cheap distance table accesses in layer 1. For ranks 2^{12}–2^{16}, we need additional look-ups in the table of layer 2 so that the queries get somewhat more expensive. In this range, outliers can be considerably more costly, indicating that occasional local searches are needed. For small ranks we usually need local searches and additional look-ups in the table of layer 3. Still, the combination of a local search in a very small area and table look-ups in all three layers usually results in query times of only about 20 μs.

In the economical approach, we observe a high variance in query times for ranks 2^{15}–2^{16}. In this range, all types of queries occur and the difference between the layer-1 queries and the local queries is rather big since the economical variant does not make use of a third layer. For smaller ranks, we see a picture very similar to basic highway hierarchies with query time growing logarithmically with Dijkstra rank.

7.4.3 Complete Description of the Shortest Path

For a given node pair (s, t), in order to get a complete description of the shortest s-t-path, we first perform a transit node query and determine the layer i that is used to obtain the shortest path distance. Then, we have to determine the path from s to the forward access point u to layer i, the path from the backward access point v to t, and the path from u to

v. In case of a local query, we can fall back on a normal highway search.

Currently, we provide an efficient implementation only for the case that the path goes through the top layer. In all other cases, we just perform a normal highway search. The effect on the average times is very small since more than 99% of the queries are correctly answered using only the top search (in case of the travel time metric; cp. Tab. 7.4).

When a node s and one of its access points u are given, we can determine the next node on the shortest path from s to u by considering all adjacent nodes s' of s and checking whether $d(s, s') + d(s', u) = d(s, u)$. In most cases, the distance $d(s', u)$ is directly available since u is also an access point of s'. In a few cases—when u is not an access point of s'—, we have to consider all access points u' of s' and check whether $d(s, s') + d(s', u') + d(u', u) = d(s, u)$. Note that $d(u', u)$ can be looked up in the top distance table. Using this subroutine, we can determine the path from s to the forward access point u and from the backward access point v to t.

A similar procedure can be used to find the path from u to v. However, in this case, we consider only adjacent nodes u' of u that belong to the top layer as well because only for these nodes we can look up $d(u', v)$. Since there are shortest paths between top layer nodes that leave the top layer—we call such paths *hidden paths*—, we execute an additional preprocessing step that determines all hidden paths and stores them in a special data structure (after the used shortcuts have been expanded). Whenever we cannot find the next node on the path to v considering only adjacent nodes in the top layer, we look for the right hidden path that leads to the next node in the top layer. In Tab. 7.5 we give the additional preprocessing time and the additional disk space for the hidden paths and the unpacking data structures. Furthermore, we report the additional time that is needed to determine a complete description of the shortest path and to traverse[5] it summing up the weights of all edges as a sanity check—assuming that the distance query has already been performed. That means that the total average time to determine a shortest path is the time given in Tab. 7.5 plus the query time given in Tab. 7.4.

7.5 Dynamic Shortest Path Computation

The successful methods we saw until now are *static*, i.e., they assume that the network—including its edge weights—does not change. This makes it possible to *preprocess* some information *once and for all* that can be used to accelerate *all* subsequent point-to-point *queries*. However, real road networks change all the time. In this section, we address two such *dynamic* scenarios: individual edge weight updates, e.g., due to traffic jams, and switching between different cost functions that take vehicle type, road restrictions, or driver preferences into account.

[5]Note that we do *not* traverse the path in the original graph, but we directly scan the assembled description of the path.

	preproc. [min]	space [MB]	query [μs]	# hops (avg.)
USA	4:04	193	258	4 537
EUR	7:43	188	155	1 373

Table 7.5: Additional preprocessing time, additional disk space and query time that is needed to determine a complete description of the shortest path and to traverse it summing up the weights of all edges—assuming that the query to determine its lengths has already been performed. Moreover, the average number of hops—i.e., the average path length in terms of number of nodes—is given. These figures refer to experiments on the graphs with the travel time metric using the generous variant.

7.5.1 Covering Nodes

We now introduce the concept of "covering nodes", which will be useful later.

Problem Definition.

During a Dijkstra search from s, we say that a settled node u is *covered* by a node set V' if there is at least one node $v \in V'$ on the path from the root s to u. A queued node is *covered* if its tentative parent is covered. The current partial shortest-path tree T is *covered* if all currently queued nodes are covered. All nodes $v \in V' \cap T$ that have no parent in T that is covered are *covering nodes*, forming the set $\mathcal{C}_G(V', s)$.

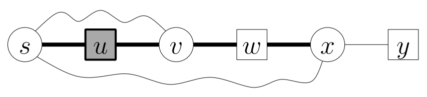
The crucial subroutine of all algorithms in the subsequent sections takes a graph G, a node set V', and a root s and determines all covering nodes $\mathcal{C}_G(V', s)$. We distinguish between four different ways of doing this.

Conservative Approach.

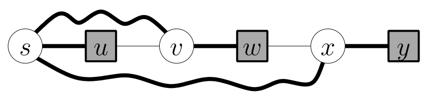
The *conservative* variant (Fig. 7.13 (a)) works in the obvious way: a search from s is stopped as soon as the current partial shortest-path tree T is covered. Then, it is straightforward to read off all covering nodes. However, if the partial shortest-path tree contains one path that is not covered for a long time, the tree can get very big even though all other branches might have been covered very early. In our application, this is a critical issue due to long-distance ferry connections.

Aggressive Approach.

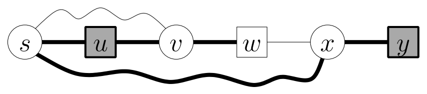
As an overreaction to the above observation, we might want to define an *aggressive* variant that does not continue the search from any covering node, i.e., some branches might



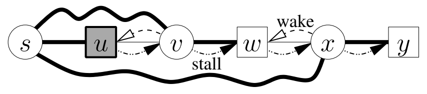
(a) conservative



(b) aggressive



(c) stall-in-advance



(d) stall-on-demand

Figure 7.13: Simple example for the computation of covering nodes. We assume that all edges have weight 1 except for the edges (s, v) and (s, x), which have weight 10. In each case, the search process is started from s. The set V' consists of all nodes that are represented by a square. Thick edges belong to the search tree T. Nodes that belong to the computed superset $\overline{\mathcal{C}}_G(V', s)$ of the covering nodes are highlighted in grey. Note that the actual covering node set contains only one node, namely u.

be terminated early, while only the non-covered paths are followed further on. Unfortunately, this provokes two problems. First, we can no longer guarantee that T contains only shortest paths. As a consequence, we get a superset $\overline{\mathcal{C}}_G(V', s)$ of the covering nodes, which still can be used to obtain correct results. However, the performance will be impaired. In Section 7.5.2, we will explain how to reduce a given superset rather efficiently in order to obtain the exact covering node set. Second, the tree T can get even bigger since the search might continue *around* the covering nodes where we pruned the search.[6] In our example (Fig. 7.13 (b)), the search is pruned at u so that v is reached using a much longer path that leads around u. As a consequence, w is superfluously marked as a covering node.

Stall-in-Advance Technique.

If we decide not to prune the search immediately, but to go on 'for a while' in order to *stall* other branches, we obtain a compromise between the conservative and the aggressive variant, which we call *stall-in-advance*. One heuristic we use prunes the search at node z when the path explored from s to z contains p nodes of V' for some tuning parameter p. Note that for $p := 1$, the stall-in-advance variant corresponds to the aggressive variant. In our example (Fig. 7.13 (c)), we use $p := 2$. Therefore, the search is pruned not until w is settled. This stalls the edge (s, v) and, in contrast to (b), the node v is covered. Still, the search is pruned too early so that the edge (s, x) is used to settle x.

[6]Note that the query algorithm of the separator-based approach virtually uses the aggressive variant to compute covering nodes. This is reasonable since the search can never 'escape' the component where it started.

Stall-on-Demand Technique.

In the stall-in-advance variant, relaxing an edge leaving a covered node is based on the 'hope' that this might stall another branch. However, our heuristic is not perfect, i.e., some edges are relaxed in vain, while other edges which would have been able to stall other branches, are not relaxed. Since we are not able to make the perfect decision in advance, we introduce a fourth variant, namely *stall-on-demand*. It is an extension of the aggressive variant, i.e., at first, edges leaving a covered node are not relaxed. However, if such a node u is reached later via another path, it is *woken up* and a breadth-first search (BFS) is performed from that node: an adjacent node v that has already been reached by the main search is inserted into the BFS queue if we can prove that the best path P found so far is suboptimal. This is certainly the case if the path from s via u to v is shorter than P. All nodes encountered during the BFS are marked as *stalled*. The main search is pruned at stalled nodes. Furthermore, stalled nodes are never marked as covering nodes. The stalling process cannot invalidate the correctness since only nodes are stalled that otherwise would contribute to suboptimal paths. In our example (Fig. 7.13 (d)), the search is pruned at u. When v is settled, we assume that the edge (v, w) is relaxed first. Then, the edge (v, u) wakes the node u up. A stalling process (a BFS search) is started from u. The nodes v and w are marked as stalled. When w is settled, its outgoing edges are not relaxed. Similarly, the edge (x, w) wakes the stalled node w and another stalling process is performed.

7.5.2 Static Highway-Node Routing

Multi-Level Overlay Graph.

For given *highway-node sets* $V =: V_0 \supseteq V_1 \supseteq ... \supseteq V_L$, we give a definition of the *multi-level overlay graph* $\mathcal{G} = (G_0, G_1, ..., G_L)$: $G_0 := G$ and for each $\ell > 0$, we have $G_\ell := (V_\ell, E_\ell)$ with $E_\ell := \{(s,t) \in V_\ell \times V_\ell \mid \exists \text{ shortest path } P = \langle s, u_1, u_2, \ldots, u_k, t\rangle \text{ in } G_{\ell-1} \text{ s.t. } \forall i : u_i \notin V_\ell\}$.

Node Selection

We can choose any highway node sets to get a correct procedure. However, the efficiency of both the preprocessing and the query very much depends on the highway node sets. Roughly speaking, a node that lies on a lot of shortest paths should belong to the node set of a high level. In a first implementation, we use the set of level-ℓ core nodes of the highway hierarchy of G as highway node set V_ℓ. In other words, we let the construction procedure of the highway hierarchies decide the importance of the nodes.

7.5.3 Construction

The multi-level overlay graph is constructed in a bottom-up fashion. In order to construct level $\ell > 0$, we perform for each node $s \in V_\ell$ a Dijkstra search in $G_{\ell-1}$ that is stopped as soon as the partial shortest-path tree is covered by $V_\ell \setminus \{s\}$. For each path $P = \langle s, u_1, u_2, \ldots, u_k, t\rangle$ in T with the property that $\forall i : u_i \notin V_\ell$, we add an edge (s, t) with weight $w(P)$ to E_ℓ.

Theorem 11 *The construction algorithm yields the multi-level overlay graph.*

Faster Construction Heuristics. Using the above construction procedure, we encounter the same performance problems and provide similar solutions than for the Highway Hierarchies: if the partial shortest-path tree contains a path that is not covered by a highway node for a long time, the tree can get very big even though all other branches might have been covered very early. In particular, we observed this behaviour in the European road network for long-distance ferry connections and for some long dead-end streets in the Alps. It is possible to prune the search at any settled node that is covered by $V_\ell \setminus \{s\}$. However, applying this aggressive pruning technique has two disadvantages. First, we can no longer guarantee that T contains only shortest paths. As a consequence, we obtain a superset of E_ℓ, which does not invalidate the correctness of the query, but which slows it down. Second, the tree T can get even bigger since the search might continue on slower roads *around* the nodes where we pruned the search.

It turns out that a good compromise is to prune only some edges at some nodes. We use two heuristic pruning rules. First, if for the current covered node u and some constant Δ, we have $d(s, u) + \Delta < \min\{\delta(v) \mid v \text{ reached, not settled, not covered by } V_\ell \setminus \{s\}\}$, then u's edges are not relaxed. Second, if on the path from s to the current node u, there are at least p nodes in some level ℓ (for some constant p), then all edges (u, v) in levels $< \ell$ are pruned.

After efficiently computing a superset of an overlay edge set E_ℓ, we can apply a fast reduction step to get rid of the superfluous edges: for each node $u \in V_\ell$, we perform a search in G_ℓ (instead of $G_{\ell-1}$) till all adjacent nodes have been settled. For any node v that has been settled via a path that consists of more than one edge, we can remove the edge (u, v) since a (better) alternative that does not require this edge has been found.

7.5.4 Query

The query algorithm is a bidirectional procedure: the backward search works completely analogously to the forward search so that it is sufficient to describe only the forward search. The search is performed in a bottom-up fashion. We perform a Dijkstra search from s in G_0 and stop the search as soon as the search tree is covered by V_1. From all covering nodes, the search is continued in G_1 until it is covered by V_2, and so on. In the topmost level, we can abort when forward and backward search meet. Figure 7.14 contains the pseudo-code of the query algorithm for the forward direction.

```
input: source node $s$;
$V_{L+1} := \emptyset$;                                    // to avoid case distinctions
$S_0 := \{s\}$; $\delta_0(s) := 0$;
for $\ell := 0$ to $L$ do
    $V'_\ell := V_\ell \cup \{s'\}$;                       // $s'$ is a new artificial node
    $E'_\ell := E_\ell \cup \{(s',u) \mid u \in S_\ell\}$; $w(s',u) := \delta_\ell(u)$;
    perform Dijkstra search from $s'$ in $G'_\ell := (V'_\ell, E'_\ell)$,
        stop when search tree is covered by $V_{\ell+1}$;
    $S_{\ell+1} := \emptyset$;
    foreach covering node $u$ do
        add $u$ to $S_{\ell+1}$;
        $\delta_{\ell+1}(u) := d(s',u)$;
```

Figure 7.14: The query algorithm for the forward direction.

Theorem 12 *The query algorithm always finds a shortest path.*

7.5.5 Analogies To and Differences From Related Techniques

Transit Node Routing. Let us consider a Dijkstra search from some node in a road network. We observe that some branches are very important—they extend through the whole road network—, while other branches are *stalled* by the more important branches at some point. For instance, there might be all types of roads (motorways, national roads, rural roads) that leave a certain region around the source node, but usually the branches that leave the region via rural roads end at some point since all further nodes are reached on a faster path using motorways or national roads. The transit node routing exploits this observation: not all nodes that separate different regions are selected as transit nodes, but only the nodes on the important branches.

Multi-level highway node routing uses the same argument to select the highway nodes. However, the distances from each node to the neighbouring highway nodes are not precalculated but computed during the query (using an algorithm very similar to the preprocessing algorithm for transit node routing). Moreover, the distances between all highway nodes are not represented by tables, but by overlay graphs. The algorithms to construct the overlay graphs and to compute the distance tables for transit node routing (except for the topmost table) are very similar. The fact that multi-level highway node routing relies on less precomputed data allows the implementation of an efficient update operation.

Multi-Level Overlay Graphs. In contrast to transit node and multi-level highway node routing, in the original multi-level approach *all* nodes that separate different regions are selected, which leads to a comparatively high average node degree. This has a negative impact on the performance. Let us consider the original approach with the new selection strategy, i.e., only 'important' nodes are selected. Then, the graph is typically not decomposed into many small components so that the following performance problem arises in the query algorithm. From the highway/separator nodes, only edges of the overlay graph are relaxed. As a consequence, the unimportant branches are not stalled by the important branches. Thus, since the separator nodes on the unimportant branches have not been selected, the search might extend through large parts of the road network.

To sum up, there are two major steps to get from the original to the new multi-level approach: first, select only 'important' nodes and second, at highway/separator nodes, do not switch immediately to the next level, but keep relaxing low-level edges 'for a while' until you can be sure that slow branches have been stalled.

Highway Hierarchies. We use the preprocessing of the highway hierarchies in order to select the highway nodes for our new approach. However, this is not the sole connection between both methods. In fact, we can interpret multi-level highway node routing as a modification of the highway hierarchy approach. (In particular, our actual implementation is a modification of the highway hierarchy program code.) An overlay graph can be represented by shortcut edges that belong to the appropriate level of the hierarchy. There are two main differences.

First, the neighbourhood of a node is defined in different ways. In case of the highway hierarchies, for a given parameter H, the H closest nodes belong to the neighbourhood. In case of multi-level highway node routing, all nodes belong to the neighbourhood that are settled by a search that is stopped when the search tree is covered by the highway node set.

Second, in case of the highway hierarchies, we decide *locally* when to switch to the next level, namely when the neighbourhood is left at some node. In case of multi-level highway node routing, we decide *globally* when to switch to the next level, namely when the complete search tree (not only the current branch) is covered by the highway node set[7]. By this modification, important branches can stall slow branches.

7.5.6 Dynamic Multi-Level Highway Node Routing

Various Scenarios

We could consider several types of changes in road networks, e.g.,

[7]This is a simplified description. As mentioned in Section 7.5.4, we enhance the query algorithm by some rules in order to deal with special cases like long-distance ferry connections more efficiently.

a) The structure of the road network changes: new roads are built, old roads are demolished. That means, edges can be added and removed.

b) A different cost function is used, which means that potentially all edge weights change. For example, a cost function can take into account different weightings of travel time, distance, and fuel consumption. With respect to travel time, we can think of different profiles of average speeds for each road category. In addition, for certain vehicle types there might be restrictions on some roads (e.g., bridges and tunnels). For many 'reasonable' cost functions, properties of the road network (like the inherent hierarchy) are possibly weakened, but not completely destroyed or even inverted. For instance, both a truck and a sports car—despite going different speeds—drive faster on a motorway than on an urban street.

c) An unexpected incident occurs: the travel time of a certain road or several roads in some area changes, e.g., due to a traffic jam. That means, a single or a few edge weights change. While a traffic jam causes a slow-down, the cancellation of a traffic jam causes a speed-up so that we have to deal with both increasing and decreasing edge weights.

d) The edge weights depend on the time of day according to some function known in advance. For example, such a function takes into account the rush hours.

The following paragraphs deal with type b) and c), respectively. We do not (explicitly) handle type a) since the addition of a new edge is a comparatively rare event in practical applications and the removal can be emulated by an edge weight change to infinity. Type d) is not (yet) covered by our work.

In case of type c), we can think of a server and mobile scenario: in the former, a server has to react to incoming events by updating its data structures so that *any* point-to-point query can be answered correctly; in the latter, a mobile device has to react to incoming events by (re)computing a *single* point-to-point query taking into account the new situation. In the server scenario, it pays to invest some time to perform the update operation since a lot of queries depend on it. In the mobile scenario however, we do not want to waste time for updating parts of the graph that are irrelevant to the current query. In this paper, we concentrate on the server scenario.

Complete Recomputation

The more time-consuming part of the preprocessing is the determination of the highway node sets. As stated above, we assume that the application of a different profile of average speeds will not completely invalidate the hierarchical properties of the road network, i.e., a node that has been very important usually will not get completely unimportant and vice versa when a different vehicle type is used. Thus, we can still expect a good query performance when keeping the highway node sets and recomputing only the overlay graphs. In order to do so, we do not need any additional data structures. We can directly use the static approach omitting the first preprocessing step (the determination of the highway

```
input: set of edges $E^m$ with modified weight;
define set of modified nodes: $V_0^m := \{u \mid (u,v) \in E^m\}$;
foreach level $\ell \geq 1$ do
    $V_\ell^m := \emptyset$;
    foreach node $v \in \bigcup_{u \in V_{\ell-1}^m} A_u^\ell$ do
        repeat construction step from $v$;
        if something changes, put $v$ to $V_\ell^m$;
```

Figure 7.15: The update algorithm that deals with a set of edge weight changes.

node sets).

Updating a Few Edge Weights

Similar to the previous paragraph, when a single or a few edge weights change, we keep the highway node sets and update only the overlay graphs. In contrast to the previous scenario, we do not have to repeat the complete construction from scratch, but it is sufficient to perform the construction step only from nodes that might be affected by the change. Certainly, a node v whose partial shortest-path tree of the initial construction did not contain any node u of a modified edge (u, x) is *not* affected: if we repeated the construction step from v, we would get exactly the same partial shortest-path tree and, consequently, the same result.

During the first construction (and all subsequent update operations), we manage sets A_u^ℓ of nodes whose level-ℓ preprocessing might be affected when an outgoing edge of u changes: when a level-ℓ construction step from some node v is performed, for each node u in the partial shortest-path tree, we add v to A_u^ℓ. Note that these sets can be stored explicitly (as we do it in our current implementation) or we could store a superset, e.g., by some kind of *geometric container* (a disk, for instance). Figure 7.15 contains the pseudo-code of the update algorithm.

Theorem 13 *After the update operation, we have the same situation as if we had repeated the complete construction procedure from scratch.*

7.5.7 Experiments

Environment and Instances.

The experiments were done on one core of a single AMD Opteron Processor 270 clocked at 2.0 GHz with 8 GB main memory and 2 $\times$ 1 MB L2 cache, running SuSE Linux

10.0 (kernel 2.6.13). The program was compiled by the GNU C++ compiler 4.0.2 using optimisation level 3.

