Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Peter Sanders
Dr. Christian Schulz, Dr. Simon Gog
Michael Axtmann

4. Übungsblatt zu Algorithmen II im WS 2016/2017

http://algo2.iti.kit.edu/AlgorithmenII_WS16.php
{christian.schulz,michael.axtmann,sanders,simon.gog}@kit.edu

Musterlösungen

Aufgabe 1 (*Analyse: Eigenschaften von Flüssen*)

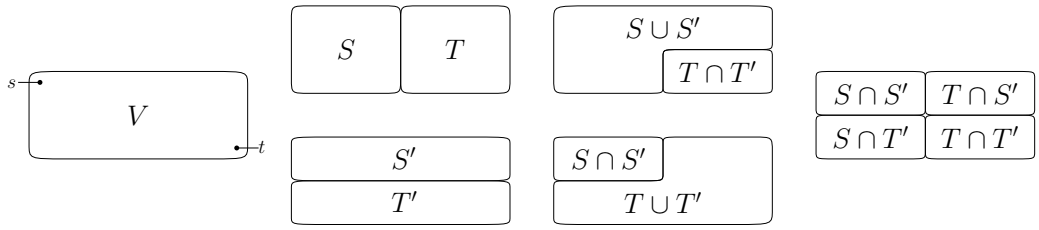
a) Seien (S, T) und (S', T') zwei minimale (s, t) Schnitte in einem Flußgraphen G. Zeigen oder widerlegen Sie, dass $(S \cup S', T \cap T')$ und $(S \cap S', T \cup T')$ auch minimale (s, t) Schnitte sind.

b) Sei (S, T) ein minimaler (s, t) Schnitt in einem Flussgraphen G. Zeigen oder widerlegen Sie, dass (S, T) ein minimaler (x, y) Schnitt ist f.a. $(x, y) \in S \times T$.

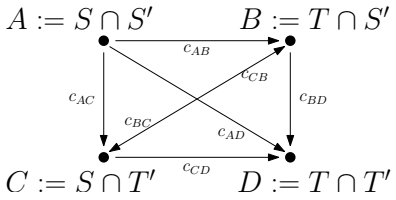
c) Zeigen Sie, dass für den *preflow-push Algorithmus* aus der Vorlesung mit beliebiger Wahl des nächsten Knotens $\mathcal{O}(n^2)$ die bestmögliche obere Schranke ist.

Musterlösung:

a) Der Graph wird in vier Bereiche aufgeteilt, $S \cap S'$, $T \cap S'$, $T \cap T'$ und $S \cap T'$ (siehe Abbildung).



Zur Anschauung definiert man einen Graphen, dessen Knoten der obigen Aufteilung entsprechen und dessen Kanten die Schnitte zwischen den Bereichen darstellen.



Damit ergeben sich folgende Werte für die Schnitte:

- (S, T) Schnitt: $c_{ST} = c_{AB} + c_{AD} + c_{CB} + c_{CD}$
- (S', T') Schnitt: $c_{S'T'} = c_{AC} + c_{AD} + c_{BC} + c_{BD}$
- $(S \cup S', T \cap T')$ Schnitt: $c_{S \cup S', T \cap T'} = c_{AD} + c_{BD} + c_{CD}$
- $(S \cap S', T \cup T')$ Schnitt: $c_{S \cap S', T \cup T'} = c_{AB} + c_{AC} + c_{AD}$

Da (S, T) und (S', T') minimale Schnitte sind, gilt $c_{ST} = c_{S'T'}$. Außerdem sind $c_{S \cup S', T \cap T'}$ und $c_{S \cap S', T \cup T'}$ jeweils größer gleich c_{ST} bzw. $c_{S'T'}$. Löst man die sich ergebenden Ungleichungen, erhält man $c_{BC} = c_{CB} = 0$ und damit $c_{AB} = c_{BD}$, $c_{AC} = c_{CD}$ (Rechnung siehe nächste Seite).

Es ergibt sich $c_{S \cup S', T \cap T'} = c_{ST} = c_{S \cup S', T \cap T'} = c_{S'T'}$.

Musterlösung:

a) (fortgesetzt)
Auflösen der Ungleichungen:

$$\begin{aligned}
c_{S\cup S',T\cap T'} &\geq c_{ST} && (1)\\
c_{S\cup S',T\cap T'} &\geq c_{S'T'} && (2)\\
c_{S\cap S',T\cup T'} &\geq c_{ST} && (3)\\
c_{S\cap S',T\cup T'} &\geq c_{S'T'} && (4)
\end{aligned}$$

$$\begin{aligned}
c_{AD}+c_{BD}+c_{CD} &\geq c_{AB}+c_{AD}+c_{CB}+c_{CD} && (1)\\
c_{AD}+c_{BD}+c_{CD} &\geq c_{AC}+c_{AD}+c_{BC}+c_{BD} && (2)\\
c_{AB}+c_{AC}+c_{AD} &\geq c_{AB}+c_{AD}+c_{CB}+c_{CD} && (3)\\
c_{AB}+c_{AC}+c_{AD} &\geq c_{AC}+c_{AD}+c_{BC}+c_{BD} && (4)
\end{aligned}$$

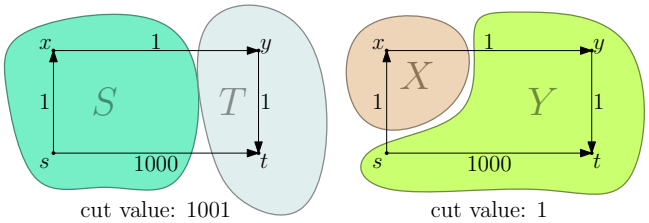
$$\begin{aligned}
c_{BD} &\geq c_{AB}+c_{CB} && (1)\\
c_{CD} &\geq c_{AC}+c_{BC} && (2)\\
c_{AC} &\geq c_{CB}+c_{CD} && (3)\\
c_{AB} &\geq c_{BC}+c_{BD} && (4)
\end{aligned}$$

Setze (2) in (3) ein und erhalte $c_{AC} \geq c_{CB}+c_{AC}+c_{BC} \Leftrightarrow 0 \geq c_{CB}+c_{BC}$. Da Kapazitäten nicht negativ sein können, gilt $c_{CB} = c_{BC} = 0$. Dies eingesetzt in die anderen Formeln liefert

$$\begin{aligned}
c_{BD} &\geq c_{AB} && (1)\\
c_{CD} &\geq c_{AC} && (2)\\
c_{AC} &\geq c_{CD} && (3)\\
c_{AB} &\geq c_{BD} && (4)
\end{aligned}$$

und damit $c_{AB} = c_{BD}$, $c_{AC} = c_{CD}$.

b) Unten abgebildetes Flussnetzwerk mit dem eingezeichneten Schnitt ist ein Gegenbeispiel. Links ist ein minimaler (s,t) Schnitt mit Wert 1001 abgebildet. Der minimale (x,y) Schnitt mit dem Wert 1 ist rechts zu sehen.

