Prof. Dr. Leif Kobbelt
Stefan Dollase, Ira Fesefeldt, Alexandra Heuschling, Gregor Kobsik

Lösung - Übung 10

Tutoraufgabe 1 (Einfach Verbundene Graphen):

a) Sei G ein gerichteter Graph. Wir nennen G *einfach verbunden*, wenn es zwischen jedem Knotenpaar $u, v \in V$ höchstens einen einfachen Pfad zwischen u und v gibt.

Geben Sie einen möglichst effizienten Algorithmus an, um herauszufinden ob ein gerichteter Graph einfach verbunden ist. Begründen Sie die Korrektheit und die Laufzeit Ihres Algorithmus.

Lösung

Der Algorithmus führt für jeden Knoten $v \in V$ eine Tiefensuche aus. Sobald diese Tiefensuche dem Knoten v eine Austrittszeit zuordnet, wird diese abgebrochen. Die Tiefensuche besucht also nur die von v erreichbaren Knoten. Der Graph ist genau dann nicht einfach verbunden, falls bei einer dieser Tiefensuchen eine forward- oder cross edge gefunden wird.

Nun zeigen wir die Korrektheit des Algorithmus. Angenommen der Graph ist einfach verbunden. In diesem Fall kann die Tiefensuche keine forward edge finden. Dies zeigen wir per Widerspruch. Angenommen die Tiefensuche findet eine forward edge, welche im Knoten v' endet, dann gibt es zwei verschiedene einfache Pfade von v zu v'. Einer davon nutzt nur die Baumkanten, der zweite nutzt die forward edge und ggf. weitere Baumkanten. Dies steht im Widerspruch zur Annahme, dass der Graph einfach verbunden ist. Es kann also in einem einfach verbundenen Graphen keine forward edge gefunden werden. Analog lässt sich zeigen, warum in einem einfach verbundenen Graphen mit obigem Algorithmus keine cross edge gefunden werden kann. Dabei ist zu beachten, dass jede gefundene cross edge tatsächlich einen zweiten einfachen Pfad erzeugen würde, da obiger Algorithmus nur im von v erreichbaren Graphen nach cross edges sucht.

Nehmen wir nun an der Graph ist nicht einfach verbunden. In diesem Fall gibt es mindestens zwei Knoten v und v', sodass die Tiefensuche von v einen einfachen Pfad $v_0 v_1 \dots v_{n-1} v_n$ mit $v_0 = v$ und $v_n = v'$ aus Baumkanten findet und sodass es einen weiteren einfachen Pfad $u_0 u_1 \dots u_{m-1} u_m$ mit $u_0 = v$ und $u_m = v'$ gibt, sodass alle $v_1 \dots v_{n-1} u_1 \dots u_{m-1}$ paarweise verschieden sind. Sei i nun die größte Zahl mit $0 \leq i < m$, sodass es keinen Pfad von v_1 zu u_i gibt, welcher nicht über v läuft (d.h. alle Pfade von v_1 zu u_i laufen über v). Dies gilt immer für $i = 0$, aber es kann auch für größere i gelten. Wir wissen nun, dass die Kante (v_0, v_1) vor der Kante (u_0, u_1) besucht wird, sonst würde $v_0 v_1 \dots v_{n-1} v_n$ nicht aus Baumkanten bestehen. Das bedeutet, alle Knoten, welche von v_1 erreichbar sind werden abgeschlossen, bevor ein Knoten angefangen wird, welcher von u_1 aber nicht von v_1 erreichbar ist. Insbesondere ist u_{i+1} von v_1 erreichbar über einen Pfad, der nicht über v läuft, sonst wäre i um 1 größer. Somit wird u_{i+1} abgeschlossen bevor u_i angefangen wird. Die Kante (u_i, u_{i+1}) muss bei der von v ausgehenden Tiefensuche also entweder eine forward edge sein (für $i = 0$) oder eine cross edge sein (für $i > 0$).

