Robert Elsässer
u.v.a.

Paderborn, den 19. Juni 2008

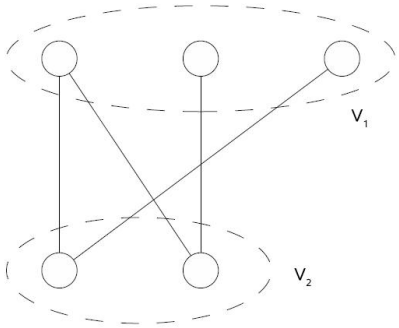
Beispiellösung zu den Übungen

Datenstrukturen und Algorithmen

SS 2008

Blatt 10

AUFGABE 1 (6 Punkte)**:**
Sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph. G heißt *bipartit*, falls sich V in zwei disjunkte Teilmengen V_1, V_2 mit $V = V_1 \cup V_2$ und $V_1 \cap V_2 = \emptyset$ zerlegen lässt, so dass für alle Kanten $(u, v) \in E$ gilt: $(u \in V_1 \wedge v \in V_2) \vee (v \in V_1 \wedge u \in V_2)$
Ein Beispiel eines bipartiten Graphen ist in der folgenden Abbildung dargestellt:



Gesucht ist ein Algorithmus, der bei Eingabe eines ungerichteten, zusammenhängenden Graphen G in Adjazenzlistendarstellung in Zeit $\mathcal{O}(|V| + |E|)$ bestimmt, ob der Graph bipartit ist oder nicht.

a) Beschreiben Sie einen Algorithmus, der das beschriebene Problem löst.

Lösung: (3 Punkte)
Wir modifizieren die Tiefensuche so, dass sie jedem Knoten eine der beiden Mengen V_1 bzw. V_2 zuordnet. Dabei ordnet die Tiefensuche zunächst dem ersten Knoten V_1 zu. Den Nachbarknoten eines Knotens der Menge V_1 wird dann jeweils, sofern diese das erste Mal besucht wurden, die Menge V_2 zugeordnet und vice versa.
Sollte ein Knoten v nun einer Menge zugeordnet sein und es wird festgestellt, dass ein Nachbarknoten, der bereits besucht wurde, in der selben Menge wie v ist, so wird ausgegeben, dass der Graph nicht bipartit ist. Werden hingegen keine solchen Konflikte festgestellt, so ist der Graph offensichtlich bipartit.

Pseudocode:

DFS(G):

```
1     for each vertex u ∈ V do
2         color[u] ← WHITE
3         π[u] ← nil
4     time ← 0
5     G_bipartit ← true
6     for each vertex u ∈ V do
7         if color[u] = WHITE then
8             BIP_set[u] ← V_1
9             if DFS-Visit(u) = false then
10                G_bipartit ← false
11    return G_bipartit
```

DFS-Visit(u):

```
1     color[u] ← GRAY
2     time ← time + 1
3     d[u] ← time
4     for each vertex v ∈ Adj[u] do
5         if color[v] = WHITE then
6             if BIP_set[u] = V_1 then
7                 BIP_set[v] = V_2
8             else
9                 BIP_set[v] = V_1
10            π[v] ← u
11            DFS-Visit(v)
12        else
13            if BIP_set[u] = BIP_set[v] then return false
14    color[u] ← BLACK
15    time ← time + 1
16    f[u] ← time
17    return true
```

b) Beweisen Sie die Korrektheit und analysieren Sie die Laufzeit Ihres Algorithmus.

Lösung: (3 Punkte)
Korrektheit:

Behauptung: G ist bipartit $\Leftrightarrow$ Algorithmus gibt G_bipartit = true aus.
Beweis per Fallunterscheidung:

Fall 1: G ist bipartit
zu Zeigen: Algorithmus gibt G_bipartit = true aus.

Es gilt: Algorithmus weist dem ersten Knoten V_1 zu. Jeder weitere Knoten hat eine vom Vaterknoten unterschiedliche Zuweisung (Zeile 5 - 9). Hierdurch wird eine Zuweisung in die gleiche Menge verhindert.
$\Rightarrow$ Jeder Knoten bekommt eine vom Vaterknoten verschiedene Menge zugewiesen.
$\Rightarrow$ der Algorithmus gibt in Zeile 11 korrekt G_bipartit = true aus.

