Aufgabensammlung ADS-Repetitorium WS 25/26

Graphen – Graphenalgorithmen – Greedy-Algorithmen

Aufgabe 1: Modellierung als Graphen

Viele algorithmische Fragestellungen können als Graphen modelliert werden. Bearbeiten Sie folgende Punkte für jede Fragestellung:

1. Geben Sie einen Algorithmus in Worten an, der die Fragestellung als Graph modelliert. Achten Sie auf eine präzise Definition der Knoten- und Kantenmenge.
2. Mit welchem Graph-Algorithmus kann die Fragestellung auf dem Graph gelöst werden? Wie müssen die Algorithmen gegebenfalls modifiziert werden?
3. Von welchen Parametern hängt die Laufzeit ab? Können Sie die Laufzeit genau angeben?

Einige der Probleme lassen sich eventuell auch einfacher ohne Graph lösen. In dieser Aufgabe sollen sie aber explizit den Umgang mit Graphen üben.

(a) Sie arbeiten im Marketing einer Firma, die Kameras herstellt. Auf einer Veranstaltung, die von n Menschen besucht wird, bieten Sie folgende Werbekampagne an:
Sie verteilen m Einwegkameras, mit denen man je genau ein Bild schießen kann. Die Kameras sollen dazu benutzt werden, Twofies (= Bilder, auf denen genau zwei Personen abgebildet sind) zu schießen. Die Person, die am Ende der Veranstaltung mit den meisten anderen Menschen auf Twofies abgebildet ist, hat gewonnen.
Wie können Sie die Person ermitteln, die gewinnt?

> **Lösung:** Alle Menschen repräsentieren die Knoten. Zwei Menschen s_1 und s_2 werden durch eine ungerichtete Kante verbunden, falls sie gemeinsam auf einem Twofie abgebildet sind. Wir suchen den Knoten mit dem höchsten Grad. Zur Berechnung der Knotengrade können wir die Breitensuche verwenden und in der **for**-Schleife die Nachbarn zählen. Das Der Knoten mit dem maximalen Knotengrad wird zurückgegeben. Die Laufzeit liegt damit in $\mathcal{O}(m+n)$.

(b) Das Kommunalunternehmen eines Landkreises mit n Gemeinden möchte jede Gemeinde an ein Radwegnetz anbinden. Um Kosten zu sparen, sollen die Radwege nur an den *breitesten* Straßen im Landkreis angelegt werden und Rundfahrten zwischen den Gemeinden sollen nicht möglich sein.

> **Lösung:** Es wird offenbar ein maximaler Spannbaum gesucht. Jarník-Prim und Kruskal liefern minimale Spannbäume. Daher suchen wir zunächst in $\mathcal{O}(|E|)$ Zeit die breiteste Straße mit Breite b und setzen das Gewicht jeder Kante e auf $b - \mathrm{w}(e)$. Nun können wir Jarník-Prim oder Kruskal ausführen und den berechneten, minimalen Spannbaum nutzen, um das Radwegenetz auszubauen. Die Laufzeit des gesamten Algorithmus liegt in $\mathcal{O}(|E| + n \log n)$ (Jarník-Prim) bzw. $\mathcal{O}(|E| \log |E|)$ (Kruskal).

(c) Ein Software-Projekt besteht aus n Modulen. Jedes Modul kann von anderen Modulen abhängig sein. Ein Modul kann nur gestartet werden, falls alle Module, von denen das Modul abhängt, bereits gestartet wurden. Gesucht ist also die Reihenfolge, in der die Module gestartet werden können.

> **Lösung:** Jedes Modul ist ein Knoten. Wir fügen eine gerichtete Kante von m_1 nach m_2 ein, falls m_2 von m_1 abhängig ist. Mit der Tiefensuche erhalten wir eine topoligische Sortierung der Module, anhand der wir die Module starten können. Die Laufzeit ist in $\mathcal{O}(n + |E|)$.

Aufgabe 2: Graphen-Cliquen

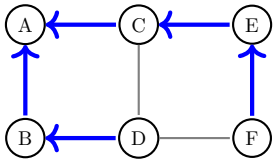
Sei $G = (V, E)$ ein ungerichteter Graph. Eine Teilmenge $C \subseteq V$ heißt *Clique*, wenn jedes Knotenpaar $e, d \in E$ durch eine Kante verbunden ist.
Schreiben Sie einen Algorithmus, der für einen Graphen $G = (V, E)$ und eine Teilmenge $C \subseteq E$ prüft, ob C eine Clique in G ist.

Lösung: Für jeden Knoten $e \in C$: Überprüfe für jeden anderen Knoten $d \in C$, ob es eine Kante $\{e, d\}$ gibt. Wenn nicht, gibt `false` zurück.
Gib am Ende `true` zurück.

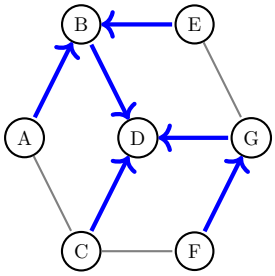
Aufgabe 3: Korrektheit von Breitensuchen erkennen

In folgenden Graphen sind die π-Zeiger der Knoten nach einer Breitensuche in blau eingezeichnet. Geben Sie an, ob die folgenden Graphen durch eine Breitensuche entstanden sein können. Begründen Sie Ihre Antworten.

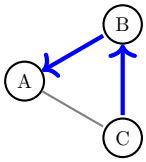
(a)



(b)



(c)



Lösung:

(a) Diese Konfiguration der π-Zeiger kann nicht durch eine Breitensuche entstanden sein. Da A keinen π-Zeiger hat, ist A der Startknoten. Der π-Zeiger von D zeigt auf B. Da B und C gleichweit von A entfernt sind und beide eine Kante zu D haben, muss B vor C entdeckt worden sein. Damit muss auch D vor E entdeckt worden sein. Somit müsste der π-Zeiger auf D und nicht auf E zeigen.

(b) Diese Breitensuche ist korrekt. D ist der Startknoten, von den Nachbarn wurde zuerst B, dann G und dann C entdeckt. Die π-Zeiger der Knoten A, E und F zeigen demnach auf die richtigen Knoten.

(c) Dies ist keine korrekte Breitensuche. A ist der Startknoten der Breitensuche. Damit müsste der π-Zeiger von C auf A zeigen, da die Knoten benachbart sind. Da dieser aber auf B zeigt ist die Breitensuche nicht korrekt.

Aufgabe 4: Tiefensuche iterativ

Sie haben in der Vorlesung zwei Durchlaufstrategien für Graphen kennengelernt: die Breitensuche und die Tiefensuche. Während die Breitensuche „iterativ“ mit einer Schlange funktioniert, benutzt die Tiefensuche Rekursion, um den Graphen zu explorieren. Da rekursive Algorithmen manchmal langsamer sind und außerdem durch die Größe der maximalen Rekursionstiefe limitiert ist, möchte man gelegentlich die Tiefensuche ebenfalls „iterativ“ implementieren.

Hinweis: Sie müssen die Farben nicht setzen und können eine einzige flag $u.visited$ für einen Knoten u benutzen.