Nun zeigen wir, dass der Algorithmus in der Laufzeit $O(n(n+m))$ arbeitet, wobei $n = |V|$ und $m = |E|$. Der Algorithmus führt n mal eine Tiefensuche aus. Die Laufzeit der Tiefensuche ist bekannt und liegt in $O(n+m)$. Zusammen ergibt dies die gewünschte Laufzeit.

b) Sei G ein gerichteter azyklischer Graph. Geben Sie einen möglichst effizienten Algorithmus an, um herauszufinden, ob G einfach verbunden ist. Begründen Sie die Korrektheit und die Laufzeit Ihres Algorithmus.

Lösung

Falls es eine Kante vom Knoten u zum Knoten v gibt, so nennen wir u einen Vorgänger von v. Wir nennen einen Knoten ohne Vorgänger einen Startknoten. Zuerst berechnet der Algorithmus eine topologische Sortierung der Knoten. Anschließend berechnet der Algorithmus für jeden Knoten im Graphen von welchen Startknoten er erreichtbar ist wie folgt. Zunächst werden alle Startknoten mit der Menge

markiert, welche nur eben diesen einen Startknoten enthält. Anschließend werden alle weiteren Knoten in aufsteigender Reihenfolge nach der topologischen Sortierung wie folgt markiert. Aufgrund der Bearbeitungsreihenfolge wurden alle Vorgänger des zu markierenden Knoten bereits markiert. Für den zu markierenden Knoten v wird überprüft, ob es zwei Vorgänger u und u' von v gibt, sodass der Schnitt deren Markierungen nicht leer ist. Ist dies der Fall, so ist der Graph nicht einfach verbunden. Ansonsten wird der Knoten v mit der Vereinigung der Markierungen der Vorgänger von v markiert. War der Schnitt immer leer, so ist der Graph einfach verbunden.

Wir zeigen nun die Korrektheit des Algorithmus. Angenommen der Graph ist nicht einfach verbunden. Dann gibt es ein Knotenpaar $v, v' \in V$, sodass es zwei verschiedene einfache Pfade von v zu v' gibt. O.b.d.A. seien nun (a, v') und (b, v') die beiden letzten Kanten auf diesen beiden verschiedenen einfachen Pfaden von v zu v' mit $a \neq b$. Wenn der Algorithmus v' markieren will, so wurden a und b bereits markiert und beide Markierungen enthalten v, da es einen Pfad von v nach a bzw. b gibt. Damit ist der Schnitt der beiden Vorgänger a und b von v' nicht leer, denn er enthält v und der Algorithmus meldet, dass der Graph nicht einfach verbunden ist.

Nehmen wir nun an der Graph ist einfach verbunden. Dann ist jeder Knoten von jedem Startknoten nur maximal über einen Pfad erreichbar. Für einen beliebigen Knoten v muss damit die Markierungen aller Vorgänger $a_1 \dots a_k$ auch Schnitt frei sein. Anderenfalls gäbe es einen Pfad von einem Startknoten zu a_i und a_j und damit auch zwei Pfade zu v. Damit kann der Algorithmus stets Markierungen Schnitt frei vereinigen und gibt zurück, dass der Graph einfach verbunden ist.

Nun betrachten wir die Laufzeit des Algorithmus. Die topologische Sortierung liegt bekanntlich in $O(n+m)$. Die Startknoten finden wir in $O(n+m)$, da wir für jeden Knoten prüfen müssen ob es mindestens eine eingehende Kante gibt. Die Markierung aller weiterer Knoten liegt in $O(n \cdot m)$, da wir für jeden Knoten alle seine Vorgänger betrachten müssen. Damit ist die Gesamtlaufzeit des Algorithmus in $O(n \cdot m)$.

c) Ein einfacher Kreis $v_0 \dots v_{k-1} v_0$ ist ein Kreis, für den der Pfad $v_0 \dots v_{k-1}$ einfach ist. Die Länge eines Kreises $v_0 \dots v_k$ ist k.