Fall 2: G ist nicht bipartit
zu Zeigen: Algorithmus gibt G_bipartit = false aus.

Es gilt: G nicht bipartit
$\Rightarrow$ Es existiert ein Kreis C mit ungerader Kantenanzahl n + 1 und
gerader Knotenanzahl n.
$\Rightarrow$ Algorithmus weist n - 1 Knoten eine korrekte Mengenzuweisung zu. Der n. Knoten bekommt eine vom Vaterknoten verschiedene Zuweisung und wird besucht.
$\Rightarrow$ Sobald DFS-Visit auf den n. Knoten des Kreises aufgerufen wird, wird eine Kante zu einem Knoten in der gleichen Menge entdeckt.
$\Rightarrow$ Es existiert eine Kante zwischen zwei Knoten, die in der gleichen Menge sind.
$\Rightarrow$ Algorithmus gibt G_bipartit = false aus. ■

Laufzeit:

Die Modifikation der Tiefensuche beschränkt sich auf Schritte mit konstanter Laufzeit. Die asymptotische Gesamtlaufzeit ergibt sich somit aus der Laufzeit der Tiefensuche und ist folglich $\mathcal{O}(|V| + |E|)$.

AUFGABE 2 (6 Punkte)**:**
Beweisen Sie die folgenden Aussagen über Spannbäume. Wir betrachten dabei stets einen gewichteten, ungerichteten, zusammenhängenden Graphen $G = (V, E)$ mit Gewichtsfunktion $w : E \to \mathbb{N}$.

a) Sei $e \in E$ eine Kante, die auf keinem Kreis in G liegt (d.h. es existiert kein Kreis in G, der e enthält). Dann ist e in jedem minimalen Spannbaum von G enthalten.

Lösung: (3 Punkte)
Beweis durch Widerspruch: Angenommen, es existiert ein minimaler Spannbaum von G, der die Kante e nicht enthält. Sei $S = (V, E')$ ein solcher minimaler Spannbaum. Durch Hinzufügen der Kante e erhalten wir nun den Graphen $G' = (V, E'')$ mit $E'' = E' \cup \{e\}$. Da S ein Spannbaum und somit zusammenhängend ist, enthält G' nun einen Kreis $C \subseteq E''$ mit $e \in C$. Da $E' \subseteq E$ und $e \in E$ gilt nun $C \subseteq E'' = E' \cup \{e\} \subseteq E$, d.h. der Kreis C, der die Kante e enthält, ist auch in $G = (V, E)$ enthalten. $\Rightarrow$ Widerspruch, da e nach Voraussetzung auf keinem Kreis in G liegt. ■

b) Sei G ein Graph mit paarweise verschiedenen Kantengewichten. Dann existiert nur ein einziger minimaler Spannbaum von G (d.h. der min. Spannbaum von G ist eindeutig).

Lösung: (3 Punkte)
Beweis durch Widerspruch: Wir nehmen an, es existieren zwei verschiedene minimale Spannbäume $S_1 = (V, E_1)$, $S_2 = (V, E_2)$ mit $S_1 \neq S_2$.
Seien die Kanten $E_1 = \{e_{1_1}, e_{1_2}, \ldots, e_{1_{n-1}}\}$ und $E_2 = \{e_{2_1}, e_{2_2}, \ldots, e_{2_{n-1}}\}$ jeweils aufsteigend sortiert, d.h. es gilt: $\forall i, j : i < j \Rightarrow w(e_{1_i}) < w(e_{1_j}) \wedge w(e_{2_i}) < w(e_{2_j})$
Da $E_1 \neq E_2$ muss es einen Index geben, an dem sich E_1 und E_2 unterscheiden. Sei k der kleinste Index, an dem dies der Fall ist, d.h. sei $k := min\{i \mid 1 \leq i \leq n-1 \wedge e_{1_i} \neq e_{2_i}\}$.
Wegen $e_{1_k} \neq e_{2_k}$ gilt nach Voraussetzung $w(e_{1_k}) \neq w(e_{2_k})$.