(a) Implementieren Sie die Tiefensuche mit Hilfe eines Stapels in Pseudocode. Ignorieren Sie hierfür die *time stamps* $u.d$ und $u.f$ eines Knotens u. Die Entdeckungsreihenfolge muss nicht identisch mit der rekursiven Version der Tiefensuche sein!

Lösung: Idee: Wir nehmen den Pseudocode von DFS und DFS-Visit und ersetzen rekursive Aufrufe durch einen Push auf einen Stack S

Algorithmus 1: DFS-iterativ$(G = (V, E))$

```
1  Stack S = new Stack()
2  for s ∈ V do
3      if s.visited == false then
4          s.visited = true
5          S.Push(s)
6
7          while S ≠ ∅ do
8              u = S.Pop()
9
10             for v ∈ Adj[u] do
11                 if v.visited == false then
12                     v.visited = true
13                     S.Push(v)
```

(b) Können Sie Ihren Pseudocode so zu modifizieren, dass Sie auch die *time stamps* korrekt setzen?

Lösung: Hier besteht die Schwierigkeit nicht nur die Enteckungszeiten richtig zu setzen sondern auch die „finish“-Zeiten zum richtigen Zeitpunkt zu setzen. Wir lösen das Problem, indem wir einen callStack simulieren. Dieser enthält drei Informationen: der derzeitige Knoten, einen Index auf den nächsten Knoten in seiner Adjazenzliste und einen Zustand, ob wir noch explorieren (EXPLORING) oder ob wir mit diesem Knoten fertig sind (FINISHED). Damit können wir jeweils die rekursiven Aufrufe simulieren:

Algorithmus 2: DFS-iterativ($G = (V, E)$)

```
1  Stack callStack = new Stack()
2  time = 0
3  for s ∈ V do
4      if s.visited == false then
5          time = time + 1
6          s.d = time
7          s.visited = true
8          callStack.Push(⟨s, 1, EXPLORING⟩)
9
10         while callStack ≠ ∅ do
11             u, index, state = callStack.Pop()
12             if state == FINISHED then
13                 time = time + 1
14                 u.f = time
15             else
16                 if index ≤ |Adj[u]| then
17                     callStack.Push(⟨u, index + 1, EXPLORING⟩)
18                     v = Adj[u].get(index)
19
20                     if v.visited == false then
21                         v.π = u
22                         v.visited = true
23                         time = time + 1
24                         v.d = time
25                         callStack.Push(⟨v, 0, EXPLORING⟩)
26                 else
27                     callStack.Push(⟨u, index, FINISHED⟩)
```

Bemerkung: Die Implementierung der Adjazenzliste aus der Vorlesung unterstützt Anfragen wie Adj[u].get() nicht, da jeder Eintrag der Adjazenzliste lediglich eine einfach verkettete Liste ist. Wir können diese aber leicht augmentieren mit einem Zeiger, der am Anfang auf das erste Element zeigt und nach jedem Aufruf von get auf das nächste Element in der jeweiligen Liste zeigt. Somit ist der Zugriff auf das jeweils nächste Element in konstanter Zeit möglich.
Laufzeit: Die Laufzeit hängt maßgeblich davon ab, wie viele Elemente auf den Stack gepusht werden, denn jede Iteration der While-Schleife in den Zeilen 10-27 ist konstant. Jeder Knoten wird 1x auf den Stack gepusht, wenn dieser entdeckt wurde. Anschließend wird er für jeden Knoten in seiner Adjazenzliste erneut auf den Stack gepusht. Als letztes wird ein Knoten noch einmal auf den Stack gepusht, nachdem die Adjazenzliste abgearbeitet wurde. Danach ist sein Status FINISHED und wird nicht mehr auf den Stack gepusht. Das heißt pro Knoten v haben wir einen Aufwand von $\Theta(\text{Adj}[u])$. Wir iterieren außerdem über jeden Knoten einmal und haben somit eine Gesamtlaufzeit von $\Theta(V + E)$

Aufgabe 5: Dijkstra's Algorithmus

Gegeben sei der Graph in Abbildung 1. Bearbeiten Sie auf diesem Graph die folgenden Aufgaben.

(a) Führen Sie Dijkstra's Algorithmus mit Startknoten s aus. Erstellen Sie dazu eine Tabelle mit drei Spalten: *Iteration*, *Schwarze Knoten*, *Graue Knoten*, zu der sie nach jeder Iteration der While-Schleife eine Zeile hinzufügen. Schreiben Sie hinter jeden Knoten in Klammern die aktuelle Distanz und den Vorgänger-Knoten.

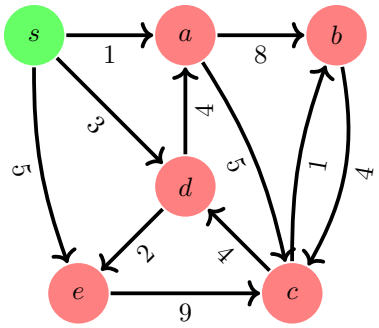


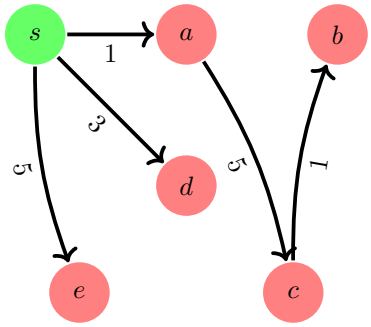
Abbildung 1: Beispiel-Graph für Dijkstras-Algorithmus.

Lösung:

Iteration	Schwarze Knoten	Graue Knoten
1	$s(0, \emptyset)$	$a(1, s), d(3, s), e(5, s)$
2	$a(1, s)$	$d(3, s), e(5, s), b(9, a), c(6, a)$
3	$d(3, s)$	$e(5, s), b(9, a), c(6, a)$
4	$e(5, s)$	$b(9, a), c(6, a)$
5	$c(6, a)$	$b(7, c)$
6	$b(7, c)$	

(b) Zeichnen Sie den entstandenen Kürzeste-Wege-Baum.

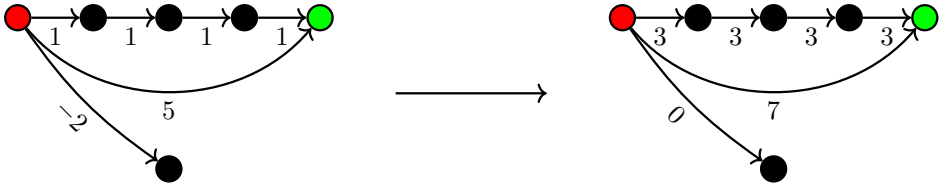
Lösung:



(c) Seien nun negative Kantengewichte erlaubt. Verändern Sie den gegeben Graphen, sodass Dijkstra nach Beendigung kein richtiges Ergebnis liefert, obwohl der kürzeste Weg definiert ist. Warum kann dies nicht behoben werden, indem wir das minimale Kantengewicht $g < 0$ identifizieren und alle Gewichte um $|g|$ erhöhen? Dann wären alle Kantengewichte wieder positiv und wir könnten Dijkstra verwenden. Begründen Sie allgemein, warum diese Vorgehensweise nicht korrekt ist.
Vorschau: Morgen werden wir uns im Rahmen der Dynamischen Programmierung mit einem Algorithmus beschäftigen, der mit negativen Kanten zurecht kommt.

