Aufgabensammlung ADS-Repetitorium WS 24/25

Graphen – Graphenalgorithmen – Greedy-Algorithmen

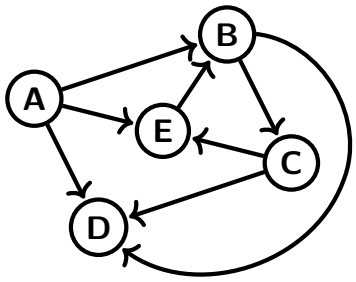


Abbildung 1: Ein gerichteter Graph für Aufgabe 1

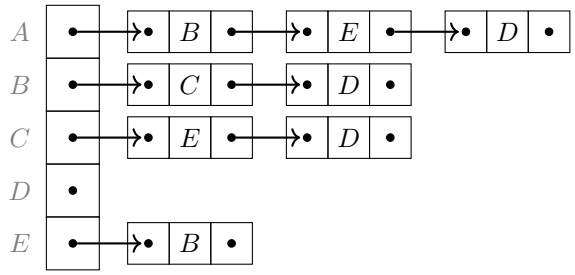
Aufgabe 1: Repräsentation von Graphen
Gegeben sei der Graph in Abbildung 1.

(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 2: Fehlende Kanten

Auf folgendem Graph in Abbildung 2 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.

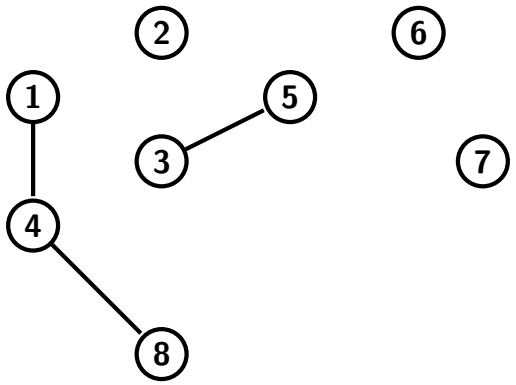
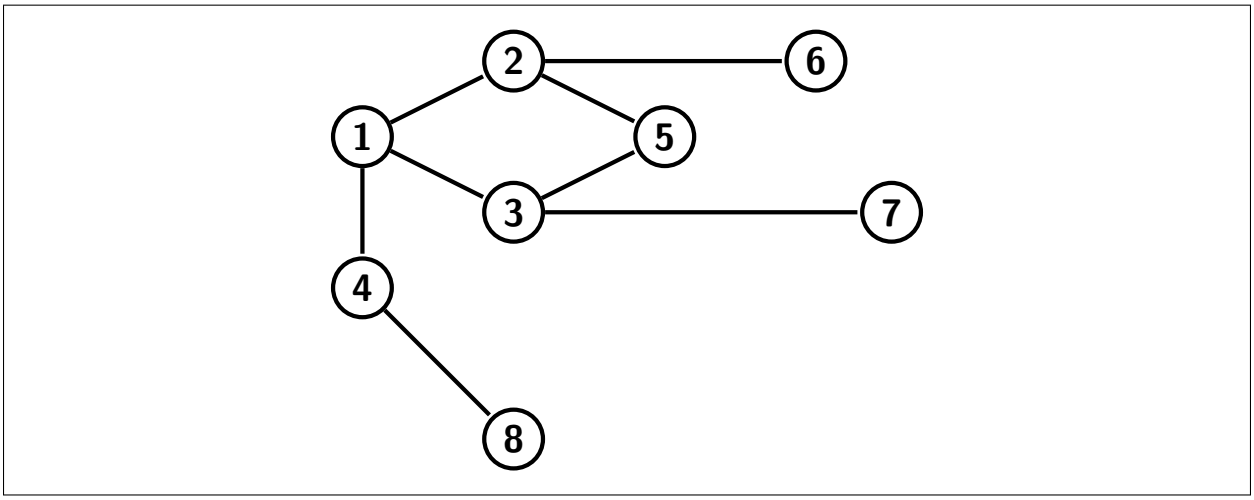


