Aufgabensammlung ADS-Repetitorium 2021

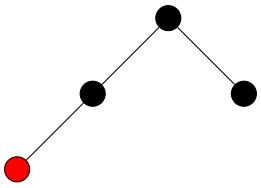
Bäume und Graphen

Aufgabe 1: Rot-Schwarz-Bäume

Zeichnen Sie die folgenden Beispiele. Im Folgenden werden die *nil*-Blätter nicht mitgezählt. Ein Baum, der nur aus einer Wurzel besteht, hat demnach einen Knoten und die Höhe eins.

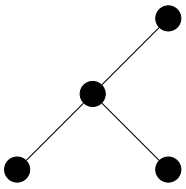
(a) Gesucht ist ein gültiger Rot-Schwarz-Baum der Höhe drei, bei dem die Anzahl der Knoten minimal ist.

Lösung:



(b) Gesucht ist ein binärer Suchbaum der Höhe drei mit maximaler Knotenzahl, für den es keine gültige Rot-Schwarz-Färbung gibt.

Lösung:



(c) Gesucht ist eine Permutation von sieben paarweise verschiedenen Zahlen, sodass diese beim Einfügen in einen Binärbaum alle Ebenen komplett ausfüllen (der Binärbaum ist also balanciert).

Lösung: Eine mögliche Permutation ist $\langle 4, 2, 1, 3, 6, 5, 7 \rangle$. Diese ist nicht eindeutig. Wichtig ist, das jeweils der Median der noch nicht eingfügten, zusammenhängenden Teillisten eingefügt wird.

Aufgabe 2: Modellierung als Graphen

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

i. Geben Sie einen Algorithmus in Worten an, der die Fragestellung als Graph modelliert. Achten Sie auf eine präzise Definition der Knoten- und Kantenmenge.

ii. Mit welchem Graph-Algorithmus kann die Fragestellung auf dem Graph gelöst werden? Wie müssen die Algorithmen gegebenfalls modifiziert werden?

iii. 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 3: Definition eines Graphen

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

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

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

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.

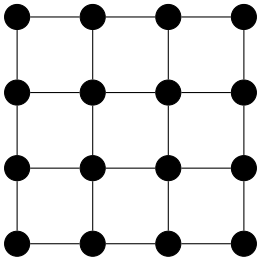
Lösung: Der Graph kann als ungerichteter Graph aufgefasst werden, da die Gewichtsfunktion symmetrisch ist, unabhängig davon, in welcher Reihenfolge u und v als Parameter übergeben werden.

(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:

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

(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 4: Augmentierung von Rot-Schwarz-Bäumen

Ein Rot-Schwarz-Baum zur Verwaltung einer dynamischen Menge verschiedener ganzer Zahlen soll so augmentiert werden, dass man zu jeder Zeit bestimmen kann, für welche zwei Zahlen i, j der Menge mit $i < j$ die Differenz $j - i$ am kleinsten ist. Geben Sie die Methode minGap(RedBlackTree T) in Pseudocode an, die das gesuchte Zahlenpaar in konstanter Zeit liefert. Welche Extrainformationen speichern Sie im Baum und wie lassen sich diese beim Einfügen, Löschen und Suchen aufrechterhalten, ohne die Laufzeiten der entsprechenden Methoden zu verschlechtern? Lässt sich das Problem auch mit konstantem Zusatzspeicher lösen, wenn man auf die Delete-Methode verzichtet?

Lösung: Wir augmentieren den Rot-Schwarz-Baum so, dass jeder Knoten v zusätzlich seinen Successor und den Knoten im Teilbaum mit Wurzel v, welcher unter allen Knoten im Teilbaum die minimale betragsmäßige Differenz zu seinem Successor hat. Dann löst folgender Algorithmus in $\mathcal{O}1$ die Aufgabe:

Algorithmus 1: minGap(RedBlackTree T)

```
1 root = T.root
2 return (root.minGapNode, root.minGapNode.successor)
```

Beim Einfügen eines neuen Knotens y wird der Successor in $\mathcal{O}(n \log n)$ Zeit und der Predecessor x von y in $\mathcal{O}(n \log n)$ berechnet. Anschließend werden in konstanter Zeit die successor-Attribute von x und y aktualisiert. Vor RBInsertFixUp ist y ein Blatt und y.minGapNode zeigt auf y. Der Wert von x.minGapNode ist nun x oder der bisherige Wert. Dies lässt sich in $\mathcal{O}(1)$ Zeit feststellen. Nun müssen

die Werte nach oben bis zur Wurzel getragen werden und gegebenfalls aktualisiert werden. Das kann in $\mathcal{O}(\log n)$ geschehen. Bei den nun folgenden Rotationen können die minGapNode in konstanter Zeit neu gesetzt werden. Daher ändert sich die asymptotische Laufzeit der Einfügeoperation nicht. Das Löschen funktioniert analog.