Lösung: Wir können auf dem gegeben Graphen das Gewicht der Kante (d, a) auf -1 und das Gewicht der Kante (s, a) auf 2.5 setzen. Dann hat der kürzeste Weg nach a die Länge 2. Die Wege nach c und b verkürzen sich ebenfalls um 1. Dijkstra erkennt dies nicht, da zuerst a behandelt wird und danach schwarz gefärbt ist. Knoten, die einmal schwarz sind, werden von Dijsktra nicht mehr verändert, weshalb der neue kürzeste Weg von d nach a nicht gefunden wird. Wenn wir allgemein

alle Kanten um den Betrag des minimalen Kantengewichts erhöhen, verlängen wir die Wege mit vielen Kanten mehr als die Wege mit wenigen Kanten, wie folgendes Beispiel zeigt:

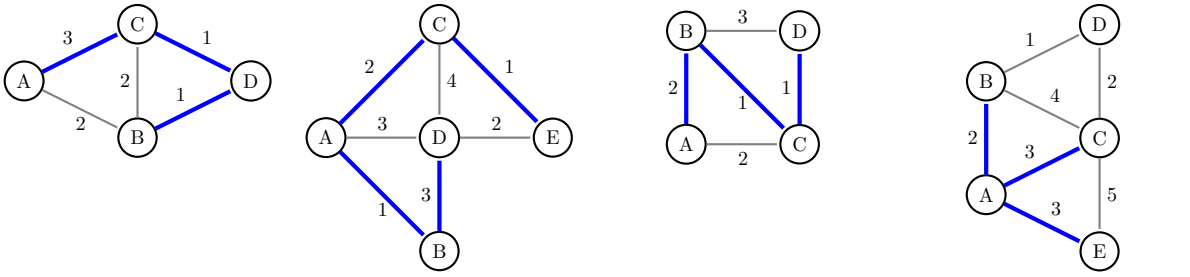


Gesucht ist ein Weg vom roten linken Knoten zum grünen rechten Knoten. Der kürzeste Weg geht über die drei Knoten in der Mitte. Da ein Kantengewicht negativ ist, wird der Graph in den rechten Graph transformiert. Nun ist der direkte Weg von links nach rechts ohne Zwischenstation der kürzeste. Folglich bleibt der kürzeste Weg bei der Transformation nicht derselbe.

Aufgabe 6: Erkennen von minimalen Spannbäumen

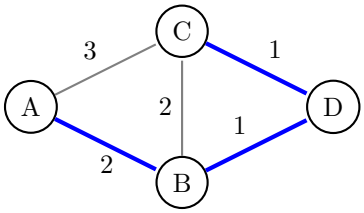
Entscheiden Sie für die folgenden blau markierten Teilgraphen, ob diese minimale Spannbäume sind. Geben Sie für jeden Graphen, dessen blau markierter Teilgraph kein minimaler Spannbaum ist, einen minimalen Spannbaum an und begründen Sie Ihre Antworten.

(a) (b) (c) (d)

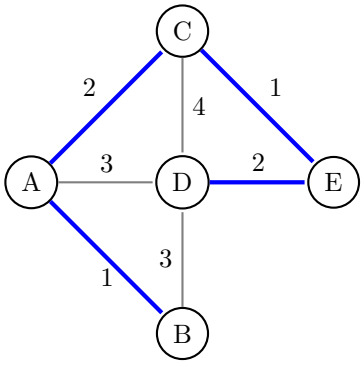


Lösung:

(a) Dies ist zwar ein Spannbaum, aber das Gewicht ist nicht minimal. Ein minimaler Spannbaum wäre:

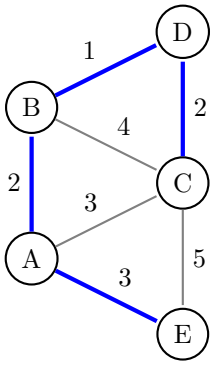


(b) Dies ist zwar ein Spannbaum, aber das Gewicht ist nicht minimal. Ein minimaler Spannbaum wäre:



(c) Dies ist ein korrekter minimaler Spannbaum.

(d) Knoten D wird von den Kanten nicht besucht. Damit bilden die Kanten keinen Spannbaum für den Graphen. Folgendes ist ein korrekter minimaler Spannbaum:



Geben Sie für jedes der folgenden Probleme einen Greedy-Algorithmus an. Denken Sie daran, die folgenden Punkte zu beweisen:

Aufgabe 7: Greedy-Algorithmen

- Beweisen Sie, dass die Lösung des Greedy-Algorithmus zulässig ist.
- Beweisen Sie, dass die Lösung des Greedy-Algorithmus optimal ist, oder geben Sie ein Beispiel an, in dem der Greedy-Algorithmus nicht die optimale Lösung findet.
- Geben Sie die Laufzeit an.

(a) Sei $G = (V, E)$ ein ungerichteter Graph. Gesucht ist eine möglichst große Teilmenge U der Knoten V, sodass für keine zwei Knoten $u, v \in U$ die Kante uv in E ist.

Lösung:

Algorithmus 3: greedyIndependentSet(V, E)

```
1 result = ∅
2 while V ≠ ∅ do
3 |   v = beliebiger Knoten aus V
4 |   Lösche Nachbarn von v aus V
5 |   result = result ∪{v}
6 return result
```

Die Lösung des Algorithmus ist per Definition zulässig, ist aber nicht immer optimal. Gegenbeispiel: Sterngraph. Auch die Modifikation, immer den Knoten mit kleinstem Grad zu löschen funktioniert nicht. Das sieht man an einem Graphen, der ein Kreis ist und in den eine ShortCut-Kante eingefügt ist, sodass genau ein Dreieck im Kreis vorkommt.

Die Laufzeit des Algorithmus ist – großzügig abgeschätzt – $O(|V| \cdot |E|)$.

(b) Gegeben ein Baum $T = (V, E)$, in dem jeder Knoten v einen positiven Wert $v.w$ hat. Gesucht ist ein Pfad von der Wurzel zu einem Blatt, sodass die Summe der auf dem Weg liegenden Knoten möglichst groß ist.

Lösung:

Algorithmus 4: greedyMaxPath$(V, E,$ root$)$

```
1 result = new List(root)
2 while result.head ist kein Blatt do
3 |   v = arg max_{v∈ result.head.children} v.w
4 |   result.insert(v)
5 return reverse(result)
```

Die Lösung des Algorithmus ist zulässig, der per Definition des Algorithmus immer ein valider Pfad zurückgegeben wird. Sie ist nicht immer optimal, ein einfach Beispiel dafür darf sich der Übungsleiter aus den Fingern ziehen, ich habe keine Lust mehr auf tikz.
Die Laufzeit ist –großzügig – $O(|E| + |V|)$

(c) Gegeben sind n positive Zahlen in einem Feld. Selektieren Sie $n/2$ Zahlen so, dass dass deren Summe möglichst klein ist.

