Δ-Stepping: A Parallel Single-Source Shortest-Path Algorithm

Ulrich Meyer and Peter Sanders

ESA Test-of-Time Award 2019

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Karlsruhe Institute of Technology

Thank You for the ESA Test-of-Time Award 2019 for

Δ-Stepping: A Parallel Single-Source Shortest-Path Algorithm.

You have honored small and simple steps
in a long, difficult and important Odyssey.


CC BY-SA 3.0, Hamilton Richards

CC BY 2.0, ORNL and Carlos Jones

From Dijkstra's algorithm to parallel shortest paths

Overview

- Motivation — Peter
- Problem statement and previous work — Uli
- Parallel Dijkstra — Uli
- Basic Δ-stepping — Uli
- Average case linear time sequential algorithm — Uli
- Multiple Δs — Uli
- Implementation experiences — Peter
- Subsequent work — Peter
- Conclusions and open problems — Peter

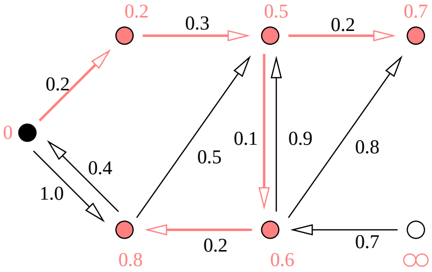
time	algorithmics	hardware
1970s	new	new
1980s	intensive work	ambitious/exotic projects
1990s	rapid decline	bankruptcies / triumps of single proc. performance
2000s	almost dead	beginning multicores
2010s	slow comeback ?	ubiquitous, exploding parallelism:
		smartPhone, GPGPUs, cloud, Big Data,...
2020s	up to us	

see also: [S., "Parallel Algorithms Reconsidered", STACS 2015, invited talk]

- Large graphs,
  e.g., huge implicitly defined state spaces
- Stored distributedly
- Many iterations, edge weights may change every time
- Even when independent SSSPs are needed:
  memory may be insufficient for running all of them

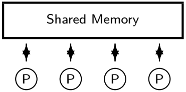
Single-Source Shortest Path (SSSP)

- Digraph: $G = (V, E)$, $|V| = n$, $|E| = m$
- Single source: s
- Non-negative edge weights: $c(e) \geq 0$
- Find: $\mathrm{dist}(v) = \min\{c(p)\ ;\ p$ path from s to $v\}$



Average-case setting:

independent random edge weights uniformly in $[0, 1]$.



PRAM

- Shared memory
- Uniform access time
- Synchronized
- Concurrent access

- Work $=$ total number of operations $\leq$ number of processors $\cdot$ parallel time

Key results:

Time:	Work:	Ref:
$\mathcal{O}(\log n)$	$\mathcal{O}(n^{3+\epsilon})$	[Han, Pan, and Reif, *Algorithmica* 17(4), 1997]
$\mathcal{O}(n \cdot \log n)$	$\mathcal{O}(n \cdot \log n + m)$	[Paige, Kruskal, *ICPP*, 1985]
$\mathcal{O}(n^{2\epsilon} + n^{1-\epsilon})$	$\mathcal{O}(n^{1+\epsilon})$, planar graphs	[TrÃďff, Zaroliagis, *JPDC* 60(9), 2000]

Goal:

$o(n)$ $\qquad$ $\mathcal{O}(n \cdot \log n + m)$

Search for hidden parallelism in sequential SSSP algorithms !

Sequential SSSP: What else was common 20 years ago?

1. Dijkstra with specialized priority queues:
   - (small) integer or float weights
   - Bit operations: RAM with word size w
2. Component tree traversal (label-setting):
   - rather involved
   - undirected: $\mathcal{O}(n+m)$ time [Thorup, *JACM* 46, 1999]
   - directed: $\mathcal{O}(n+m\log w)$ time [Hagerup, *ICALP*, 2000]
3. Label-correcting algorithms:
   - rather simple
   - bad in the worst case, but often great in practice
   - average-case analysis largely missing

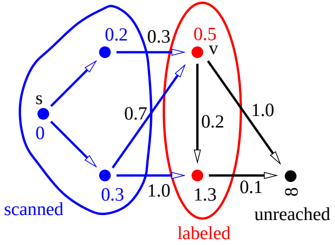
Our ESA-paper in 1998:

Simple label-correcting algorithm for directed SSSP with theoretical analysis.

Basis for various sequential and parallel extensions.