We deal with the road network of Western Europe which was already used in the last sections. It consists of 18 029 721 nodes and 42 199 587 directed edges. The original graph contains for each edge a length and a road category. There are four major road categories (motorway, national road, regional road, urban street), which are divided into three subcategories each. In addition, there is one category for forest and gravel roads. We assign average speeds (130, 120, ..., 10 km/h)[8] to the road categories, compute for each edge the average travel time, and use it as weight. We call this our *default* speed profile. Experiments which we did on a US and Canadian road network of roughly the same size (provided by PTV as well) show exactly the same relative behaviour as in section 7.3.4, namely that it is slightly more difficult to handle North America than Europe (e.g., 20% slower query times). We give detailed results only for Europe.

For now, we report the times needed to compute the shortest-path distance between two nodes without outputting the actual route. Note that we could also output full path descriptions. The query times are averages based on 10 000 randomly chosen (s, t)-pairs. In addition to providing average values, we use the methodology from 7.9 in order to plot query times against the 'distance' of the target from the source, where in this context, the *Dijkstra rank* is used as a measure of distance: for a fixed source s, the Dijkstra rank of a node t is the rank w.r.t. the order which Dijkstra's algorithm settles the nodes in. Such plots are based on 1 000 random source nodes. After performing a lot of preliminary experiments, we decided to apply the stall-in-advance technique to the construction and update process (with $p := 1$ for the construction of level 1 and $p := 5$ for all other levels) and the stall-on-demand technique to the query.

Highway Hierarchy Construction.

In order to determine the highway node sets, we construct seven levels of the highway hierarchy using our default speed profile and neighbourhood size $H = 70$. This can be done in 16 minutes. For *all* further experiments, these highway-node sets are used.

Static Scenario.

The first data column of Tab. 7.6 contains the construction time of the multi-level overlay graph and the average query performance for the default speed profile. Figure 7.16 shows the query performance against the Dijkstra rank. The disk space overhead of the static variant is 8 bytes per node to store the additional edges of the multi-level overlay graph and the level data associated with the nodes. Note that this overhead can be further reduced to as little as 2.0 bytes per node yielding query times of 1.55 ms (Tab. 7.9). The

[8] we call this our *speed profile*

speed profile	default	(reduced)	fast car	slow car	slow truck	distance
constr. [min]	1:40	(3:04)	1:41	1:39	1:36	3:56
query [ms]	1.17	(1.12)	1.20	1.28	1.50	35.62
#settled nodes	1 414	(1 382)	1 444	1 507	1 667	7 057

Table 7.6: Construction time of the overlay graphs and query performance for different speed profiles using the same highway-node sets. For the default speed profile, we also give results for the case that the edge reduction step (Section 7.5.2) is applied.

	any road type				motorway				national		regional		urban	
|change set|	$+$	$-$	∞	$\times$	$+$	$-$	∞	$\times$	$+$	∞	$+$	∞	$+$	∞
1	2.7	2.5	2.8	2.6	40.0	40.0	40.1	37.3	19.9	20.3	8.4	8.6	2.1	2.1
1000	2.4	2.3	2.4	2.4	8.4	8.1	8.3	8.1	7.1	7.1	5.3	5.3	2.0	2.0

Table 7.7: Update times per changed edge [ms] for different road types and different update types: add a traffic jam ($+$), cancel a traffic jam ($-$), block a road (∞), and multiply the weight by 10 ($\times$). Due to space constraints, some columns are omitted.

	affected	#settled nodes		query time [ms]		
|change set|	queries	absolute	relative	init	search	total
1	0.6 %	2 347	(1.7)	0.3	2.0	2.3
10	6.3 %	8 294	(5.9)	1.9	7.2	9.1
100	41.3 %	43 042	(30.4)	10.6	36.9	47.5
1 000	82.6 %	200 465	(141.8)	62.0	181.9	243.9
10 000	97.5 %	645 579	(456.6)	309.9	627.1	937.0

Table 7.8: Query performance depending on the number of edge weight changes (select only motorways, multiply weight by 10). For ≤ 100 changes, 100 different edge sets are considered; for $\geq 1\,000$ changes, we deal only with one set. For each set, 1 000 queries are performed. We give the average percentage of queries whose shortest-path length is affected by the changes, the average number of settled nodes (also relative to zero changes), and the average query time, broken down into the init phase where the reliable levels are determined and the search phase.

method	preprocessing time [min]	preprocessing space [B/node]	static queries time [ms]	static queries #settled nodes	updates compl. [min]	updates single [ms]	dynamic queries #settled nodes 10 chgs.	dynamic queries #settled nodes 1000 chgs.
HH pure	17	28	1.16	1 662	17	–	–	–
StHNR	19	8	1.12	1 382	3	–	–	–
StHNR mem	24	2	1.55	2 453	8	–	–	–
DynHNR	18	32	1.17	1 414	2	37	8 294	200 465
DynALT-16	(85)	128	(53.6)	74 441	(6)	(2 036)	75 501	255 754

Table 7.9: Comparison between pure highway hierarchies, three variants of highway-node routing (HNR), and dynamic ALT-16 [36]. 'Space' denotes the average disk space *overhead*. We give execution times for both a complete recomputation using a similar cost function and an update of a single motorway edge multiplying its weight by 10. Furthermore, we give search space sizes after 10 and 1 000 edge weight changes (motorway, $\times 10$) for the mobile scenario. Time measurements in parentheses have been obtained on a similar, but not identical machine.

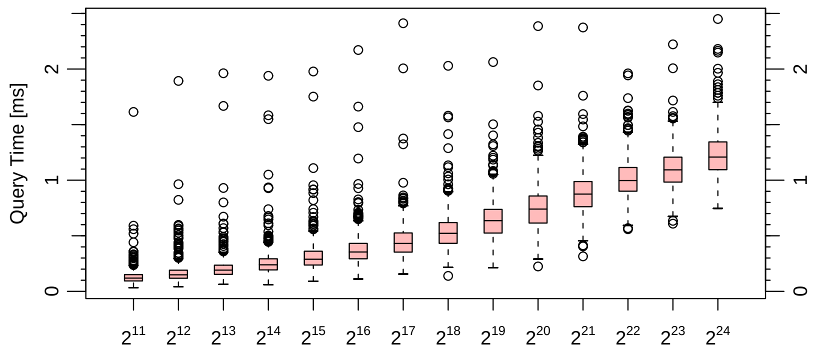


Figure 7.16: Query performance against Dijkstra rank for the default speed profile, with edge reduction. Each box represents the three quartiles box-and-whisker plot

total disk space[9] of 32 bytes per node also includes the original edges and a mapping from original to internal node IDs (that is needed since the nodes are reordered by level).

Changing the Cost Function.

In addition to our default speed profile, Tab. 7.6 also gives the construction and query times for a few other selected speed profiles (which have been provided by the company PTV AG) using the same highway-node sets. Note that for most road categories, our profile is slightly faster than PTV's fast car profile. The last speed profile ('distance') virtually corresponds to a distance metric since for each road type the same constant speed is assumed. The performance in case of the three PTV travel time profiles is quite close to the performance for the default profile. Hence, we can switch between these profiles without recomputing the highway-node sets. The constant speed profile is a rather difficult case. Still, it would not completely fail, although the performance gets considerably worse. We assume that any other 'reasonable' cost function would rank somewhere between our default and the constant profile.

Updating a Few Edge Weights (Server Scenario).

In the dynamic scenario, we need additional space to manage the affected node sets A_u^ℓ. Furthermore, the edge reduction step is not yet supported in the dynamic case so that the total disk space usage increases to 56 bytes per node. In contrast to the static variant, the main memory usage is considerably higher than the disk space usage (around a factor of two) mainly because the dynamic data structures maintain vacancies that might be filled during future update operations.

We can expect different performances when updating very important roads (like motorways) or very unimportant ones (like urban streets, which are usually only relevant to very few connections). Therefore, for each of the four major road categories, we pick 1 000 edges at random. In addition, we randomly pick 1 000 edges irrespective of the road type. For each of these edge sets, we consider four types of updates: first, we add a traffic jam to each edge (by increasing the weight by 30 minutes); second, we cancel all traffic jams (by setting the original weights); third, we block all edges (by increasing the weights by 100 hours, which virtually corresponds to 'infinity' in our scenario); fourth, we multiply the weights by 10 in order to allow comparisons to [36]. For each of these cases, Tab. 7.7 gives the average update time per changed edge. We distinguish between two change set sizes: dealing with only one change at a time and processing 1 000 changes simultaneously.

[9]The main memory usage is somewhat higher. However, we cannot give exact numbers for the static variant since our implementation does not allow to switch off the dynamic data structures.

As expected, the performance depends mainly on the selected edge and hardly on the type of update. The average execution times for a single update operation range between 40 ms (for motorways) and 2 ms (for urban streets). Usually, an update of a motorway edge requires updates of most levels of the overlay graph, while the effects of an urban-street update are limited to the lowest levels. We get a better performance when several changes are processed at once: for example, 1 000 random motorway segments can be updated in about 8 seconds. Note that such an update operation will be even more efficient when the involved edges belong to the same local area (instead of being randomly spread), which might be a common case in real-world applications.

Updating a Few Edge Weights (Mobile Scenario).

Table 7.8 shows for the most difficult case (updating motorways) that using our modified query algorithm we can omit the comparatively expensive update operation and still get acceptable execution times, at least if only a moderate amount of edge weight changes occur. Additional experiments have confirmed that, similar to the results in Tab. 7.7, the performance does not depend on the update type (add 30 minutes, multiply by 10, . . .), but on the edge type (motorway, urban street, . . .) and, of course, on the number of updates.

Comparisons.

Highway-node routing has similar preprocessing and query times as pure highway hierarchies, but (in the static case) a significantly smaller memory overhead. Table 7.9 gives detailed numbers, and it also contains a comparison to the dynamic ALT approach [36] with 16 landmarks. We can conclude that as a stand-alone method, highway-node routing is (clearly) superior to dynamic ALT w.r.t. all studied aspects.[10]

[10]Note that our comparison concentrates on only one variant of dynamic ALT: different landmark sets can yield different tradeoffs. Also, better results can be expected when a lot of very small changes are involved. Moreover, dynamic ALT can turn out to be very useful in combination with other dynamic speedup techniques yet to come.

Chapter 8

Minimum Spanning Trees

The section on the I-Max-Filteralgorithm is based on [37]. The external memory algorithm is described in [38] and the addition on connected components was taken from [6].

8.1 Definition & Basic Remarks

Consider a connected[1] undirected graph $G = (V, E)$ with positive edge weights $c : E \to \mathbb{R}_+$. A *minimum spanning tree (MST)* of G is defined by a set $T \subseteq E$ of edges such that the graph (V, T) is connected and $c(T) := \sum_{e \in T} c(e)$ is minimized. It is not difficult to see that T forms a tree[2] and hence contains $n - 1$ edges.

Because MSTs are such a simple concept, they also show up in many seemingly unrelated problems such as clustering, finding paths that minimize the maximum edge weight used, or finding approximations for harder problems like TSP.

8.1.1 Two important properties

The following two properties are the base for nearly every MST algorithm. On an abstract level they even suffice to formulate the algorithms by Kruskal and Prim presented later.

Cut Property: Consider a proper subset S of V and an edge $e \in \{(s,t) : (s,t) \in E, s \in S, t \in V \setminus S\}$ with minimal weight. Then there is an MST T of G that contains e.

Proof: Consider any MST T' of G. Since T' is a tree, T' contains a unique edge $e' \in T'$ connecting a node from S with a node from $V \setminus S$. Furthermore, $T' \setminus \{e'\}$ defines a spanning trees for S and $V \setminus S$ and hence $T = (T' \setminus \{e'\}) \cup \{e\}$ defines a spanning

[1]If G is not connected, we may ask for a *minimum spanning forest* — a set of edges that defines an MST for each connected component of G.

[2]In this chapter we often identify a set of edges T with a subgraph of (V, T).

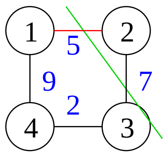


Figure 8.1: The cut property

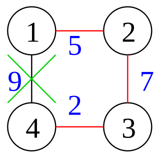


Figure 8.2: The cycle property

tree. By our assumption, $c(e) \leq c(e')$ and therefore $c(T) \leq c(T')$. Since T' is an MST, we have $c(T) = c(T')$ and hence T is also an MST.

Cycle Property: Consider any cycle $C \subseteq E$ and an edge $e \in C$ with maximal weight. Then any MST of $G' = (V, E \setminus \{e\})$ is also an MST of G.

Proof: Consider any MST T of G. Since trees contain no cycles, there must be some edge $e' \in C \setminus T$. If $e = e'$ then T is also an MST of G' and we are done. Otherwise, $T' = \{e'\} \cup T \setminus \{e\}$ forms another tree and since $c(e') \leq c(e)$, T' must also form an MST of G.

8.2 Classic Algorithms

The well known Jarnik-Prim algorithm starts from an (arbitrary) source node s and grows a minimum spanning tree by adding one node after the other, using the cut property. The set S is the set of nodes already added to the tree. This choice guarantees that the smallest edge leaving S is not in the tree yet.

This high-level description is of course not suited for implementation. The main challenge is to find (u, v) from the cut property efficiently. To this end, the algorithm 8.4 maintains the shortest connection between any node $v \in V \setminus S$ to S in an (adresseable) priority queue q. The smallest element in q gives the desired edge. To add a new node to S, we have to check its incident edges whether they give improved connections to nodes in $V \setminus S$. Note that by setting the distance of nodes in S to zero, edges connecting s with a node $v \in S$ will be ignored as required by the cut property. This small trick saves a comparison in the innermost loop.

It may be interesting to study the form of graph representation we need for the Jarnik-Prim algorithm. The graph is accessed when we add a new node to the tree and scan

```
T:= $\emptyset$
S:= $\{$s$\}$ for arbitrary start node $s$
repeat $n-1$ times
    find $(u,v)$ fulfilling the cut property for $S$
    $S$:= $S \cup \{v\}$
    $T$:= $T \cup \{(u,v)\}$
```

Figure 8.3: Abstract description of the Jarnik-Prim algorithm

```
Function jpMST(V, E, w) : Set of Edge
    dist=$[\infty,\ldots,\infty]$ : Array $[1..n]$          // $dist[v]$ is distance of $v$ from the tree
    pred : Array of Edge                    // pred$[v]$ is shortest edge between $S$ and $v$
    q : PriorityQueue of Node with dist$[\cdot]$ as priority
    dist$[s]$ := 0;   q.insert$(s)$ for any $s \in V$
    for $i$ := 1 to $n-1$ do  do
        $u$ := q.deleteMin()                                   // new node for $S$
        dist$[u]$ := 0
        foreach $(u,v) \in E$ do
            if $c((u,v)) < dist[v]$ then
                dist$[v]$ := $c((u,v))$; pred$[v]$ := $(u,v)$
                if $v \in q$ then q.decreaseKey$(v)$ else q.insert$(v)$
    return $\{pred[v] : v \in V \setminus \{s\}\}$
```

Figure 8.4: The Jarnik-Prim algorithm using priority queues

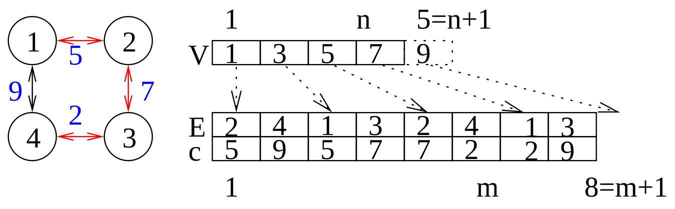


Figure 8.5: Adjacency Array

```
$T := \emptyset$                                              // subforest of the MST
**foreach** $(u, v) \in E$ in ascending order of weight **do**
    **if** $u$ and $v$ are in *different subtrees of* $T$ **then**
        $T := T \cup \{(u, v)\}$                              // Join two subtrees
**return** $T$
```

Figure 8.6: An abstract description of Kruskal's algorithm

its edges for new or cheaper connections to nodes outside the tree. An adjacency array (a static variant of the well-known adjacency list) supports this mapping from nodes to incident edges: We maintain the edges in a sorted array, first listing all neighbors of node 1 (and the costs to reach them), then all neighbors of node 2, etc. A second array maintains a pointer for every node leading to the first incident edge.

This representation is very cache efficient for our application (in contrast to e.g. a linked list). On the downside, we have to store every edge twice and receive a very static data structure.

For analysing the algorithm's runtime, we have to study the number of priority queue operations (all other instructions run in $\mathcal{O}(n + m)$). We obviously have n `deleteMin` operations, costing $\mathcal{O}(\log n)$ each. As every node is regarded exactly once, every edge is regarded exactly once resulting in $\mathcal{O}(m)$ `decreaseKey` operations. The latter can be implemented in amortized time $\mathcal{O}(1)$ using Fibonacci Heaps. In total, we have costs of $\mathcal{O}(m + n \log n)$. This result is partly theoretical as practical implementations will often resort to simpler pairing heaps for which the analysis is still open.

Another classic algorithm is due to Kruskal:

Again, correctness follows from from the cut property (set S as one of the subtrees connected by (u, v)).

For an efficient implementation of this algorithm we need a fast way to determine whether two nodes are in the same subtree. We use the Union-Find data structure for this task: It maintains disjoint sets (in our case containing the subtrees of T) whose union is V. It allows near-constant operations to identify the subtree a node is in (via the `find`-operaton) and to merge two subtrees using `link`. A more general overview over Union-

```
$T$ : *UnionFind*$(n)$
*sort* $E$ *in ascending order of weight*
*kruskal*$(E)$

**Procedure** *kruskal*$(E)$
    **foreach** $(u, v) \in E$ **do**
        $u'$:= $T.find(u)$
        $v'$:= $T.find(v)$
        **if** $u' \neq v'$ **then**
            *output* $(u, v)$
            $T.link(u', v')$
```

Figure 8.7: Kruskal's algorithm using union-find

Find is given in 8.2.1.

Using Union-Find, we have a running time of $\mathcal{O}(sort(m) + m\alpha(m, n)) = \mathcal{O}(m \log m)$ where α is the inverse Ackermann function.

The necessary graph representation is very simple: An array of edges is enough and can be sorted and scanned very cache efficiently. Every edge is represented only once.

Which of these two algorithms is better? As often, there is no easy answer to this question. Kruskal wins for very sparse graphs while Prim's algorithm is more suited for dense graphs. The switching point is unclear and is heavily dependant on the input representation, the structure of the graphs, etc. Systematic experimental evaluation is required.

8.2.1 Excursus: The Union-Find Data Structure

A *partition* of a set M into subsets $M_1, \ldots, M_k$ has the property that the subsets are disjoint and cover M, i.e., $M_i \cup M_j = \emptyset$ for $i \neq j$ and $M = M_1 \cup \cdots \cup M_k$. For example, in Kruskal's algorithm the forest T partitions V into subtrees — including trivial subsets of size one for isolated nodes. Kruskal's algorithms performs two operations on the partition: Testing whether two elements are in the same subset (subtree) and joining two subsets into one (inserting an edge into T).

The *union-find data structure* maintains a partition of the set $1..n$ and supports these two operations. Initially, each element is in its own subset. Each subset is assigned a *leader* element (or representative). The function `find`(i) finds the leader of the subset containing i; `link`(i, j) applied to leaders of different partitions joins these two subsets. Figure 8.8 gives an efficient implementation of this idea. The most important part of the data structure is the array $parent$. Leaders are their own parents. Following parent

```
Class UnionFind($n : \mathbb{N}$)                                    // Maintain a partition of $1..n$
    parent=$\langle 1, 2, \ldots, n\rangle$ : Array [$1..n$] of $1..n$
    gen=$\langle 0, \ldots, 0\rangle$ : Array [$1..n$] of $0..\log n$                  // generation of leaders
Function find($i : 1..n$) : $1..n$                                           // picture 'before'
    if parent[$i$] $= i$ then return $i$
    else $i'$ := find(parent[i])
        parent[$i$] := $i'$                                                // path compression
        return $i'$                                                          // picture 'after'
Procedure link($i, j : 1..n$)                                                // picture 'before'
    assert $i$ and $j$ are leaders of different subsets
    if gen[$i$] $<$ gen[$j$] then parent[$i$] := $j$                                   // balance
    else
        parent[$j$] := $i$
        if gen[$i$] $=$ gen[$j$] then gen[$i$]++
Procedure union($i, j : 1..n$)
    if find($i$) $\neq$ find($j$) then link(find($i$), find($j$))
```

Figure 8.8: An efficient Union-Find data structure maintaining a partition of the set $\{1, \ldots, n\}$.

references leads to the leaders. The `parent` references of a subset form a *rooted tree*, i.e., a tree with all edges directed towards the root.[3] Additionally, each root has a self-loop. Hence, `find` is easy to implement by following the `parent` references until a self-loop is encountered.