Lösung:

Algorithmus 5: greedyMinSum(A[])

```
1 result = new List()
2 for i = 1 to n/2 do
3 |   Finde kleinste Zahl k in A
4 |   Ersetze k in A durch max A
5 |   result.insert(k)
6 return result
```

Die Lösung des Algorithmus ist zulässig, der Algorithmus nur Zahlen, die tatsächlich in A vorkommen verwendet. Die Lösung ist immer optimal. Angenommen es gibt eine bessere Auswahl A^* an Zahlen als der vom Algorithmus berechneten A. Also $\sum A^* < \sum A$. Unser Algorithmus wählt die $n/2$ kleinsten Zahlen aus, also muss es eine Zahl k' in A^* geben, die nicht unter den $n/2$ kleinsten Zahlen ist. Und es muss eine Zahl k'' unter den $n/2$ kleinsten Zahlen geben, die nicht in A^* ist. Tausche k' und k'' in A^* aus. A^* wird kleiner. Widerspruch zur Optimalität von A^*.
Die Laufzeit ist in $O(n^2)$.

(d) Gegeben eine Liste von Jobs $j_1, \ldots, j_n$ sowie die Zeiten t_i, die zum Abarbeiten des i. Jobs benötigt wird. Es stehen zwei Maschinen zur Verfügung. Verteile die Jobs so auf beide Maschinen, dass alle Jobs möglichst schnell bearbeitet wurden.

Lösung:

Algorithmus 6: jobGreedy($t_1, \ldots, t_n$)

```
1 machine1 = new List()
2 machine2 = new List()
3 Sortiere Jobs absteigend
4 for i = 1 to n do
5 |   m = Maschine, die aktuell am frühesten fertig ist
6 |   j = längster Job, der noch nicht zugeordnet ist
7 |   m.insert(j)
8 return machine1, machine2
```

Die Lösung ist zulässig, da alle Jobs auf eine Maschine verteilt werden und nur Jobs der Eingabe verteilt werden. Aber die Lösung ist nicht optimal. Betrachte fünf Jobs mit Laufzeiten 5, 5, 4, 4, 2. Unser Algorithmus ordnet sie folgendermaßen an: $m_1 = 5, 4, 2$ und $m_2 = 5, 4$. Die Jobs sind also nach 11 Einheiten abgearbeitet. Die optimale Lösung ist aber $m_1 = 5, 5$ und $m_2 = 4, 4, 2$ mit einer Bearbeitungszeit von 10.
Die Laufzeit ist, wenn die Jobs als Liste daherkommen, $O(n \log n)$ durch das Sortieren der Liste. In Zeile 5 können wir dank der Liste den längsten Job in konstanter Zeit löschen.

(e) Sie wollen eine Wüste durchqueren. In der Wüste sind $n + 1$ Oasen, Sie starten in Oase 0 und Ihr

Ziel ist Oase n. Dazwischen sind $n-1$ Oasen, in denen Sie Ihre Wasserflasche auffüllen können. Ihre Wasserflasche reicht für r Kilometer. Oase i und Oase $i+1$ sind d_i Kilometer voneinander entfernt. Finden Sie eine Folge von Oasen, sodass die möglichst selten nachfüllen müssen. Sie können davon ausgehen, dass alle Distanzen zwischen den Oasen kleiner als r sind.

Lösung:

Algorithmus 7: desertGreedy(Distances $d_0, \ldots, d_n$, r)

```
1 walked = d_0
2 refill = ∅
3 for i = 1 to n − 1 do
4     if walked + d_i > r then
5         refill = refill ∪ {i}
6         walked = d_i
7     else
8         walked = walked + d_i
9 return refill
```

Die Lösung ist zulässig, da wir niemals weiterlaufen, als wir mit unserem Wasservorrat noch laufen könnten. Die Lösung ist auch optimal. Angenommen, es gibt eine Lösung L^*, die mit weniger Refills auskommt als unsere Lösung L. Betrachte die erste Oase i, die in L^* vorkommt, aber nicht in L und die erste Oase i' nach i, die in L vorkommt. Da L zulässig ist, können wir in in L^* die Oase i durch Oase i' austauschen, ohne die Zulässigkeit und die Güte von L^* zu beeinträchtigen. Wiederhole diesen Schritt für die nächste Oase, die nicht in L, aber in L^* vorkommt. Sobald es keine solche Oase mit gibt, sind L und L^* identisch und gleich gut. Widerspruch zu „L^* ist besser als L“.
Die Laufzeit ist in $O(n)$.

Aufgabe 8: Definition eines Graphen
Die Definition eines Graphen $G = (V, E, w)$ ist gegeben durch die Knotenmenge V und die Kantenmenge E:

$$V = \{(x, y) \mid x, y \in \mathbb{N} \wedge x < n \wedge y < n\}, \ n \in \mathbb{N}^+$$

$$E = \{((x_1, y_1), (x_2, y_2)) \in V \mid w((x_1, y_1), (x_2, y_2)) < a\}, \ a \in \mathbb{R}^+$$

Die Funktion w ist durch die Zuordnungsvorschrift $(x_1, y_1), (x_2, y_2) \mapsto \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ gegeben. Falls $((x_1, y_1), (x_2, y_2)) \in E$, so repräsentiert w die Gewichtsfunktion der Kanten.

(a) Kann der Graph als ungerichteter Graph aufgefasst werden? Begründen Sie Ihre Antwort.

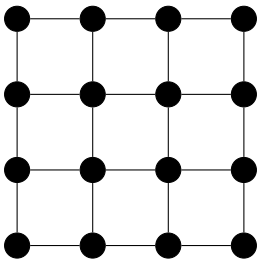
Lösung: Der Graph kann als ungerichteter Graph aufgefasst werden, da die Gewichtsfunktion symmetrisch ist, unabhängig davon, in welcher Reihenfolge u und v als Parameter übergeben werden.

(b) Welches reale Szneario könnte durch diesen Graphen mit einem festen n und a beschrieben werden?

Lösung: Denkbar ist ein Szenario, in dem n^2 Entitäten gitterartig auf einer Landkarte angeordnet werden. Die Entitäten, deren euklidische Distanz kleiner als a ist, werden direkt miteinander verbunden, zum Beispiel Straßenkreuzungen in Manhatten für $a = \sqrt{2}$.

(c) Zeichnen Sie den Graph so, dass sich keine Kanten überkreuzen, für den Fall, dass $n = 5$ und $a = \sqrt{2}$.

Lösung:



(d) Geben Sie für $a = \sqrt{2}$ eine äquivalente Definition der Kantenmenge E an. In Ihrer Definition soll die Funktion w nicht vorkommen.

Lösung: Bei $a = \sqrt{2}$ kommt ein Gitter zustande. Deshalb sind genau die Knoten miteinander verbunden deren x bzw. y Koordinate jeweils genau eine Differenz von 1 aufweisen:

$$E = \left\{((x_1, y_1), (x_2, y_2)) \in V^2 \mid x_1 = x_2 \wedge |y_1 - y_2| = 1 \vee y_1 = y_2 \wedge |x_1 - x_2| = 1\right\}$$

(e) Geben Sie ein $0 < a < 3\pi$ an, sodass die Berechnung der *Länge* des kürzesten Weges zwischen zwei Knoten bei beliebigem n in konstanter Zeit möglich ist. Wie funktioniert dann die Berechnung der Länge des kürzesten Weges?