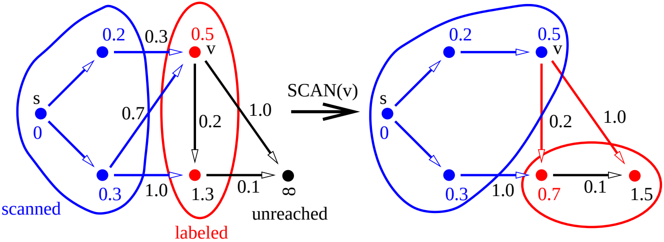
Dijkstra's Sequential Label-Setting Algorithm

- Partitioning: settled, queued, unreached nodes
- Store tentative distances $\mathrm{tent}(v)$ in a **priority-queue** Q.
- Settle nodes one by one in priority order:
  v selected from $Q \Rightarrow \mathrm{tent}(v) = \mathrm{dist}(v)$
- Relax outgoing edges
- $\mathcal{O}(n \log n + m)$ time (comparison model)



Dijkstra's Sequential Label-Setting Algorithm

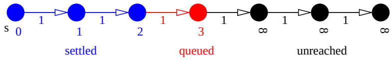
- Partitioning: settled, queued, unreached nodes
- Store tentative distances $\mathrm{tent}(v)$ in a **priority-queue** Q.
- Settle nodes one by one in priority order:
  v selected from $Q \Rightarrow \mathrm{tent}(v) = \mathrm{dist}(v)$
- Relax outgoing edges
- $\mathcal{O}(n \log n + m)$ time (comparison model)

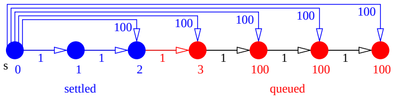


Hidden Parallelism in Dijkstra's Algorithm?

Question: Is there always more than one settled vertex in Q with $\mathrm{tent}(v) = \mathrm{dist}(v)$?

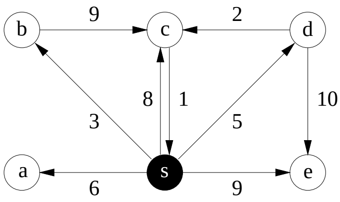
Answer: Not in the worst case:

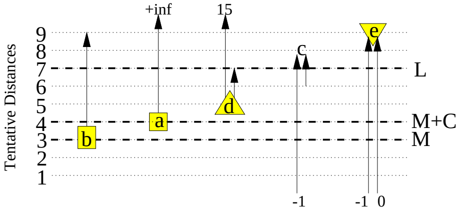




Lower Bound: At least as many phases as depth of shortest path tree.
In practice such trees are often rather flat ...

Challenge: Find provably good identification criteria for settled vertices.





Random graphs: $\mathcal{D}(n, \bar{d}/n)$

- Edge probability $\bar{d}/n$
- Weights indep. & uniform in $[0, 1]$

Analysis:

OUT: $\mathcal{O}(\sqrt{n})$ phases whp.
INOUT: $\mathcal{O}(n^{1/3})$ phases whp.

Simulation:

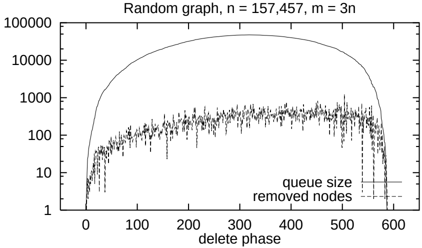
OUT: $2.5 \cdot \sqrt{n}$ phases on av.
INOUT: $6.0 \cdot n^{1/3}$ phases on av.

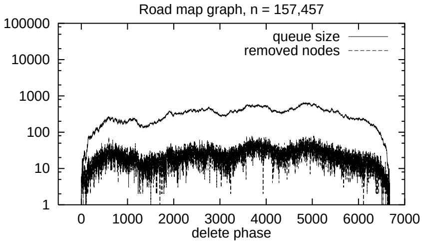
Road maps:

Southern Germany: $n = 157457$.

INOUT: 6647 phases.

$n \to 2 \cdot n$:
The number of phases is multiplied by approximately $1.63 \approx 2^{0.7}$.





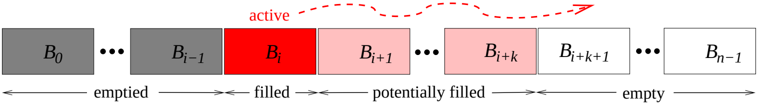
Promising approach but (at that time) still too many phases.