Geben Sie einen möglichst effizienten Algorithmus an, um herauszufinden ob ein ungerichteter Graph einen einfachen Kreis der Länge mindestens drei enthält. Begründen Sie die Korrektheit und die Laufzeit Ihres Algorithmus.

Lösung

Wir modifizieren die Tiefensuche für ungerichtete Graphen wie folgt: Sobald die Tiefensuche eine Kante in die eine Richtung besucht, löscht sie die Kante in die Gegenrichtung. Wenn also beispielsweise von v ausgehend die Kante (v, w) besucht wird, dann wird direkt die Kante (w, v) gelöscht. Eine Ausnahme bilden Selbstkanten, welche nicht gelöscht werden. Der ungerichtete Graph wird durch die Tiefensuche also gerichtet.

Der Algorithmus führt die modifizierte Tiefensuche auf dem Graphen aus und nutzt dazu einen beliebigen Startknoten. Der Algorithmus meldet genau dann einen einfachen Kreis gefunden zu haben, falls er bei der Tiefensuche eine back edge findet.

Wir zeigen nun die Korrektheit. Dazu stellen wir zunächst fest, dass bei der Tiefensuche keine foward- oder cross edges auftreten können, da der Graph ungerichtet ist. Es gibt also nur Baum- und back edges.

Angenommen der Graph enthält einen einfachen Kreis $v_0 \dots v_{k-1} v_0$. Wir nehmen O.b.d.A. an, dass v_0 die geringste Eintrittszeit besitzt. Die Eintrittszeit von v_{k-1} ist also größer als die Eintrittszeit von v_0. Da es einen Pfad von v_0 zu v_{k-1} gibt, ist die Austrittszeit von v_{k-1} kleiner als die Austrittszeit von v_0. Dies stellt eine back edge dar und somit meldet der Algorithmus einen einfachen Kreis gefunden zu haben.

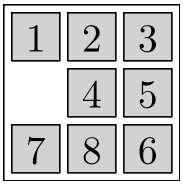
Nehmen wir nun an, der Algorithmus meldet einen einfachen Kreis gefunden zu haben. Dann hat der Algorithmus eine back edge e gefunden. Wir nehmen nun an, dass diese back edge die beiden Knoten v und v' miteinander verbindet. Nehmen wir weiter O.b.d.A. an, dass v die geringere Eintrittszeit hat. Damit die Kante e eine back edge ist muss nun die Eintrittszeit von v' größer sein als die von v und die

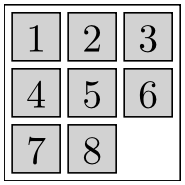
Austrittszeit von v' muss kleiner sein als die von v. Daraus folgt, dass es neben der Kante e noch einen weiteren einfachen Pfad zwischen v und v' geben muss. Wird dieser Pfad um e erweitert, so haben wir einen einfachen Kreis im Graphen gefunden.

Nun betrachten wir die Laufzeit des Algorithmus. Der Algorithmus führt zwar eine Tiefensuche auf dem Graphen aus, jedoch terminiert er sofort sobald er eine Kante findet, welche keine Baumkante ist. Es gibt höchstens $n-1$ viele Baumkanten. Damit liegt die Laufzeit des Algorithmus in $O(n)$.

Tutoraufgabe 2 (Neunerschiebepuzzle):

Das „Neunerschiebepuzzle“ besteht aus acht beweglichen Feldern in einer 3×3-Matrix. Jeweils eine der neun Positionen ist frei und ein an die freie Position angrenzendes Feld kann in diese hineingeschoben werden. Dies nennen wir einen „Zug“.





Ziel des Spiels ist es, die rechts gezeigte Position zu erreichen. Wir können uns nun einen ungerichteten Graphen vorstellen, dessen Knoten die Positionen dieses Spiels sind. Zwei Positionen sind durch eine Kante verbunden, wenn ein Zug sie ineinander überführt.
Wir können eine Breitensuche auf diesen Graphen beginnen, ohne ihn vorher komplett zu konstruieren. Führen Sie eine solche Breitensuche auf dem links gezeigten Startknoten aus und brechen Sie sie ab, sobald die Zielposition erscheint.