Lösung: Wähle $a = \sqrt{2}$, dann ist die Distanz zwischen zwei Knoten u und v die Manhatten-Distanz ihrer Koordinaten, also $\delta(u, v) = |u.x - v.x| + |u.y - v.y|$. Diese Berechnung ist in konstanter Zeit möglich.

Aufgabe 9: DNS-Sequenzen und k-mere
Eine DNS-Sequenz ist ein String aus den vier Zeichen `G`, `T`, `C` und `A`. In der Biologie vergleicht man die

DNS-Sequenzen verschiedener Organismen. Dazu muss der DNS-String zunächst aus der Zelle extrahiert werden, wozu Sequenzierautomaten verwendet werden. Technisch bedingt ist die Länge k der Ausgabe der Automaten beschränkt, sodass eine DNS-Sequenz nicht komplett sequenziert wird. Stattdessen werden alle Substrings der Länge k in beliebiger Reihenfolge ausgegeben, wobei auch doppelte Substrings enhalten sind. Diese Substrings der Länge k werden als k-mere bezeichnet. Gesucht ist nun nach einer Möglichkeit, aus den k-meren die ursprüngliche DNS-Sequenz herzustellen. Ein *Präfix* eines k-mers ist der String des k-mers ohne das letzte Zeichen. Äquivalent dazu ist der *Suffix* eines k-mers der String des k-mers ohne das erste Zeichen.

(a) Ein Automat mit $k = 3$ sequenziert ein Genom mit der Sequenz `GTCCAGTCCAC`. Was ist die Ausgabe?

Lösung: Die Ausgabe ist `GTC`, `TCC`, `CCA`, `CAG`, `AGT`, `GTD`, `TDD`, `DDA`, `DAC` in beliebiger Reihenfolge.

(b) Wie viele k-mere gibt es in einer Sequenz der Länge $n > k$?

Lösung: Die Anzahl der k-mere ist $n - k + 1$.

(c) Gegeben ist der Graph G_{H} in Abbildung 2, der aus den 3-meren `ACG`, `ACT`, `CGT`, `CTA`, `GTA`, `GTT`, `TAC` und `TAC` entstanden ist. Leiten Sie die Vorgehensweise ab, mit der aus beliebigen k-meren ein Graph G_{H} gebildet wird.

Lösung: Man betrachte alle k-mere als Knoten. Zwischen zwei k-meren s_1 und s_2 befindet sich dann eine gerichtete Kante von s_1 nach s_2, falls der Suffix von s_1 der Präfix von s_2 ist.

(d) Beschreiben Sie in einem Satz, wie man mit G_{H} eine Sequenz findet, die die gegebenen k-mere aufweist. In der Bioinformatik ist die Anzahl der k-mere häufig in der Größenordnung einiger Millionen. Warum können Sie die Methode mit dem Graphen G_{H} nicht empfehlen?

Lösung: Im Graphen G_{H} muss ein Weg gefunden werden, der jeden Knoten genau einmal traviersiert, mit anderen Worten, ein Hamilton-Weg. Es ist kein effizienter Algorithmus zum Berechnen eines Hamiltonwegs bekannt, deshalb ist der Einsatz dieser Methode bei der gegebenen Anzahl an k-meren nicht ratsam.

(e) Geben Sie eine Sequenz s an, deren Sammlung der k-mere identisch zu denen aus c) ist.

Lösung: Durch ein wenig Probieren findet man zum Beispiel folgenden Hamiltonweg: `GTA` – `TAD` – `ADT` – `DTA` – `TAD` – `ADG` – `DGT` – `GTT`. Durch diesen Weg ergibt sich die Sequenz `GTADTADGTT`. Diese Sequenz ist unter Umständen nicht eindeutig. In der Realität untersucht man das zu sequenzierende Genom mit weiteren Methoden, um die eindeutige Sequenz zu finden. Das geht aber über das Ziel der Aufgabe hinaus.

(f) Glücklicherweise gibt es eine andere Möglichkeit, einen Graphen G_{E} aufzubauen, um eine Sequenz aus gegebenen k-meren zu berechnen: Die Kantenmenge besteht aus allen k-meren der Eingabemenge, wobei Duplikate erlaubt sind. Diese Kanten schreiben wir untereinander, beschriften sie mit dem jeweiligen k-mer und richten sie nach rechts. An die linke Seite jeder Kante fügen wir nun einen Knoten mit dem Präfix der Kante an und an der rechten Seite einen Knoten mit dem Suffix der Kante. Nun werden die Kanten analog zu einem Dominospiel an passenden Knoten aneinandergefügt.

Zeichnen Sie diesen Graph für die 3-mere aus c).

Lösung:

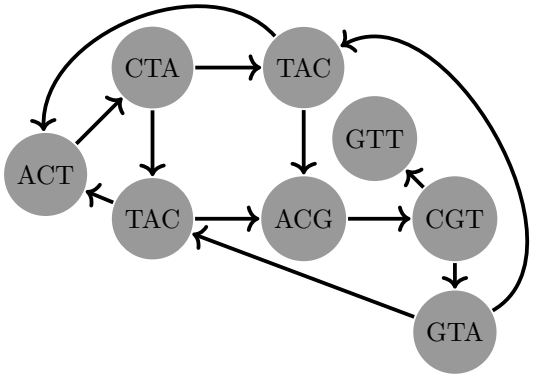
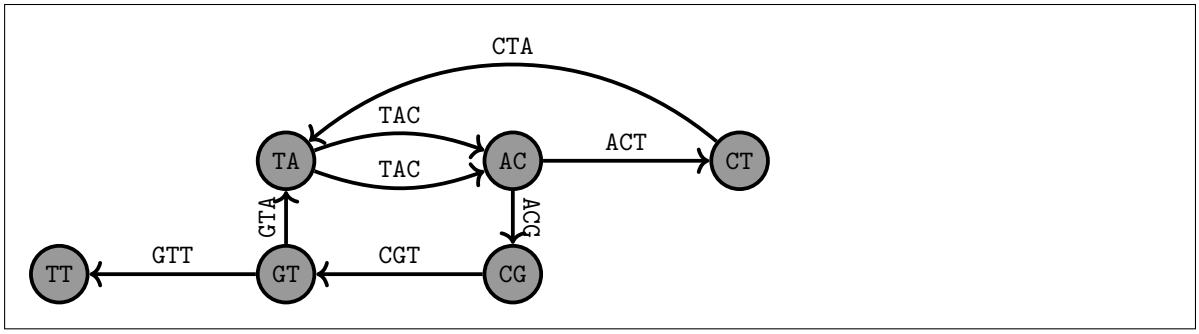


Abbildung 2: 3-mere-Graph für die Sequenz-Aufgabe.



(g) Beschreiben Sie in einem Satz, wie man mit G_{E} eine Sequenz findet, die die gegebenen k-mere aufweist.

Lösung: Man muss in G_{E} einen Weg finden, der alle Kanten genau einmal traversiert, also einen Eulerweg. Dieser ist nicht unbedingt eindeutig. In der Realität bedient man sich weiterer biologischer Hilfsmittel, um den richtigen Eulerweg zu finden. Hier ist der Eulerweg `GT` – `TA` – `AC` – `CT` – `TA` – `AC` – `CG` – `GT` – `TT`. Dies führt zurselben Sequenz wie in e).