Recent revival: V. K. Garg 2018, Krainer/TrÃďff 2019.

Basic Δ-Stepping

Q is replaced by array $B[\cdot]$ of buckets having width Δ each.

Source $s \in B[0]$ and $v \in Q$ is kept in $B[\,\lfloor \mathrm{tent}(v)/\Delta \rfloor\,]$.



In each phase: Scan all nodes from first nonempty bucket ("current bucket", B_{cur}) but only relax their outgoing light edges $(c(e)) < \Delta)$.

When B_{cur} finally remains empty: Relax all heavy edges of nodes settled in B_{cur} and search for next nonempty bucket.

Difference to *Approximate Bucket Implementation** of Dijkstra's Algorithm:

- No FIFO order in buckets assumed.
- Distinction between light and heavy edges.

* [Cherkassky, Goldberg, and Radzik, *Math. Programming* 73:129–174, 1996]

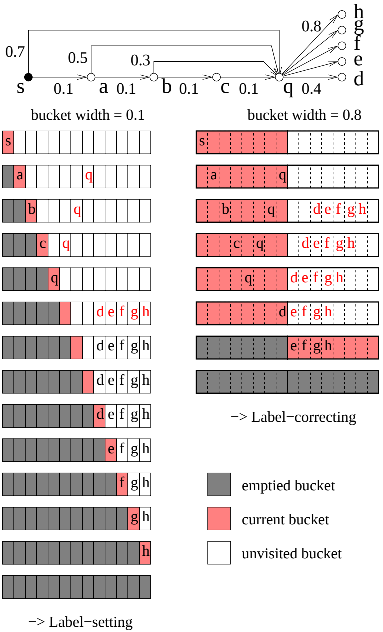
Choice of the Bucket Width Δ

Extreme cases:

- Δ = min edge weight in G
  → label-setting (no re-scans)
  → potentially many buckets traversed *(Dinic-Algorithm*)*
- $\Delta = \infty$: $\simeq$ Bellman-Ford
  → label-correcting (potentially many re-inserts)
  → less buckets traversed.

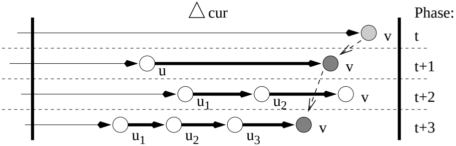
Is there a provably good choice for Δ that always beats Dijkstra?

- not in general :-(
- but for many graph classes :-)



* [Dinic, *Transportation Modeling Systems*, 1978]

Δ-Stepping with i.i.d. Random Edge Weights Uniformly in $[0,1]$



Lemma: $\#$ re-insertions(v) $\leq \#$ paths into v of weight $< \Delta$ ("Δ-paths").
If $d :=$ max. degree in $G \Rightarrow \leq d^l$ paths of l edges into v.

Lemma: Prob [path of l edges has weight $\leq \Delta$] $\leq \Delta^l/l!$

$\Rightarrow$ $\text{E}[\#$ re-ins.$(v)] \leq \sum_l d^l \cdot \Delta^l/l! = \mathcal{O}(1)$ for $\Delta = \mathcal{O}(1/d)$

$\mathcal{L} :=$ max. shortest path weight, graph dependent !

Thm: Sequential $\Theta(\frac{1}{d})$-Stepping needs $\mathcal{O}(n + m + d \cdot \mathcal{L})$ time on average.

Linear if $d \cdot \mathcal{L} = \mathcal{O}(n+m)$ e.g. $\mathcal{L} = \mathcal{O}(\log n)$ for random graphs whp.

BUT: $\exists$ sparse graphs with random weighs where any fixed Δ causes $\omega(n)$ time.

Number of Phases for $\Theta(1/d)$-Stepping with Random Edge Weights

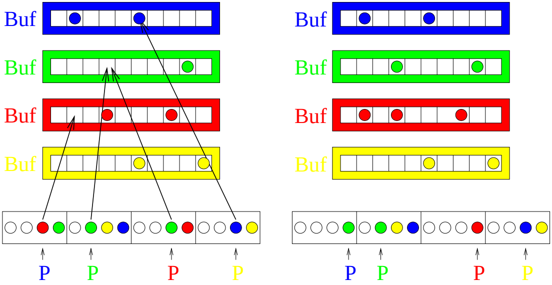
Lemma: For $\Delta = \mathcal{O}(1/d)$, no Δ-path contains more than $l_\Delta = \mathcal{O}(\log n / \log\log n)$ edges whp.