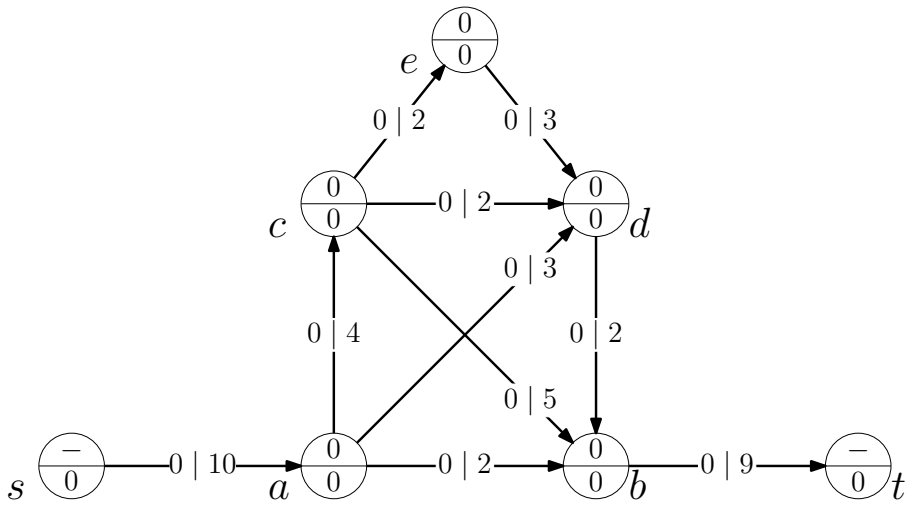


c) Wir betrachten folgenden Graphen:

Um einen Fluss auf diesem Graphen zu berechnen, muss der Fluss einmal durch den ganzen Graphen und wieder zurückfließen. Damit der zusätzliche Fluss vom vorletzten Knoten wieder in die Quelle zurückfließen kann, muss der Knoten mindestens Level $n+1$ haben. Somit ergibt sich über alle Knoten $\#relabel \geq \sum_{i=1}^{n-1} n+1 = n^2-1$.

Aufgabe 2 (*Rechnen: preflow-push Algorithmus*)

Gegeben sei folgender Flussgraph:



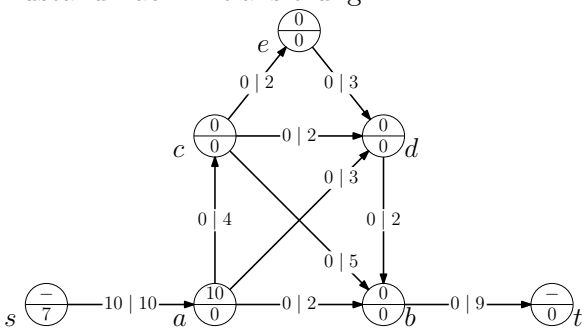
(das Haus vom Nikolaus mit Vorgarten und Hinterhof)

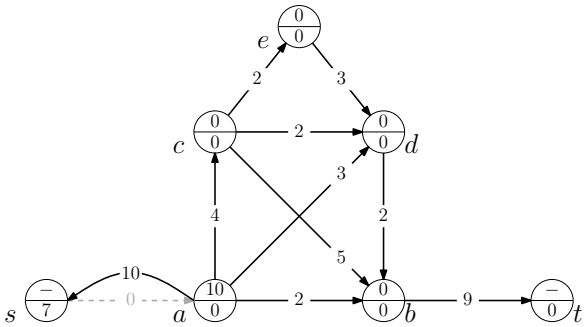
Knotenbeschriftung: Level (unten), Überschuss (oben)
Kantenbeschriftung: Fluss (vorne), Kapazität (hinten)

Bestimmen Sie den maximalen Fluss von s nach t mit dem generischen *preflow-push* Algorithmus.

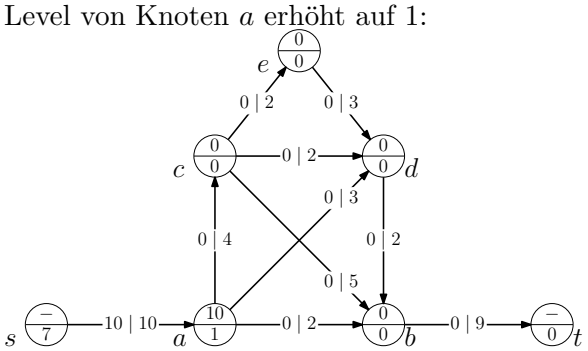
Musterlösung:
Im Folgenden wird der *preflow-push* Algorithmus aus der Vorlesung auf den Flussgraphen angewendet. Aktive Knoten werden zufällig ausgewählt. Es wird bei einem Knoten geblieben bis dessen gesamter Überschuss weggeschoben wurde. Dieser Ablauf ist *nicht* der schnellstmögliche!
Links ist der Zustand des Flussgraphen nach jedem Schritt zu sehen, rechts der des Residualgraphen.

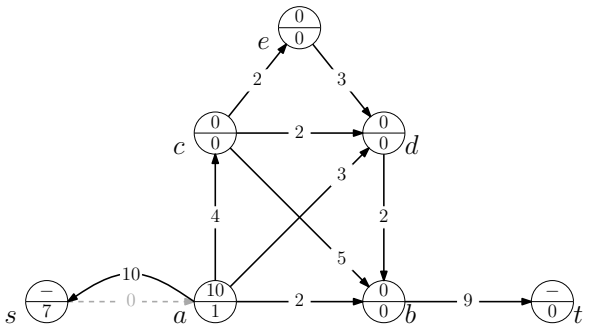
Zustand nach Initialisierung:





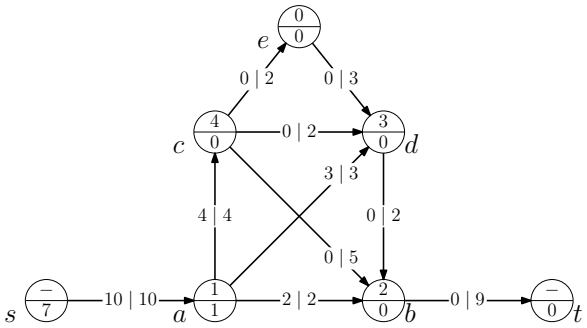
Level von Knoten a erhöht auf 1:

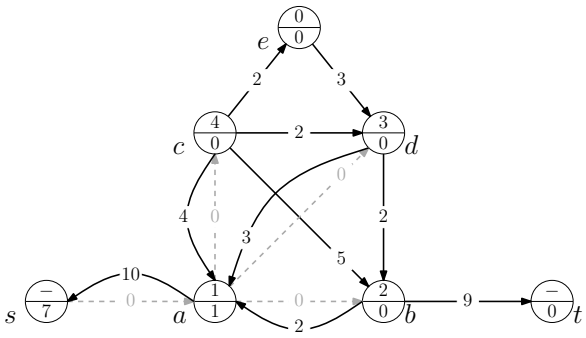




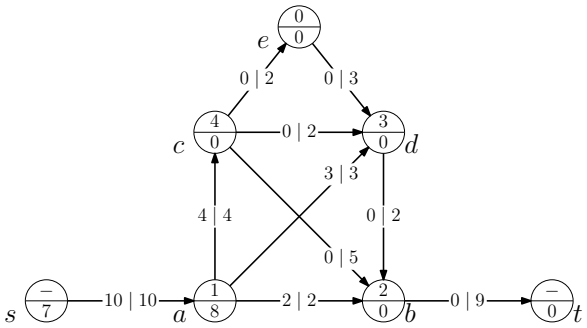
Musterlösung:

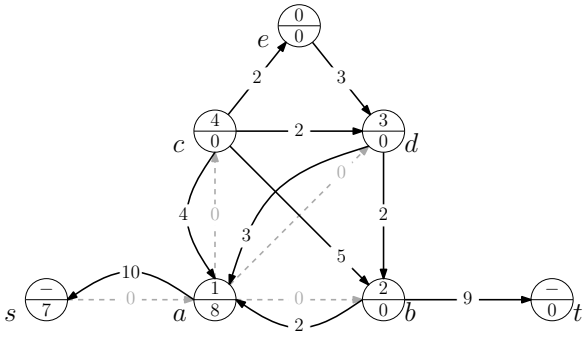
Fluss von a nach b, c und d geschoben:



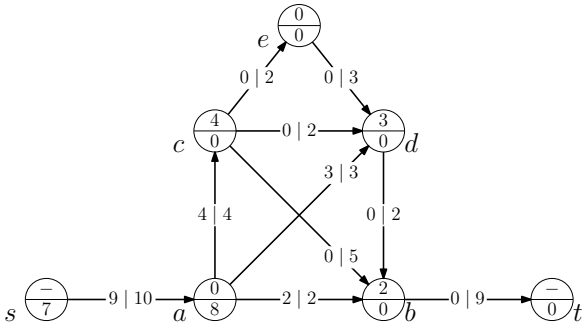


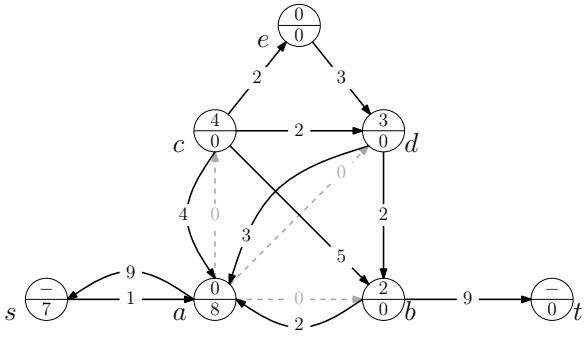
Level von Knoten a erhöht auf 8:



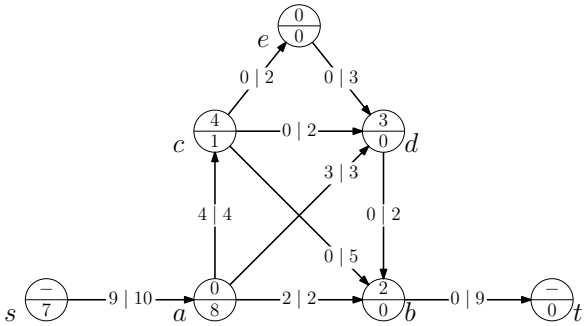


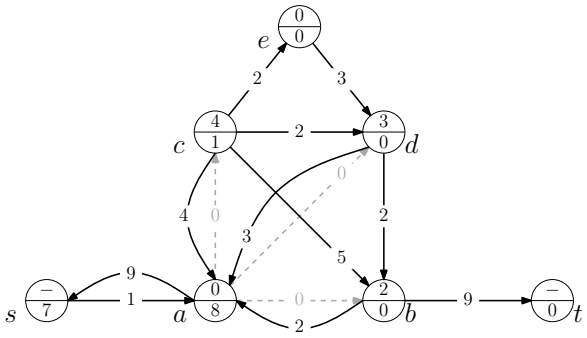
Fluss von a nach s geschoben:





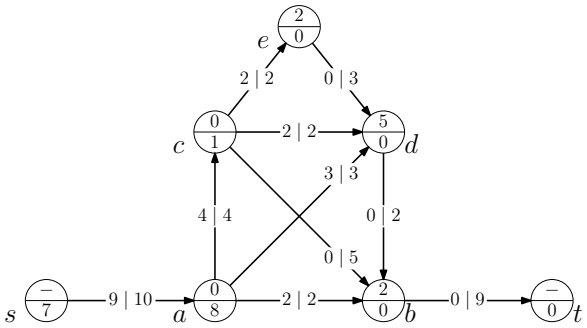
Level von Knoten c erhöht auf 1:

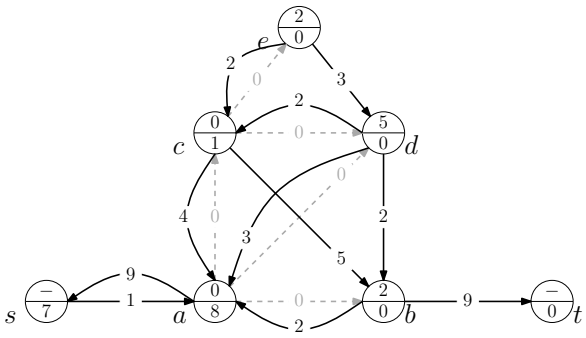




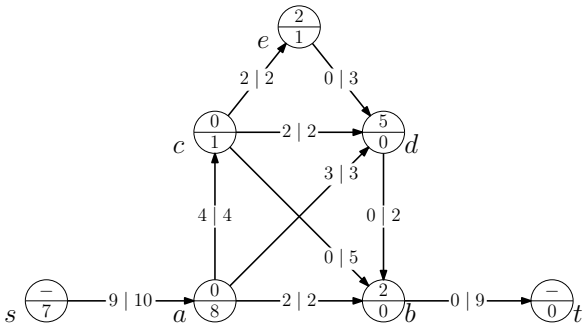
Musterlösung:

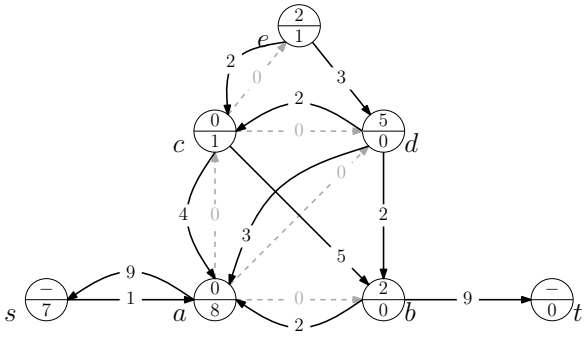
Fluss von c nach d und e geschoben:



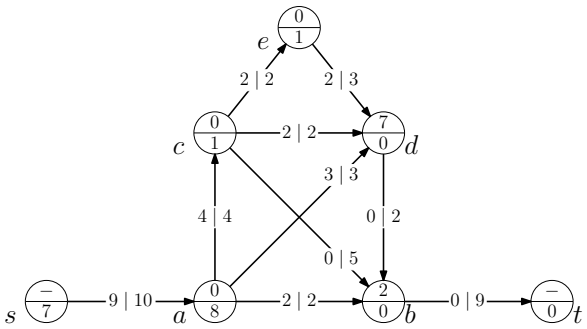


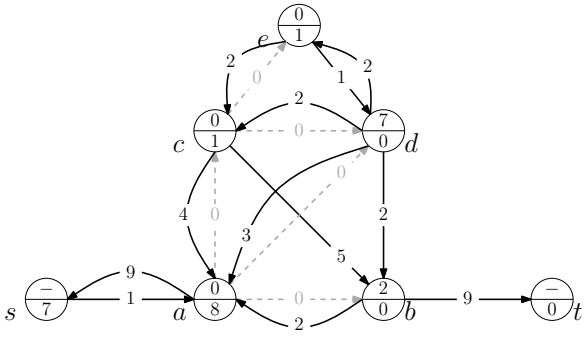
Level von Knoten e erhöht auf 1:



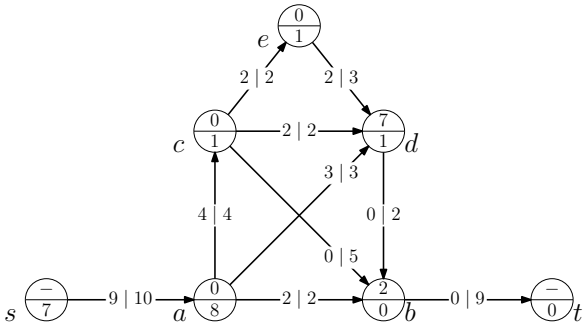


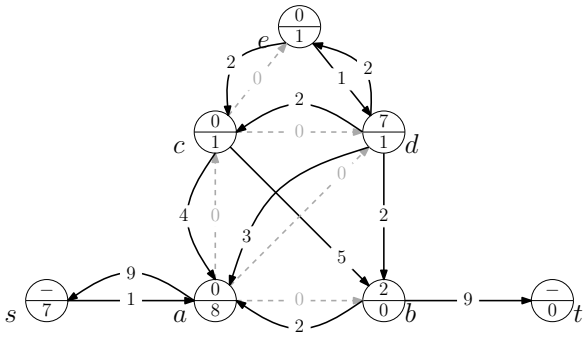
Fluss von e nach d geschoben:





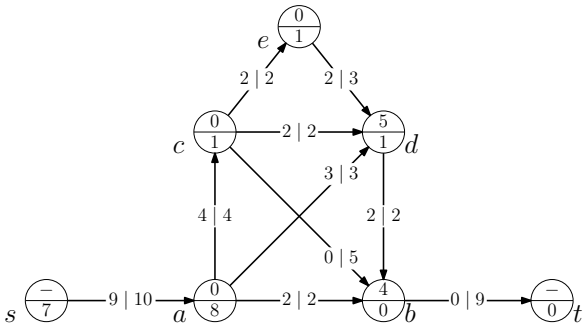
Level von Knoten d erhöht auf 1:

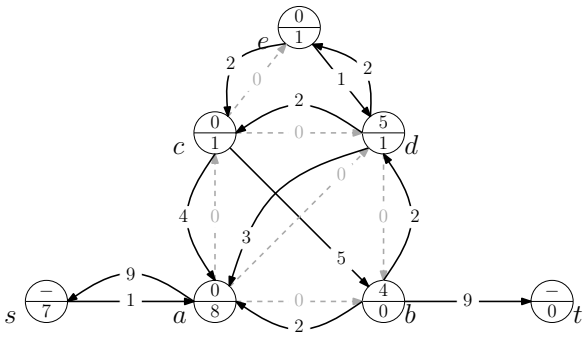




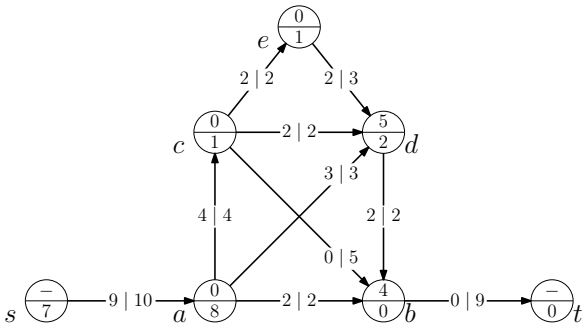
Musterlösung:

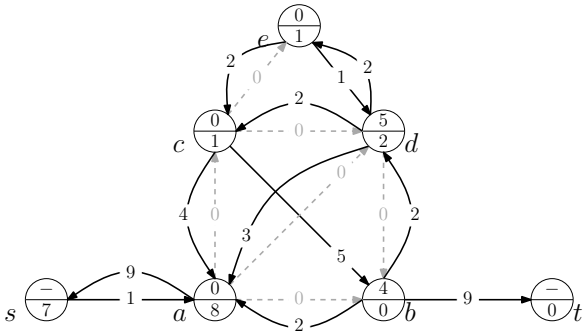
Fluss von d nach b geschoben:



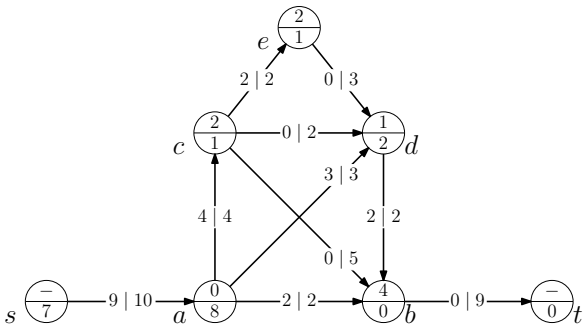


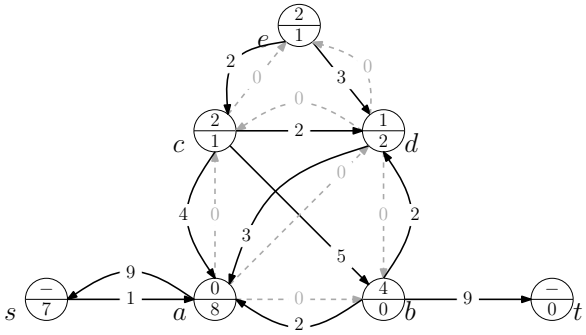
Level von Knoten d erhöht auf 2:



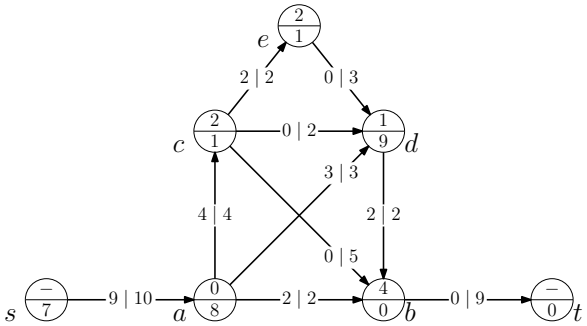


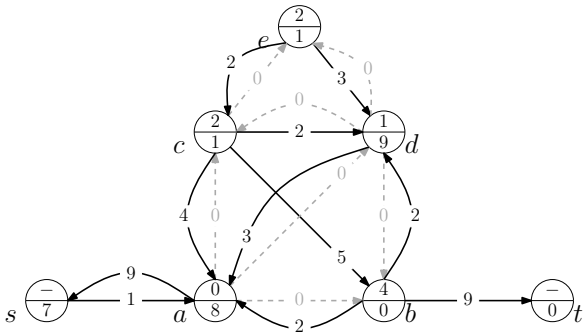
Fluss von d nach c und e geschoben:





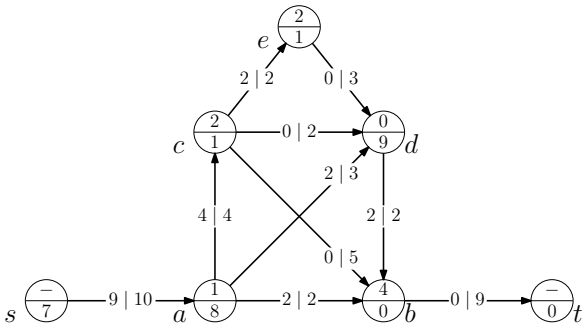
Level von Knoten d erhöht auf 9:

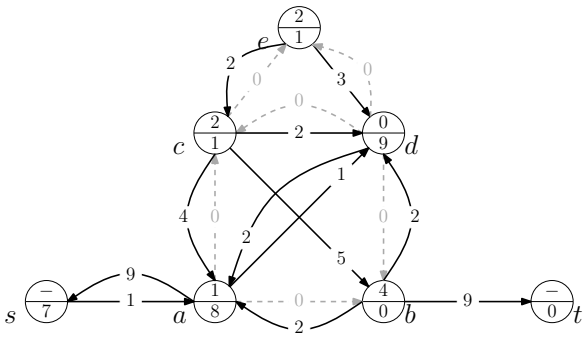




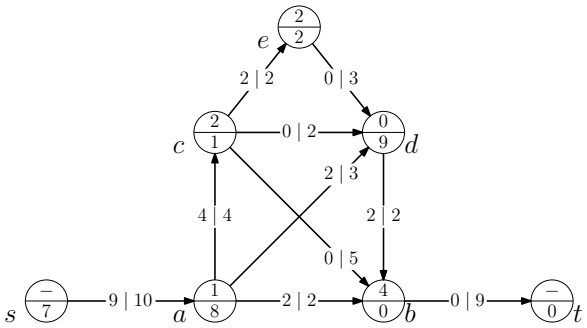
Musterlösung:

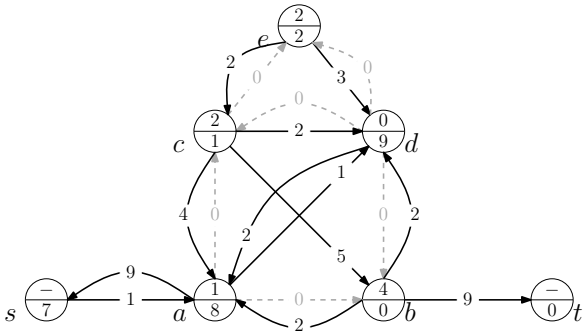
Fluss von d nach a geschoben:



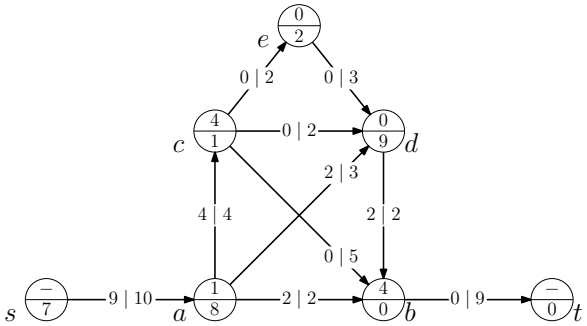


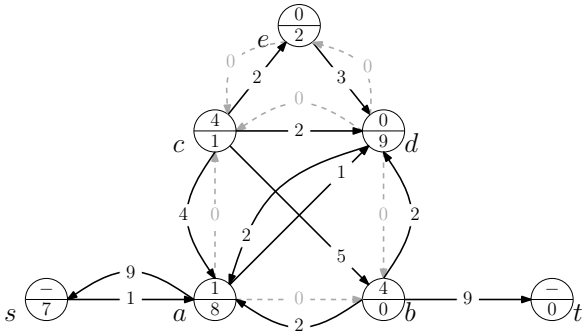
Level von Knoten e erhöht auf 2:



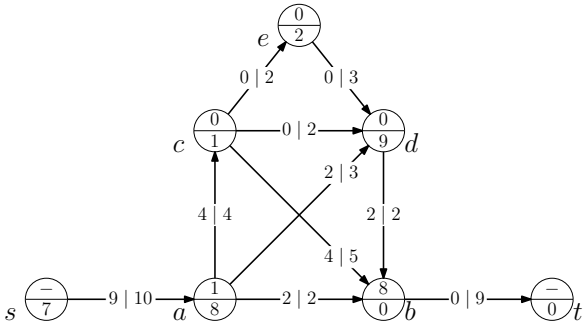


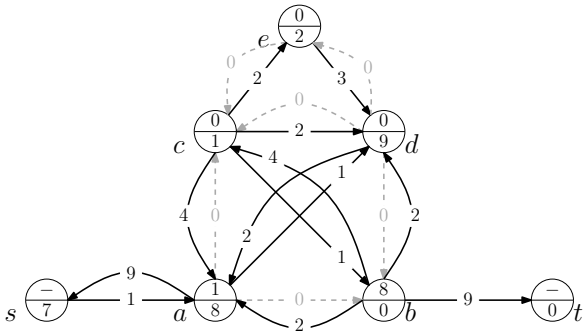
Fluss von e nach c geschoben:





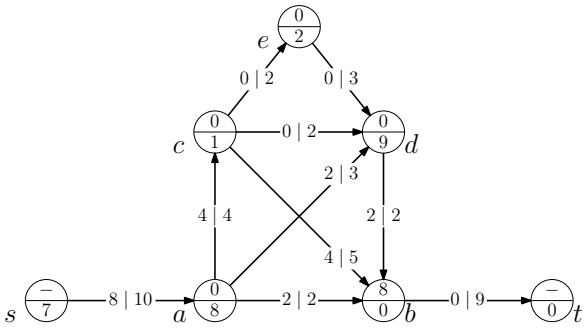
Fluss von c nach b geschoben:

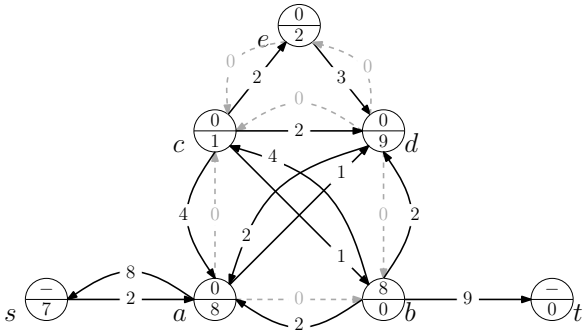




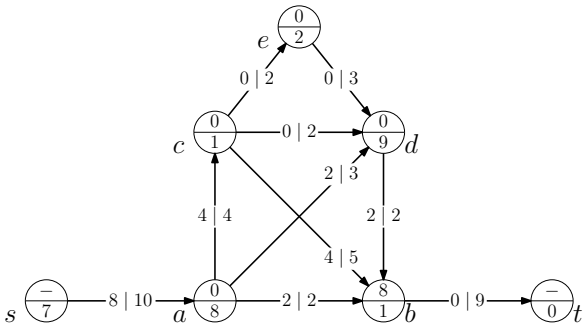
Musterlösung:

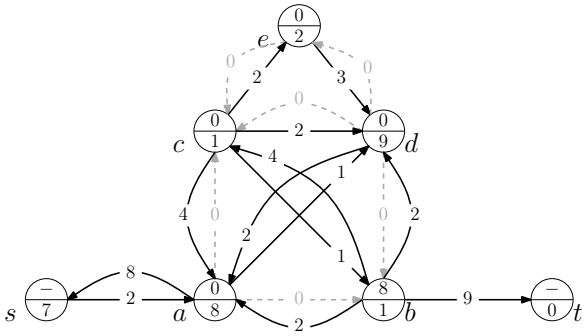
Fluss von a nach s geschoben:



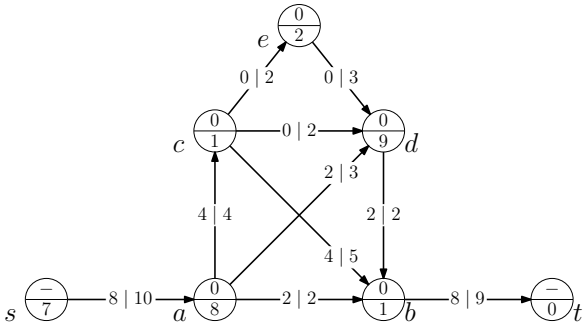


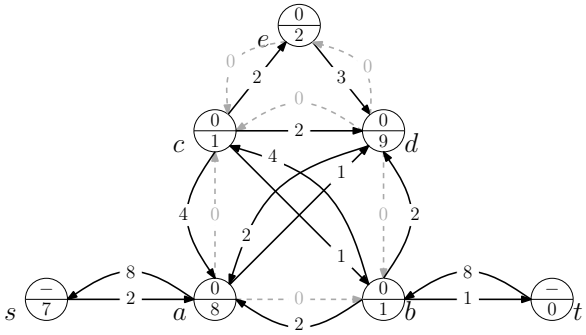
Level von Knoten b erhöht auf 1:





Fluss von b nach t geschoben:





Fertig!

Aufgabe 3 (*Entwurf+Analyse: Qualitätskontrolle)*)

Sie sind beauftragt worden, einen Algorithmus für die Abteilung zur Qualitätskontrolle zu entwerfen, der die Validierung der wöchentlichen Resultate übernimmt.
Am Ende jeder Woche erhalten Sie dafür eine Liste L der in dieser Woche produzierten Bauteile mit den drei Angaben: (Typ; ID des Bauteils selbst; ID des Bauteils, in dem es verbaut wurde). Zudem erhalten Sie eine Liste D mit den IDs aller Bauelemente, die in derselben Woche von der Qualitätskontrolle als defekt markiert worden sind. Beide Listen sind potentiell zu groß für den Hauptspeicher und enthalten die Daten in unsortierter Reihenfolge.
Ihre Aufgabe ist es zu überprüfen, ob alle Bauteile, die selbst defekte Bauteile enthalten, als defekt markiert worden sind und falls nicht, dies zu korrigieren. Zu Ihrer Übersicht steht Ihnen ein Schema zur Verfügung, aus dem man ablesen kann, welche Bauteiltypen in welchen anderen verbaut werden. Dieses Schema passt in den Hauptspeicher.

a) Erweitern Sie Liste L um die Angabe aus Liste D, ob das jeweilige Bauteil defekt ist. Sie können davon ausgehen, dass diese Information keinen zusätzlichen Speicher benötigt.

b) Definieren Sie eine totale Ordnung $\prec_b$ auf den Bauteilen, die sich für jedes Bauteil aus lokalen Informationen und dem Bauteilschema berechnen lässt. Die Ordnung soll dabei erfüllen, dass in einer nach $\prec_b$ sortierten Liste L jedes Bauteil nach allen in ihm verbauten Bauteilen steht.

c) Verwenden Sie die sortiere und um Angaben zu Defekten erweiterte Liste L, um alle Bauteile zu identifizieren, die defekte Bauteile enthalten.

   Hinweis: Verwalten Sie die als defekt identifizierten aber noch zu betrachtenden Bauteile in einer geeigneten Datenstruktur.

Musterlösung:

a) Sortiere L und D nach der ID der Bauteile und scanne beide Listen. Erhöhe dazu jeweils den Index der Liste, der auf das Bauteil mit der kleineren ID zeigt. Zeigen beide auf die gleiche ID, setze das Defekt-Flag und erhöhe beide Indizes.

b) Erstelle aus den Abhängigkeiten der Bauteiltypen einen DAG. Dies ist möglich, da kein Bauteil in sich selbst verbaut worden sein kann. Sortiere L nach der durch den DAG induzierten Teilordnung, verwende die ID der Bauteile als Sortierkriterium bei Unvergleichbarkeit.

c) Verwalte defekte Bauteile in einer *Ausschlußliste* in Form einer externen Prioritätswarteschlange (Schlüssel ist die ID des Bauteils, kleinere IDs haben höhere Priorität). Durchlaufe die sortierte Liste L. Falls beim Durchlaufen der Liste L ein defektes Bautteil t_0 gefunden wird und das defekte Bauteil in einem anderen Bauelement e_0 verbaut wurde, so füge das Bauelement e_0 in die Prioritätswarteschlange ein. Wird beim Durchlaufen der Liste L ein Bauteil t_1 gefunden, welches in der Prioritätswarteschlange die höchste Priorität hat, so wird das Bauteil t_1 ebenfalls als defekt markiert. Ist das Bauteil t_1 in einem anderen Bauelement e_1 verbaut, so füge das Bauelement e_1 in die Prioritätswarteschlange ein.