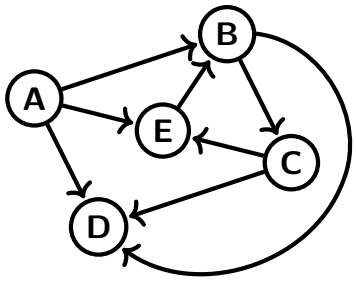


Abbildung 3: Ein gerichteter Graph für Aufgabe 10

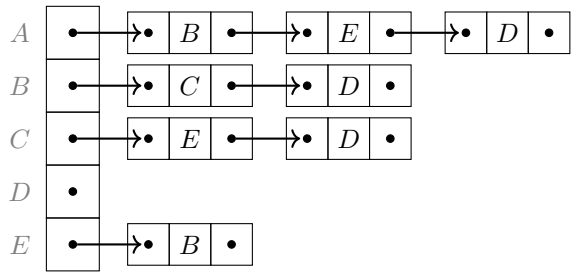
Aufgabe 10: Repräsentation von Graphen
Gegeben sei der Graph in Abbildung 3.

(a) Enthält der Graph Kreise? Wenn ja, geben Sie einen Kreis an.

Lösung: Der Graph enthält in der Tat einen Kreis, nämlich $\langle B, C, E \rangle$.

(b) Repräsentieren Sie diesen Graphen mit einer Adjazenzliste.

Lösung:



(c) Repräsentieren Sie diesen Graphen mit einer Adjazenzmatrix.

Lösung:

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

(d) Sei nun ein beliebiger simpler Graph G gegeben, dessen Maximalgrad $\Delta(G) = \lfloor \sqrt{|V|} \rfloor$ ist. In welcher Zeit kann man den Grad eines Knotens bestimmen oder testen, ob zwei Knoten benachbart sind, wenn G mit einer Adjazenzliste oder eine Adjazenzmatrix dargestellt ist?

Lösung: Sei $n = |V|$. Für einen Knoten u ist die Adjazenzliste $\mathsf{Adj}[u]$ eine einfach verkettete Liste. Deshalb müssen wir die gesamte Liste durchlaufen, um den Knotengrad von u zu bestimmen. Nachdem $\Delta(G) \in \mathcal{O}(\sqrt{n})$ ist, benötigen wir auch höchstens so viel Zeit.
Um zu testen, ob ein Knoten u mit einem Knoten v benachbart ist, müssen wir den Knoten v in der Adjazenzliste $\mathsf{Adj}[u]$ suchen. Im worst case durchlaufen wir die gesamte Liste, sodass wir $\mathcal{O}(\sqrt{n})$ Zeit benötigen.

Im Falle der Repräsentation durch eine $n \times n$ Adjazenzmatrix müssen wir die gesamte Zeile durchgehen und alle 1 Einträge zählen, um einen Knotengrad zu bestimmen. Das benötigt also $\mathcal{O}(n)$ Zeit.
Wenn u und v benachbart sind, so ist der Eintrag a_{uv} in der Adjazenzmatrix 1. Diesen Test können wir in konstanter Zeit durchführen.

(e) Beweisen oder widerlegen Sie folgende Aussage: Für einen beliebigen zusammenhängenden, einfachen Graphen $G = (V, E)$ mit $|V| \geq 2$ gilt: $\log(|E|) \in \Theta(\log |V|)$.

Lösung: Da G zusammenhängend ist, gilt $|E| \geq |V| - 1$ und weil G einfach ist, gilt $|E| \leq \binom{|V|}{2}$. Daraus folgt, dass $|V| - 1 \leq |E| \leq \binom{|V|}{2} < |V|^2 \Rightarrow \log(\frac{1}{2}|V|) \leq \log(|V| - 1) \leq \log(|V|) \leq \log(|E|) \leq 2\log(|V|)$. Mit anderen Worten $\log(|E|) \in \Theta(\log |V|)$.

Aufgabe 11: Matchings in Graphen

Sei $G = (V, E)$ ein ungerichteter Graph. Eine Teilmenge $M \subseteq E$ der Kantenmenge heißt *Matching*, wenn keine zwei Kanten in M einen Knoten gemeinsam haben. Ein Matching heißt *nicht erweiterbar*, wenn es keine Kante e in $E \setminus M$ gibt, sodass $e \cup M$ ein Matching ist.

(a) Überlegen Sie sich ein Szenario, das man als Graph modellieren und mit einem Algorithmus, der ein maximales Matching findet, lösen kann.

Lösung: Beispielsweise: Menschen in Zweiergruppen aufteilen, die sich untereinander nicht mit allen verstehen (eventuell werden nicht alle Menschen dabei berücksichtigt.)

(b) Schreiben Sie einen Algorithmus in Pseudocode, der für einen gegebenen Graphen $G = (V, E)$ und eine Teilmenge $M \subseteq E$ bestimmt, ob M ein Matching ist.

Lösung: Für jede Kante $e \in E$ mit den Endknoten $u, v \in V$: Gibt es eine Kante $d \in E$, sodass $e \neq d$ und v ist Endknoten von d oder u ist Enknoten von d, gibt `false` zurück
Gib am Ende `true` zurück.

(c) Entwickeln Sie einen Algorithmus in Pseudocode, der für einen gegebenen Graphen $G = (V, E)$ ein gültiges, nicht erweiterbares Matching berechnet.

Lösung: Idee: Nimm eine beliebige Kante $uv \in E$, füge sie zu M hinzu und lösche alle Nachbarknoten von u, v, bis E leer ist.

Algorithmus 8: maxMatching(Graph $G = (V, E)$)

```
1 M = ∅
2 while E ≠ ∅ do
3 |   Wähle zufällige Kante uv ∈ E
4 |   Lösche alle zu u, v adjazenten Knoten aus V
```

Aufgabe 12: Fehlende Kanten

Auf folgendem Graph in Abbildung 4 wurde eine Breitensuche ausgeführt. Dabei steht die Zahl in den Knoten für den Zeitpunkt, zu dem sie entdeckt wurden. Einige Kanten in dem Graph sind hier nicht dargestellt. Zeichnen Sie diese ein. Dabei darf jeder Knoten maximal den Grad 3 haben.