$\Rightarrow$ At most $\lceil d \cdot \mathcal{L} \cdot l_\Delta \rceil$ phases whp.

- Active insertion of shortcut edges [M.,S., *EuroPar*, 2000] in a preprocessing can reduce the number of phases to $\mathcal{O}(d \cdot \mathcal{L})$:
  Insert direct edge (u, v) for each simple Δ-path $u \to v$ with same weight.
- For random graphs from $\mathcal{D}(n, \bar{d}/n)$ we have $d = \mathcal{O}(\bar{d} + \log n)$ and $\mathcal{L} = \mathcal{O}(\log n / \bar{d})$ whp. yielding a polylogarithmic number of phases.
- Time for a phase depends on the exact parallelization.
- We maintain linear work.

- Randomized assignment of vertex indices to processors.
- Problem: Requests for the same target queue must be transfered and performed in *some* order, standard sorting is too expensive.
- Simple solution: Use commutativity of requests in a phase: Assign requests to their appropriate queues in random order.
- Technical tool: *Randomized dart-throwing.*

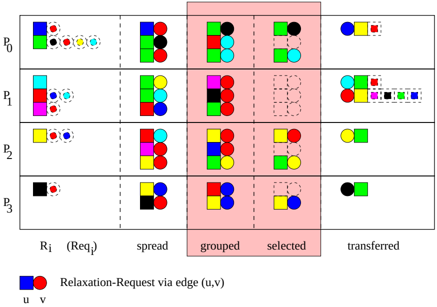
$\mathcal{O}(d \cdot \log n)$ time per $\Theta(1/d)$-Stepping phase.



Improved PRAM Parallelization [M.,S., *EuroPar*, 2000]

Central Tool: Grouping

- Group relaxations concerning target nodes (blackbox: hashing & integer sorting).
- Select strictest relaxation per group.
- Transfer selected requests to appropriate Q_i.
- For each Q_i, perform selected relaxation.



At most one request per target node $\Rightarrow$ Improved Load-Balancing.

$\mathcal{O}(\log n)$ time per $\Theta(1/d)$-Stepping phase.

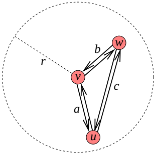
Intermediate Conclusions – Δ-Stepping with Fixed Bucket Width

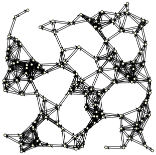
Δ-Stepping works provably well with random edge weights on small to medium diameter graphs with small to medium nodes in-degrees, e.g.:

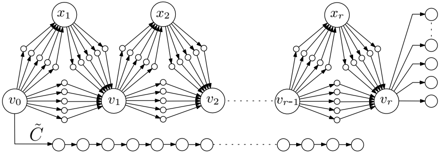
- Random Graphs from $\mathcal{D}(n, \bar{d}/n)$: $\mathcal{O}(\log^2 n)$ parallel time and linear work.
- Random Geometric Graphs with threshold parameter $r \in [0, 1]$:
  Choosing $\Delta = r$ yields linear work.

There are classes of sparse graphs with random edge weight where no good fixed choice for Δ exists [M., Negoescu, Weichert, *TAPAS*, 2011]:

- Δ-Stepping: $\Omega(n^{1.1-\epsilon})$ time on average.
- ABI-Dijkstra: $\Omega(n^{1.2-\epsilon})$, Dinic & Bellman-Ford: $\Omega(n^{2-\epsilon})$







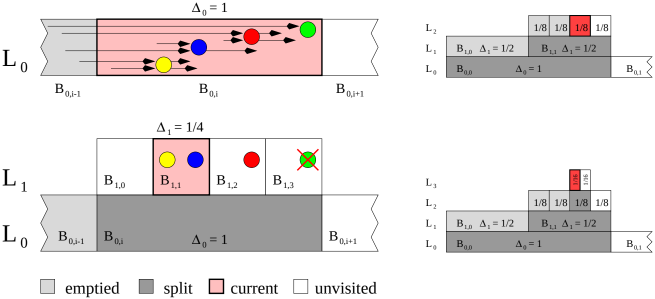
$\Rightarrow$ Develop algorithms with dynamically adapting bucket width Δ.

Linear Average-Case Sequential SSSP for Arbitrary Degrees [M., *SODA*, 2001]

Run Δ-Stepping with initial bucket width $\Delta_0 = 1$.
$d^* :=$ max. degree in current bucket B_{cur} at phase start.