Linking two leaders i and j is also easy to implement by promoting one of the leaders to overall leader and making it the parent of the other. What we have said so far yields a correct but inefficient union-find data structure. The `parent` references could form long chains that are traversed again and again during `find` operations.

Therefore, Figure 8.8 makes two optimizations. The `link` operation uses the array `gen` to limit the depth of the `parent` trees. Promotion in leadership is based on the seniority principle. The older generation is always promoted. It can be shown that this measure alone limits the time for `find` to $\mathcal{O}(\log n)$. The second optimization is *path compression*. A long chain of parent references is never traversed twice. Rather, `find` redirects all nodes it traverses directly to the leader. It is possible to prove that these two optimizations together make the union-find data structure "breath-takingly" efficient —

[3]Note that this tree may have very different structure compared to the corresponding subtree in Kruskal's algorithm.

```
Procedure quickKruskal($E$ : Sequence of Edge)
    if $m \leq \beta n$ then kruskal($E$)                         // for some constant $\beta$
    else
        pick a pivot $p \in E$
        $E_{\leq}$:= $\langle e \in E : e \leq E \rangle$                  // partitioning a la
        $E_{>}$:= $\langle e \in E : e > E \rangle$                        // quicksort
        quickKruskal($E_{\leq}$)
        $E'_{>}$:= filter($E_{>}$)
        quickKruskal($E'_{>}$)
Function filter($E$)
    make sure that leader[$i$] gives the leader of node $i$        // $\mathcal{O}(n)$!
    return $\langle (u, v) \in E : leader[u] \neq leader[v] \rangle$
```

Figure 8.9: The QuickKruskal algorithm

the amortized cost of any operation is almost constant.

8.3 QuickKruskal

As Kruskal's algorithm becomes less attractive for dense graphs, we propose a variant that uses a quicksort-like recursion to deal with those instances:

When the average degree is bounded by some constant β (i.e. the graph is sparse) we know that Kruskal's algorithm performs well. Else, we determine a MST recursively on the smallest edges of the graph, resulting in a set of connected components. Now the second recursion only has to regard those heavy edges connecting two different components, the others are filtered out. The filtering subroutine makes again use of the Union-Find data structure: We have $leader[v]$:= `find(v)` to determine the connected component node v is in. Note that n find operations have a running time of $\mathcal{O}(n)$[4].

We can now attempt an average-case analysis of QuickKruskal: We assume that the weights are unique and randomly chosen, the pivot has median weight. Let $T(m)$ denote the expected execution time for m edges. For $m \leq \beta n$, our base case, we have $T(m) = \mathcal{O}(m \log m) = \mathcal{O}(n \log n)$. In the general case, we have costs of $\mathcal{O}(m + n) = \Omega(m)$ for partitioning and filtering. $E_{\leq}$ has a size of $m/2$ for an optimal pivot. The key observation here is that the number of edges surviving the filtering is only linear in n.

[4]this can be shown using amortized analysis: Every element accessed once during a find operation will be a direct successor of its root node, resulting in constant costs for subsequent requests. $\mathcal{O}(n)$ is less than the general bound on n union-find operations

```
$R$:= random sample of $r$ edges from $E$
$F$ :=MST($R$)                                        // Wlog assume that $F$ spans $V$
$L$:= $\emptyset$                                     // “light edges” with respect to $R$
foreach $e \in E$ do                                  // Filter
    $C$ := the unique cycle in $\{e\} \cup F$
    if $e$ is not heaviest in $C$ then
        $L$:= $L \cup \{e\}$
return MST($(L \cup F)$)
```

Figure 8.10: A simplified filtering algorithm using random samples

This leads to $T(m) = \Omega(m) + T(m/2) + T(2n)$. Since for $\beta \geq 2$, the second recursion will already fall back to the base case, we have the linear recurrence $T(m) = \Omega(m) + n \log n + T(m/2)$ which solves (using stanard techniques) to $\mathcal{O}\big(m + n \log n \log \frac{m}{n}\big)$.

A hard instance for QuickKruskal would consist of several very dense components consisting of light edges which are connected by heavy edges which would not be sorted out during filtering as the first recursion will concentrate on local MSTs within the components. More concrete: Consider the fully connected graph K_{x^2} (for $x \in \mathbb{N}$) where every node is replaced with another K_x. This graph has $\mathcal{O}(2x^4)$ edges. Let the “outer“ edges have weight 2 whereas the “inner“ edges have weight 1. The first recursion of QuickKruskal will only regard edges within the K_x components but completely ignore the heavy edges so none of them is filtered out.

8.4 The I-Max-Filter algorithm

A similar approach also resorts to filtering, but the subgraph does not consist of the lightest edges but of a random sample:

While Kruskal’s and Prim’s algorithms make use of the Cut Property, this code’s correctness is guaranteed by the Cycle Property. Its performance depends on the size of L and F. It can be shown that if r edges are chosen, we expect only $\frac{mn}{r}$ edges to survive the filtering[5].

The tricky part in implementing this algorithm is how to determine the heaviest edge in C. We exploit that by renumbering nodes according to the order in which they are added to the MST by the Jarnik-Prim algorithm, heaviest edge queries can be reduced

[5]this is because for every edge not to be filtered has to be in $MST(\{e\} \cup R)$, which has $\leq n$ elements. The probability of survival therefore is $\leq \frac{n}{r}$, as r edges are regarded

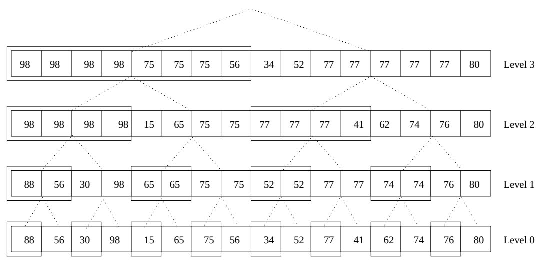


Figure 8.11: Example of a layers array for interval maxima. The suffix sections are marked by an extra surrounding box.

to simple interval maximum queries. A proof for this claim can be found in 12.5. We therefore reduced the problem to efficiently compute interval maxima:

Given an array $a[0] \ldots a[n-1]$, we explain how $\max a[i..j]$ can be computed in constant time using preprocessing time and space $\mathcal{O}(n \log n)$. The emphasis is on very simple and fast queries since we are looking at applications where many more than $n \log n$ queries are made. This algorithm might be of independent interest for other applications. Slight modifications of this basic algorithm are necessary in order to use it in the I-Max-Filter algorithm. They will be described later. In the following, we assume that n is a power of two. Adaption to the general case is simple by either rounding up to the next power of two and filling the array with $-\infty$ or by introducing a few case distinctions while initializing the data structure.

Consider a complete binary tree built on top of a so that the entries of a are the leaves (see level 0 in Figure 8.11). The idea is to store an array of prefix or suffix maxima with every internal node of the tree. Left successors store suffix maxima. Right successors store prefix maxima. The size of an array is proportional to the size of the subtree rooted at the corresponding node. To compute the interval maximum $\max a[i..j]$, let v denote the least common ancestor of $a[i]$ and $a[j]$. Let u denote the left successor of v and let w denote the right successor of v. Let $u[i]$ denote the suffix maximum corresponding to leaf i in the suffix maxima array stored in u. Correspondingly, let $w[j]$ denote the prefix maximum corresponding to leaf j in the prefix maxima array stored in w. Then $\max a[i..j] = \max(u[i], w[j])$.

We observed that this approach can be implemented in a very simple way using a $\log(n) \times n$ array preSuf. As can be seen in Figure 8.11, all suffix and prefix arrays in one layer can be assembled in one array as follows

$$\mathrm{preSuf}[\ell][i] = \begin{cases} \max(a[2^\ell b..i]) & \text{if } b \text{ is odd} \\ \max(a[i..(2^\ell+1)b-1]) & \text{otherwise} \end{cases}$$

where $b = \lfloor i/2^\ell \rfloor$.

```
//Compute MST of $G = (\{0, \ldots, n-1\}, E)$
Function I-Max-Filter-MST($E$) : set of Edge
    $E'$ := random sample from $E$ of size $\sqrt{mn}$
    $E''$ := JP-MST($E'$)
    Let $\mathrm{jpNum}[0..n-1]$ denote the order in which JP-MST added the nodes
    Initialize the table $\mathrm{preSuf}[0..\log n][0..n-1]$
    //Filtering loop
    forall edges $e = (u, v) \in E$ do
        $\ell$ := $\mathrm{msbPos}(\mathrm{jpNum}[u] \oplus \mathit{jpNum}[v])$
        if $w_e < \mathrm{preSuf}[\ell][\mathrm{jpNum}[u]]$ and $w_e < \mathrm{preSuf}[\ell][\mathrm{jpNum}[v]]$ then add $e$ to $E''$
return JP-MST($E''$)
```

Figure 8.12: The I-Max-Filter algorithm.

Furthermore, the interval boundaries can be used to index the arrays. We simply have $\max a[i..j] = \max(\mathrm{preSuf}[\ell][i], \mathrm{preSuf}[\ell][j])$ where $\ell = \mathrm{msbPos}(i \oplus j)$; $\oplus$ is the bitwise exclusive-or operation and $\mathrm{msbPos}(x) = \lfloor \log_2 x \rfloor$ is equal to the position of the most significant nonzero bit of x (starting at 0). Some architectures have this operation in hardware[6]; if not, $\mathrm{msbPos}(x)$ can be stored in a table (of size n) and found by table lookup. Layer 0 is identical to a. A further optimization stores a pointer to the array $\mathrm{preSuf}[\ell]$ in the layer table. As the computation is symmetric, we can conduct a table lookup with indices i, j without knowing whether $i < j$ or $j < i$.

To use this data structure for the I-Max-Filter algorithm we need a small modification since we are interested in maxima of the form $\max a[\min(i,j) + 1..\max(i,j)]$ without knowing which of two endpoints is the smaller. Here we simply note that the approach still works if we redefine the suffix maxima to exclude the first entry, i.e., $\mathrm{preSuf}[\ell][i] = \max(a[i+1..(2^\ell + 1)\lfloor i/2^\ell \rfloor - 1])$ if $\lfloor i/2^\ell \rfloor$ is even.

We can now return to the original problem of finding an MST. Figure 8.12 gives a detailed implementation of the I-Max-Filteralgorithm:

The I-Max-Filter algorithm computes MSTs in expected time
$mT_{\text{filter}} + \mathcal{O}(n \log n + \sqrt{nm})$ where T_{filter} is the time required to query the filter about one edge.

The algorithms we saw until now all had specific requirements for the graph representation. The I-Max-Filter algorithm can be implemented to work well with any representation that allows sampling edges in time linear in the sample size and that allows fast iteration over all edges. In particular, it is sufficient to store each edge once. Our implementation for I-Max-Filter uses an array in which each edge appears once as (u, v) with $u < v$ and the edges are sorted by source node (u).[7]

[6]One trick is to use the exponent in a floating point representation of x.

[7]These requirements could be dropped at very small cost. In particular, I-Max-Filter can work efficiently

Experiments

I-Max-Filter should work well for dense graphs where $m \gg n \log n$. We try to prove this claim in experiments.

Both algorithms, JP and I-Max-Filter were implemented in C++ and compiled using GNU `g++ version 3.0.4` with optimization level `-O6`. We use a SUN-Fire-15000 server with 900 MHz UltraSPARC-III+ processors.

We performed measurements with four different families of graphs, each with adjustable *edge density* $\rho = 2m/n(n-1)$. A test instance is defined by three parameters: the graph type, the number of nodes and the density of edges (the number of edges is computed from these parameters). Each reported result is the average of ten executions of the relevant algorithm; each on a different randomly generated graph with the given parameters. Furthermore, the I-Max-Filter algorithm is randomized because the sample graph is selected at random. Despite the randomization, the variance of the execution times within one test was consistently very small (less than 1 percent), hence we only plot the averages.

Worst-Case: $\rho \cdot n(n-1)/2$ edges are selected at random and the edges are assigned weights that cause JP to perform as many Decrease Key operations as possible.

Linear-Random: $\rho \cdot n(n-1)/2$ edges are selected at random. Each edge (u, v) is assigned the weight $w(u, v) = |u - v|$ where u and v are the integer IDs of the nodes.

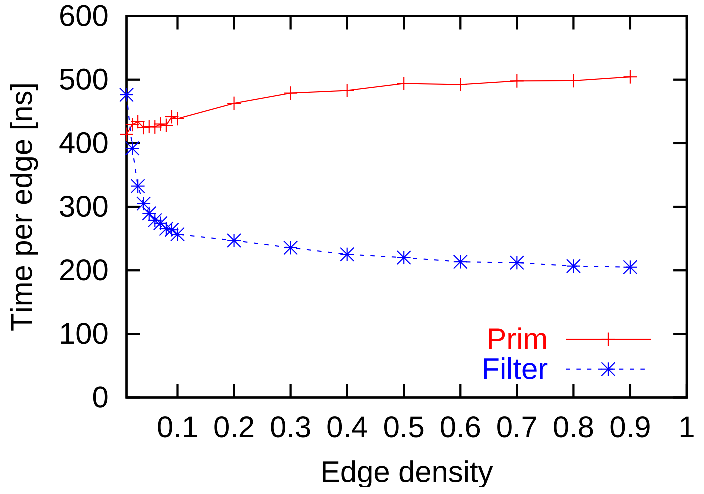
Uniform-Random: $\rho \cdot n(n-1)/2$ edges are selected at random and each is assigned an edge weight which is selected uniformly at random.

Random-Geometric: Nodes are random 2D points in a $1 \times y$ rectangle for some stretch factor $y > 0$. Edges are between nodes with Euclidean distance at most α and the weight of an edge is equal to the distance between its endpoints. The parameter α indirectly controls density whereas the stretch factor y allows us to interpolate between behavior similar to class Uniform-Random and behavior similar to class Linear-Random.

Fig. 8.13 shows execution times per edge on the SUN for two graph families Worst-Caseand Uniform-Random for $n = 10000$ nodes and varying density. We can see that I-Max-Filter is up to 2.46 times faster than JP. The speedup is smaller for Uniform-Random graphs. The reason is that for "average" inputs JP needs to perform only a sublinear number of decrease-key operations so that the part of code dominating the execution time of JP is scanning adjacency lists and comparing the weight of each edge with the distance of the target node from the current MST. There is no hope to be significantly faster than that. Hence, when we say that I-Max-Filter outperforms JP this is with respect to space consumption, simplicity of input conventions and worst-case performance guarantees rather than average case execution time.

On very sparse graphs, I-Max-Filter is up to two times slower than JP, because

with a completely unsorted edge array or with an adjacency array representation that stores each edge only in one direction. The latter only needs space for $m + n$ node indices and m edge weights.



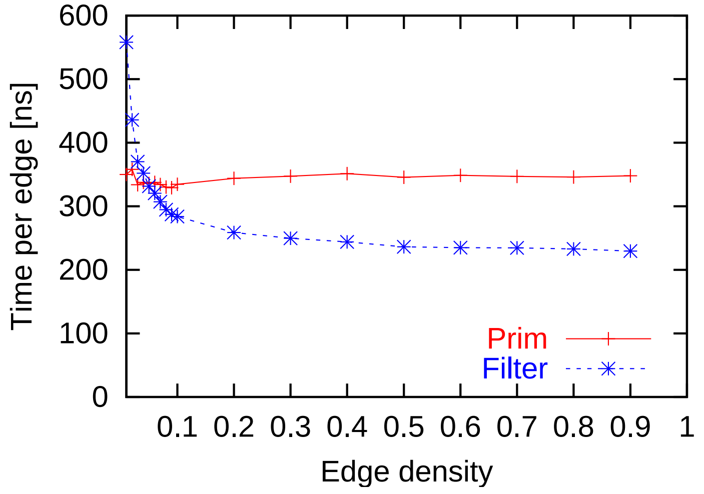


Figure 8.13: Worst-Case and Uniform-Random graphs, 10000 nodes on a SUN machine.

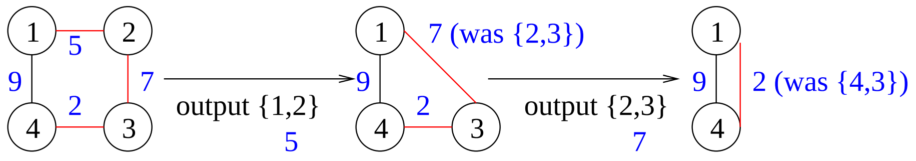


Figure 8.14: Example for node contraction

$\sqrt{mn} = \Theta(m)$ and as a result both the sample graph and the graph that remains after the filtering stage are not much smaller than the original graph. The runtime is therefore comparable to two runs of JP on the input.

8.5 External MST

After studying and extending some classic algorithms for graphs in main memory, we now consider another approach. We start with a simple randomized algorithm using a graph contraction operation, develop an even simpler variant and step by step generate an external algorithm for huge graphs.

Contracting is defined as follows: If $e = (u, v) \in E$ is known to be an MST edge, we can remove u from the problem by outputting e and identifying u and v, e.g., by removing node u and renaming an edge of the form (u, w) to a new edge (v, w). By remembering where (v, w) came from, we can reconstruct the MST of the original graph from the MST of the smaller graph.

With this operation, we can use Boruvka's Algorithm which consists of repeated execution of *Boruvka Phases*: In each phase, find the lightest incident edge for each node. The set C of these edges can be output as part of the MST (because of the Cut property). Now contract these edges, i.e., find a representative node for each connected component of (V, C) and rename an edge $\{u, v\}$ to $\{\text{componentId}(u), \text{componentId}(v)\}$. This routine at least halves the number of nodes (as every edge is picked at most twice) and runs in $\mathcal{O}(m)$[8]. In total, if we contract our graph until only one node is left, we have a runtime of $\mathcal{O}(m \log n)$.

On our way to an external MST algorithm we will use a simpler variant of Boruvka's Algorithm which does not use phases — Sibeyn's algorithm:

In the iteration when i nodes are left (note that $i = n$ in the first iteration), the expected degree of a random node is at most $2m/i$. Hence, the expected number of edges, X_i, inspected in iteration i is at most $2m/i$. By the linearity of expectation, the total expected

[8]We can use Union-Find again: m operations to construct all components by merging and another m operations to find the edges between different components

```
**for** $i$ := $n$ **downto** $n' + 1$ **do**
    *pick a random node* $v$
    *find the* lightest *edge* $(u, v)$ *out of* $v$ *and output it*
    *contract* $(u, v)$
```

Figure 8.15: High level version of Sibeyn's MST algorithm.

```
*Factor 8 node reduction (*$3\times$ *Boruvka or sweep algorithm) //* $\mathcal{O}(m + n)$
$R \Leftarrow m/2$ *random edges*
$F \Leftarrow MST(R)$ *[Recursively]*
*Find light edges* $L$ *(edge reduction) //* $\mathcal{O}(m + n)$, $\mathrm{E}[|L|] \leq \frac{mn/8}{m/2} = n/4$
$T \Leftarrow MST(L \cup F)$ *[Recursively]*
```

Figure 8.16: Outline of a randomized linear time MST algorithm.

number of edges processed is

$$\mathrm{E}[\sum_{n'<i\leq n} X_i] = \sum_{n'<i\leq n} \mathrm{E}[X_i] \leq \sum_{n'<i\leq n} \frac{2m}{i} = 2m \sum_{n'<i\leq n} \frac{1}{i} = 2m \left(\sum_{1\leq i\leq n} \frac{1}{i} - \sum_{1\leq i\leq n'} \frac{1}{i} \right)$$

$$= 2m(H_n - H_{n'}) \leq 2m(\ln n - \ln n') = 2m \ln \frac{n}{n'}$$

where $H_n = \ln n + 0.577 \cdots + \mathcal{O}(1/n)$ is the n-th harmonic number.

The techniques of sampling and contraction can lead to a (impractical) randomized linear time algorithm, developed by Karger, Klein and Tarjan. It is presented in 8.16 but not studied in detail. Its analysis depends again on the observation that clever sampling will lead to an expected number of unfiltered (light) edges linear in n. The complicated step is the fourth, done in linear time using table lookups. The expected runtime for this algorithm is given by $T(n, m) \leq T(n/8, m/2) + T(n/8, n/4) + c(n + m)$, which is fulfilled by $T(n, m) = 2c(n + m)$.