Abbildung 2: Abbildung Graphens für Aufgabe 2

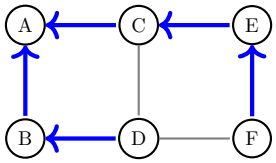
Lösung: Die Lösung ist nicht eindeutig. Eine mögliche Lösung ist die folgende:



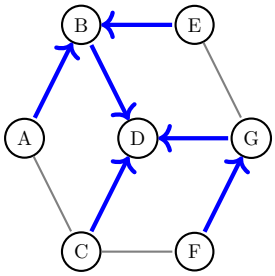
Aufgabe 3: Korrektheit von Breitensuchen erkennen

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.

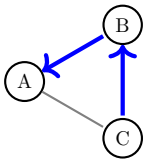
(a)



(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: Dijkstra's Algorithmus

Gegeben sei der Graph in Abbildung 3. 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

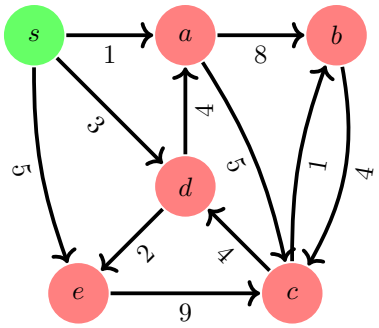


Abbildung 3: Beispiel-Graph für Dijkstras-Algorithmus.

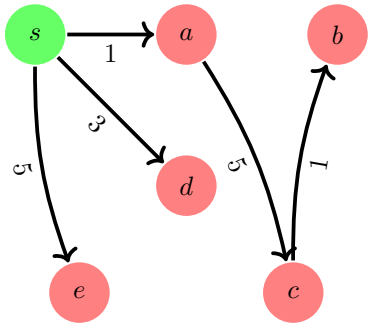
eine Zeile hinzufügen. Schreiben Sie hinter jeden Knoten in Klammern die aktuelle Distanz und den Vorgänger-Knoten.

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.

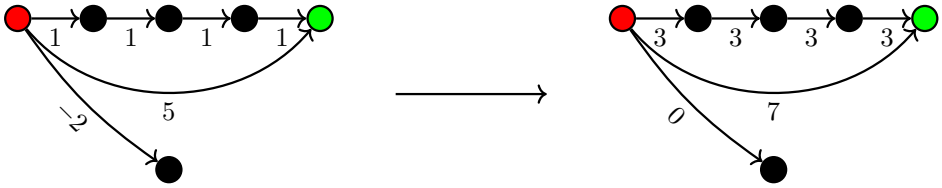
Lösung:



(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 5: 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($G = (V, E)$)