Aufgabe 4 (*Analyse: Speicherbandbreite (*)*)

Die Effizienz eines Algorithmus, der auf externem Speicher arbeitet, hängt von der gewählten Blockgröße B in Zusammenspiel mit der maximalen Bandbreite W_{max} und der durchschnittlichen Zugriffszeit T_{seek} des externen Speichers ab.
Bestimmen Sie für die folgenden Fälle die Blockgröße, für die 90% der maximalen Bandbreite ausgereizt werden kann. Sie können davon ausgehen, dass ohne Unterbrechung auf ganze Blöcke in zufälliger Reihenfolge zugegriffen wird. Etwaige Berechnungen können als asynchron angenommen werden. Daher muss für diese keine Zeit berücksichtigt werden.

a) $W_{max} = 144$ MByte$/s$, $T_{seek} = 12ms$ (*Lesen von Festplatte*)

b) $W_{max} = 550$ MByte$/s$, $T_{seek} = 100\mu s$ (*Lesen von SSD*)

c) $W_{max} = 68$ MByte$/s$, $T_{seek} = 60s$ (*Lesen von LTO Streamer*)

Musterlösung:
Ein Block kann in Zeit $T = T_{seek} + B/W_{max}$ eingelesen werden.
Mit der effektiven Bandbreite $W = B/T \stackrel{!}{=} 0.9 \cdot W_{max}$ ergibt sich

$$B = 9 \cdot W_{max} \cdot T_{seek}$$

für die gesuchte Blockgröße. Damit folgt:

a) $B = 15.552$ MByte ≈ 15 MByte

b) $B = 0.495$ MByte ≈ 500 kByte

c) $B = 36\,720$ MByte ≈ 37 GByte

Aufgabe 5 (*Analyse: Externer Stack*)

In der Vorlesung wurde eine Implementierung von *Stack* als externe Datenstruktur vorgestellt. Eine äquivalente Implementierung besitzt folgende Struktur: Im Speicher wird ein Puffer P der Größe $2B$ gehalten – B sei die Blockgröße beim Zugriff auf externen Speicher. Der Puffer ist in Form eines (internen) Stacks organisiert und enthält die neuesten gespeicherten Elemente. Folgende Operationen sind für die externe Datenstruktur definiert:

`pop` Falls P nicht leer, entferne das neueste Element aus P. Ansonsten, lese einen Block ein, um die Hälfte von P zu füllen bevor `pop` auf P ausgeführt wird.

`push` Falls P nicht voll, füge das neue Element direkt zu P. Ansonsten, schreibe die ältere Hälfte von P in den externen Speicher und verschiebe die aktuellere Hälfte an diese Stelle im Speicher. Anschließend führe ein `push` auf P aus.

Für die Analyse können Sie davon ausgehen, dass ein Block B Elemente des Stacks halten kann.

a) Zeigen Sie, dass die Operationen `push` und `pop` amortisiert $O(1/B)$ I/O Operationen benötigen.

b) Warum genügt es nicht, nur einen Puffer mit Größe B zu verwenden?

Musterlösung:

a) Betrachte die minimale Anzahl an Operationen (`push` oder `pop`) bis zur nächsten I/O Operation: Nach einer I/O Operation ist die Hälfte des Puffers leer – bei einem `push` auf den vollen Puffer wurde die Hälfte in den externen Speicher verlagert bzw. bei einem `pop` auf einen leeren Puffer wurde der halbe Puffer aufgefüllt. Von diesem Zustand ausgehend werden mindestens B Operationen einer Art ausgeführt, bevor erneut ein I/O Zugriff erfolgt – entweder, um bei einem `push` Daten in den externen Speicher zu schreiben, da der Puffer voll ist, oder um bei einem `pop` Daten aus dem externen Speicher zu laden, weil der Puffer leer ist.

b) Bei nur einem Puffer der Größe B könnte man sich folgende maximal schlechte Folge an Operationen überlegen: $B + 1$ `push` Operationen, gefolgt von einer Reihe von je 2 `pop` und 2 `push` Operationen. Jeweils die zweite dieser Anweisungen löst eine I/O Operation aus, da das `pop` auf einem leeren und das `push` auf einem vollen Puffer stattfindet. Amortisiert ergeben sich $O(1)$ I/O Operationen.

Aufgabe 6 (*Entwurf+Analyse: Telekommunikationsgesellschaft*)

Eine Telekommunikationsgesellschaft beauftragt Sie eine Anwendung zu schreiben, die monatlich die k Kunden bestimmt, bei denen sich die Rechnung im Vergleich zum Vormonat am meisten verändert hat. Diese Kunden will sich die Telekommunikationsgesellschaft noch einmal genau anschauen, um ihnen eventuell einen neuen Vertrag anzubieten.
Die zu bearbeitenden Daten werden Ihnen auf (langsamen) Bandspeichern zur Verfügung gestellt. Sie erhalten eine Liste mit den aneinandergefügten Datensätzen jeder Zweigstelle ihres Auftraggebers für den aktuellen Monat. Außerdem haben Sie eine entsprechende Liste für den Vormonat zur Verfügung. Gespeichert sind jeweils Tupel $(Kundennummer, Kosten)$.

a) Geben Sie einen Algorithmus an, der die geforderte Aufgabe erfüllt. Geben Sie außerdem die Laufzeit Ihres Algorithmus an und begründen Sie diese. Sie können davon ausgehen, dass die k zu bestimmenden Kunden in den Hauptspeicher passen.

b) Seien nun die k zu bestimmenden Kunden zu groß, um im Hauptspeicher gehalten zu werden. Ändern Sie Ihren Algorithmus so ab, dass er mit der erhöhten Datenmenge zurecht kommt. Geben Sie die Laufzeit Ihres neuen Algorithmus an und begründen Sie diese.

Hinweis: Diese Aufgabe war ursprünglich für die Klausur vorgesehen.

Musterlösung:

a) In einem ersten Schritt werden beide Listen nach aufsteigender Kundennummer sortiert mit externem MergeSort. Anschließend scannt der Algorithmus linear über beide sortierte Listen M, N. Zu diesem Zweck wird für jede Liste ein Zeiger i_M bzw. i_N mit der aktuellen Position gespeichert und ein Teil der Liste mit Größe B im Speicher gehalten, der bei Bedarf durch den folgenden ersetzt wird. Beide Zeiger starten am Anfang der jeweiligen Liste. Stimmen die Kundennummern von $M[i_M]$ und $N[i_N]$ überein, wird die Differenz ihrer Kosten gebildet. Diese wird in einer lokalen Prioritätswarteschlange PQ gespeichert. Hat PQ nach dem Einfügen mehr als k Elemente, so wird das kleinste entfernt. Stimmen die Kundennummern M_{i_M} und N_{i_N} nicht überein, wird der Zeiger um eins erhöht, der auf den Eintrag mit der kleineren Kundennummer zeigt. Nachdem die kürzere von beiden Listen vollständig abgearbeitet wurde, enthält PQ die gesuchten Elemente.

   Die Laufzeit wird von den benötigten I/O Operationen dominiert. Sei $n := |N| + |M|$ und H die Größe des Hauptspeichers. Sortieren benötigt $\Theta(\frac{n}{B} log_{\frac{H}{B}} \frac{n}{H})$, der lineare Scan über beide Listen $O(n/B)$ I/O Operationen. Der gesamte Algorithmus ist also vom Sortieren dominiert.

   Für die Blockgröße B beim linearen Scan gilt: $B < (S - \text{maximaler Speicher für PQ})/2$, wobei S den verfügbaren Hauptspeicher angibt. Die Blockgröße beim MergeSort kann größer gewählt werden, da die PQ zu diesem Zeitpunkt noch leer ist. Dies ändert die Anzahl I/O Operationen aber nur um einen konstanten Faktor.

b) Verwende eine externe Prioritätswarteschlange statt einer internen. Diese benötigt bis zu $O(\frac{n}{B} \log_{\frac{H}{B}} \frac{n}{H})$ I/O Operationen, falls jeder Wert eingefügt werden muss (Werte löschen ist amortisiert kostenlos). Dies entspricht der Anzahl, die für das initiale Sortieren benötigt wird. Die Laufzeit ändert sich also nicht.