Verzichtet man auf die Delete Methode, kann man das Problem mit konstanten Zusatzspeicher lösen. Man speichert sich einfach immer die Knoten minGapNode1 ,minGapNode2, die in dem Moment den geringsten Abstand haben. Beim Einfügen muss man lediglich in $\mathcal{O}(\log n)$ Zeit, den Predecessor und Successor des neuen Knoten bestimmen und ggf. minGapNode1 und minGapNode2 aktualisieren.

Aufgabe 5: 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 2: 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 6: 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 7: 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 8: 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.

Graph G ist bipartit $\Leftrightarrow$

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

(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 4: boolean checkIfBipartite(Graph G, Vertex u = **nil**)

```
1  if u = nil 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 9: 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 5: countPaths(s,t)

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

Aufgabe 10: 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 1a, 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.

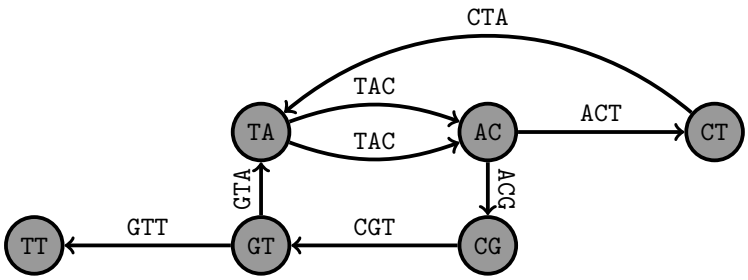
(e) Geben Sie eine Sequenz s an, deren Sammlung der k-mere identisch zu denen aus c) ist.

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:



(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).

Aufgabe 11: Dijkstra's Alogrithmus

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

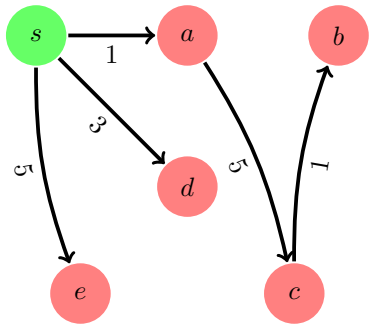
(a) Führen Sie Dijkstra's Algorithmus mit Startknoten s aus. Erstellen Sie dazu eine Tabelle mit drei Spalten: *Iteration*, *Schwarze Knoten*, *Graue Knoten*, zu der sie nach jeder Iteration der While-Schleife eine Zeile hinzufügen. Schreiben Sie hinter jeden Knoten in Klammern die aktuelle Distanz und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

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	$b(7, c)$	

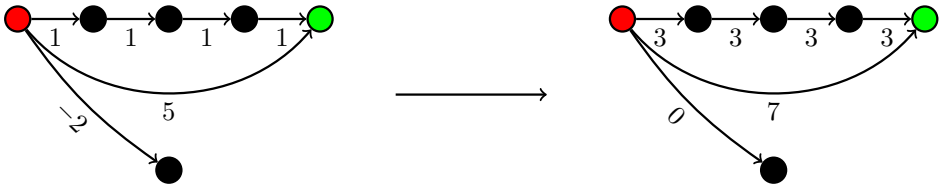
(b) Zeichnen Sie den entstandenen Kürzeste-Wege-Baum.

Lösung:

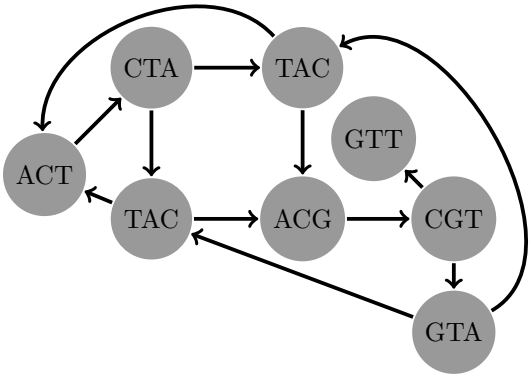


(c) Seien nun negative Kantengewichte erlaubt. Verändern Sie den gegeben Graphen, sodass Dijkstra nach Beendigung kein richtiges Ergebnis liefert, obwohl der kürzeste Weg definiert ist. Warum kann dies nicht behoben werden, indem wir das minimale Kantengewicht $g < 0$ identifizieren und alle Gewichte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

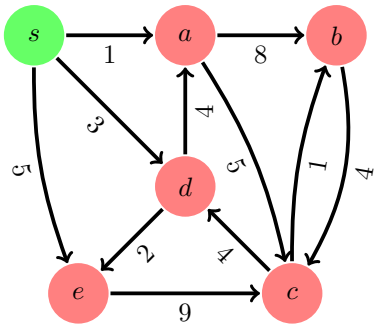
Lösung: Wir können auf dem gegeben Graphen die das Gewicht Kante (d, a) auf -1 setzen. Dann hat der kürzeste Weg nach a die Länge 2. Die Wege nach c und b verkürzen sich ebenfalls um 1. Dijkstra erkennt dies nicht, da zuerst a behandelt wird und danach schwarz gefärbt ist. Knoten, die einmal schwarz 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.



(a) 3-mere-Graph für die Sequenz-Aufgabe.



(b) Beispiel-Graph für Dijkstras-Algorithmus.

Abbildung 1: Graphen für die Aufgaben 8 und 9.