O.B.d.A. gelte nun $w(e_{1_k}) < w(e_{2_k})$. Damit kann e_{1_k} nicht in E_2 enthalten sein (wäre dies der Fall, so müsste ein i existieren mit $e_{2_i} = e_{1_k}$ und $w(e_{2_i}) = w(e_{1_k}) < w(e_{2_k})$ - hieraus würde aufgrund der aufsteigenden Sortierung $i < k$ folgen, d.h. es würde $w(e_{2_i}) = w(e_{1_k}) > w(e_{1_i})$ gelten, was im Widerspruch zur Definition von k stünde).
In Worten: Wäre e_{1_k} in E_2 enthalten, so müsste dies aufgrund der aufsteigenden Sortierung einen kleineren Index als i haben, womit k nicht mehr der kleinste Index wäre, an dem sich E_1 und E_2 unterscheiden.

Wir konstruieren uns nun den Graphen $G' = (V, E')$ mit $E' = E_2 \cup \{e_{1_k}\}$. Da S_2 ein Spannbaum und somit zusammenhängend ist, enthält G' nun einen Kreis $C \subseteq E'$ mit $e_{1_k} \in C$. Durch Entfernen einer beliebigen anderen Kante e aus diesem Kreis C enthalten wir offensichtlich einen neuen Spannbaum. Wir unterscheiden daher zwei Fälle:

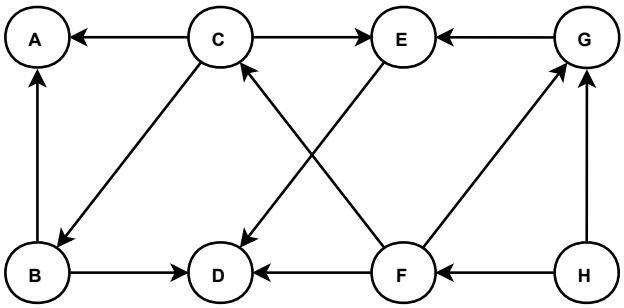
1. Fall: C enthält Kante e mit $w(e) > w(e_{1_k})$
   Damit bildet $S_2' = (V, E_2')$ mit $E_2' = E_2 \setminus \{e\} \cup \{e_{1_k}\}$ wieder einen Spannbaum. Wegen $w(e) > w(e_{1_k})$ hat S_2' jedoch geringeres Gewicht als S_2, was im Widerspruch zur Voraussetzung, dass S_2 ein minimaler Spannbaum ist, steht.
2. Fall: Für alle Kanten $e \in C$ gilt: $w(e) \leq w(e_{1_k})$

Da die Kantengewichte paarweise verschieden sind, gilt für alle $e \in C \setminus \{e_{1_k}\}$: $w(e) < w(e_{1_k})$. Aufgrund der Definition von k müssen damit aber alle Elemente aus $C \setminus \{e_{1_k}\}$ auch in E_1 enthalten sein. Da außerdem $e_{1_k} \in E_1$ gilt, würde damit $C \subseteq E_1$ gelten, d.h. S_1 würde einen Kreis enthalten, was im Widerspruch zur Voraussetzung steht, dass S_1 ein Spannbaum ist. *Anmerkung:* Man kann hier alternativ auch analog zum 1. Fall argumentieren, dass durch Entfernen einer Kante e mit $e \neq e_{1_k}$ nur Spannbäume erzeugt werden können, die ein größeres Gewicht als S_2 selbst haben und somit nicht minimal sind.

Beide Fälle führen zu einem Widerspruch, womit die Behauptung bewiesen ist. ■

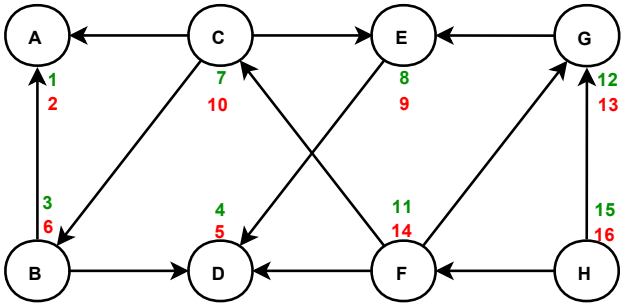
AUFGABE 3 (6 Punkte)**:**

a) Gegeben sei der folgende Graph $G = (V, E)$:



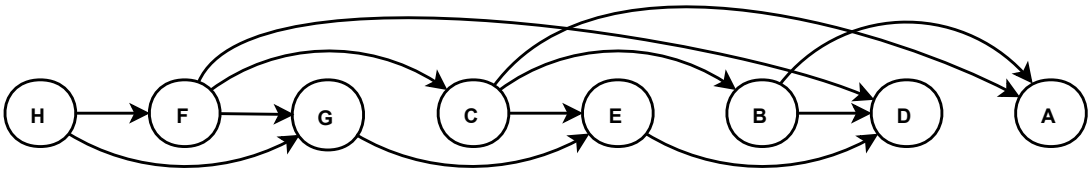
Geben Sie eine topologische Sortierung des Graphen nach Anwendung des TopSort-Algorithmus an. Führen Sie dazu zunächst eine Tiefensuche durch und geben Sie für jeden Knoten v die Discovery- und Finishing-Times ($d[v]$ bzw. $f[v]$) an.