8.5.1 Semiexternal Algorithm

A first step for an algorithm that can cope with huge graphs stored on disk is a semiexternal algorithm: We use Kruskal's Algorithm but incorporate an external sorting algorithm. We then just have to scan the edges and maintain the union-find array in main memory.

```
$\pi$ : random permutation $V \to V$
sort edges $(u, v)$ by $\min(\pi(u), \pi(v))$
for $i := n$ downto $n' + 1$ do
    pick the node $v$ with $\pi(v) = i$
    find the lightest edge $(u, v)$ out of $v$ and output it
    contract $(u, v)$
```

Figure 8.17: High level implementation for graph contraction with sweeping

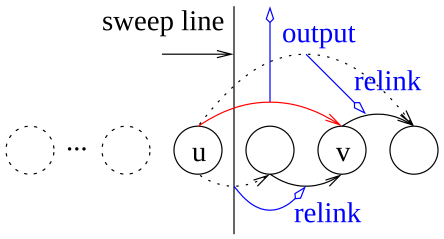


Figure 8.18: Sweeping scans through the randomly ordered nodes, removes one, outputs its lightest edge and relinks the others

This only requires one 32 bit word per node to store up to $0..2^{32} - 32 = 4\,294\,967\,264$ nodes. There exist asymptotically better algorithms but these come with discouraging constant factors and significantly larger data structures.

8.5.2 External Sweeping Algorithm

We can use the semiexternal algorithm while $n < M - 2B$. For larger inputs, we cannot store the additional data structures in main memory. To deal with those graphs, we use again the technique of node reduction via contraction until we can resort to our semiexternal algorithm.

This algorithm is a more concrete implementation of Sibeyn's algorithm from Figure 8.5. We replace random selection of nodes by *sweeping* the nodes in an order fixed in advance. We assume that nodes are numbered $0..n-1$. We first rename the node indices using a random permutation $\pi : 0..n-1 \to 0..n-1$ and then remove renamed nodes in the order $n-1$, $n-2$,..., n'. This way, we replace random access by sorting and scanning the nodes once. The appendix (in section 12.6) describes a procedure to create a random permutation on the fly without additional I/Os.

There is a very simple external realization of the sweeping algorithm based on priority queues of edges. Edges are stored in the form $((u, v), c, e_{\text{old}})$ where (u, v) is the edge in the current graph, c is the edge weight, and e_{old} identifies the edge in the original graph.

```
Q: priority queue                    // Order: max node, then min edge weight
foreach $(\{u,v\},c) \in E$ do Q.insert($(\{\pi(u),\pi(v)\},c,\{u,v\})$)
current := $n+1$
loop
    $(\{u,v\},c,\{u_0,v_0\})$ := Q.deleteMin()
    if current $\neq \max\{u,v\}$ then
        if current $= M+1$ then return
        output $\{u_0,v_0\},c$
        current := $\max\{u,v\}$
        connect := $\min\{u,v\}$
    else Q.insert($(\{\min\{u,v\},connect\},c,\{u_0,v_0\})$)
```

Figure 8.19: Sweeping algorithm implementation using Priority Queues

The queue normalizes edges (u,v) in such a way that $u \geq v$. We define a priority order $((u,v),c,e_{\text{old}}) < ((u',v'),c',e'_{\text{old}})$ iff $u > u'$ or $u = u'$ and $c < c'$. With these conventions in place, the algorithm can be described using the simple pseudocode in Figure 8.19. If e_{old} is just an edge identifier, e.g. a position in the input, an additional sorting step at the end can extract the actual MST edges. If e_{old} stores both incident vertices, the MST edge and its weight can be output directly.

With optimal external priority queues, this implementation performs $\approx sort(10m\log\frac{n}{M})$ I/Os.

The priority queue implementation unnecessarily sorts the edges adjacent to a node where we really only care about the smallest edge coming first. We now describe an implementation of the sweeping algorithm that has internal work linear in the total I/O volume. We first make a few simplifying assumptions to get closer to our implementation.

The representation of edges and the renaming of nodes works as in the priority queue implementation. As before, in iteration i, node i is removed by outputting the lightest edge incident to it and relinking all the other edges. We split the node range $n'..n-1$ into $k = \mathcal{O}(M/B)$ equal sized *external buckets*, i.e., subranges of size $(n-n')/k$ and we define a special external bucket for the range $0..n'-1$. An edge (u,v) with $u > v$ is always stored in the bucket for u. We assume that the current bucket (that contains i) completely fits into main memory. The other buckets are stored externally with only a write buffer block to accommodate recently relinked edges.

When i reaches a new external bucket, it is distributed to *internal buckets* — one for each node in the external bucket. The internal bucket for i is scanned twice. Once for finding the lightest edge and once for relinking. Relinked edges destined for the current external bucket are immediately put into the appropriate internal bucket. The remaining edges are put into the write buffer of their external bucket. Write buffers are flushed to

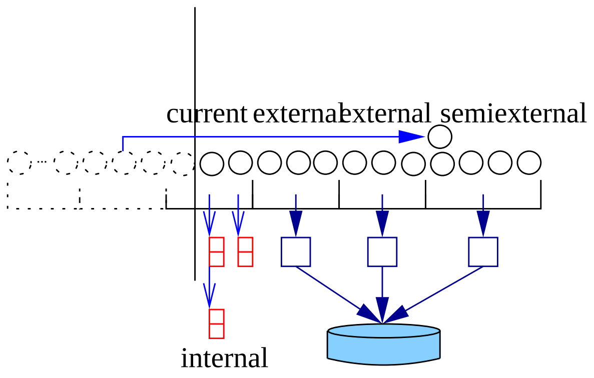


Figure 8.20: Sweeping with buckets

disk when they become full.

When only n' nodes are left, the bucket for range $0..n' - 1$ is used as input for the semi-external Kruskal algorithm.

Nodes with very high degree ($> M$) can be moved to the bucket for the semiexternal case directly. These nodes can be assigned the numbers $n' + 1$, $n' + 2$,... without danger of confusing them with nodes with the same index in other buckets. To accomodate these additional nodes in the semiexternal case, n' has to be reduced by at most $\mathcal{O}(M/B)$ since for $m = \mathcal{O}(M^2/B)$ there can be at most $\mathcal{O}(M/B)$ nodes with degree $\Omega(M)$.

8.5.3 Implementation & Experiments

Our external implementation makes extensive use of the Stxxl[9] and uses many techniques and data structures we already saw in earlier chapters. The semiexternal Kruskal and the priority queue based sweeping algorithm become almost trivial using external sorting and external priority queues. The bucket based implementation uses external stacks to represent external buckets. The stacks have a single private output buffer and they share a common pool of additional output buffers that facilitates overlapping of output and internal computation. When a stack is switched to reading, it is assigned additional private buffers to facilitate prefetching.

The internal aspects of the bucket implementation are also crucial. In particular, we need a representation of internal buckets that is space efficient, cache efficient, and can grow adaptively. Therefore, internal buckets are represented as linked lists of small blocks that can hold several edges each. Edges in internal buckets do not store their source node because this information is redundant.

For experiments we use three families of graphs: Instance families for *random* graphs with random edge weights and random *geometric* graphs where random points in the unit

[9]see chapter 5

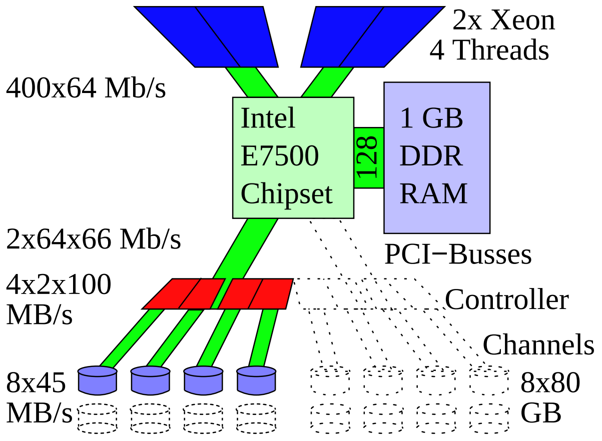


Figure 8.21: Setup for experiments on external MST

square are connected to their d closest neighbors. In order to obtain a simple family of planar graphs, we use *grid* graphs with random edge weights where the nodes are arranged in a grid and are connected to their (up to) four direct neighbors[10].

The experiments have been performed on a low cost PC-server (around 3000 Euro in July 2002) with two 2 GHz Intel Xeon processors, 1 GByte RAM and 4×80 GByte disks (IBM 120GXP) that are connected to the machine in a bottleneck-free way. This machine runs Linux 2.4.20 using the XFS file system. Swapping was disabled. All programs were compiled with `g++` version 3.2 and optimization level `-O6`. The total computer time spend for the experiments was about 25 days producing a total I/O volume of several dozen Terabytes.

Figure 8.22 summarizes the results for the bucket implementation. The curves only show the internal results for random graphs — at least Kruskal's algorithm shows very similar behavior for the other graph classes.

Our implementation can handle up to 20 million edges. Kruskal's algorithm is best for very sparse graphs ($m \leq 4n$) whereas the Jarník-Prim algorithm (with a fast implementation of pairing heaps) is fastest for denser graphs but requires more memory. For $n \leq 160\,000\,000$, we can run the semiexternal algorithm and get execution times within a factor of two of the internal algorithm.[11] The curves are almost flat and very similar for all three graph families. This is not astonishing since Kruskal's algorithm is not very dependent on the structure of the graph. Beyond 160 000 000 nodes, the full external al-

[10]Note that for planar graphs we can give a bound of $\mathcal{O}(sort(n))$ if we deal with parallel edges: When scanning the internal bucket for node i, the edges (i, v) are put into a hash table using v as a key. The corresponding table entry only keeps the lightest edge connecting i and v seen so far.

[11]Both the internal and the semiexternal algorithm have a number of possibilities for further tuning (e.g., using integer sorting or a better external sorter for small elements). But none of these measures is likely to yield more than a factor of 2.

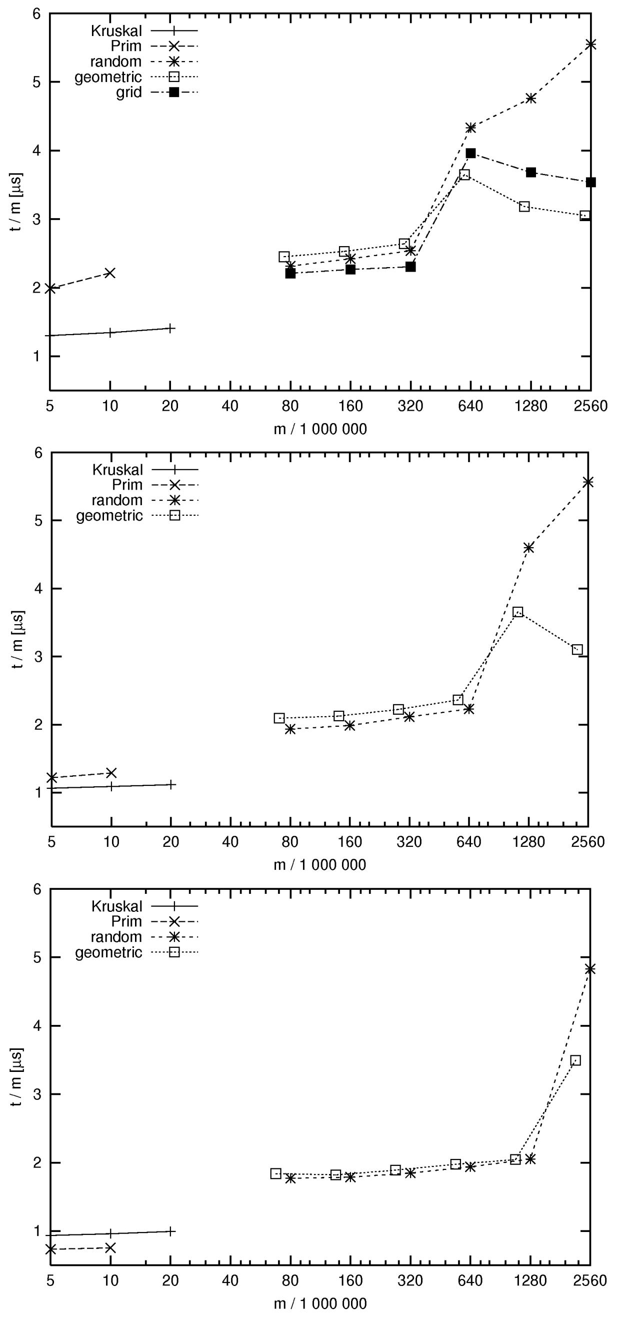


Figure 8.22: Execution time per edge for $m \approx 2 \cdot n$ (top),$m \approx 4 \cdot n$ (center), $m \approx 8 \cdot n$ (bottom). “Kruskal“ and “Prim“ denote the results of these internal algorithms on the “random“ input.

gorithm is needed. This immediately costs us another factor of two in execution time: We have additional costs for random renaming, node reduction, and a blowup of the size of an edge from 12 bytes to 20 bytes (for renamed nodes). For random graphs, the execution time keeps growing with n/M.

The behavior for grid graphs is much better than predicted. It is interesting that similar effects can be observed for geometric graphs. This is an indication that it is worth removing parallel edges for many nonplanar graphs.[12] Interestingly, the time per edge *decreases* with m for grid graphs and geometric graphs. The reason is that the time for the semiexternal base case does not increase proportionally to the number of input edges. For example, $5.6 \cdot 10^8$ edges of a grid graph with $640 \cdot 10^6$ nodes survive the node reduction, and $6.3 \cdot 10^8$ edges of a grid graph with twice the number of edges.

Another observation is that for $m = 2560 \cdot 10^6$ and random or geometric graphs we get the worst time per edge for $m \approx 4n$. For $m \approx 8n$, we do not need to run the node reduction very long. For $m \approx 2n$ we process less edges than predicted even for random graphs simply because one MST edge is removed for each node.

8.6 Connected Components

We modify and extend the bucket version of the bucket algorithm to get the connected components of an exernal graph. As in the spanning forest algorithm, the input is an unweighted graph represented as a list of edges. The output of the algorithm is a list of entries (v, c), $v \in V$, where c is the connected component id of node v, at the same time c is the id of a node belonging to the connected component. This special node c is sometimes called the representative node of a component. The algorithm makes two passes over adjacency lists of nodes (left-to-right pass $v = n-1..0$ and right-to-left $v = 0..n-1$, $v \in V$), relinking the edges such that they connect node v with the representative node of its connected component.

If there are $k = \mathcal{O}(M/B))$ external memory buckets then bucket $i \in \{0..k-1\}$ contains the adjacent edges (u, v), $u > v$ of nodes $u_{i-1} < u < u_i$, where u_i is the upper (lower) bound of node ids in bucket $i(i+1)$. Additionally, there are k question buckets and k answer buckets with the same bounds. A question is a tuple $(v, r(v))$ that represents the assignment of node v to a preliminary representative node $r(v)$. An answer is a tuple $(v, r(v))$ that represents the assignment of node v to an ultimate representative node. Function $b : V \rightarrow \{0..k-1\}$ maps a node id to the coresponding bucket id according to the bucket bounds. The bucket implementation is complemented with the following steps. During the processing of node v, the algorithm assigns $r(v)$ tentatively the id of its neighbor with the smallest id. If no neighbor exists then $r(v) := v$. After processing

[12]Very few parallel edges are generated for random graphs. Therefore, switching off duplicate removal gives about 13 % speedup for random graphs compared to the numbers given.

the bucket i we post the preliminary assignments $(v, r(v))$ of nodes v, $u_{i-1} < v$. u_i to question bucket $b(r(v))$ if $r(v)$ does not belong to bucket i. Otherwise we can update $r(v)$ with $r(r(v))$. If the new r(v) belongs to bucket i than it is the ultimate representative node of v and $(v, r(v))$ can be written to the answer bucket $b(v)$, otherwise we post question $(v, r(v))$ to the appropriate question bucket. Note that the first answer bucket is handled differently as it is implemented as the union-find data structure in the base case. For v in the union-data structure $r(v)$ is the id of the leader node of the union where v belongs to. The connected component algorithm needs an additional right-to-left scan to determine the ultimate representatives which have not been determined in the previous left-to-right scan. The buckets are read in the order $0..k-1$. For each $(v, r(v))$ in question bucket i we update $r(v)$ with the ultimate representative $r(r(v))$ looking up values in answer bucket i. The final value $(v, r(v))$ is appended to answer bucket $b(v)$. After answering all questions in bucket i, the content of answer bucket i is added to the output of the connected component algorithm. If one only needs to compute the component ids and no spanning tree edges then the implementation does not keep the original edge id in the edge data structure. It is sufficient to invert randomization for the node ids in the output, which can be done with the chosen randomization scheme without additional I/Os. Due to this measure the total I/O volume and the memory requirements of the internal buckets are reduced such that the block size of the external memory buckets can be made larger. All this leads to an overall performance improvement.

Chapter 9

String Sorting

This chapter is based on [39].

9.1 Introduction

The task is to sort a set $R = \{s_1, s_2, \ldots, s_n\}$ of n (non-empty) strings into the lexicographic order. N is the total length of strings, D the total length of *distinguishing prefixes*. The distinguishing prefix of a string s in R is the shortest prefix of s that is not a prefix of another string (or s if s is a prefix of another string). It is the shortest prefix of s that determines the rank of s in R. A sorting algorithm needs to access every character in the distinguishing prefixes, but no character outside the distinguishing prefixes.

We can evaluate algorithms using different alphabet models: In an ordered alphabet, only comparisons of characters are allowed. In an ordered alphabet of *constant* size, a multiset of characters can be sorted in linear time using counting sort. An integer alphabet is $\{1, \ldots, \sigma\}$ for integer $\sigma \geq 2$. Here, sorting a multiset of k characters can be done in $\mathcal{O}(k + \sigma)$ time with the same algorithm.
We have the following simple lower bounds for sorting using these models:

If we use a standard sorting algorithm for strings, the worst case requires $\Theta(n \log n)$ string comparisons. Let $s_i = \alpha\beta_i$, where $|\alpha| = |\beta_i| = \log n$. This means $D = \Theta(n \log n)$.

alignment
all
allocate
alphabet
alternate
alternative

Figure 9.1: Example on distinguishing prefixes.

alphabet	lower bound
ordered	$\Omega(D + n \log n)$
constant	$\Omega(D)$
integer	$\Omega(D)$

Table 9.1: Simple lower bounds for string sorting using different alphabet models

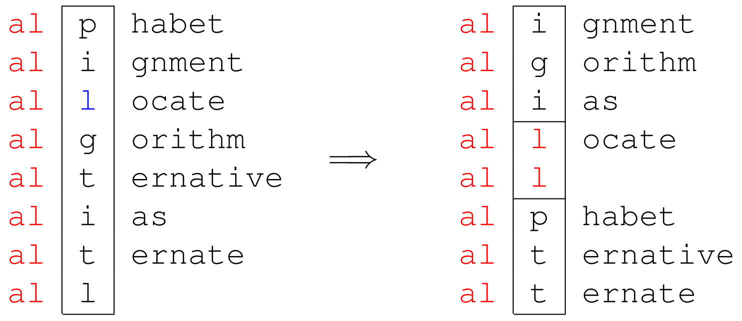


Figure 9.2: One partitioning step in multikey quicksort, with pivot ’l’ in ’allocate’

Our lower bound is $\Omega(D + n \log n) = \Omega(n \log n)$, but standard sorting has costs of $\Theta(n \log n) \cdot \Theta(\log n) = \Theta(n \log^2 n)$. In the next sections, we try to approach the lower bound for string sorting.

9.2 Multikey Quicksort

Multikey Quicksort [40] performs in every recursion level a ternary partitioning of the data elements. In contrast to the standard algorithm, the pivot is not a whole key (which would be a complete word), but only the first character following the common prefix shared by all elements.

We will now analyse the algorithm given in pseudo code in figure 9.3. The running time is dominated by the comparisons done in the partitioning step. We will use amortized analysis to count these comparisons. If $s[\ell + 1] \neq p[\ell + 1]$, we charge the comparison on s. Assuming a perfect choice for the pivot element, we see that the total charge on s for this type of comparison is $\leq \log n$, as the parition containing s is at least halved.
If we have $s[\ell + 1] = p[\ell + 1]$, we charge the comparison on $s[\ell + 1]$. After that, $s[\ell + 1]$ becomes part of the common prefix in its partition and will never again be chosen as pivot character. Therefore, the charge on $s[\ell + 1]$ is ≤ 1 and the total charge on all characters is $\leq D$. Combining this with the above result, we get a total runtime of $\mathcal{O}(D + n \log n)$.
The only flaw in the above analysis is the assumption of a perfect pivot. Like in the analysis of standard quicksort, we can show that the expected number of $\neq$ comparisons is $2n \ln n$ when using a random pivot character.

```
Function Multikey-quicksort($R$ : Sequence of String, $\ell$ : Integer) : Sequence of String
    // $\ell$ is the length of the common prefix in $R$
    if $|R| \leq 1$ then return $R$
    choose pivot $p \in R$
    $R_< := \{s \in R \mid s[\ell+1] < p[\ell+1]\}$
    $R_= := \{s \in R \mid s[\ell+1] = p[\ell+1]\}$
    $R_> := \{s \in R \mid s[\ell+1] > p[\ell+1]\}$
    Multikey-quicksort($R_<$, $\ell$)
    Multikey-quicksort($R_=$, $\ell+1$)
    Multikey-quicksort($R_>$, $\ell$)
    return concatenation of $R_<$, $R_=$, and $R_>$
```

Figure 9.3: Pseudocode for Multikey Quicksort

9.3 Radix Sort

Another classic string sorting algorithm is radix sort. There exist two main variants: LSD-first radix sort starts from the end of the strings (Least Significant Digit first) and moves backward by one position in each step, until the first character is reached. In every phase, it partitions all strings according to the character at the current position (one group for every possible character). When this is done, the strings are recollected, starting with the group corresponding to the "smallest" character. For correct sorting, this has to be done in a stable way within a group. The LSD variant accesses all characters (as we have to reach the first character of each word for correct sorting), which implies costs of $\Omega(N)$ time. This is poor when $D \ll N$.
MSD-first radix sort on the other hand starts from the beginning of the strings (Most Significant Digit first). It distributes the strings (using counting sort) to groups according to the character at the current position and sorts these groups recursively (increasing the position of the relevant character by 1). Then, all groups are concatenated, in the order of the corresponding characters[1]. This variant accesses only distinguishing prefixes.