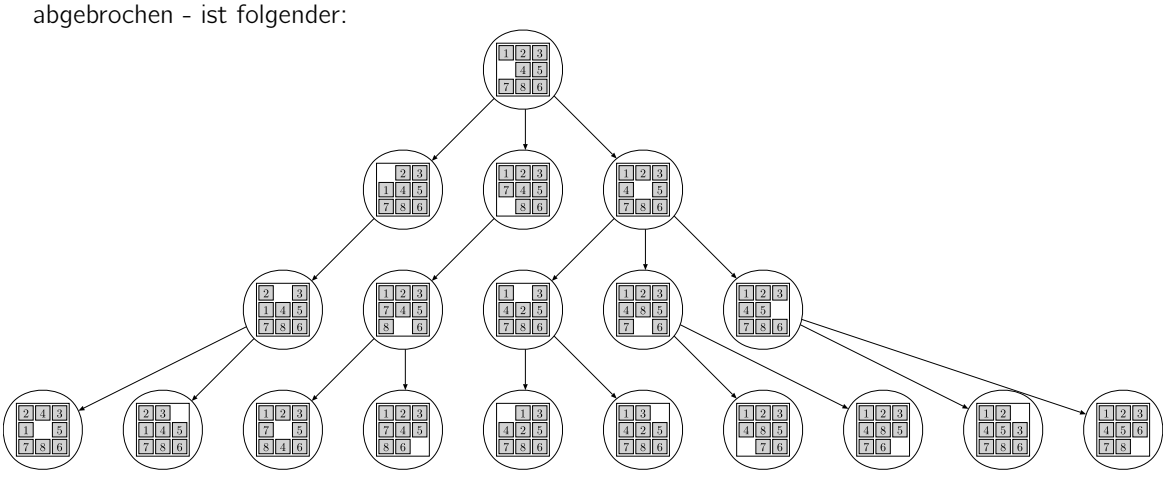
a) Zeichnen Sie den bis dahin entstandenen Breitensuchbaum auf.

b) Können Sie jetzt eine kürzestmögliche Lösung des Rätsels aus diesem Baum ablesen?

c) Ist es in dieser und ähnlichen Situationen Ihrer Meinung nach besser eine Tiefen- oder eine Breitensuche durchzuführen?

Lösung

a) Ein möglicher Baum der aufgebaut ist bis zu dem Zielknoten - ab diesem Punkt haben wir die Berechnung abgebrochen - ist folgender:

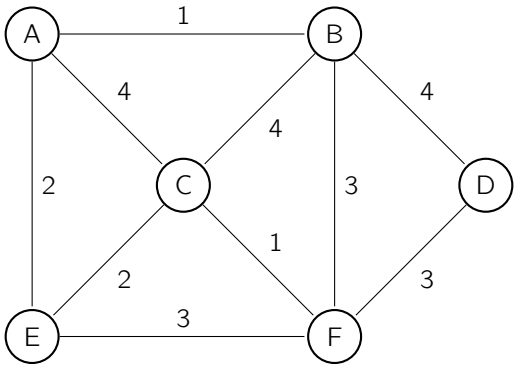


b) In der Tat gibt der Breitensuchbaum nun den kürzesten Pfad von der Startkonfiguration zu jeder anderen Konfiguration damit an. Damit können wir nun den Pfad von der Startkonfiguration zur Lösungskonfiguration im Breitensuchbaum suchen. Dies ist garantiert ein kürzeste Pfad im Graph.

c) Eine Breitensuche scheint hier die bessere Wahl, da bei einer Tiefensuche sehr viele weit entfernte Konfiguration besucht werden könnten, bevor der relativ nahe Zielknoten gefunden wird.