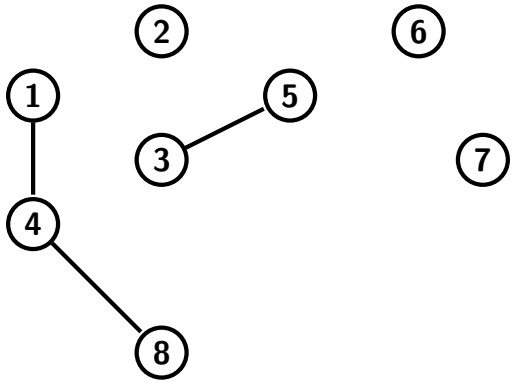
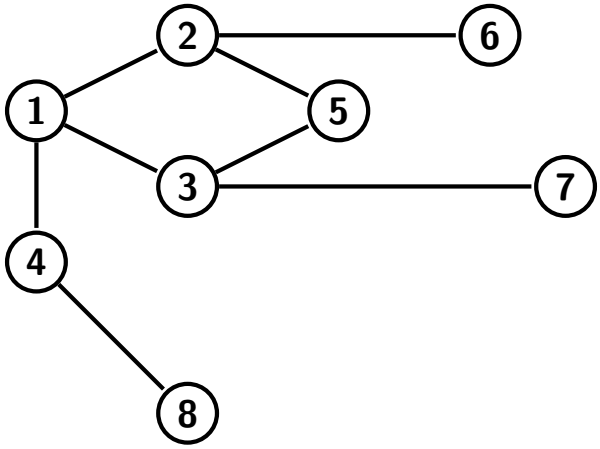


Abbildung 4: Abbildung Graphens für Aufgabe 12

Lösung: Die Lösung ist nicht eindeutig. Eine mögliche Lösung ist die folgende:



Aufgabe 13: Graphen-Theorie

Als *bipartit* wird ein Graph $G = (V, E)$ bezeichnet, falls sich seine Knotenmenge V in zwei Mengen S_1 und S_2 aufteilen lässt, sodass alle Kanten einen Knoten in S_1 mit einem Knoten in S_2 verbinden.

(a) Vervollständigen Sie die Definition, sodass sie äquivalent zur obigen textuellen Definition ist. Füllen Sie in jede Lücke genau ein Zeichen.

$$\text{Graph } G \text{ ist bipartit} \Leftrightarrow$$

$$\exists\ \underline{\ S_1\ } \subseteq V\ \forall\ (\underline{\ u\ }, v) \in \underline{\ \mathbf{E}\ } : (\underline{\ v\ } \in S_1\ \wedge\ \underline{\ u\ } \in V \setminus \underline{\ S_1\ }) \vee (\underline{\ v\ } \in V \setminus S_1 \underline{\ \wedge\ } u \in S_1)$$

(b) Beweisen Sie folgende Aussage: Graph G ist bipartit $\Leftrightarrow$ Graph G ist zweifärbbar.

Lösung:

Beweis. Wir zeigen beide Richtungen:

$\Rightarrow$ Sei G bipartit. Dann lässt sich die Knotenmenge V in zwei disjunkte Mengen S_1 und S_2 aufteilen, sodass keine Kante innerhalb S_1 bzw. S_2 liegt. Folglich können alle Knoten in S_1 rot und alle Knoten in S_2 blau gefärbt werden. Da aufgrund der bipartiten Eigenschaft von G keine Kanten innerhalb von S_1 und S_2 existieren, ist damit die Zwei-Färbbarkeits-Eigenschaft nicht verletzt.

$\Leftarrow$ Sei G zweifärbbar. Dann gibt es eine Färbung, sodass keine Kante zwei Knoten mit derselben Farbe verbindet. Wir legen alle roten Knoten in S_1 und ale blauen Knoten in S_2. Dann sind S_1 und S_2 disjunkt und es gibt keine Kante innerhalb von S_1 bzw. S_2. □

(c) Der folgende Algorithmus überprüft, ob ein gegebener Graph bipartit ist. Ergänzen Sie die Lücken. Zu Beginn des Algorithmus hat das marker-Attribut aller Knoten den Wert 0.

Algorithmus 9: boolean checkIfBipartite(Graph G, Vertex u = **null**)

```
1  if u = null then
2      u = beliebiger Knoten
3      u.marker = ____1____
4  foreach v ∈ Adj[u] do
5      if v.marker ≠ 0 then
6          if ____v.marker = u.marker____ then return false
7      else
8          if u.marker = 1 then v.marker = 2
9          if u.marker = 2 then v.marker = 1
10         flag = checkIfBipartite( G , v )
11         if flag = false then __return false__
12 return ____true____
```

(d) Welcher Algorithmus aus der Vorlesung liegt checkIfBipartite(G,v) zu Grunde?

Lösung: Der Tiefensuche-Algorithms liegt checkIfBipartite(G,v) zu Grunde.

Aufgabe 14: Terminale verbinden

Sie sind Betreiber eines Routernetzwerkes, wobei R die Menge Ihrer Router darstellt. Die Router sind untereinander verbunden, sodass Ihr gesamtes Netzwerk zusammenhängend ist. Dabei muss nicht notwendigerweise jeder Router mit jedem anderen Router verbunden sein. Damit die Verbindungen funktionieren, muss jede Verbindung mit Strom versorgt werden.

Im Falle eines Stromausfalls kann jede Verbindung mit je einem teurem Notstromaggregat betrieben werden. Die Kosten für dieses Notstromaggregat sind je Verbindung verschieden.

In Ihrem Netzwerk befinden sich einige besonders wichtige Knotenrouter K. Falls der Strom ausfällt, sollen weiterhin alle Router in K miteinander verbunden sein. Alle anderen Router $R \setminus K$ dürfen, aber müssen nicht unbedingt, ans Netzwerk angeschlossen sein. Sie sind nun an einer Auswahl an Verbindungen interessiert, die möglichst günstig mit Notstromaggregaten betrieben werden können und alle Knotenrouter K verbindet.

(a) Modellieren Sie das Problem als Graph-Problem. *Tipp:* Machen Sie sich mit einer Skizze die Aufgabenstellung klar.

Lösung: Knoten: Router; es existiert eine Kante zwischen Router u und v, falls u und v miteinander verbunden sind. Der Graph ist ungerichtet. Die Gewichte der Kanten entsprechen den Kosten für das Notstromaggregat. Gesucht ist ein Baum, der mindestens alle Knoten aus K enthält/aufspannt und minimal unter allen solchen Bäumen ist.

(b) Angenommen $|K| = 2$. Geben Sie einen effizienten Algorithmus an, der das Problem löst.

Lösung: Dijsktra löst das Problem.

(c) Angenommen $K = R$, mit anderen Worten: Alle Router sind wichtig. Geben Sie einen effizienten Algorithmus an, der das Problem löst.

Lösung: Jarnik/Prim bzw. Kruskal lösen das Problem.

(d) Das Software-Unternehmen PISNP bietet einen Algorithmus an, der das Problem für alle K löst. Dieser Algorithmus berechnet einen Graph T, der angeblich die oben beschrieben Anforderungen erfüllt. Geben Sie einen effizienten Algorithmus an, der überprüft, ob T tatsächlich gültig ist. Ihr Algorithmus erhält als Eingabe ihr Routernetzwerk, wie in a) modelliert, sowie K und T.

Lösung:

Algorithmus 10: testTree(G, K, T)

```
1 V, E = G
2 V', E' = T
3 if K ⊈ V' or E' ⊈ E then
4     return false
5 for v ∈ K do
6     for u ∈ K do
7         überprüfe mit BFS, ob u von v in T erreichbar ist, falls nicht return false.
8 return true
```

Die Laufzeit ist $O(|K|^2 \cdot (|V'| + |E'|))$

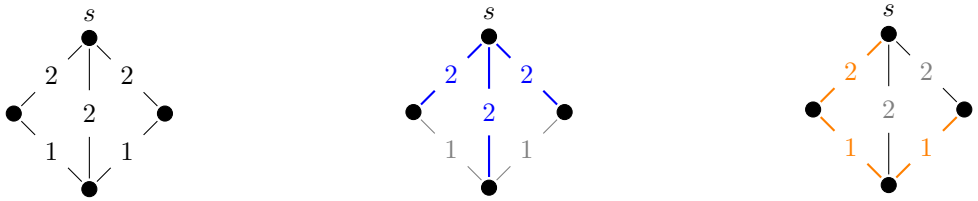
Aufgabe 15: Kürzeste-Wege-Bäume und minimale Spannbäume

Die Algorithmen von Dijkstra und Jarnik-Prim gehen ähnlich vor. Beide berechnen, ausgehend von einem Startknoten s, einen Baum. Allerdings brechnet der Algorithmus von Dijkstra einen Kürzesten-Wege-Baum, während der Algorithmus von Jarnik-Prim einen minimalen Spannbaum berechnet.