What is the running time of MSD-first radix sort? Partitioning a group of k strings in σ buckets takes $\mathcal{O}(k + \sigma)$ time. As the total size of the partitioned groups is D, we have $\mathcal{O}(D)$ total time on constant alphabets.

The total number of times any string is assigned to a group is D (the total size of all groups created while sorting). For every non-trivial partitioning step (where not all

[1]When implementing this algorithm, many ideas used for Super Scalar Sample Sort (e.g. the two-pass-approach to determine the optimal bucket size) will also help for MSD-first radix sort. In fact, MSD-first radix sort inspired the development of SSSS

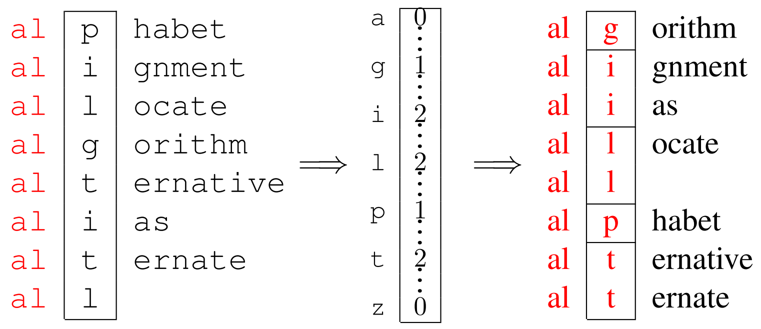


Figure 9.4: Example of one partitioning phase in MSD-first radix sort using counting sort for allocation

alphabet	lower bound	upper bound	algorithm
ordered	$\Omega(D + n \log n)$	$\mathcal{O}(()\,D + n \log n)$	multikey quicksort
constant	$\Omega(D)$	$\mathcal{O}(()\,D)$	radix sort
integer	$\Omega(D)$	$\mathcal{O}(()\,D + n \log \sigma)$	radix sort + multikey quicksort

Figure 9.5: Overview on upper and lower bounds using different alphabet models

characters are equal), additional costs of $\mathcal{O}(\sigma)$ for creating groups occur. Obviously, the number of non-trivial partitionings is $\leq n$. We therefore have costs of $\mathcal{O}(D + n\sigma)$, which becomes $\mathcal{O}(D)$ for constant alphabets. When dealing with integer alphabets, another improvement helps lowering the running time: When $k < \sigma$, where k is the number of strings to be partitioned in a certain step, switch to multikey quicksort. This results in a running time of $\mathcal{O}(D + n \log \sigma)$.

Table 9.5 gives an overview over the results of this chapter. Some gaps could be closed, others require more elaborated techniques beyond this text.

Chapter 10

Suffix Array Construction

The description of the DC3 algorithm was taken from [41]. Material on external suffix array construction is from [42].

10.1 Introduction

The suffix *tree* of a string is a compact trie of all the suffixes of the string. It is a powerful data structure with numerous applications in computational biology and elsewhere. One of the important properties of the suffix tree is that it can be constructed in linear time in the length of the string. The classical linear time algorithms require a constant alphabet size, but Farach's algorithm works also for integer alphabets, i.e., when characters are polynomially bounded integers.

The suffix *array* is a lexicographically sorted array of the suffixes of a string. For several applications, a suffix array is a simpler and more compact alternative to suffix trees. The suffix array can be constructed in linear time by a lexicographic traversal of the suffix tree, but such a construction loses some of the advantage that the suffix array has over the suffix tree. We introduce the *DC3 algorithm*, a linear-time direct suffix array construction algorithm for integer alphabets. The DC3 algorithm is simpler than any suffix tree construction algorithm. In particular, it is much simpler than linear time suffix tree construction for integer alphabets.

10.2 The DC3 Algorithm

The DC3 algorithm has a following structure:

1. Recursively construct the suffix array of the suffixes starting at positions $i \bmod 3 \neq 0$. This is done by reduction to the suffix array construction of a string of two thirds the length, which is solved recursively.

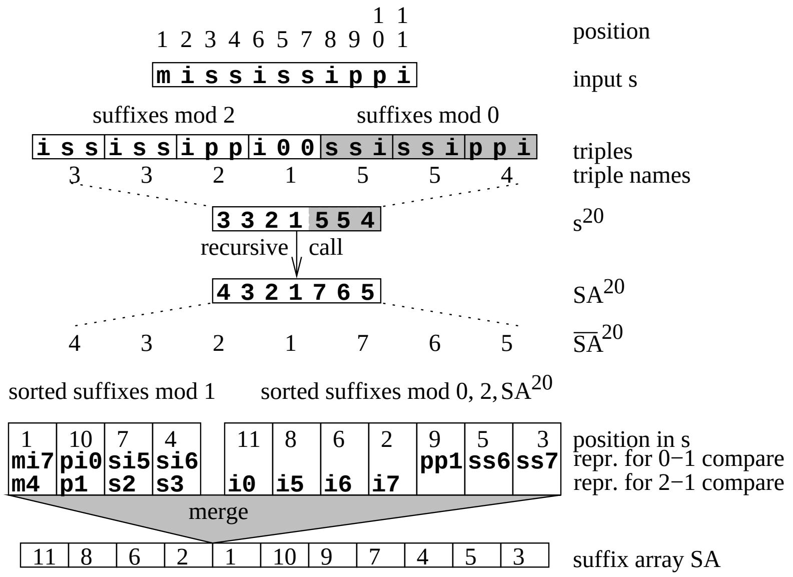


Figure 10.1: The DC3 algorithm applied to $s = \texttt{mississippi}$. First get all suffixes with index $\bmod 3 = 0, 2$. Group their characters to triples and map these meta-characters to an integer alphabet. Use the resulting string as input for a recursive call. The result contains at position i the index of the suffix with rank i. Sort the suffixes with index $\bmod 3 = 1$ (using the rank of the following suffix). Merge both results.

2. Construct the suffix array of the remaining suffixes using the result of the first step.

3. Merge the two suffix arrays into one.

If we would halve the string length for recursion, step three would be very difficult and costly. Surprisingly, the use of two thirds instead of half of the suffixes in the recursion makes the last step almost trivial: a simple comparison-based merging is sufficient. For example, to compare suffixes starting at i and j with $i \bmod 3 = 0$ and $j \bmod 3 = 1$, we first compare the initial characters, and if they are the same, we compare the suffixes starting at $i + 1$ and $j + 1$ whose relative order is already known from the first step.

Algorithm DC3

Input The input is a *string* $T = T[0, n) = t_0 t_1 \cdots t_{n-1}$ over the alphabet $[1, n]$, that is, a sequence of n integers from the range $[1, n]$. For convenience, we assume that $t_n = t_{n+1} = t_{n+2} = 0$.

The restriction to the alphabet $[1, n]$ is not a serious one. For a string T over any alphabet, we can first sort the characters of T, remove duplicates, assign a rank to each character, and construct a new string T' over the alphabet $[1, n]$ by renaming the characters of T with their ranks. Since the renaming is order preserving, the order of the suffixes does not change.

Output For $i \in [0, n]$, let S_i denote the *suffix* $T[i, n) = t_i t_{i+1} \cdots t_{n-1}$. We also extend the notation to sets: for $C \subseteq [0, n]$, $S_C = \{S_i \mid i \in C\}$. The goal is to sort the set $S_{[0,n]}$ of suffixes of T lexicographically. The output is the *suffix array* $\mathrm{SA}[0, n]$ of T, a permutation of $[0, n]$ defined by

$$\mathrm{SA}[i] = |\{j \in [0, n] \mid S_j < S_i\}| .$$

Step 0: Construct a sample For $k = 0, 1, 2$, define

$$B_k = \{i \in [0, n] \mid i \bmod 3 = k\}.$$

Let $C = B_1 \cup B_2$ be the set of *sample positions* and S_C the set of *sample suffixes*.

Step 1: Sort sample suffixes For $k = 1, 2$, construct the strings

$$R_k = [t_k t_{k+1} t_{k+2}][t_{k+3} t_{k+4} t_{k+5}] \ldots [t_{\max B_k} t_{\max B_k + 1} t_{\max B_k + 2}]$$

whose characters are triples $[t_i t_{i+1} t_{i+2}]$. Note that the last character of R_k is always unique because $t_{\max B_k + 2} = 0$. Let $R = R_1 \odot R_2$ be the concatenation of R_1 and R_2. Then the (nonempty) suffixes of R correspond to the set S_C of sample suffixes: $[t_i t_{i+1} t_{i+2}][t_{i+4} t_{i+5} t_{i+6}] \ldots$ corresponds to S_i. The correspondence is order preserving, i.e., by sorting the suffixes of R we get the order of the sample suffixes S_C.

To sort the suffixes of R, first radix sort the characters of R. If all characters are different, the order of characters gives directly the order of suffixes. Otherwise, we use the technique of renaming the characters with their ranks, and then sort the suffixes of the resulting string using Algorithm DC3.

Once the sample suffixes are sorted, assign a rank to each suffix. For $i \in C$, let $\mathrm{rank}(S_i)$ denote the rank of S_i in the sample set S_C. Additionally, define $\mathrm{rank}(S_{n+1}) = \mathrm{rank}(S_{n+2}) = 0$. For $i \in B_0$, $\mathrm{rank}(S_i)$ is undefined.

Step 2: Sort nonsample suffixes Represent each nonsample suffix $S_i \in S_{B_0}$ with the pair $(t_i, rank(S_{i+1}))$. Note that $rank(S_{i+1})$ is always defined for $i \in B_0$. Clearly we have, for all $i, j \in B_0$,

$$S_i \le S_j \iff (t_i, rank(S_{i+1})) \le (t_j, rank(S_{j+1})).$$

The pairs $(t_i, rank(S_{i+1}))$ are then radix sorted.

Step 3: Merge The two sorted sets of suffixes are merged using a standard comparison-based merging. To compare suffix $S_i \in S_C$ with $S_j \in S_{B_0}$, we distinguish two cases:

$$\begin{array}{ll} i \in B_1: & S_i \leq S_j \iff (t_i, \mathrm{rank}(S_{i+1})) \leq (t_j, \mathrm{rank}(S_{j+1})) \\ i \in B_2: & S_i \leq S_j \iff (t_i, t_{i+1}, \mathrm{rank}(S_{i+2})) \leq (t_j, t_{j+1}, \mathrm{rank}(S_{j+2})) \end{array}$$

Note that the ranks are defined in all cases.

Theorem 14 *The time complexity of Algorithm DC3 is $\mathcal{O}(n)$.*

Proof: Excluding the recursive call everything can clearly be done in linear time. The recursion is on a string of length $\lceil 2n/3 \rceil$. Thus the time is given by the recurrence $T(n) = T(2n/3) + \mathcal{O}(n)$, whose solution is $\mathcal{O}(n)$. ▮

10.3 External Suffix Array Construction

In this section we are trying to engineer algorithms for suffix array construction that work on huge inputs using the external memory model.

The Doubling Algorithm

Figure 10.2 gives pseudocode for the doubling algorithm. The basic idea is to replace characters $T[i]$ of the input by *lexicographic names* that respect the lexicographic order of the length 2^k substring $T[i, i + 2^k)$ in the k-th iteration. In contrast to previous variants of this algorithm, our formulation never actually builds the resulting string of names. Rather, it manipulates a sequence P of pairs (c, i) where each name c is tagged with its position i in the input. To obtain names for the next iteration $k + 1$, the names for $T[i, i + 2^k)$ and $T[i + 2^k, i + 2^{k+1})$ together with the position i are stored in a sequence S and sorted. The new names can now be obtained by scanning this sequence and comparing adjacent tuples. Sequence S can be build using consecutive elements of P if we sort P using the pair $(i \ \mathbf{mod}\ 2^k, i\ \mathbf{div}\ 2^k)$. Previous formulations of the algorithm use i as a sorting criterion and therefore have to access elements that are 2^k characters apart. Our approach saves I/Os and simplifies the pipelining optimization described in Section 10.3.

The algorithm performs a constant number of sorting and scanning operations for sequences of size n in each iteration. The number of iterations is determined by the logarithm of the longest common prefix.

Theorem 15 *The doubling algorithm computes a suffix array using $\mathcal{O}(\mathrm{sort}(n) \lceil \log \mathrm{maxlcp} \rceil)$ I/Os where* $\mathrm{maxlcp} := \max_{0 \leq i < n} \mathrm{lcp}(i, i + 1)$.

```
**Function** $doubling(T)$
    $S$:= $\langle((T[i], T[i+1]), i) : i \in [0, n)\rangle$                         1
    **for** $k$ := 1 **to** $\lceil \log n \rceil$ **do**
        *sort* $S$                                                                   2
        $P$:= $name(S)$                                                              3
        **invariant** $\forall (c, i) \in P$ :
            $c$ *is a lexicographic name for* $T[i, i + 2^k)$
        **if** *the names in* $P$ *are unique* **then**
            **return** $\langle i : (c, i) \in P\rangle$                            4
        *sort* $P$ *by* $(i \bmod 2^k, i \operatorname{div} 2^k))$                   5
        $S$:= $\langle((c, c'), i) : j \in [0, n),$                                  6
                $(c, i) = P[j], (c', i + 2^k) = P[j + 1]\rangle$

**Function** $name(S : Sequence$ **of** $Pair)$
    $q$:= $r$:= 0;    $(\ell, \ell')$:= ($, $)
    $result$:= $\langle\rangle$
    **foreach** $((c, c'), i) \in S$ **do**
        $q$++
        **if** $(c, c') \neq (\ell, \ell')$ **then** $r$:= $q$;    $(\ell, \ell')$:= $(c, c')$
        *append* $(r, i)$ *to* $result$
    **return** $result$
```

Figure 10.2: The doubling algorithm.

Pipelining

The I/O volume of the doubling algorithm from Figure 10.2 can be reduced significantly by observing that rather than writing the sequence S to external memory, we can directly feed it to the sorter in Line (1). Similarly, the sorted tuples need not be written but can be directly fed into the naming procedure in Line (2) which can in turn forward it to the sorter in Line (4). The result of this sorting operation need not be written but can directly yield tuples of S that can be fed into the next iteration of the doubling algorithm.

To motivate the idea of pipelining let us first analyze the constant factor in a naive implementation of the doubling algorithm from Figure 10.2. For simplicity assume for now that inputs are not too large so that sorting m words can be done in $4m/DB$ I/Os using two passes over the data. For example, one run formation phase could build sorted runs of size M and one multiway merging phase could merge the runs into a single sorted sequence.

Line (1) sorts n triples and hence needs $12n/DB$ I/Os. Naming in Line (2) scans the triples and writes name-index pairs using $3n/DB + 2n/DB = 5n/DB$ I/Os. The naming procedure can also determine whether all names are unique now, hence the test in Line (3) needs no I/Os. Sorting the pairs in P in Line (4) costs $8n/DB$ I/Os. Scanning the pairs and producing triples in Line (5) costs another $5n/DB$ I/Os. Overall, we get $(12 + 5 + 8 + 5)n/DB = 30n/DB$ I/Os for each iteration.

This can be radically reduced by interpreting the sequences S and P not as files but as pipelines similar to the pipes available in UNIX. In the beginning we explicitly scan the input T and produce triples for S. We do not count these I/Os since they are not needed for the subsequent iterations. The triples are not output directly but immediately fed into the run formation phase of the sorting operation in Line (1). The runs are output to disk ($3n/DB$ I/Os). The multiway merging phase reads the runs ($3n/DB$ I/Os) and directly feeds the sorted triples into the naming procedure called in Line (2) which generates pairs that are immediately fed into the run formation process of the next sorting operation in Line (3) ($2n/DB$ I/Os). The multiway merging phase ($2n/DB$ I/Os) for Line (3) does not write the sorted pairs but in Line (4) it generates triples for S that are fed into the pipeline for the next iteration. We have eliminated all the I/Os for scanning and half of the I/Os for sorting resulting in only $10n/DB$ I/Os per iteration — only one third of the I/Os needed for the naive implementation.

Note that pipelining would have been more complicated in the more traditional formulation where Line (3) sorts P directly by the index i. In that case, a pipelining formulation would require a FIFO of size 2^k to produce shifted sequences. When $2^k > M$ this FIFO would have to be maintained externally causing $2n/DB$ additional I/Os per iteration, i.e., our modification simplifies the algorithm and saves up to 20 % I/Os.

Let us discuss a more systematic model: The computations in many external memory algorithms can be viewed as a data flow through a directed acyclic graph $G = (V = F \cup S \cup R, E)$. The *file nodes* F represent data that has to be stored physically on disk. When a file node $f \in F$ is accessed we need a buffer of size $b(f) = \Omega(BD)$. The *streaming nodes* $s \in S$ read zero, one or several sequences and output zero, one or several new sequences using internal buffers of size $b(s)$.[1] The *Sorting nodes* $r \in R$ read a sequence and output it in sorted order. Sorting nodes have a buffer requirement of $b(r) = \Theta(M)$ and outdegree one[2]. Edges are labeled with the number of machine words $w(e)$ flowing between two nodes.

Theorem 16 *The doubling algorithm from Figure 10.2 can be implemented to run using* $\mathrm{sort}(5n) \lceil \log(1 + \mathrm{maxlcp}) \rceil + \mathcal{O}(\mathrm{sort}(n))$ *I/Os.*

Proof: The following flow graph shows that each iteration can be implemented using

[1]Streaming nodes may cause additional I/Os for internal processing, e.g., for large FIFO queues or priority queues. These I/Os are not counted in our analysis.

[2]We could allow additional outgoing edges at an I/O cost n/DB. However, this would mean to perform the last phase of the sorting algorithm several times.

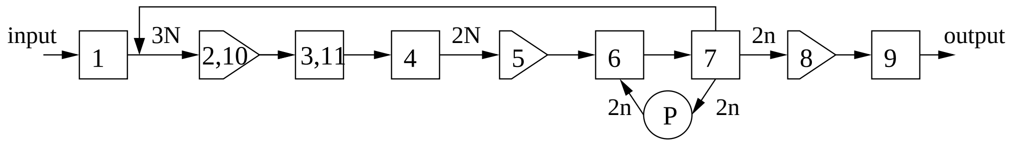
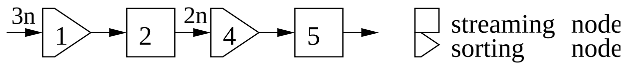


Figure 10.3: Data flow graph for the $doubling + discarding$. The numbers refer to line numbers in Figure 12.4. The edge weights are sums over the whole execution with $N = n \log \mathrm{dps}$.

$\mathrm{sort}(2n) + \mathrm{sort}(3n) \leq \mathrm{sort}(5n)$ I/Os. The numbers refer to the line numbers in Figure 10.2



After $\lceil \log(1 + \mathrm{maxlcp}) \rceil$ iterations, the algorithm finishes. The $\mathcal{O}(\mathrm{sort}(n))$ term accounts for the I/Os needed in Line 0 and for computing the final result. Note that there is a small technicality here: Although naming can find out "for free" whether all names are unique, the result is known only when naming finishes. However, at this time, the first phase of the sorting step in Line 4 has also finished and has already incurred some I/Os. Moreover, the convenient arrangement of the pairs in P is destroyed now. However we can then abort the sorting process, undo the wrong sorting, and compute the correct output. ∎

Discarding

Let c_i^k be the lexicographic name of $T[i, i + 2^k)$, i.e., the value paired with i at iteration k in Figure 10.2. Since c_i^k is the number of strictly smaller substrings of length 2^k, it is a non-decreasing function of k. More precisely, $c_i^{k+1} - c_i^k$ is the number of positions j such that $c_j^k = c_i^k$ but $c_{j+2^k}^k < c_{i+2^k}^k$. This provides an alternative way of computing the names given in Figure 12.3.