Tutoraufgabe 3 (Prim(-Jarnik-Dijkstra) Algorithmus und Heaps):

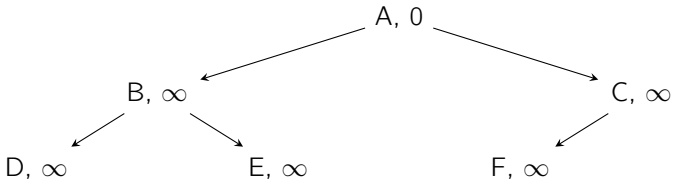
Gegeben ist folgender Graph G:



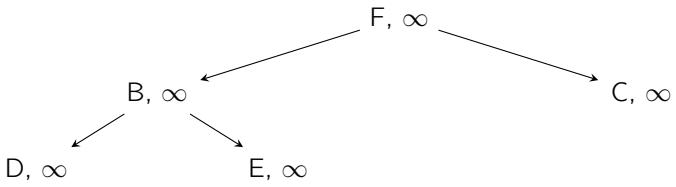
Führen Sie Prims Algorithmus auf den Graphen G mit Startknoten A aus, um einen Minimalen Spannbaum zu berechnen. Geben Sie dabei nach jeder Operation auf dem Heap den neuen Zustand des Heaps, sowie die gespeicherten Kosten jedes Knotens an. Es ist nicht nötig, alle Zwischenschritte beim initialen Erstellen des Heaps anzugeben. Gehen Sie außerdem davon aus, dass über Knoten stets in alphabetischer Reihenfolge iteriert wird. Sie dürfen den Heap sowohl als Baum, als auch als Array angeben. Geben Sie außerdem den resultierenden Minimalen Spannbaum an.

Lösung

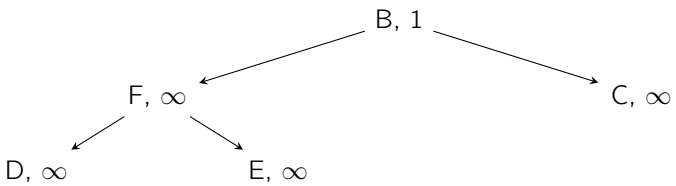
Wir geben den Heap stets mit Knotennamen und entsprechenden gespeicherten Kosten an. Nach dem Aufbau des Heaps, haben wir folgenden Heap:



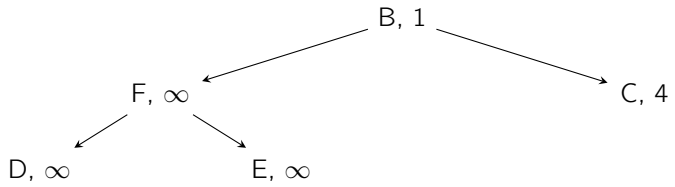
Nachdem A ausgewählt wurde, erhalten wir:



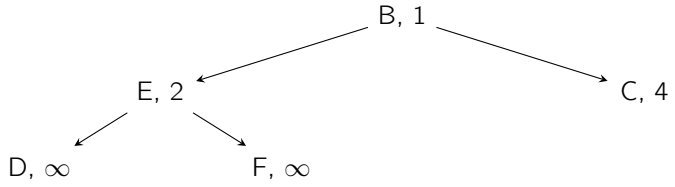
Nun updaten wir erst B:



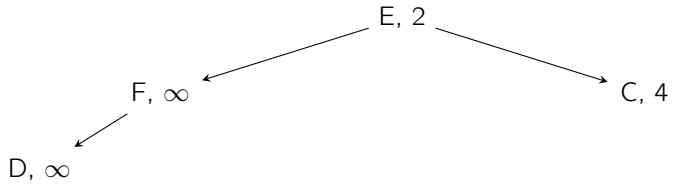
Dann C:



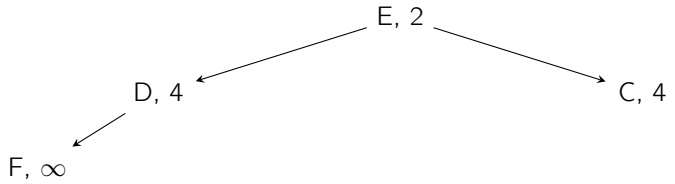
Und schließlich E:



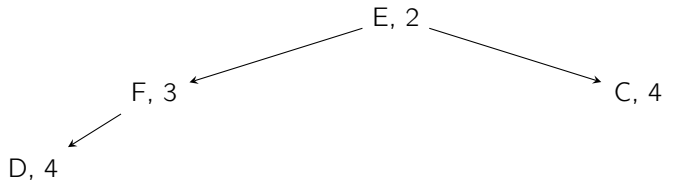
Nun wählen wir B aus:



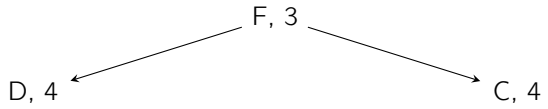
A führt zu keinem Update des Heaps, C führt zu keinem Update des Heaps, D dagegen führt zum Heap:



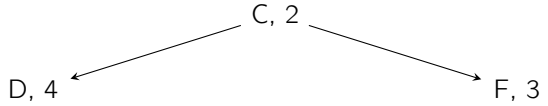
F führt zum Heap:



Nun wählen wir E aus:

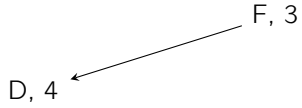


A führt zu keinem Update, C führt zu dem Update:

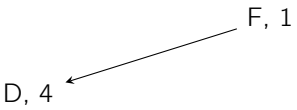


Und F führt zu keinem Update.

Wir wählen nun C aus und erhalten:



A führt zu keinem Update, B führt zu keinem Update, E führt zu keinem Update und F führt zu dem Update:

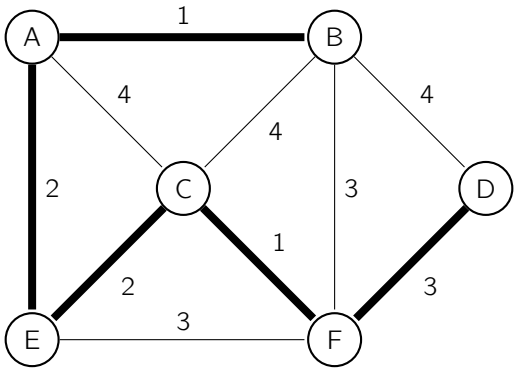


F wird nun ausgewählt, der Heap hat anschließend die Form:
D, 4

B führt zu keinem Update, C führt zu keinem Update, D führt zu dem Update:
D, 3

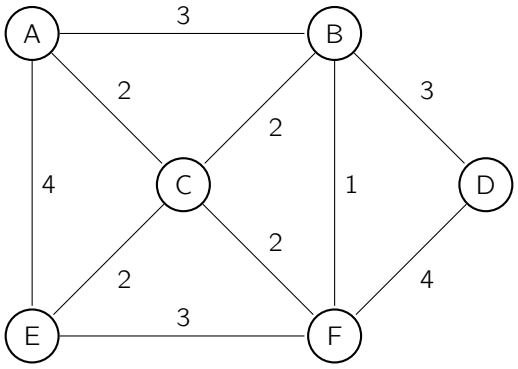
Und E führt ebenfalls zu keinem Update.

D wird nun ausgewählt und damit ist der Heap leer. Der resultierende Minimale Spannbaum sieht wie folgt aus:



Aufgabe 4 (Kruskals Algorithmus und Union-Find): 10 Punkte

Gegeben ist folgender Graph G:



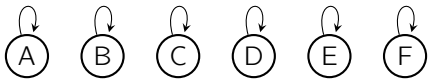
Führen Sie Kruskals Algorithmus auf den Graphen G aus, um einen Minimalen Spannbaum zu berechnen. Geben Sie dabei nach jeder Operation auf der Union-Find Struktur die entsprechende Operationen und den neuen Zustand der Union-Find Struktur an falls diese sich verändert hat. Dabei soll die Union-Operation bei **gleicher Höhe der Wurzeln immer die Wurzel des zweiten Parameters** als neue Wurzel wählen. Benutzen Sie hierfür sowohl Pfadkompression als auch Höhenbalancierung. Es ist nicht nötig, alle Zwischenschritte beim Ausführen der MakeSet Operationen anzugeben. Benutzen Sie die folgende Sortierung der Kanten:

$$(B, F), (A, C), (B, C), (C, E), (C, F), (A, B), (B, D), (E, F), (A, E), (D, F)$$

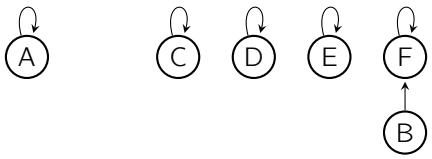
Geben Sie außerdem den resultierenden Minimalen Spannbaum an.

Lösung

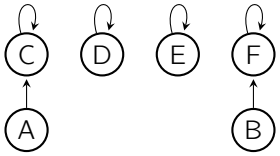
Die Union-Find Struktur nach allen MakeSet Operationen sieht wie folgt aus:



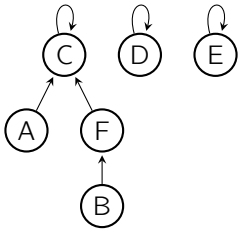
Find(B) keine Änderung, Find(F) keine Änderung, Union(B,F):



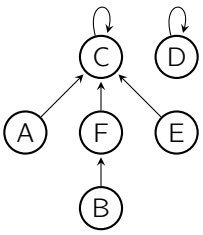
Find(A) keine Änderung, Find(C) keine Änderung, Union(A,C):



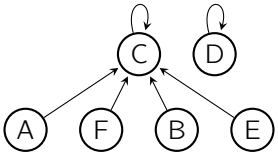
Find(B) keine Änderung, Find(C) keine Änderung, Union(B,C):



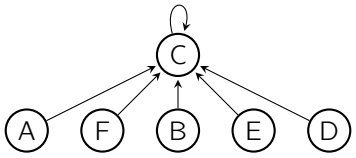
Find(C) keine Änderung, Find(E) keine Änderung, Union(C,E):



Find(C) keine Änderung, Find(F) keine Änderung
Find(A) keine Änderung, Find(B):

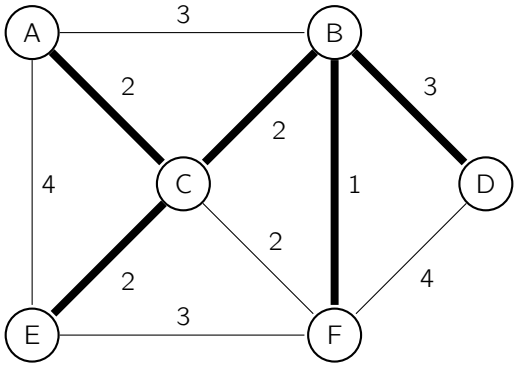


Find(B) keine Änderung, Find(D) keine Änderung, Union(B,D):



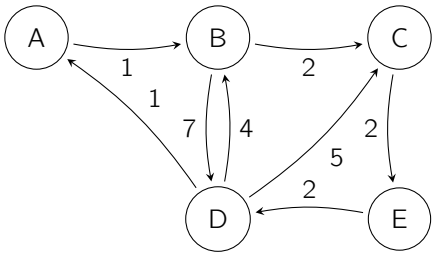
Find(E) keine Änderung, Find(F) keine Änderung
Find(A) keine Änderung, Find(E) keine Änderung
Find(D) keine Änderung, Find(F) keine Änderung

Der resultierende Minimale Spannbaum ist folgender:



Aufgabe 5 (Floyd-Warshall Algorithmus): 10 Punkte

Betrachten Sie den folgenden Graphen:



Führen Sie den *Algorithmus von Floyd* auf diesem Graphen aus. Geben Sie dazu nach jedem Durchlauf der äußeren Schleife die aktuellen Entfernungen in einer Tabelle an. Die erste Tabelle enthält bereits die Adjazenzmatrix nach Bildung der reflexiven Hülle. Der Eintrag in der Zeile i und Spalte j ist also ∞, falls es keine Kante vom Knoten der Zeile i zu dem Knoten der Spalte j gibt, und sonst das Gewicht dieser Kante. Beachten Sie, dass in der reflexiven Hülle jeder Knoten eine Kante mit Gewicht 0 zu sich selbst hat.