(a) Geben Sie einen ungerichteten gewichteten Graphen $G = (V, E)$ mit höchstens 5 Knoten und einen Startknoten $s \in V$ an, sodass Dijkstra und Jarnik-Prim ausgehend von s verschiedene Bäume in G liefern. Geben Sie beide Bäume an.

Lösung: In diesem Graphen ist der Kürzeste-Wege-Baum von Dijkstra und der minimale Spannbaum von Jarnik-Prim nicht gleich:

Graph $G = (V, E)$: Dijkstra: Jarnik-Prim:



(b) Geben Sie eine Familie von Graphen an, auf denen der Algorithmus von Jarnik-Prim asymptotisch schneller als der Algorithmus von Kruskal ist.

Lösung: Wenn Jarnik-Prim mit einem Fibonacci Heap implementiert ist, hat er eine Laufzeit von $\mathcal{O}(m + n \log n)$ und Kruskal hat eine Laufzeit von $\mathcal{O}(m \log m)$, wobei n die Anzahl der Knoten und m die Anzahl der Kanten ist. Für einen vollständigen Graphen K_n hat Jarnik-Prim eine Laufzeit von $\mathcal{O}(n^2)$ und Kruskal eine Laufzeit von $\mathcal{O}(n^2 \log n)$. Damit ist Jarnik-Prim für die Familie der vollständigen Graphen asymptotisch schneller als Kruskal.

(c) Angenommen Sie haben einen ungerichteten, gewichteten Graphen $G = (V, E)$ gegeben und alle Gewichte sind natürliche Zahlen zwischen 1 und $|V|$. Wie könnten Sie die Laufzeit von Kruskal's Algorithmus beschleunigen?

Lösung: Die Laufzeit von Kruskal ist beschränkt durch $\mathcal{O}(V + E)$ Union-Find Operationen und die Zeit, die benötigt wird, um alle Kanten zu sortieren. Nachdem die ganzzahligen Kantengewichte durch $|V|$ nach oben beschränkt sind, können die Kanten mit CountingSort in $\mathcal{O}(E)$ sortiert werden. Dadurch ist die Laufzeit nun $\mathcal{O}(E \cdot \alpha(V))$, wobei α die inverse Ackermann-Funktion ist.

Aufgabe 16: MaxCut

Gegeben sei ein ungewichteter ungerichteter Graph $G = (V, E)$. Im MaxCut Problem ist ein Schnitt $(C, V \setminus C)$ gesucht, sodass die Anzahl der Kanten, die diesen Schnitt schneiden maximal ist.

Algorithmus 11: `RandomizedMaxCut`(Graph $G = (V, E)$)

```
1 C ← ∅
2 foreach v ∈ V do
3 |   m = throwCoin()          // Gibt mit Wahrscheinlichkeit 1/2 Kopf oder Zahl in O(1)
4 |   if m == Kopf then
5 |   |   C ← C ∪ {v}
6 return (C, V \ C)
```

(a) Um welche Art von Algorithmus handelt es sich? Ist die Lösung immer valide?

Lösung: Der Algorithmus ist ein randomisierter Greedy Algorithmus. Die Lösung ist dabei immer valide, da der Algorithmus immer eine Partition von V zurückgibt. Allerdings ist die Lösung des Algorithmus nicht notwendigerweise optimal.

(b) Mit welcher Datenstruktur könnte man die Menge C darstellen?

Lösung: Nachdem wir nur Elemente zur Datenstruktur hinzufügen, sollte eine Datenstruktur gewählt werden, die `Insert` Operationen schnell erledigt. Zum Beispiel ein Array der Länge n oder eine Liste kann in $\mathcal{O}(1)$ Elemente einfügen. Damit eigenen sich diese beiden Datenstrukturen gut.

(c) Welche Laufzeit hat dieser Algorithmus?

Lösung: Mit der Wahl einer geeigneten Datenstruktur in Teilaufgabe (b) haben wir einen Aufwand von $\mathcal{O}(1)$ pro Iteration, wobei wir $\mathcal{O}(V)$ Iterationen haben. Damit ist die Gesamtlaufzeit $\mathcal{O}(V)$.

(d) Sei $\{u, v\} \in E$. Mit welcher Wahrscheinlichkeit liegt diese Kante auf dem Schnitt?

Lösung: Die Kante $\{u, v\}$ liegt auf dem Schnitt, wenn $u \in C$ und $v \notin C$ oder wenn $u \notin C$ und $v \in C$. Laut Angabe ist die Wahrscheinlichkeit $\Pr[v \in C] = 1/2$.

$$\mathbf{Pr}[u \in C \text{ und } v \notin C] = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$$
$$\mathbf{Pr}[u \notin C \text{ und } v \in C] = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$$

Das heißt $\mathbf{Pr}[(u \in C \text{ und } v \notin C) \text{ oder } (u \notin C \text{ und } v \in C)] = 1/4 + 1/4 = 1/2$.

(e) Sei $(C, V \setminus C)$ ein Schnitt, den `RandomizedMaxCut` zurückgegeben hat. Wie viele Kanten liegen erwartet auf diesem Schnitt?

Lösung: Sei X_e die Indikatorzufallsvariable, die angibt, ob die Kante $e \in E$ auf dem Schnitt liegt ($X_e = 1$) oder nicht und sei X die Zufallsvariable, die die Anzahl der Kanten angibt, die auf dem Schnitt liegen.

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{e \in E} X_e\right] = \sum_{e \in E} \mathbf{E}[X_e] \overset{\text{(d)}}{=} \sum_{e \in E} \frac{1}{2} = \frac{|E|}{2}$$

Die erwartete Anzahl der Kanten auf dem Schnitt sind also $|E|/2$.

(f) Wie viele Kanten können maximal auf einem Schnitt liegen?

Lösung: Es können alle Kanten auf dem Schnitt liegen.

(g) Welchen erwarteten Approximationsfaktor hat `RandomizedMaxCut`?

Lösung: Mithilfe der vorherigen Teilaufgaben wissen wir, dass $\mathbf{E}[X] = |E|/2$ und eine optimale Lösung höchstens $|E|$ Kanten auf dem Schnitt hat.

$$\frac{\mathsf{ALG}}{\mathsf{OPT}} \overset{\text{(e)}}{=} \frac{|E|/2}{\mathsf{OPT}} \overset{\text{(f)}}{\geq} \frac{|E|/2}{|E|} = \frac{1}{2}$$

Damit haben wir einen erwarteten Approximationsfaktor von $1/2$.