Another consequence of the above observation is that if c_i^k is unique, i.e., $c_j^k \neq c_i^k$ for all $j \neq i$, then $c_i^h = c_i^k$ for all $h > k$. The idea of the discarding algorithm is to take advantage of this, i.e., discard pair (c, i) from further iterations once c is unique. A key to this is the new naming procedure in Figure 12.3, because it works correctly even if we exclude from S all tuples $((c, c'), i)$, where c is unique. Note, however, that we cannot exclude $((c, c'), i)$ if c' is unique but c is not. Therefore, we will *partially* discard (c, i) when c is unique. We will *fully* discard $(c, i) = (c_i^k, i)$ when also either $c_{i-2^k}^k$ or $c_{i-2^{k+1}}^k$ is unique, because then in any iteration $h > k$, the first component of the tuple $((c_{i-2^h}^h, c_i^h), i - 2^h)$ must be unique. The final algorithm is given in Figure 12.4.

Theorem 17 *Doubling with discarding can be implemented to run using* $\mathrm{sort}(5n \log \mathrm{dps}) + \mathcal{O}(\mathrm{sort}(n))$ *I/Os.*

The proof for theorem 17 and pseudocode can be found in 12.7 and 12.8 in the appendix.

Chapter 11

Presenting Data from Experiments

11.1 Introduction

A paper in experimental algorithmics will often start by describing the problem and the experimental setup. Then a substantial part will be devoted to presenting the results together with their interpretation. Consequently, compiling the measured data into graphs is a central part of writing such a paper. This problem is often rather difficult because several competing factors are involved. First, the measurements can depend on many parameters: problem size and other quantities describing the problem instance; variables like number of processors, available memory describing the machine configuration used; and the algorithm variant together with tuning parameters such as the cooling rate in a simulated annealing algorithm.

Furthermore, many quantities can be measured such as solution quality, execution time, memory consumption and other more abstract complexity measures such as the number of comparisons performed by a sorting algorithm. Mathematically speaking, we sample function values of a mapping $f : A \to B$ where the domain A can be high-dimensional. We hope to uncover properties of f from the measurements, e.g., an estimate of the time complexity of an algorithm as a function of the input size. Measurement errors may additionally complicate this task.

As a consequence of the the multitude of parameters, a meaningful experimental setup will often produce large amounts of data and still cover only a tiny fraction of the possible measurements. This data has to be presented in a way that clearly demonstrates the observed properties. The most important presentation usually takes place in conference proceedings or scientific journals where limited space and format restriction further complicate the task.

This paper collects rules that have proven to be useful in designing good graphs. Although the examples are drawn from the work of the author, this paper owes a lot to discussions with colleagues and detailed feedback from several referees. Sections 11.3–

11.7 explains the rules. The stress is on Section 11.4 where two-dimensional figures are discussed in detail.

Instead of an abstract conclusion, Section 11.8 collects all the rules in a check list that can possibly be used when looking for teaching and as a source of ideas for improving graphs.

11.2 The Process

In a simplified model of experimental algorithmics a paper might be written using a "waterfall model". The experimental design is followed by a description of the measurement which is in turn followed by an interpretation. In reality, there are numerous feedbacks involved and some might even remain visible in a presentation. After an algorithm has been implemented, one typically builds a simple yet flexible tool that allows many kinds of measurements. After some explorative measurements the researcher gets a basic idea of interesting parameter settings. Hypotheses are formed which are tested using more extensive measurements using particular parameter ranges. This phase is the scientifically most productive phase and often leads to new insights which lead to algorithmic changes which influence the entire setup.

It should be noted that most algorithmic problems are so complex that one cannot expect to arrive at an ultimate set of measurements that answers all conceivable questions. Rather, one is constantly facing a list of interesting open questions that require new measurements. The process of selecting the measurements that are actually performed is driven by risk and opportunity: The researcher will usually have a set of hypotheses that have some support from measurements but more measurements might be important to confirm them. For example, the hypothesis might be "my algorithm is better than all the others" then a big risk might be that a promising other algorithm or important classes of problem instances have not been tried yet. A small risk might be that a tuning parameter has so far been set in an ad hoc fashion where it is clear that it can only improve a precomputation phase that takes 20 % of the execution time.

An opportunity might be a new idea of the authors' that an algorithm might be useful for a new application where it was not originally designed for. In that case, one might consider to include problem instances from the new application into the measurements.

At some point, a group of researchers decides to cast the current state of results into a paper. The explorative phase is then stopped for a while. To make the presentation concise and convincing, alternative ways to display the data are designed that are compact enough to meet space restrictions and make the conclusions evident. This might also require additional measurements giving additional support to the hypotheses studied.

11.3 Tables

Tables are easier to produce than graphs and perhaps this advantage causes that they are often overused. Tables are more difficult to interpret and too large for large data sets. A more detailed explanation why tables are often a bad idea has been given by McGeoch and Moret [51]. Nevertheless, tables have their place. Tufte [59] gives the rule of thumb that "tables usually outperform a graph for small data sets of 20 numbers or less". Tables give very accurate values which make it easier to check whether some experiments can be reproduced. Furthermore, one sometimes wants to present some quantities, e.g., solution quality, as a function of problem instances which cannot be meaningfully arranged on the axis of a graph. In that case, a graph or bar chart may look nicer but does not add utility compared to a more accurate and compact table. Often a paper will contain small tables with particularly important results and graphs giving results in an abstract yet less accurate way. Furthermore, there may be an appendix or a link to a web page containing larger tables for more detailed documentation of the results.

11.4 Two-dimensional Figures

As our standard example we will use the case that execution time should be displayed as a function of input size. The same rules will usually apply for many other types of variables. Sometimes we mention special examples which should be displayed differently.

The x-Axis

The first question one can ask oneself is what unit one chooses for the x-axis. For example, assume we want to display the time it takes to broadcast a message of length k in some network where transmitting k' bytes of data from one processor to another takes time $t_0 + k'$. Then it makes sense to plot the execution time as a function of k/t_0 because for many implementations, the shape of the curve will then become independent of t_0. More generally, by choosing an appropriate unit, we can sometimes get rid of one degree of freedom. Figure 11.1 gives an example.

The variable defining the x-axis can often vary over many orders of magnitude. Therefore one should always consider whether a logarithmic scale is appropriate for the x-axis. This is an accepted way to give a general idea of a function over a wide range of values. One will then choose measurement values such that they are about evenly spaced on the x-axis, e.g., powers of two or powers of $\sqrt{2}$. Figures 11.3, 11.5, and 11.6 all use powers of two. In this case, one should also choose tic marks which are powers of two and not powers of ten. Figures 11.1 and 11.4 use the "default" base ten because there is no choice of input sizes involved here.

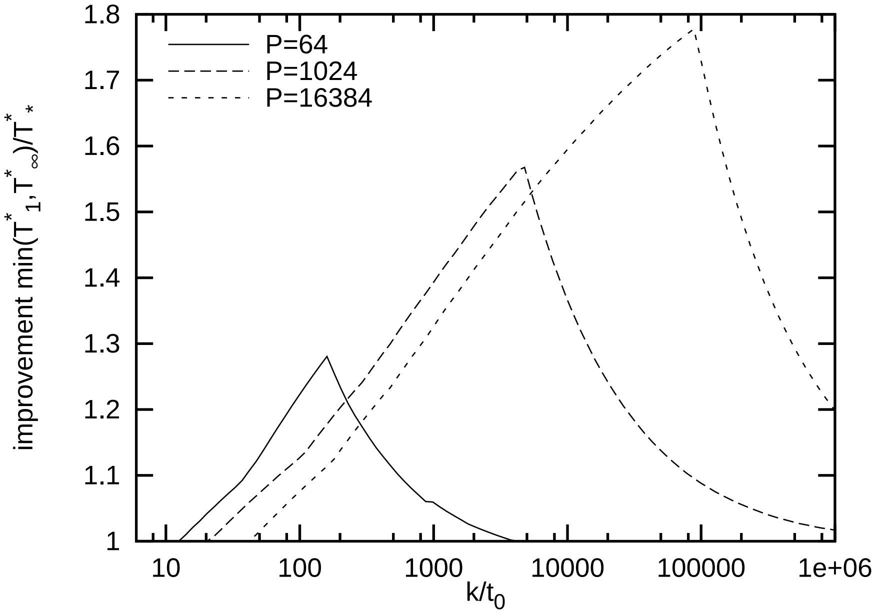


Figure 11.1: Improvement of the fractional tree broadcasting algorithm [57] over the best of pipelined binary tree and sequential pipeline algorithm as a function of message transmission time k over startup overhead t_0. P is the number of processors. (See also Section 11.4 and 11.4)

Sometimes it is appropriate to give more measurements for small x-values because they are easily obtained and particularly important. Conversely, it is not a good idea to measures using constant offsets ($x \in \{x_0 + i\Delta : 0 \leq i < i_{\max}\}$) as if one had a linear scale and then to display the values on a logarithmic scale. This looks awkward because points are crowded for large values. Often there will be too few values for small x and one nevertheless wastes a lot of measurement time for large inputs.

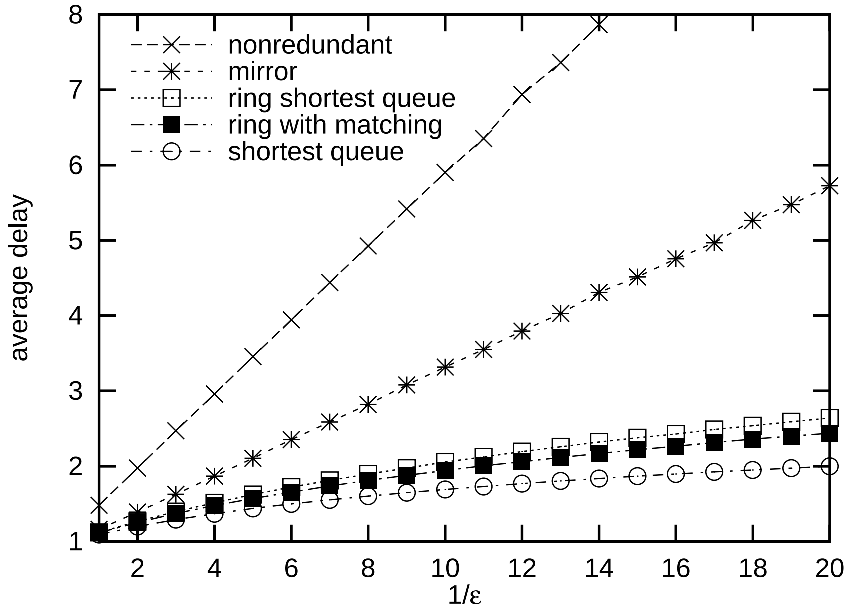
A plain linear scale is adequate if the interesting range of x-values is relatively small, for example if the x-axis is the number of processors used and one measures on a small machine with only 8 processors. A linear scale is also good if one wants to point out periodic behavior, for example if one wants to demonstrate that slow-down due to cache conflicts get very large whenever the input size is a multiple of the cache size. However, one should resist the temptation to use a linear scale when x-values over many orders of magnitude are important but the own results look particularly good for large inputs.

Sometimes, transformations of the x-axis other than linear or logarithmic make sense. For example, in queuing systems one is often interested in the delay of requests as the system load approaches the maximum performance of the system. Figure 11.2 gives an example. Assume we have a disk server with 64 disks. Data is placed randomly on these disks using a hash function. Assume that retrieving a block from a disk takes one time unit and that there is a periodic stream of requests — one every $(1 + \epsilon)/64$ time units. Using queuing theory one can show that the delay of a request is approximately proportional to $1/\epsilon$ if only one copy of every block is available. Therefore, it makes sense to use $1/\epsilon$ as the x-value. First, this transformation makes it easy to check whether the system measured also shows this behavior linear in $1/\epsilon$. Second, one gets high resolution for arrival rates near the saturation point of the system. Such high arrival rates are often more interesting than low arrival rates because they correspond to very efficient uses of the system.

The y-Axis

Given that the x-axis often has a logarithmic scale, we often seem to be forced to use a logarithmic scale also for the y-axis. For example, if the execution time is approximately some power of the problem size, such a double-logarithmic plot will yield a straight line.

However, plots of the execution time can be quite boring. Often, we already know the general shape of the curve. For example, a theoretical analysis may tell us that the execution time is between $T(n) = \Omega(n)$ and $T(n) = \mathcal{O}(n\text{Polylog}(n))$. A double-logarithmic plot will show something very close to a diagonal and discerns very little about the Polylog term we are really interested in. In such a situation, we transform the y-axis so that a priori information is factored out. In our example above we could better display $T(n)/n$ and then use a linear scale for the y-axis. A disadvantage of such transformations is that they may be difficult to explain. However, often this problem can be solved by finding a good term describing the quantity displayed. For example, "time per element" when



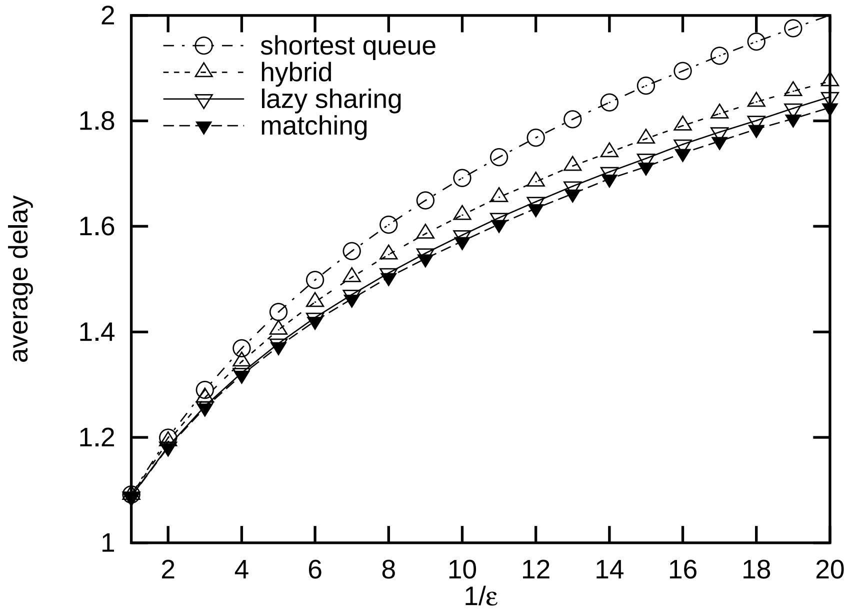


Figure 11.2: Comparison of eight algorithms for scheduling accesses to parallel disks using the model described in the text (note that "shortest queue" appears in both figures). Only the two algorithms "nonredundant" and "mirror" exhibit a linear behavior of the access delay predicted by queuing theory. The four best algorithms are based on *random duplicate allocation* — every block is available on two randomly chosen disks and a scheduling algorithm [55] decides which copy to retrieve. (See also Section 11.4)

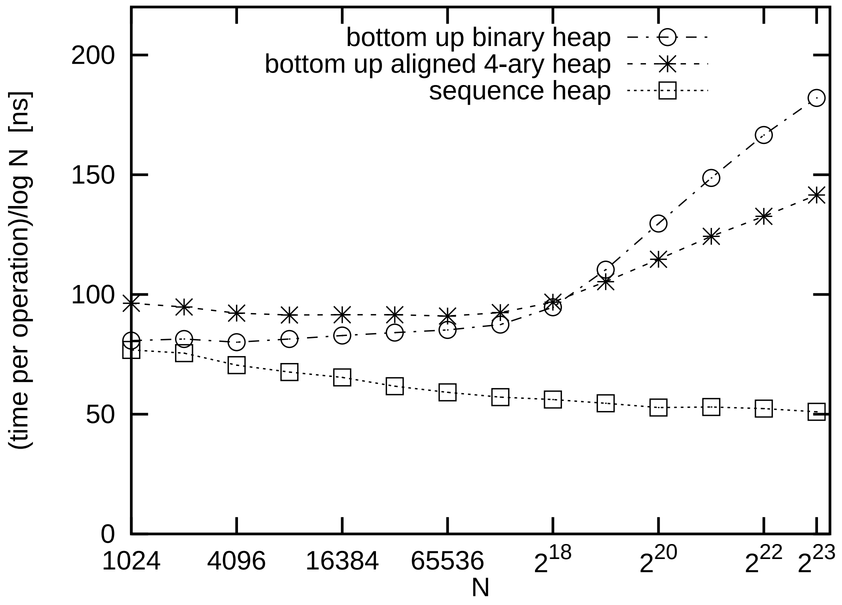


Figure 11.3: Comparison of three different priority queue algorithms [58] on a MIPS R10000 processor. N is the size of the queue. All algorithms use $\Theta(\log N)$ key comparisons per operation. The y-axis shows the total execution time for some particular operation sequence divided by the number of deletion/insertion pairs and $\log N$. Hence the plotted value is proportional to the execution time per key comparison. This scaling was chosen to expose cache effects which are now the main source of variation in the y-value. (See also Sections 11.4 and 11.4.)

one divides by the input size, “competitive ratio” when one divides by a lower bound, or “efficiency” when one displays the ratio between an upper performance bound and the measured performance. Figure 11.3 gives an example for using such a ratio.

Another consideration is the range of y-values displayed. Assume $y_{\min} > 0$ is the minimal value observed and $y_{\max}$ is the maximal value observed. Then one will usually choose $[y_{\min}, y_{\max}]$ or (better) a somewhat larger interval as the displayed range. In this case, one should be careful however with overinterpreting the resulting picture. A change of the y-value by 1 % will look equal to a change of y-value of 400 %. If one wants to support claims such as “for large x the improvements due to the new algorithm become very large” using a graph, choosing the range $[0, y_{\max}]$ can be a more sound choice. (At least if $y_{\max}/y_{\min}$ is not too close to one. Some of the space “wasted” this way can often be used for placing curve labels.) In Figure 11.2, using $y_{\min} = 1$ is appropriate since no request can get an access delay below one in the model used.

The choice of the the maximum y value displayed can also be nontrivial. In particular, it may be appropriate to clip extreme values if they correspond to measurement points which are clearly useless in practice. For example, in Figure 11.2 it is not very interesting to see the entire curve for the algorithm "nonredundant" since it is clearly outclassed for large $1/\epsilon$ anyway and since we have a good theoretical understanding of this particular curve.

A further degree of freedom is the vertical size of the graph. This parameter can be used to achieve the above goals and the rule of "banking to $45°$": The weighted average of the slants of the line segments in the figure should be about $45°$.[1] Refer to [47] for a detailed discussion. The weight of a segment is the x-interval bridged. There is good empirical and mathematical evidence that graphs using this rule make changes in slope most easily visible.

If banking to $45°$ does not yield a clear insight regarding the graph size, a good rule of thumb is to make the graph a bit wider than high [59]. A traditional choice is to use the golden ratio, i.e., a graph that is 1.62 times wider than high.

Arranging Multiple Curves

An important feature of two-dimensional graphs is that we can place several curves in a single graph as in Figures 11.1, 11.2, and 11.3. In this way we can obtain a high information density without the disadvantages of three-dimensional plots. However, one can easily overdo it resulting in a chaos of undecipherable points and lines. How many curves fit into one pictures depends on the information density. When curves are very smooth, and have few points where they cross each other, as in Figure 11.2, up to seven curves may fit in one figure. If curves are very complicated, even three curves may be too much. Often one will start with a straight-forward graph that turns out to be too ugly for publication. Then one can use a number of techniques to improve it:

- Remove unnecessary curves. For example, Figure 11.2 from [55] compares only eight algorithms out of eleven studied in this paper. The remaining three are clearly outclassed or equivalent to other algorithms for the measurement considered.
- If several curves are too close together in an important range of x-values, consider using another y range or scale. If the small differences persist and are important, consider to use a separate graph with a magnification. For example, in Figure 11.2 the four fastest algorithms were put into a separate plot to show the differences between them.
- Check whether several curves can be combined into one curve. For example, assume we want to compare a new improved algorithm with several inferior old algorithms for input sizes on the x-axis. Then it might be sufficient to plot the speedup

[1]This is one of the few things described here which are are not easy to do with gnuplot. But even keeping the principle of banking to $45°$ in mind is helpful.

of the new algorithm over the best of the old algorithms; perhaps labeling the sections of the speedup curve so that the best of the old algorithms can be identified for all x-values. Figure 11.1 gives an example where the speeup of one algorithm over two other algorithms is shown.

- Decrease noise in the data as described in Section 11.4.
- Once noise is small, replace error bars with specifications of the accuracy in the caption as in Figure 11.6.
- Connect points belonging to the same curves using straight lines.
- Choose different point styles and line styles for different curves.
- Arrange labels explaining point and line styles in the "same order"[2] as they appear in the graph. Sometimes one can also place the labels directly at the curves. But even then the labels should not obscure the curves. Unfortunately, gnuplot does not have this feature so that we could not use it in this paper.
- Choose the x-range and the density of x-values appropriately.