If $\Delta_{cur} > 1/d^*$

1. Split B_{cur} into buckets of width $\leq 1/d^*$ each.
2. Settle nodes with “obvious” final distances.
3. Find new current bucket on next level.

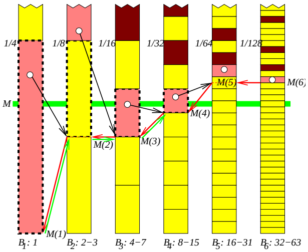


$\Rightarrow$ creates at most $\sum_v 2 \cdot \text{degree}(v) = \mathcal{O}(m)$ new buckets.

$\Rightarrow$ High-degree nodes treated in narrow current buckets.
$\rightarrow$ Linear average-case bound for arbitrary graphs.

Parallel Independent Stepwidths [M., *IPDPS*, 2002]

Direct parallelization of the splitting idea still takes $\Omega(\text{maxdegree})$ phases. Better:



- $\Theta(\log n)$ cyclically traversed bucket arrays with exponentially decreasing Δ.
- All nodes v of degree d_v treated in buckets of width $\simeq 2^{-d_v}$, no splitting.
- Parallel scanning from selected buckets.
- Fast traversal of empty buckets.

Improves the parallel running time from

$$T = \mathcal{O}(\log^2 n \cdot \min_i\{2^i \cdot \mathbf{E}[\mathcal{L}] + \textstyle\sum_{v\in G,\text{degree}(v)>2^i} \text{degree}(v)\})$$ to

$$T = \mathcal{O}(\log^2 n \cdot \min_i\{2^i \cdot \mathbf{E}[\mathcal{L}] + \textstyle\sum_{v\in G,\text{degree}(v)>2^i} 1\})$$

Ex: Low-diameter graphs where vertex degrees follow a power law $(\beta = 2.1)$:

Δ-Stepping: $\Omega(n^{0.90})$ time and $\mathcal{O}(n+m)$ work on average.

Parallel Indep. Step Widths: $\mathcal{O}(n^{0.48})$ time and $\mathcal{O}(n+m)$ work on average.

Beyond Parallelism

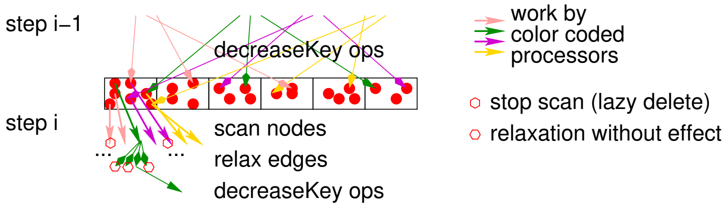
The linear average-case SSSP result from [M., *SODA*, 2001] has triggered various alternative sequential solutions:

- [A.V. Goldberg, *A simple shortest path algorithm with linear average time.* ESA, 2001]
  - for integer weights
  - based on radix heaps
- [A.V. Goldberg, *A Practical Shortest Path Alg. with Linear Expected Time.* SIAM J. Comput., 2008]
  - optimized code for realistic inputs with integer/float weights.
  - implementation is nearly as efficient as plain BFS.
- [T. Hagerup, *Simpler Computation of SSSP in Linear Average Time.* STACS, 2004]
  - combination of heaps and buckets
  - focus on simple common data structures and analysis

All approaches use some kind of special treatment for vertices with small incoming edge weights ($\simeq$ IN-criterion).

Implementing Δ-Stepping – Shared Memory

- graph data structure as in seq. case
- lock-free edge relaxations (e.g., use CAS/fetch_and_min) with little contention (few updates on average)
- possibly replace decrease-key by insertion and lazy deletion
- synchronized phases simplify concurrent bucket-priority-queue
- load balanced traversal of current bucket



Or use shared-memory implementation of a distributed-memory algorithm

[Madduri et al., "Parallel Shortest Path Algorithms for Solving Large-Scale Instances", 9th DIMACS Impl. Challenge, 2006]

[Duriakova et al. "Engineering a Parallel Δ-stepping Algorithm", IEEE Big Data, 2019]