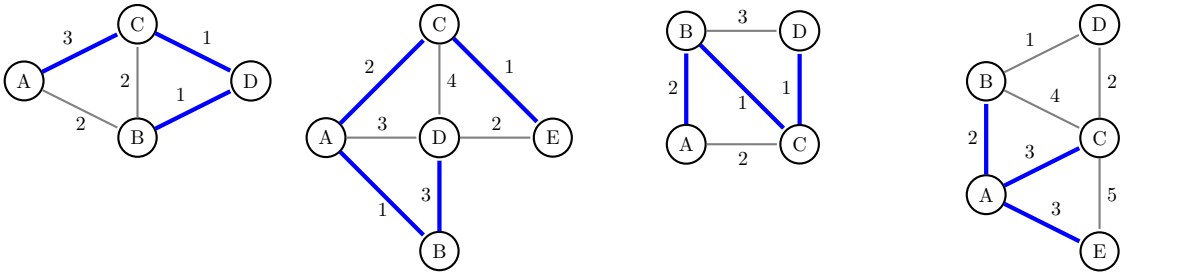
```
1  Stack callStack = new Stack()
2  time = 0
3  for s ∈ V do
4      if s.visited == false then
5          time = time + 1
6          s.d = time
7          s.visited = true
8          callStack.Push(⟨s, 1, EXPLORING⟩)
9
10         while callStack ≠ ∅ do
11             u, index, state = callStack.Pop()
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 6: Erkennen von minimalen Spannbäumen

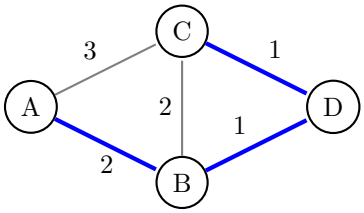
Entscheiden Sie für die folgenden blau markierten Teilgraphen, ob diese minimale Spannbäume sind. Geben Sie für jeden Graphen, dessen blau markierter Teilgraph kein minimaler Spannbaum ist, einen minimalen Spannbaum an und begründen Sie Ihre Antworten.

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

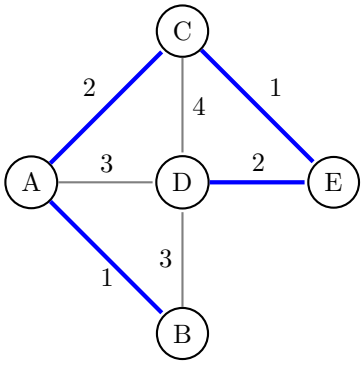


Lösung:

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

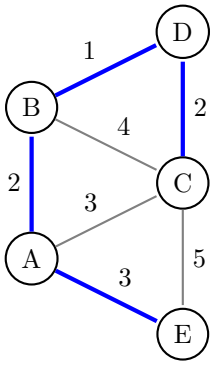


(b) Dies ist zwar ein Spannbaum, aber das Gewicht ist nicht minimal. Ein minimaler Spannbaum wäre:



(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:



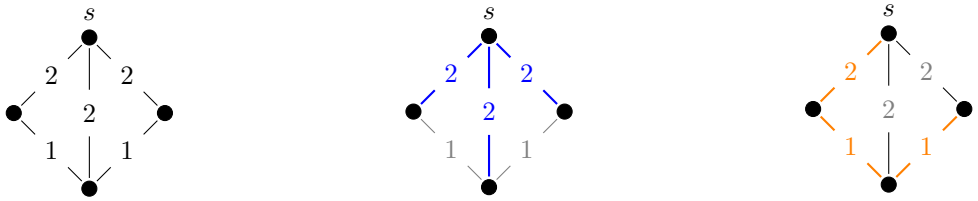
Aufgabe 7: Kürzeste-Wege-Bäume und minimale Spannbäume

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 8: 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 9: 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 3: 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 10: Greedy-Algorithmen

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

- 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 4: 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 5: 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 6: 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 7: 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 8: 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 11: Anzahl der einfachen Pfade im Graph

Verwenden Sie den Tiefensuche-Algorithmus, um die *Anzahl* der einfachen Pfade zwischen zwei Knoten in einem azyklischen, gerichteten und ungewichteten Graphen in $\mathcal{O}(|V| + |E|)$ Zeit zu berechnen. Verwendet Ihr Algorithmus das Prinzip dynamischer Programmierung oder ist er ein Greedy-Algorithmus?

Lösung: Die Idee ist, den Tiefensuche-Algorithmus zu modifizieren. Wir geben jedem Knoten u ein Attribut $u.\mathsf{paths}$, welches die Anzahl der Pfade von u zum Zielknoten t angibt. Dann ergibt sich schnell die Berechnungsvorschrift:

$$u.\mathsf{paths} = \begin{cases} \sum_{(u,v)\in E} \mathsf{countPaths}(v,t) & \text{falls } s \neq t \\ 1 & \text{sonst} \end{cases}$$

Sobald die Anzahl der Pfade eines Knotens berechnet wurden, müssen sie nicht erneut berechnet werden, sondern können direkt wiederverwendet werden. Damit verwendet der Algorithmus das Prinzip der dynamischen Programmierung.

Algorithmus 9: countPaths(s,t)

```
1 if s == t then
2     return 1
3 if s.paths = 0 then
4     for k ∈ Adj[s] do
5         s.paths = s.paths + countPaths(k, t)
6 return s.paths
```

Aufgabe 12: 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 10: 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 13: 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.