Lösung: (3 Punkte)
Die Tiefensuche kann z.B. das folgende Ergebnis liefern:



Die Discovery-Times d sind in der Abbildung grün, die Finishing-Times f rot eingezeichnet. Um eine topologische Sortierung zu erhalten, müssen wir die Knoten nur noch absteigend nach Finishing-Times sortieren.

Die resultierende topologische Sortierung kann dann wie folgt aussehen:

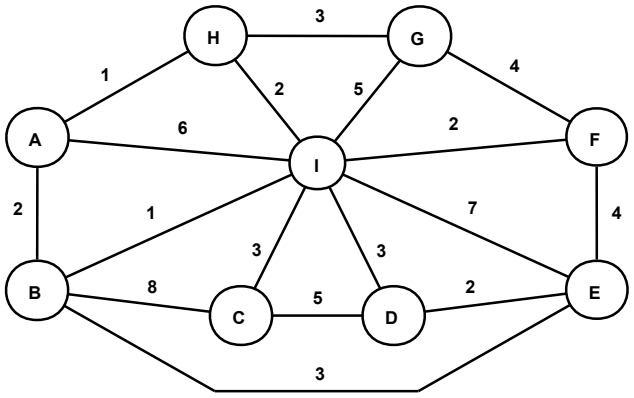


b) Wir betrachten den folgenden, halbformal beschriebenen Algorithmus, der zu einem gegebenen, gewichteten, ungerichteten, zusammenhängenden Graphen G und einem Startknoten r einen minimalen Spannbaum berechnet:

PRIM$(G = (V, E), r)$:

```
1   Q ← V − {r}
2   A ← ∅
3   while Q ≠ ∅ do
4       Finde Kante (u, v), u ∈ Q, v ∈ V − Q mit minimalem Gewicht
5       Q ← Q − {u}
6       A ← A ∪ {(u, v)}
7   return A
```

Sei nun der folgende gewichtete, ungerichtete, zusammenhängende Graph $G = (V, E)$ gegeben:

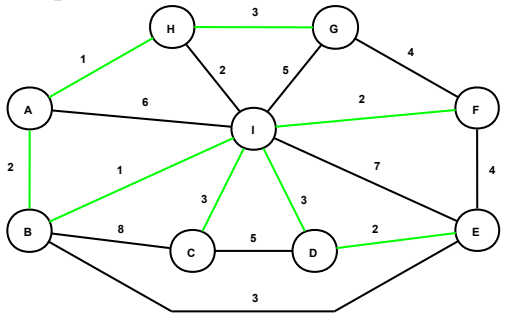


Erarbeiten Sie sich die Arbeitsweise des o.g. Algorithmus, indem Sie diesen auf den Graph G mit Startknoten A anwenden, um einen minimalen Spannbaum zu bestimmen. Geben Sie für jeden Zwischenschritt an, welche Kante dem Spannbaum hinzugefügt wird und nennen Sie außerdem jeweils den Inhalt der Queue zu Beginn jedes Schritts. Welche Größe hat der resultierende minimale Spannbaum?

Lösung: (3 Punkte)

Schritt	Queue	Kante
1	$\{B, C, D, E, F, G, H, I\}$	$\{A, H\}$
2	$\{B, C, D, E, F, G, I\}$	$\{A, B\}$
3	$\{C, D, E, F, G, I\}$	$\{B, I\}$
4	$\{C, D, E, F, G\}$	$\{F, I\}$
5	$\{C, D, E, G\}$	$\{G, H\}$
6	$\{C, D, E\}$	$\{C, I\}$
7	$\{D, E\}$	$\{D, I\}$
8	$\{E\}$	$\{D, E\}$

Graph:



Die Größe des minimalen Spannbaums beträgt 17.