Implementing Δ-Stepping – Distributed Memory

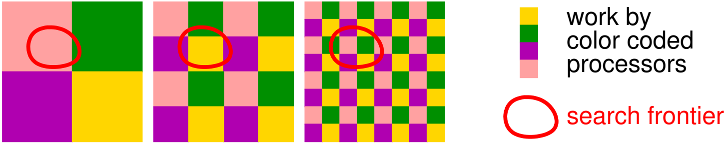
- 1D partitioning: each PE responsible for some vertices – owner computes paradigm

  Procedure relax(u, v, w)

  if v is local **then** relax locally

  else send relaxation request (v, w) to owner of v

- Two extremes in a Tradeoff:
  - use graph partitioning: high locality
  - random assignment: good load balance



- Extensive tuning on RMAT graphs (very low diameter).
  $\rightsquigarrow$ algorithms with complexity $O(n \cdot diameter)$
  (unscanned vertices pull relevant relaxations)

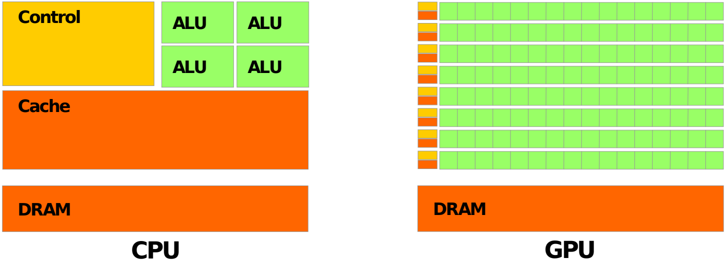
[Chakravarthy et al., *Scalable single source shortest path algorithms for massively parallel systems*, IEEE TPDS 28(7), 2016]

Implementing Δ-Stepping – GPU

[Davidson et al. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths, IPDPS 2014]:

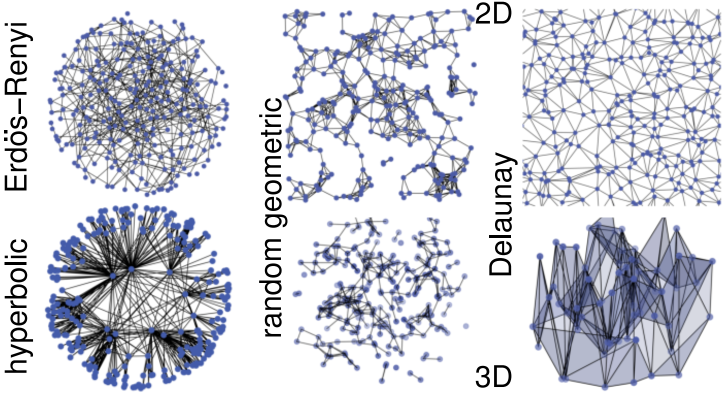
- Partition edges to be relaxed ⇝ fine-grained parallelization
- Fastest algorithm is sth like Δ-Stepping without a PQ.
  Rather, identify vertices in next bucket brute-force from a “far pile”.

[Ashkiani et al., GPU Multisplit: An Extended Study of a Parallel Algorithm, ACM TPC 4(1), 2017]:
bucket queue is now useful



Implementing Δ-Stepping – Summary

- Better than Dijkstra or Bellman-Ford
- Several implementation difficulties:
  load balancing, contention, parameter tuning,...
  $\leadsto$ implementation details can dominiate experimental performance
- Viable for low diameter graphs. Challenging for high diameter



Subsequent Work – Radius Stepping

[Blelloch et al., *Parallel shortest paths using radius stepping* SPAA 2016]

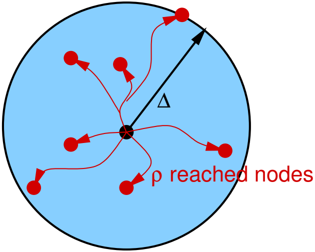
Generalization of Δ-stepping:

- choose Δ adaptively
- add shortcuts such that from any vertex ρ vertices are reached in one step

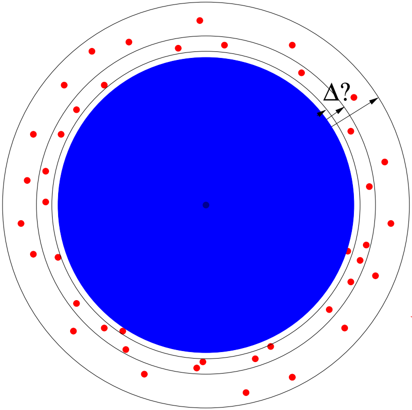
Work–Time tradeoff

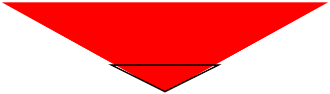
$m \log n + n\rho^2$ work versus $\frac{n}{\rho} \log n \log \rho \cdot \mathrm{maxEdgeWeight}$ time

for tuning parameter ρ



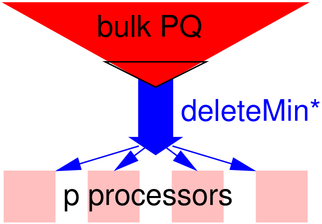
How to choose Δ in practice?
Perhaps adapt dynamically to keep a given amount of parallelism?