   Für die Blockgrößen gilt die gleiche Argumentation wie in der ersten Teilaufgabe.

Aufgabe 7 (*Implementierung: Preflow Push*)

Implementieren Sie den Algorithmus "Preflow Push" in `C++` mit dem KaHIP Framework. Laden Sie sich den KaHIP Fork[1] herunter und wechseln Sie in den Branch PreflowPush. Das KaHIP Framework stellt bereits einen Flussgraph (`lib/data_structure/flow_graph.h`) bereit. Diese Datenstruktur implementiert die Basisfunktionen von Residualgraphen. Die Funktionalität des Residualgraphen wurde bereits um folgende Funktionen erweitert:

```
// set flow of forward and reverse edge
void updateEdgeFlow(NodeID source, EdgeID e, FlowType flow);
FlowType getEdgeResCapacity(NodeID source, EdgeID e) const ;
bool isSaturated(NodeID source, EdgeID e) const;
// increases edge flow of edge and decreases flow of reverse edge
void increaseEdgeFlow(NodeID source, EdgeID e, FlowType flow);
bool isValidEdge(NodeID v, EdgeID e) const;
EdgeID get_first_flow_edge(NodeID node);
EdgeID get_next_flow_edge(NodeID node, EdgeID e);
```

Implementieren Sie die fehlenden Methoden der Klasse `PreflowPush` in der Datei `app/preflow_push/preflow_push.cpp`. Diese Klasse initialisiert bereits alle Datenstrukturen für den Algorithmus "Preflow Push". Weiter sind folgende Hilfsfunktionen ebenfalls schon implementiert:

```
EdgeID get_next_eligible_edge(const flow_graph& G, NodeID v);
void relabel(NodeID v);
// Pushes flow over edge e in the first phase.
// If the target node gets activated, we add the target node to
// the container small_active_nodes_ or the container active_nodes_
// depending on the level of the target node.
void low_level_push(flow_graph& G, NodeID v, EdgeID e, FlowType flow);
void push(flow_graph& G, NodeID v, EdgeID e, FlowType flow);
// Returns maximal residual flow based on remaining access.
FlowType max_allowed_res_flow(const flow_graph& G, NodeID v, EdgeID e) const;
FlowType get_outgoing_flow(const flow_graph& G, EdgeID e) const;
```

Implementieren Sie nun die Methode

```
// Executes the preflow push algorithm for nodes on arbitrary levels
// stored in the container active_nodes_.
FlowType remaining_level_preflow_push(flow_graph& G, NodeID source, NodeID sink);
```

Diese Methode soll in jeder Iteration einen aktiven Knoten v aus dem Container `active_nodes_` entnehmen und Überschluss an Fluss im Knoten v über "eligible" Kanten fließen lassen. Sobald es keine "eligible" Kanten mehr gibt, aber weiterhin Überschluss an Fluss im Knoten v existiert, wird der Level von Knoten v erhöht und wieder zu den aktiven Knoten hinzugefügt (Fifo Heuristik). Rufen Sie am Ende die Methode `get_next_flow_edge` auf, um den ausgehenden Fluss vom Startknoten zu berechnen (und somit den maximalen Fluss). Compilieren Sie Ihre Implementierung danach mithilfe des Skripts `compile.sh` und führen Sie den Algorithmus auf der Kommandozeile aus:

```
> ./optimized/preflow ./examples/flow_graphs/af_shell9.graph.hierarchy.5.h.inp
```

Aktivieren Sie nun die Heuristik "highest level first" indem sie die Codezeile

```
#define HIGHEST_LEVEL_FIRST
```

einkommentieren. Passen Sie Ihre Implementierung so an, dass der Algorithmus "Preflow Push" mit der Fifo Heuristik, aber auch mit der "highest level first" Heuristik maximale Flüsse berechnet. Führen Sie den Algorithmus mit beiden Heuristiken und verschiedenen Flussgraphen (`./examples/flow_graphs/`) aus. Wie unterscheidet sich die Laufzeit? Vergleichen Sie ebenfalls die Laufzeit vom Algorithmus "Preflow Push" mit der Laufzeit vom Algorithmus von Ford Fulkerson (`> ./optimized/maxflow`).

[1] `https://github.com/MichaelAxtmann/KaHIP`

Aktivieren Sie nun zusätzlich die Heuristik "aggressive local relabeling" indem Sie die Codezeile

```
#define AGGRESSIVE_RELABELING
```

einkommentieren. Wie ändert sich die Laufzeit vom Algorithmus "Preflow Push"?

Um zusätzlich die Heuristik "two phase preflow push" aktivieren zu können (`TWO_PHASE_APPROACH`), müssen Sie die Methode

```
void low_level_preflow_push(flow_graph& G, NodeID source, NodeID sink);
```

implementieren. Diese Methode erfüllt die gleiche Funktionalität wie die Methode

```
FlowType remaining_level_preflow_push(flow_graph& G, NodeID source, NodeID sink);
```

prozessiert jedoch ausschließlich Knoten aus dem Container `small_active_nodes_`. Knoten aus diesem Container liegen auf Level $0 \ldots n-1$. Überschreiten Knoten Level $n-1$, so sollen diese im Container `active_nodes_`, zur späteren Bearbeitung durch die Methode `remaining_level_preflow_push`, abeglegt werden. Führen Sie den Algorithmus nun gleichzeitig mit allen drei Heuristiken "aggressive local relabeling", "two phase preflow push" und "highest level first" aus. Vergleichen Sie die Laufzeiten von dieser Variante mit den bisherigen Laufzeiten. Welche Heuristik/Heuristiken bewähren sich?

Bis zur Veröffentlichung der Musterlösung können Studenten Implementierungen mit Heuristiken "highest level first" und/oder "two phase preflow push" per E-Mail bei Übungsleiter einreichen. Die ersten zehn Studenten, dessen Implementierungen ähnlich schnell oder schneller als die Musterlösung (gleiche Heuristik) sind, werden belohnt :) Sind die Implementierung höchstens 20% langsamer als die Musterlösung, so bekommt der Student eine Tafel Ritter Sport! Ist die Implementierung schneller als die Musterlösung, so bekommt der Student eine weitere Tafel Ritter Sport! Verglichen wird die Laufzeit der Instanz `af_shell9.graph.hierarchy.5.h.inp`. Als Referenzrechner dient ein Laptop mit einem i5-5200U Prozessor (4 Kerne mit jeweils 2.20GHz). Tabelle 1 zeigt die Referenzzeiten.

Tabelle 1: Referenzzeiten.

Heuristics	Time [ms]
AR+HLF+2FA	12771
AR+2FA	15833
AR+HLF	13364

Musterlösung:
Eine Referenzimplementierung ist im Branch PreflowPushSolution bereitgestellt.