①	A	B	C	D	E
A	0	1	∞	∞	∞
B	∞	0	2	7	∞
C	∞	∞	0	∞	2
D	1	4	5	0	∞
E	∞	∞	∞	2	0

②	A	B	C	D	E
A					
B					
C					
D					
E					

③	A	B	C	D	E
A					
B					
C					
D					
E					

④	A	B	C	D	E
A					
B					
C					
D					
E					

⑤	A	B	C	D	E
A					
B					
C					
D					
E					

⑥	A	B	C	D	E
A					
B					
C					
D					
E					

Lösung

①	A	B	C	D	E
A	0	1	∞	∞	∞
B	∞	0	2	7	∞
C	∞	∞	0	∞	2
D	1	4	5	0	∞
E	∞	∞	∞	2	0

②	A	B	C	D	E
A	0	1	∞	∞	∞
B	∞	0	2	7	∞
C	∞	∞	0	∞	2
D	1	2	5	0	∞
E	∞	∞	∞	2	0

③	A	B	C	D	E
A	0	1	3	8	∞
B	∞	0	2	7	∞
C	∞	∞	0	∞	2
D	1	2	4	0	∞
E	∞	∞	∞	2	0

④	A	B	C	D	E
A	0	1	3	8	5
B	∞	0	2	7	4
C	∞	∞	0	∞	2
D	1	2	4	0	6
E	∞	∞	∞	2	0

⑤	A	B	C	D	E
A	0	1	3	8	5
B	8	0	2	7	4
C	∞	∞	0	∞	2
D	1	2	4	0	6
E	3	4	6	2	0

⑥	A	B	C	D	E
A	0	1	3	7	5
B	7	0	2	6	4
C	5	6	0	4	2
D	1	2	4	0	6
E	3	4	6	2	0

Aufgabe 6 (Programmierung in Python - Graphalgorithmen):

2 + 3 + 5 + 6 + 4 = 20 Punkte

Bearbeiten Sie die Python Programmieraufgaben. In dieser praktischen Aufgabe werden Sie sich mit Graphalgorithmen auseindersetzen. Diese Aufgabe dient dazu einige Konzepte der Vorlesung zu wiederholen und zu vertiefen. Zum Bearbeiten der Programmieraufgabe können Sie einfach den Anweisungen des Notebooks *blatt10-python.ipynb* folgen. Das Notebook steht in der .zip-Datei zum Übungsblatt im Lernraum zur Vergügung.
Ihre Implementierung soll immer nach dem # *YOUR CODE HERE* Statement kommen. Ändern Sie keine weiteren Zellen.
Laden Sie spätestens bis zur Deadline dieses Übungsblatts auch Ihre Lösung der Programmieraufgabe im Lernraum hoch. Die Lösung des Übungsblatts und die Lösung der Programmieraufgabe muss im Lernraum an derselben Stelle hochgeladen werden. Die Lösung des Übungsblatts muss dazu als .pdf-Datei hochgeladen werden. Die Lösung der Programmieraufgabe muss als .ipynb-Datei hochgeladen werden.

Übersicht der zu bearbeitenden Aufgaben:

a) Implementierung von Dijkstra
- initialize_single_source()
- relax()
- dijkstra()

b) Labyrinthe: Generieren und Lösen
- generate_maze()
- solve_maze()

Lösung

Die Lösung der Programmieraufgaben finden Sie im Lernraum. Die Datei trägt den Namen *blatt10-python-solution.ipynb*.