Sometimes we need so many curves that they cannot fit into one figure. For example, when the cross-product of several parameter ranges defines the set of curves needed. Then we may finally decide to use several figures. In this case, the same y-ranges should usually be chosen so that the results remain comparable. Also one should choose the same point styles and line styles for related curves in different figures, e.g., for curves belonging to the same algorithm as for the "shortest queue" algorithm in Figure 11.2. Note that tools such as gnuplot cannot do that automatically.

The explanations of point and line styles should avoid cryptic abbreviations whenever possible and at the same time avoid overlapping the curves. Both requirements can be reconciled by placing the explanations appropriately. For example, in computer science, curves often go from the lower left corner to the upper right corner. In that case, the best place for the definition of line and point styles is the upper left corner.

Arranging Instances

If measurements like execution time for a small set of problem instances are to be displayed, a bar chart is an appropriate tool. If other parameters such as the algorithm used, or the time consumed by different parts of the algorithm should be differentiated, the bars can be augmented to encode this. For example, several bars can be stacked in depth using three-dimensional effects or different pieces of a bar can get different shadings.[3]

[2]For example, one could use the order of the y-values at the larges x-value as in Figure 11.3.

[3]Sophisticated fill styles give us additional opportunities for diversification but Tufte notes that they are often too distracting [59].

If there are so many instances that bar charts consume too much space, a *scatter plot* can be useful. The x-axis stands for a parameter like problem size and we plot one point for every problem instance. Figure 11.4 gives a simple example. Point styles and colors can be used to differentiate different types of instances or variations of other parameters such as the algorithm used. Sometimes these points are falsely connected by lines. This should be avoided. It not only looks confusing but also wrongly suggests a relation between the data points that does not exist.

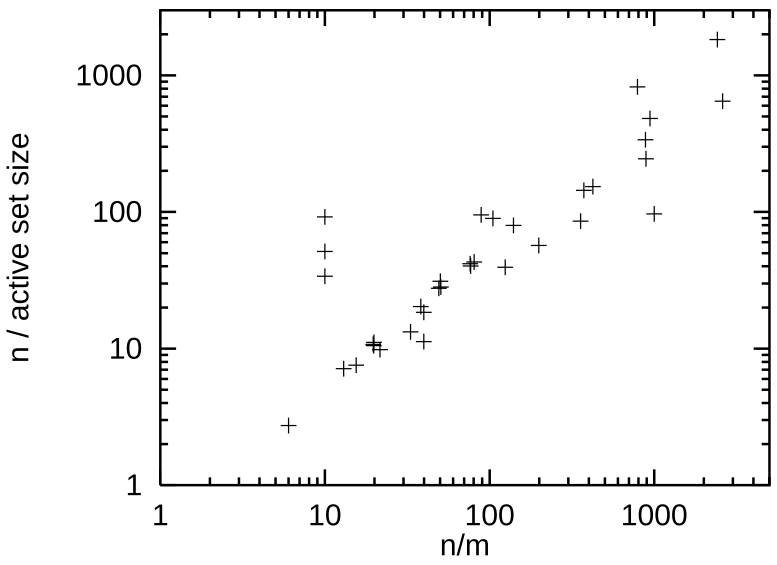


Figure 11.4: Each point gives the ratio between total problem size and "core" problem size in a fast algorithm for solving set covering problems from air line crew scheduling [43]. The larger this ratio, the larger the possible speedup for a new algorithm. The x-axis is the ratio between the number of variables and number of constraints. This scale was chosen to show that there is a correlation between these two ratios that is helpful in understanding when the new algorithm is particularly useful. The deviating points at $n/m = 10$ are artificial problems rather different from typical crew scheduling problems. (See also Section 11.4.)

How to Connect Measurements

Tools such as gnuplot allow us to associate a measured value with a symbol like a cross or a star that clearly specifies that point and encodes some additional information about the measurement. For example, one will usually choose one point symbol for each displayed curve. Additionally, points belonging to the same curve can be connected by a straight line. Such lines should usually not be viewed as a claim that they present a good interpolation of the curve but just as a visual aid to find points that belong together. In this case, it is important that the points are large enough to stand out against the connecting lines. An alternative is to plot measurements points plus curves stemming from an analytic model as in Figure 11.5.

The situation is different if only lines and no points are plotted as in Figure 11.1. In this case, it is often impossible to tell which points have been measured. Hence such a lines-only plot implies the very strong claim that the points where we measured are irrelevant and the plotted curve is an accurate representation of the true behavior for the entire x-range. This only makes sense if very dense measurements have been performed and they indeed form a smooth line. Sometimes one sees smooth lines that are weighted averages over a neighborhood in the x-coordinates. Then one often uses very small points for the actual measurements that form a cloud around this curve.

A related approach is connecting measured points with interpolated curves such as splines which are more smooth than lines. Such curves should only be used if we actually conjecture that the interpolation used is close to the truth.

Measurement Errors

Tools allow us to generalize measured points to ranges which are usually a point plus an error bar specifying positive and negative deviations from the y-value.[4] The main question from the point of view of designing graphs is what kind of deviations should be displayed or how one can avoid the necessity for error bars entirely.

Let us start with the well behaved case that we are simulating a randomized algorithm or work with randomly generated problem instances. In this case, the results from repeated runs are independent identically distributed random variables. In this case, powerful methods from statistics can be invoked. For example, the point itself may be the average of the measured values and the error bar could be the standard deviation or the standard error [53]. Figure 11.5 gives an example. Note that the latter less well known quantity is a better estimate for the difference between the average and the actual mean. By monitoring the standard error during the simulation, we can even repeat the measurement sufficiently often so that this error measure is below some prespecified value. In this case, no error bars are needed and it suffices to state the bound on the error in the caption of the graph. Figure 11.6 gives an example.

The situation is more complicated for measurements of actual running times of deterministic algorithms, since this involves errors which are not of a statistical nature. Rather, the errors can stem from hidden variables such as operating system interrupts, which we cannot fully control. In this case, points and error bars based on order statistics might be more robust. For example, the y value could be the median of the measured values and the error bar could define the minimum and the maximum value measured or values

[4]Uncertainties in both x and y-values can also be specified but this case seems to be rare in Algorithmics.

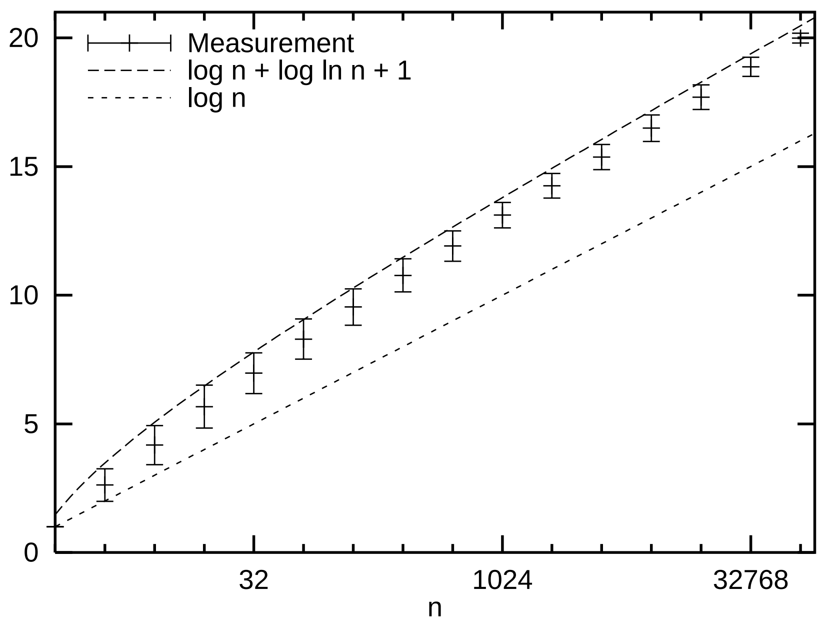


Figure 11.5: Number of iterations that the dynamic load balancing algorithm *random polling* spends in its warmup phase until all processors are busy. Hypothesized upper bound, lower bound and measured averages with standard deviation [54, 56]. (See also Sections 11.4 and 11.4.)

exceeded in less than 5 % of the measurements. The caption should explain how many measurements have been performed.

11.5 Grids and Ticks

Tools for drawing graphs give us a lot of control over how axes are decorated with numbers, tick marks and grid lines. The general rule that is often achieved automatically is to use a few round numbers on each axis and perhaps additional tick marks without numbers. The density of these numbers should not be too high. Not only should they appear well separated but they also should be far from dominating the visual appearance of the graph. When a very large range of values is displayed, we sometimes have to force the system to use exponential notation on a part of the axis before numbers get too long. Figure 11.6 gives an example for the particularly important case of base two scales. Sometimes we may decide that reading off values is so important in a particular graph that grid lines should be added, i.e., horizontal and vertical lines that span the entire range of the graph. Care must be taken that such grid lines to not dilute the visual impression of the data

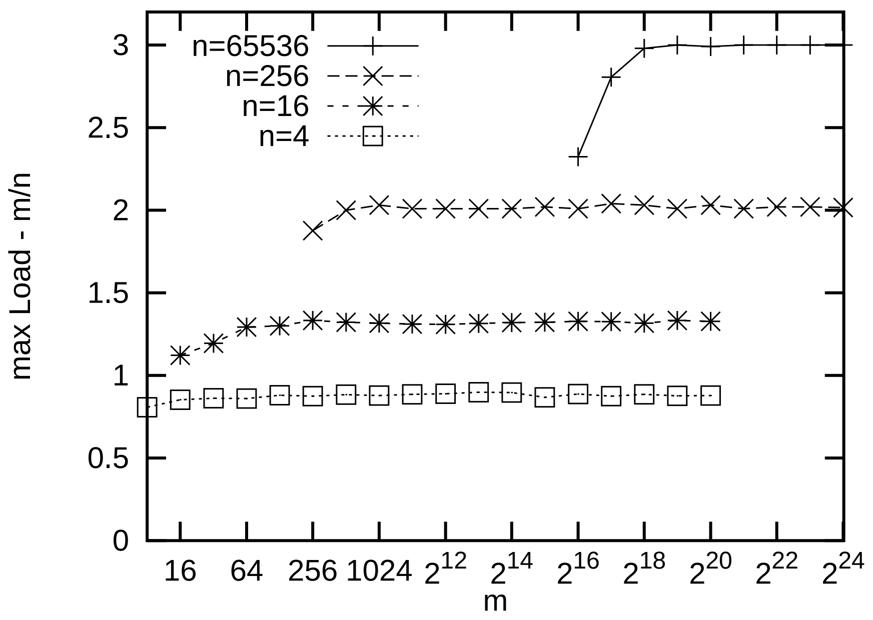


Figure 11.6: m Balls are placed into n bins using *balanced random allocation* [44, 45]. The difference between maximal and average load is plotted for different values of m and n. The experiments have been repeated at least sufficiently often to reduce the *standard error* ($\sigma/\sqrt{\text{repetitions}}$ [53]) below one percent. In order to minimize artifacts of the random number generator, we have used a generator with good reputation and very long period ($2^{19937} - 1$) [49]. In addition, some experiments were repeated with the Unix generator `srand48` leading to almost identical results. (See also Section 11.4.)

points. Hence, grid lines should be avoided or at least made thin or, even better, light gray. Sometimes grid lines can be avoided by plotting the values corresponding to some particularly important data points also on the axes.

A principle behind many of the above considerations is called Data-Ink Maximization by Tufte [59]. In particular, one should reduce non-data ink and redundant data ink from the graph. The ratio of data ink to total ink used should be close to one. This principle also explains more obvious sins like pseudo-3D bar charts, complex fill styles, etc.

11.6 Three-dimensional Figures

On the first glance, three-dimensional figures are attractive because they look sophisticated and promise to present large amounts of data in a compact way. However there are many drawbacks.

- It is almost impossible to read absolute values from the two-dimensional projection of a function.
- In complicated functions interesting parts may be hidden from view.
- If several functions are to be compared, one is tempted to use a corresponding number of three-dimensional figures. But in this case, it is more difficult to interpret differences than in two-dimensional figures with cross-sections of all the functions.

It seems that three-dimensional figures only make sense if we want to present the general shape of a single function. Perhaps three-dimensional figures become more interesting using advanced interactive media where the user is free to choose viewpoints, read off precise values, view subsets of curves, etc.

11.7 The Caption

Graphs are usually put into "floating figures" which are placed by the text formatter so that page breaks are taken into account. These figures have a caption text at their bottom which makes the figure sufficiently self contained. The captions explains *what* is displayed and *how* the measurements have been obtained. This includes the instances measured, the algorithms and their parameters used, and, if relevant the system configuration (hardware, compiler,...). One should keep in mind that experiments in a scientific paper should be reproducible, i.e., the information available should suffice to repeat a similar experiment with similar results. Since the caption should not become too long it usually contains explicit or implicit references to surrounding text, literature or web resources.

11.8 A Check List

In the following we summarize the rules discussed above. This list has the additional beneficial effect to serve as a check list one can refer to for preparing graphs and for teaching. The Section numbers containing a more detailed discussion are appended in brackets. The order of the rules has been chosen so that in most cases they can be applied in the order given.

- Should the experimental setup from the exploratory phase be redesigned to increase conciseness or accuracy? (11.2)
- What parameters should be varied? What variables should be measured? How are parameters chosen that cannot be varied? (11.2)
- Can tables be converted into curves, bar charts, scatter plots or any other useful graphics? (11.3, 11.4)
- Should tables be added in an appendix or on a web page? (11.3)
- Should a 3D-plot be replaced by collections of 2D-curves? (11.6)
- Can we reduce the number of curves to be displayed? (11.4)
- How many figures are needed? (11.4)
- Scale the x-axis to make y-values independent of some parameters? (11.4)
- Should the x-axis have a logarithmic scale? If so, do the x-values used for measuring have the same basis as the tick marks? (11.4)
- Should the x-axis be transformed to magnify interesting subranges? (11.4)
- Is the range of x-values adequate? (11.4)
- Do we have measurements for the right x-values, i.e., nowhere too dense or too sparse? (11.4)
- Should the y-axis be transformed to make the interesting part of the data more visible? (11.4)
- Should the y-axis have a logarithmic scale? (11.4)
- Is it be misleading to start the y-range at the smallest measured value? (11.4)
- Clip the range of y-values to exclude useless parts of curves? (11.4)
- Can we use banking to $45°$? (11.4)
- Are all curves sufficiently well separated? (11.4)
- Can noise be reduced using more accurate measurements? (11.4)
- Are error bars needed? If so, what should they indicate? Remember that measurement errors are usually *not* random variables. (11.4, 11.4)

- Use points to indicate for which x-values actual data is available. (11.4)
- Connect points belonging to the same curve. (11.4,11.4)
- Only use splines for connecting points if interpolation is sensible. (11.4,11.4)
- Do not connect points belonging to unrelated problem instances. (11.4)
- Use different point and line styles for different curves. (11.4)
- Use the same styles for corresponding curves in different graphs. (11.4)
- Place labels defining point and line styles in the right order and without concealing the curves. (11.4)
- Captions should make figures self contained. (11.7)
- Give enough information to make experiments reproducible. (11.7)

Chapter 12

Appendix

12.1 Used machine models

In 1945 John von Neumann introduced a basic architecture of a computer. The design was very simple in order to make it possible to build it with the limited hardware technology of the time. Hardware design has grown out of this in most aspects. However, the resulting programming model was so simple and powerful, that it is still the basis for most programming. Usually it turns out that programs written with the model in mind also work well on the vastly more complex hardware of todays machines.

The variant of von Neumann's model we consider is the *RAM (random access machine)* model. The most important features of this model are that it is *sequential*, i.e., there is a single processing unit, and that it has a uniform memory, i.e., all memory accesses cost the same amount of time. The memory consists of cells $S[0]$, $S[1]$, $S[2]$, $\ldots$ The "$\ldots$" means that there are potentially infinitely many cells although at any point of time only a finite number of them will be in use. We assume that "reasonable" functions of the input size n can be stored in a single cell. We should keep in mind however, that that our model allows us a limited form of parallelism. We can perform simple operations on $\log n$ bits in constant time.

The *external memory model* is like the RAM model except that the fast memory is limited in size to M words. Additionally, there is an external memory with unlimited size. There are special *I/O operations* that transfer B consecutive words between slow and fast memory. For example, the external memory could be a hard disk, M would then be the main memory size and B would be a block size that is a good compromise between low latency and high bandwidth. On current technology $M = 1$GByte and $B = 1$MByte could be realistic values. One I/O step would then be around 10ms which is 10^7 clock cycles of a 1GHz machine. With another setting of the parameters M and B, we could model the smaller access time differences between a hardware cache and main memory.

12.2 Amortized Analysis for Unbounded Arrays

Our implementation of unbounded arrays follows the algorithm design principle "make the common case fast". Array access with $[\cdot]$ is as fast as for bounded arrays. Intuitively, $pushBack$ and $popBack$ should "usually" be fast — we just have to update n. However, a single insertion into a large array might incur a cost of n. We now show that such a situation cannot happen for our implementation. Although some isolated procedure calls might be expensive, they are always rare, regardless of what sequence of operations we execute.

Lemma 18 *Consider an unbounded array u that is initially empty. Any sequence $\sigma = \langle \sigma_1, \ldots, \sigma_m \rangle$ of $pushBack$ or $popBack$ operations on u is executed in time $\mathcal{O}(m)$.*

Corollary 19 *Unbounded arrays implement the operation $[\cdot]$ in worst case constant time and the operations $pushBack$ and $popBack$ in amortized constant time.*

To prove Lemma 18, we use the *accounting method.* Most of us have already used this approach because it is the basic idea behind an insurance. For example, when you rent a car, in most cases you also have to buy an insurance that covers the ruinous costs you could incur by causing an accident. Similarly, we force all calls to $pushBack$ and $popBack$ to buy an insurance against a possible call of $reallocate$. The cost of the insurance is put on an account. If a $reallocate$ should actually become necessary, the responsible call to $pushBack$ or $popBack$ does not need to pay but it is allowed to use previous deposits on the insurance account. What remains to be shown is that the account will always be large enough to cover all possible costs.

Proof: Let m' denote the total number of elements copied in calls of $reallocate$. The total cost incurred by calls in the operation sequence σ is $\mathcal{O}(m + m')$. Hence, it suffices to show that $m' = \mathcal{O}(m)$. Our unit of cost is now the cost of one element copy.

We require an insurance of 3 units from each call of $pushBack$ and claim that this suffices to pay for all calls of $reallocate$ by both $pushBack$ and $popBack$.

We prove by induction over the calls of $reallocate$ that immediately after the call there are at least n units left on the insurance account.

First call of $reallocate$: The first call grows w from 1 to 2 after at least onecall of $pushBack$. We have $n = 1$ and $3 - 1 = 2 > 1$ units left on the insurance account.

For the induction step we prove that $2n$ units are on the account immediately before the current call to $reallocate$. Only n elements are copied leaving n units on the account — enough to maintain our invariant. The two cases in which $reallocate$ may be called are analyzed separately.

pushBack **grows the array:** The number of elements n has doubled since the last *reallocate* when at least $n/2$ units were left on the account by the induction hypothesis (this holds regardless of the type of operation that caused the *reallocate*). The $n/2$ new elements paid $3n/2$ units giving a total of $2n$ units for insurance.

popBack **shrinks the array:** The number of elements has halved since the last *reallocate* when at least $2n$ units were left on the account by the induction hypothesis. Since then, $n/2$ elements have been removed and $n/2$ elements have to be copied. After paying for the current *reallocate*, $2n - n/2 = 3/2n > 2(n/2)$ are left on the account. ▮

12.3 Analysis of Randomized Quicksort

To analyze the running time of quicksort for an input sequence $s = \langle e_1, \ldots, e_n \rangle$ we focus on the number of element comparisons performed. Other operations contribute only constant factors and small additive terms in the execution time.

Let $C(n)$ denote the worst case number of comparisons needed for any input sequence of size n and any choice of random pivots. The worst case performance is easily determined. Lines (A), (B), and (C) in Figure 3.1. can be implemented in such a way that all elements except for the pivot are compared with the pivot once (we allow *three-way* comparisons here, with possible outcomes 'smaller', 'equal', and 'larger'). This makes $n - 1$ comparisons. Assume there are k elements smaller than the pivot and k' elements larger than the pivot. We get $C(0) = C(1) = 0$ and

$$C(n) = n - 1 + \max\{C(k) + C(k') : 0 \le k \le n-1, 0 \le k' < n-k\} \quad .$$

By induction it is easy to verify that

$$C(n) = \frac{n(n-1)}{2} = \Theta\left(n^2\right) \quad .$$

The worst case occurs if all elements are different and we are always so unlucky to pick the largest or smallest element as a pivot.

The expected performance is much better.

Theorem 20 *The expected number of comparisons performed by quicksort is*

$$\bar{C}(n) \le 2n \ln n \le 1.4n \log n \quad .$$

We concentrate on the case that all elements are different. Other cases are easier because a pivot that occurs several times results in a larger middle sequence b that need not be processed any further.