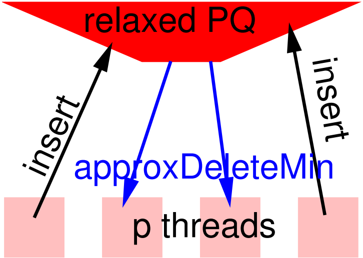
Then why not do this directly?
$\rightsquigarrow$ relaxed priority queue.

Procedure RPQ-SSSP(G, s)
  $\mathrm{dist}[v] := \infty$ for all $v \in V$; $\mathrm{dist}[s] := 0$
  RelaxedPriorityQueue $Q = \{(s, 0)\}$
  while $Q \neq \emptyset$ **do** // Globally synchronized iterations
    $L^* := Q$.deleteMin* // get the $O(p)$ smallest labels
    foreach $(v, \ell) \in L^*$ **dopar**
      if $\mathrm{dist}[v] = \ell$ **then** // still up to date?
        foreach $e = (v, w) \in E$ **do** relax$((v, w), x)$



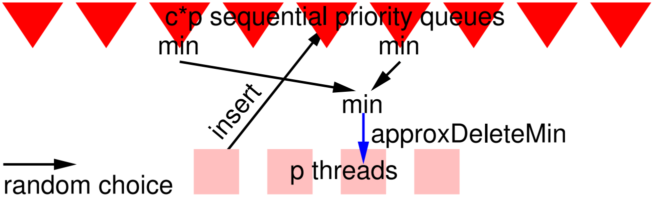
deleteMin* can be implemented with logarithmic latency
[S., *Randomized priority queues for fast parallel access* JPDC 49(1), 86–97, 1998]
and respecting the owner-computes paradigm
[Hübschle-Schneider, S., *Communication Efficient Algorithms for Top-k Selection Problems*, IPDPS 2016]

Procedure asynchronousRPQ-SSSP(G, s)
  $\text{dist}[v] := \infty$ for all $v \in V$; $\text{dist}[s] := 0$
  AsynchronousRelaxedPriorityQueue $Q = \{(s, 0)\}$
  foreach thread **dopar**
    while no global termination **do** // asynchronous parallelism
      $(v, \ell) := Q$.approxDeleteMin // get a "small" label
      if $\text{dist}[v] = \ell$ **then** // still up to date?
        foreach $e = (v, w) \in E$ **do** relax$((v, w), x)$



MultiQueues:

- $c \cdot p$ sequential queues, $c > 1$
- insert: into random queue
- approxDeleteMin: minimum of minimum of two (or more) random queues
- "Waitfree" locking



[Rihani et al., *MultiQueues: Simpler, Faster, and Better Relaxed Concurrent Priority Queues*, SPAA 2015]

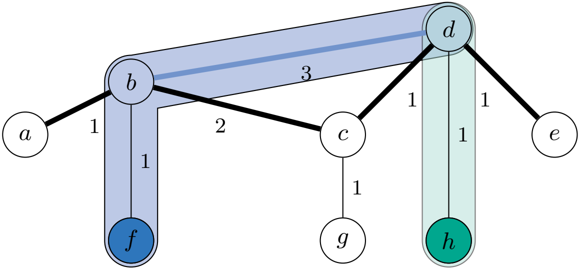
[Alistarh et al., *The power of choice in priority scheduling*, PODC 2017]

Idea: preprocess graph. Then support fast s–t queries.