AUFGABE 4 (6 Punkte)**:**
$G = (V, E)$ sei ein *gerichteter* Graph, dessen Knoten Computer und dessen Kanten Kommunikationsverbindungen sind. Für jede Kante $(u, v) \in E$ ist eine Kapazität $r(u, v) \in \mathbb{N}$ angegeben, welche die Geschwindigkeit, mit der Daten direkt vom Computer u zum Computer v gesendet werden können, in MBit/s angibt.
Gesucht ist ein effizienter Algorithmus, welcher die Geschwindigkeit der schnellsten möglichen Verbindung von einem gegebenen Computer x zu einem anderen gegebenen Computer y liefert. Nehmen Sie dabei an, dass die Kommunikation über beliebig viele Zwischenknoten möglich ist und die Geschwindigkeit der Gesamtverbindung genau gleich der Kapazität der langsamsten Leitung auf dem Pfad von x nach y ist.

a) Beschreiben Sie einen Algorithmus, der das beschriebene Problem löst.

Lösung: (3 Punkte)
Idee:

Die Idee ist, den Dijkstra-Algorithmus dahingehend zu modifizieren, dass dieser nicht einen Pfad findet, dessen Summe der Kantengewichte minimal ist, sondern einen Pfad, der das kleinste Kantengewicht auf diesem Pfad maximiert. Als Distanz d jedes Knoten v speichern wir dabei jeweils die maximale Geschwindigkeit, mit der der Computer x Daten zu v senden kann.

Algorithmus:

Zunächst initialisieren wir die Geschwindigkeiten, mit denen der Computer x Daten an einen Computer v senden kann, für jeden Knoten $v \neq x$ mit 0 und für den Startknoten x mit unendlich. Der Algorithmus arbeitet dann analog zum Dijkstra-Algorithmus - zu berücksichtigen ist nur, dass hier versucht wird, die „Weglänge“ (d.h. die Geschwindigkeit) zu maximieren statt zu minimieren und sich die „Weglänge“ vom Startknoten x zu einem Knoten v nicht aus der Summe der Kantengewichte ergibt, sondern aufgrund der Aufgabenstellung durch das Minimum aller Kantengewichte bestimmt ist.

FastestConnection$(G = (V, E), x, y)$:

```
1   for each vertex v ∈ V do
2       d[v] ← 0
3       π[v] ← nil
4   d[x] ← ∞
5   S ← ∅
6   Q ← V
7   while Q ≠ ∅ do
8       u ← Extract-Max(Q)
9       S ← S ∪ {u}
10      for each vertex v ∈ Adj[u] do
11          if d[v] < min{d[u], w(u, v)} then
12              Increase-Key(Q, v, min{d[u], w(u, v)})
13              π[v] ← u
14  return d[y]
```

Anmerkung: Die Funktion Increase-Key für Max-Heaps ist analog zur Funktion Decrease-Key für Min-Heaps aus der Vorlesung definiert: Sie erhöht den Schlüsselwert eines Elements und erhält dabei die Heapeigenschaft.

b) Beweisen Sie die Korrektheit und analysieren Sie die Laufzeit Ihres Algorithmus.

Lösung: (3 Punkte)
Korrektheit:

Beweis per Induktion über die Anzahl n der Knoten in S:

Invariante:
Für jeden Knoten $v' \in S$ bezeichnet $d[v']$ die maximale Geschwindigkeit einer Kommunikationsverbindung von x nach v' und für jeden Knoten $v \in V \setminus S$ bezeichnet $d[v]$ die maximale Geschwindigkeit einer Kommunikationsverbindung von x nach v, die nur über Knoten aus S läuft.

Induktionsanfang:
Für $n = 1$ ist nur der Startknoten x in S enthalten, da dieser wg. $d[x] = \infty$ als erstes zu S hinzugefügt wird. Die Geschwindigkeit, mit der x mit sich selbst kommunizieren kann, ist nicht durch Kommunikationsverbindungen begrenzt, daher ist $d[x] = \infty$ korrekt. Zudem setzt der Algorithmus in Zeile 11 für alle benachbarten Knoten v des Startknotens $d[v] \leftarrow w(x, v)$, was ebenfalls korrekt ist, da die Kommunikation von x nach v lediglich durch die Verbindungskante (x, v) beschränkt ist. Alle anderen Knoten sind nicht über Pfade erreichbar, die nur Knoten aus S nutzen, und haben damit korrekterweise Gewicht 0. Damit gilt die Invariante.