Let $s' = \langle e'_1, \ldots, e'_n \rangle$ denote the elements of the input sequence in sorted order. Elements e'_i and e'_j are compared at most once and only if one of them is picked as a pivot. Hence, we can count comparisons by looking at the indicator random variables X_{ij}, $i < j$ where $X_{ij} = 1$ if e'_i and e'_j are compared and $X_{ij} = 0$ otherwise. We get

$$\bar{C}(n) = \mathrm{E}[\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathrm{prob}(X_{ij} = 1) \ .$$

The middle transformation follows from the linearity of expectation. The last equation uses the definition of the expectation of an indicator random variable $\mathrm{E}[X_{ij}] = \mathrm{prob}(X_{ij} = 1)$. Before we can further simplify the expression for $\bar{C}(n)$, we need to determine this probability.

Lemma 21 *For any* $i < j$, $\mathrm{prob}(X_{ij} = 1) = \dfrac{2}{j - i + 1}$.

Proof: Consider the $j - i + 1$ element set $M = \{e'_i, \ldots, e'_j\}$. As long as no pivot from M is selected, e'_i and e'_j are not compared but all elements from M are passed to the same recursive calls. Eventually, a pivot p from M is selected. Each element in M has the same chance $1/|M|$ to be selected. If $p = e'_i$ or $p = e'_j$ we have $X_{ij} = 1$. The probability for this event is $2/|M| = 2/(j - i + 1)$. Otherwise, e'_i and e'_j are passed to different recursive calls so that they will never be compared. ▮

Now we can finish the proof of Theorem 20 using relatively simple calculations.

$$\begin{aligned}\bar{C}(n) &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathrm{prob}(X_{ij} = 1) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} = \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k} \\ &\le \sum_{i=1}^{n} \sum_{k=2}^{n} \frac{2}{k} = 2n \sum_{k=2}^{n} \frac{1}{k} = 2n(H_n - 1) \le 2n(\ln n + 1 - 1) = 2n \ln n \ .\end{aligned}$$

For the last steps, recall the properties of the harmonic number $H_n := \sum_{k=1}^{n} 1/k \le \ln n + 1$. ■

12.4 Insertion Sort

Insertion Sort maintains the invariant that the output sequence is always sorted by choosing an arbitrary element of the input sequence but taking care to insert this element at the right place in the output sequence. Figure 12.1 gives an in-place array implementation of insertion sort. This implementation is straightforward except for a small trick that allows the inner loop to use only a single comparison. When the element e to be inserted is smaller than all previously inserted elements, it can be inserted at the beginning without further tests. Otherwise, it suffices to scan the sorted part of a from right to left while e is smaller than the current element. This process has to stop because $a[1] \leq e$. Insertion Sort has a worst case running time of $\Theta(n^2)$ but is nevertheless a fast algorithm for small n.

```
**Procedure** *insertionSort*($a$ : **Array** $[1..n]$ **of** *Element*)
    **for** $i := 2$ **to** $n$ **do**
        **invariant** $a[1] \leq \cdots \leq a[i-1]$                // a: | $1..i-1$: sorted | $i..n$: unsorted |
        //Move $a[i]$ to the right place
        $e := a[i]$                                                // a: | sorted | $e$ | $i+1..n$ |
        **if** $e < a[1]$ **then**                                              // new minimum
            **for** $j := i$ **downto** $2$ **do** $a[j] := a[j-1]$     // a: |   | sorted $> e$ | $i+1..n$ |
            $a[1] := e$                                            // a: | $e$ | sorted $> e$ | $i+1..n$ |
        **else**                                                          // Use $a[1]$ as a sentinel
            **for** $j := i$ **downto** $-\infty$ **while** $a[j-1] > e$ **do** $a[j] := a[j-1]$
            $a[j] := e$                                            // a: | $\leq e$ | $e$ | $> e$ | $i+1..n$ |
```

Figure 12.1: Insertion sort

12.5 Lemma on Interval Maxima

Lemma 22 *Consider an MST $T = (\{0, \ldots, n-1\}, E_T)$ where the JP algorithm (JP) adds the nodes to the tree in the order $0, \ldots, n-1$. Let e_i, $0 < i < n$ denote the edge used to add node i to the tree by the JP algorithm. Let w_i, denote the weight of e_i. Then, for all nodes $u < v$, the heaviest edge on the path from u to v in T has weight $\max_{u<j\leq v} w_j$.*

Proof: By induction over v. The claim is trivially true for $v = 1$. For the induction step we assume that the claim is true for all pairs of nodes (u, v') with $u < v' < v$ and show that it is also true for the pair (u, v). First note that e_v is on the path from u to v because in the JP algorithm u is inserted before v and v is an isolated node until e_v is added to the tree. Let $v' < v$ denote the node at the other end of edge e_v. Edge e_v is heavier than all the edges $e_{v'+1}, \ldots e_{v-1}$ because otherwise the JP algorithm would have added v, using e_v, earlier. There are two cases to consider (see Figure 12.2).

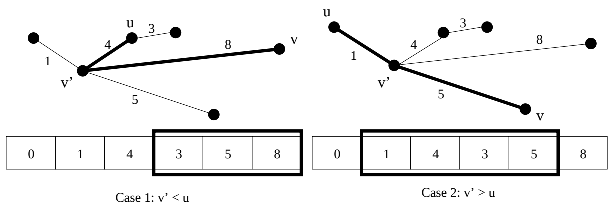


Figure 12.2: Illustration of the two cases of Lemma 22. The JP algorithm adds the nodes from left to right.

Case $v' \leq u$: By the induction hypothesis, the heaviest edge on the path from v' to u is $\max_{v'<j\leq u} w_j$. Since all these edges are lighter than e_v, the maximum over w_u,...,w_v finds the correct answer w_v.
Case $v' > u$: By the induction hypothesis, the heaviest edge on the path between u and v' has weight $\max_{u<j\leq v'} w_j$. Hence, the heaviest edge we are looking for has weight $\max\{w_v, \max_{u<j\leq v'} w_j\}$. Maximizing over the larger set $\max_{u<j\leq v} w_j$ will return the right answer since e_v is heavier than the edges $e_{v'+1}$,...e_{v-1}. ∎

Lemma 22 also holds when we have the MSF of an unconnected graph rather than the MST of a connected graph. When JP spans a connected component, it selects an arbitrary node i and adds it to the MSF with $w_i = \infty$. Then the interval maximum for two nodes that are in two different components is ∞, as it should be.

12.6 Random Permutations without additional I/Os

For renaming nodes, we need a (pseudo)random permutation $\pi : 0..n-1 \rightarrow 0..n-1$. Assume for now that n is a square so that we can represent a node i as a pair (a,b) with $i = a + b\sqrt{n}$. Our permutations are constructed from *Feistel* permutations, i.e., permutations of the form $\pi_f((a,b)) = (b, a+f(b) \bmod \sqrt{n})$ for some random mapping $f : 0..\sqrt{n}-1 \rightarrow 0..\sqrt{n}-1$. Since $\sqrt{n}$ is small, we can afford to implement f using a lookup table filled with random elements. For example, for $n = 2^{32}$ the lookup table for f would require only 128 KByte. It is known that a permutation $\pi(x) = \pi_f(\pi_g(\pi_h(\pi_l(x))))$ build by chaining four Feistel permutations is "pseudorandom" in a sense useful for cryptography. The same holds if the innermost and outermost permutation is replaced by an even simpler permutation. In our implementation we use just two stages of Feistel-Permutations. It is an interesting question what provable performance guarantees for the sweep algorithm or other algorithmic problems can be given for such permutations.

A permutation π' on $0..\lceil\sqrt{n}\rceil^2 - 1$ can be transformed to a permutation π on $0..n-1$ by iteratively applying π' until a value below n is obtained. Since π' is a permutation, this process must eventually terminate. If π' is random, the expected number of iterations is close to 1 and it is unlikely that more than three iterations are necessary for *any* input.

12.7 Proof of Discarding Theorem for Suffix Array Construction

Proof: We prove the theorem by showing that the total amount of data in the different steps of the algorithm over the whole execution is as in the data flow graph in Figure 10.3. The nontrivial points are that at most $N = n \log \mathrm{dps}$ tuples are processed in each sorting step over the whole execution and that at most n tuples are written to P. The former follows from the fact that a suffix i is involved in the sorting steps as long as it has a non-unique rank, which happens in exactly $\lceil \log(1+\mathrm{dps}(i)) \rceil$ iterations. To show the latter, we note that a tuple (c, i) is written to P in iteration k only if the previous tuple $(c', i - 2^k)$ was not unique. That previous tuple will become unique in the next iteration, because it is represented by $((c', c), i - 2^k)$ in S. Since each tuple turns unique only once, the total number of tuples written to P is at most n. ∎

12.8 Pseudocode for the Discarding Algorithm

```
Function $name2(S : Sequence$ of $Pair)$
    $q := q' := 0$;    $(\ell, \ell') := (\$, \$)$
    $result := \langle\rangle$
    foreach $((c, c'), i) \in S$ do
        if $c \neq \ell$ then $q := q' := 0$; $(\ell, \ell') := (c, c')$
        else if $c' \neq \ell'$ then $q' := q$;    $\ell' := c'$
        append $(c + q', i)$ to $result$
        $q{+}{+}$
    return $result$
```

Figure 12.3: The alternative naming procedure.

```
**Function** $doubling + discarding(T)$
    $S$:= $\langle((T[i], T[i+1]), i) : i \in [0, n)\rangle$                        1
    *sort* $S$                                                                        2
    $U$:= $name(S)$        //undiscarded                                             3
    $P$:= $\langle\rangle$               //partially discarded
    $F$:= $\langle\rangle$               //fully discarded
    **for** $k := 1$ **to** $\lceil \log n \rceil$ **do**
        *mark unique names in* $U$                                                    4
        *sort* $U$ *by* $(i \textbf{ mod } 2^k, i \textbf{ div } 2^k)$               5
        *merge* $P$ *into* $U$*;* $P$:= $\langle\rangle$                             6
        $S$:= $\langle\rangle$*;* $count$:= 0
        **foreach** $(c, i) \in U$ **do**                                             7
            **if** $c$ *is unique* **then**
                **if** $count < 2$ **then**
                    *append* $(c, i)$ *to* $F$
                **else** *append* $(c, i)$ *to* $P$
                $count$:= 0
            **else**
                *let* $(c', i')$ *be the next pair in* $U$
                *append* $((c, c'), i)$ *to* $S$
                $count{+}{+}$
        **if** $S = \emptyset$ **then**
            *sort* $F$ *by first component*                                           8
            **return** $\langle i : (c, i) \in F\rangle$                             9
        *sort* $S$                                                                    10
        $U$:= $name2(S)$                                                              11
```

Figure 12.4: The doubling with discarding algorithm.

Bibliography

[1] K. Kaligosi, P. Sanders. How Branch Mispredictions Affect Quicksort In *14th European Symposium on Algorithms (ESA)*, number 4168 in LNCS, pages 780–791, 2006.

[2] P. Sanders and S. Winkel. Super Scalar Sample Sort In *12th European Symposium on Algorithms (ESA)*, number 2625 in LNCS, pages 784–796, 2004.

[3] R. Dementiev and P. Sanders Asynchronous Parallel Disk Sorting In *15th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 138–148, San Diego, 2003.

[4] D. A. Hutchinson, P. Sanders, and J. S. Vitter. Duality Between Prefetching and Queued Writing with Parallel Disks In *9th European Symposium on Algorithms (ESA)*, number 2161 in LNCS, pages 62–73, 2001.

[5] Peter Sanders. Fast Priority Queues for Cached Memory In *ALENEX '99, Workshop on Algorithm Engineering and Experimentation*, number 1619 in LNCS, pages 312–327, 1999.

[6] Roman Dementiev. *Algorithm Engineering for Large Data Sets*. Dissertation at Universität des Saarlandes, 2006.

[7] P. Sanders, R. Dementiev. *I/O-Efficient Algorithms and Data Structures*. Slides for Algorithm Engineering course at Universität Karlsruhe, 2007.

[8] Laura Toma and Norbert Zeh I/O-Efficient Algorithms for Sparse Graphs In *Algorithms for Memory Hierarchies*, number 2625 in LNCS, pages 85–109, 2003.

[9] Piyush Kumar Cache Oblivious Algorithms In *Algorithms for Memory Hierarchies*, number 2625 in LNCS, pages 193–212, 2003.

[10] Anil Maheshwari and Norbert Zeh A Survey of Techniques for Designing I/O-Efficiet Algorithms In *Algorithms for Memory Hierarchies*, number 2625 in LNCS, pages 36–61, 2003.

[11] M. Frigo, C. E. Leiserson, H. Prokop, S. Ramachandran. Cache-Oblivious Algorithms In *40th Symposium on Foundations of Computer Science*, pages 285–298, 1999.

[12] Irit Katriel and Ulrich Meyer Elementary Graph Algorithms in External Memory In *Algorithms for Memory Hierarchies*, number 2625 in LNCS, pages 62–84, 2003.

[13] Deepak Ajwani, Ulrich Meyer, Vitaly Osipov. Improved external memory BFS implementations In *Workshop on Algorithm engineering and experiments (ALENEX 07)*, pages 3–12. New Orleans, USA, 2007.

[14] K. Munagala and A. Ranade. I/O-Complexity of Graph Algorithms In *Proc. 10th Ann. Symposium on Discrete Algorithms*, pages 687–694. ACM-SIAM, 1999.

[15] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proc. 10th Ann. European Symposium on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 723–735. Springer, 2002.

[16] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, 1995.

[17] D. Ajwani, R. Dementiev, and U. Meyer. A computational study of external-memory BFS algorithms. *SODA*, pages 601–610, 2006.

[18] Gerth Stolting Brodal and Rolf Fagerberg. Cache oblivious distribution sweeping In *Proceedings of the 29th International Colloquium on Au- tomata, Languages, and Programming*, pages 426–438, Malaga, Spain, July 2002.

[19] Erik D. Demaine. Cache-Oblivious Algorithms and Data Structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets*, 2002.

[20] L. A. Arge, G. S. Brodal, and L. Toma. On external-memory MST, SSSP and multi-way planar graph separation. In *Proc. 8th Scand. Workshop on Algorithmic Theory*, volume 1851 of *LNCS*, pages 433–447. Springer, 2000.

[21] N. Zeh. *I/O-Efficient Algorithms for Shortest Path Related Problems*. PhD thesis, School of Computer Science, Carleton University, 2002.

[22] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Information Processing Letters*, 6(3):80-82, 1977.

[23] R. Dementiev, L. Kettner, J. Mehnert, and P. Sanders. Engineering a Sorted List Data Structure for 32 Bit Keys. In *Workshop on Algorithm Engineering & Experiments*, pages 142–151, New Orleans, 2004.

[24] A. V. Goldberg and C. Harrelson. Computing the Shortest Path: A^* meets Graph Theory. In *16th ACM-SIAM Symposium on Discrete Algorithms*, pages 156–165, 2005.

[25] J. Maue and P. Sanders and D. Matijevic. Goal Directed Shortest Path Queries Using Precomputed Cluster Distances. In *5th Workshop on Experimental Algorithms (WEA)*, LNCS vol. 4007, pages 316–328, 2006.

[26] F. Schulz and D. Wagner and K. Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. In *3rd Workshop on Algorithm Engineering*, LNCS vol. 1668, pages 110–123, 1999.

[27] D. Wagner and T. Willhalm. Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs. In *11th European Symposium on Algorithms*, LNCS vol. 2832, pages 776–787, 2003.

[28] R. H. Möhring and H. Schilling and B. Schütz and D. Wagner and T. Willhalm. Partitioning Graphs to Speed Up Dijkstra's Algorithm. In *4th International Workshop on Efficient and Experimental Algorithms*, pages 189–202, 2005.

[29] E. Köhler and R. H. Möhring and H. Schilling. Acceleration of Shortest Path and Constrained Shortest Path Computation. In *4th International Workshop on Efficient and Experimental Algorithms*, 2005.

[30] P. Sanders, D. Schultes. Engineering Fast Route Planning Algorithms. In *6th Workshop on Experimental Algorithms (WEA)*, LNCS vol. 4525, pages 23–36, 2007.

[31] P. Sanders, D. Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *13th European Symposium on Algorithms (ESA)*, LNCS vol. 3669, pages 568–597, 2005.

[32] P. Sanders, D. Schultes. Engineering Highway Hierarchies. In *14th European Symposium on Algorithms (ESA)*, LNCS vol. 4168, pages 804–816, 2006.

[33] P. Sanders, D. Schultes. Robust, Almost Constant Time Shortest-Path Queries in Road Networks. In *9th DIMACS Challenge on Shortest Paths*, 2007.

[34] P. Sanders, D. Schultes. Dynamic Highway-Node Routing. In *6th Workshop on Experimental Algorithms (WEA)*, LNCS vol. 4525, pages 66–79, 2007.

[35] R. Gutman. Reach-based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *6th Workshop on Algorithm Engineering and Experiments*, pages 100–111, 2004.

[36] D. Delling and D. Wagner. Landmark-Based Routing in Dynamic Graphs. In *6th Workshop on Experimental Algorithms*, 2007.

[37] I. Katriel, P. Sander and J. L. Träff. A Practical Minimum Spanning Tree Algorithm Using the Cycle Property. In *11th European Symposium on Algorithms (ESA)*, LNCS vol. 2832, pages 679–690, 2003.

[38] Roman Dementiev, Peter Sanders, Dominik Schultes, Jop Sibeyn. A Practical Minimum Spanning Tree Algorithm Using the Cycle Property. In *3rd IFIP International Conference on Theoretical Computer Science (TSC2004)*, pages 195–208, 2004.

[39] Juha Kärkkäinen, Peter Sanders. *Sorting Strings and Suffixes*. Slides, 2003.

[40] Bentley and Sedgewick. Fast Algorithms for Sorting and Searching Strings. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, 1997.

[41] J. Kärkkäinen and P. Sanders. Simple Linear Work Suffix Array Construction. In *30th International Colloquium on Automata, Languages and Programming*, LNCS vol. 2719, pages 943–955, 2003.

[42] R. Dementiev, J. Mehnert, J. Kärkkäinen, P. Sanders. Better External Memory Suffix Array Construction. In *Workshop on Algorithm Engineering & Experiments (ALENEX05)*, pages 86–97, 2005.

[43] P. Alefragis, P. Sanders, T. Takkula, and D. Wedelin. Parallel integer optimization for crew scheduling. *Annals of Operations Research*, 99(1):141–166, 2000.

[44] Y. Azar, A. Z. Broder, A. R. Karlin, and Eli Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1):180–200, February 2000.

[45] P. Berenbrink, A. Czumaj, A. Steger, and B. Vöcking. Balanced allocations: The heavily loaded case. In *32th Annual ACM Symposium on Theory of Computing*, 2000.

[46] J. M. Chambers, W. S. Cleveland, b. Kleiner, and P. A. Tukey. *Graphical Methods for Data Analysis*. Duxbury Press, Boston, 1983.

[47] W. S. Cleveland. *Elements of Graphing Data*. Wadsworth, Monterey, Ca, 2nd edition, 1994.

[48] D. S. Johnson. A theoretician's guide to the experimental analysis of algorithms. In M. Goldwasser, D. S. Johnson, and C. C. McGeoch, editors, *Proceedings of the 5th and 6th DIMACS Implementation Challenges*. American Mathematical Society, 2002.

[49] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACMTMCS: ACM Transactions on Modeling and Computer Simulation*, 8:3–30, 1998. `http://www.math.keio.ac.jp/~matumoto/emt.html`.

[50] C. C. McGeoch, D. Precup, and P. R. Cohen. How to find big-oh in your data set (and how not to). In *Advances in Intelligent Data Analysis*, number 1280 in LNCS, pages 41–52, 1997.

[51] C.C. McGeoch and B. M. E. Moret. How to present a paper on experimental work with algorithms. *SIGACT News*, 30(4):85–90, 1999.

[52] B. M. E. Moret. Towards a discipline of experimental algorithmics. In *5th DIMACS Challenge*, DIMACS Monograph Series, 2000. to appear.

[53] W. H. Press, S.A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 2nd edition, 1992.

[54] P. Sanders. *Lastverteilungsalgorithmen für parallele Tiefensuche*. Number 463 in Fortschrittsberichte, Reihe 10. VDI Verlag, 1997.

[55] P. Sanders. Asynchronous scheduling of redundant disk arrays. In *12th ACM Symposium on Parallel Algorithms and Architectures*, pages 89–98, 2000.

[56] P. Sanders and R. Fleischer. Asymptotic complexity from experiments? a case study for randomized algorithms. In *Workshop on Algorithm Engineering*, number 1982 in LNCS, pages 135–146, 2000.

[57] P. Sanders and J. Sibeyn. A bandwidth latency tradeoff for broadcast and reduction. In *6th Euro-Par*, number 1900 in LNCS, pages 918–926, 2000.

[58] Peter Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5, 2000.

[59] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, Connecticut, U.S.A., 1983.