Successful example **Contraction Hierarchies (CHs):**
Aggressive (obviously wrong) heuristics:
Sort vertices by “importance”. Consider only up–down routes –

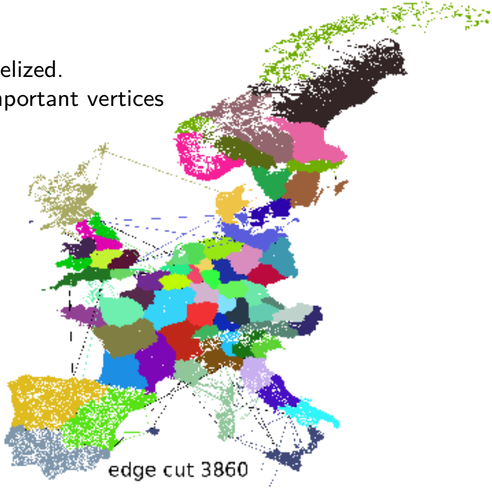
- $\nearrow$ Ascend to more and more important vertices
- $\searrow$ Descend to less and less important vertices

Make that correct by inserting appropriate shortcuts.



About 10 000 times faster than Dijkstra for large road networks.

- Construction of CHs can be parellelized.
  Roughly: Contract locally least important vertices
- Trivial parallelization of multiple point-to-point queries
- Distributable using graph partitioning
- “polylogarithmic” parallel time one-to-all/few-to-all queries using PHAST



[Geisberger et al., *Exact Routing in Large Road Networks using Contraction Hierarchies, Transportation Science 46(3), 2012*]

[Kieritz et al., *Distributed Time-Dependent Contraction Hierarchies*, SEA 2010]

[Delling et al., *PHAST: Hardware-accelerated shortest path trees*, JPDC 73(7), 2013]

Subsequent Work – Multi-objective Shortest Paths

Given d objective functions, s,
for (one/all) t, find all Pareto optimal s–t paths,
i.e., those that are not dominated by any other path wrt all objectives.

NP-hard, efficient "output-sensitive" sequential algorithms

Example: time/changes/footpaths tradeoff for public transportation

Procedure paPaSearch(G, s) // was: Dijkstra

  $\mathrm{dist}[v] := \emptyset$ for all $v \in V$; $\mathrm{dist}[s] := \{0^d\}$

  ParetoQueue $Q = \{(s, 0^d)\}$ // was: PriorityQueue

  while $Q \neq \emptyset$ **do**

    $L^* := Q$.deleteParetoOptimal // was: deleteMin

    foreach $(v, \ell) \in L^*$ **do** // was: one label

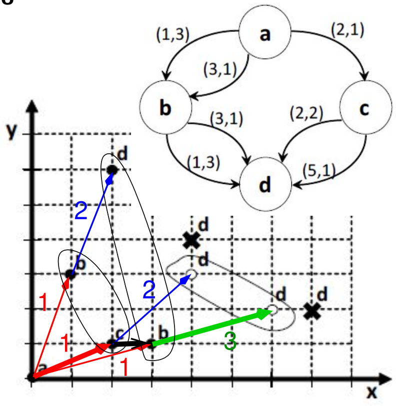
      foreach $e = (v, w) \in E$ **do**

        relax$((v, w), x)$

Theorem: $\leq n$ iterations

Theorem for $d = 2$:
efficient parallelization with
time $O(n \log p \log \text{totalWork})$
(Search trees,
geometry meets graph algorithms)

"All the hard stuff is parallelizable"
[S., Mandow, *Parallel Label-Setting Multi-Objective Shortest Path Search*, IPDPS 2013]



Conclusion and Open Problems

- Main theoretical question still open:
  work optimal SSSP (even BFS) with $o(n)$ time? (beyond bounded treewidth [Chaudhuri–Zaroliagis/Bodlaender–Hagerup])
  - special graph classes?
  - average case, smoothed analysis?
- Better relaxed priority queues (RPQs) in theory and practice:
  - small rank error: no large elements deleted very early
  - "fairness": no small elements deleted very late
  - cache efficient
  - better understand termination detection
  - analyze SSSP and other applications with RPQs (e.g., branch-and-bound)
- Algorithm engineering for (distributed-memory) SSSP
  - Δ/radius-stepping/generalized Dijkstra/Independent stepwidth/relaxed PQs
  - asynchronous algorithms
  - tradeoff partitioning versus randomization
  - diverse inputs

P.S. is hiring!

- More inputs in all experiments, e.g.,:
  - use geometric graphs with their natural distances (e.g., Delaunay, random geometric, hyperbolic)
    [Funke et al. *Communication-free massively distributed graph generation*, JPDC 2019]
  - Graph Delaunay Diagrams use low diameter SSSP in high diameter graphs
    [Mehlhorn, *A faster approximation algorithm for the Steiner problem in graphs*, IPL 1988]