Induktionsvoraussetzung:
Die Invariante gelte für $n \leq |V|$.

Induktionsschritt: $n \to n+1$
Nach Induktionsvoraussetzung ist für jeden Knoten $v_1, \ldots, v_n \in S$ die Geschwindigkeit der besten Kommunikationsverbindung von $x \in S$ nach $v_i, 1 \leq i \leq n$ in $d[v_i]$ gespeichert. Wir betrachten nun das Hinzufügen des Knotens v_{n+1} zu S:
Gemäß Zeile 8 des Algorithmus ist v_{n+1} von allen Knoten, die nicht in S liegen, am schnellsten von x aus erreichbar. Da für v_{n+1} der Pfad mit maximaler Geschwindigkeit über Knoten aus S bekannt ist und alle anderen Knoten $v \in V \setminus \{S \cup \{v_{n+1}\}\}$ nicht mit höherer Geschwindigkeit erreichbar sind, ist somit die maximale Geschwindigkeit der Kommunikationsverbindung von x nach v_{n+1} in $d[v_{n+1}]$ gespeichert und der erste Teil der Invariante somit erfüllt.
Wir zeigen nun, dass für alle Nachbarknoten (d.h. Nachfolger) von v_{n+1} die schnellste Kommunikationsverbindung ausgehend von x bestimmt wird, die nur Knoten aus $S \cup \{v_{n+1}\}$ nutzt. Sei im Folgenden z ein beliebiger, aber fester Nachbarknoten von v_{n+1}. In $d[z]$ ist bereits die Geschwindigkeit der Kommunikationsverbindung gespeichert, die nur Knoten aus S nutzt. Zusätzlich kann eine Kommunikationsverbindung von x über v_{n+1} nach z aufgebaut werden, deren Geschwindigkeit durch die Leitung mit der geringsten Kapazität auf diesem Pfad beschränkt ist. Offensichtlich ist die Geschwindigkeit dieser Verbindung dann entweder durch die Geschwindigkeit des besten Pfades von x nach v_{n+1} oder durch die Kante von v_{n+1} nach z beschränkt. In Zeile 11 des Algorithmus wird nun bestimmt, ob dieser Pfad eine schnellere Verbindung erlaubt als die bereits bekannte Verbindung, die ausschließlich Knoten aus S nutzt. Ist dies der Fall, wird die neue Geschwindigkeit gespeichert. Für jeden Nachbarknoten z von v_{n+1} bezeichnet $d[z]$ damit die maximale Geschwindigkeit einer Kommunikationsverbindung von x nach z, die nur über Kanten aus $S \cup \{v_{n+1}\}$ läuft.
Es bleibt zu zeigen, dass für alle Knoten $v' \in V \setminus \{S \cup \{v_{n+1}\}\}$, die nicht Nachbarknoten von v_{n+1} sind, keine bessere Kommunikationsverbindung über Knoten aus $S \cup \{v_{n+1}\}$ existiert, als bisher gefunden wurde. Würde eine solche schnellere Kommunikationsverbindung existieren, so müsste v' Nachfolger eines anderen Elements z' aus S sein. Damit kann die Geschwindigkeit der Kommunikationsverbindung von x nach v' aber nur dadurch verbessert werden, dass die Verbindung von x nach z' verbessert wird, was nach Voraussetzung nicht der Fall sein kann.
Damit ist gezeigt, dass die Invariante nach jedem Schleifendurchlauf gilt.

Nach $|V|$ Durchläufen terminiert die Schleife und S enthält alle Knoten aus V. Laut Invariante gilt für jedes $v \in S$: $d[v]$ ist die max. Geschwindigkeit einer Kommunikationsverbindung von x nach v. Folglich ist die Rückgabe des Wertes $d[y]$ korrekt. ■

<u>Laufzeit:</u>

Gegenüber dem Dijkstra-Algorithmus wurden nur Operationen mit konstanter Laufzeit geändert. Die Worst-Case-Laufzeit dieses Algorithmus ist damit gleich der Worst-Case-Laufzeit des Dijkstra-Algorithmus und beträgt somit $\mathcal{O}((|V| + |E|) \log |V|)$.