Kapitel 7

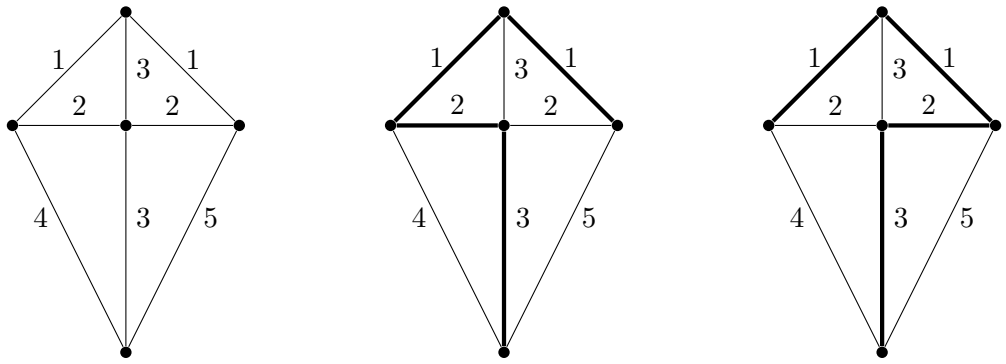
Gierige Algorithmen

Ein gieriger Algorithmus stellt die Lösung für ein Problem dadurch zusammen dass er zu jedem Zeitpunkt der Lösung jenen Teil hinzufügt der zu diesem Zeitpunkt am vielversprechendsten ist. Einmal gemachte Entscheidungen werden dabei nicht mehr revidiert. Auf diese Weise werden lokal optimale Bausteine zu einer Lösung zusammengefügt. Für gewisse Probleme wird dadurch eine global optimale Lösung erreicht. Wenn gierige Algorithmen auch nicht immer optimale Lösungen liefern, so haben sie doch den Vorteil dass sie schnell (im Sinne der Laufzeit) und einfach (im Sinne des Implementierungsaufwands) sind.

7.1 Der Algorithmus von Kruskal

Ein gutes Beispiel für einen gierigen Algorithmus is der Algorithmus von Kruskal zur Berechnung eines minimalen Spannbaums eines zusammenhängenden endlichen (gerichteten) Graphen, vgl. Abschnitt 2.3. Dieser Algorithmus ist leicht erklärt: Starte mit einer leeren Menge von Kanten B und wiederhole den folgenden Schritt so oft wie möglich: füge von den noch nicht betrachteten Kanten des Eingabegraphen G eine mit minimalen Kosten zu B hinzu falls dadurch kein Zyklus entsteht. Dieser Algorithmus vereinigt also sukzessive Bäume zu Bäumen, begonnen bei Knoten-Singletons bis daraus ein Spannbaum entsteht. Es handelt sich um einen gierigen Algorithmus da in jedem Schritt ein lokal optimales Element der Lösung (eine Kante mit minimalen Kosten) hinzugefügt wird ohne sich darum zu kümmern, ob dadurch eine global optimale Lösung entsteht (ein minimaler Spannbaum).

Beispiel 7.1. Ein Graph mit Kantenkosten die zwei minimale Spannbäume erlauben:



Satz 7.1. *Der Algorithmus von Kruskal ist korrekt.*

Wir wollen jetzt einen Korrektheitsbeweis des Algorithmus von Kruskal angeben. Der Algorithmus geht wie folgt vor: sei $e_1, \ldots, e_n$ eine Sortierung von E so dass $c(e_1) \leq \cdots \leq c(e_n)$, sei $E_0 = \emptyset$ und

$$E_{i+1} = \begin{cases} E_i \cup \{e_{i+1}\} & \text{falls } (V, E_i \cup \{e_{i+1}\}) \text{ zyklenfrei ist und} \\ E_i & \text{sonst.} \end{cases}$$

Der Algorithmus antwortet mit (V, E_n).

Satz 7.2. *Der Algorithmus von Kruskal ist korrekt.*

Beweis. Zunächst ist klar dass $B = (V, E_n)$ zyklenfrei ist. B ist aber auch zusammenhängend: Angenommen B wäre nicht zusammenhängend, seien dann $v, w \in V$ in unterschiedlichen Zusammenhangskomponenten von B. Dann gibt es einen Pfad p von v nach w in G und eine Kante e_i auf p mit der die Zusammenhangskomponente von v verlassen wird und damit auch $e_i \notin E_n$. Dann ist aber $(V, E_n \cup \{e_i\})$ zyklenfrei und damit auch $(V, E_i \cup \{e_i\})$ zyklenfrei, also $e_i \in E_n$, Widerspruch. B ist also ein Spannbaum von G.

Für die Minimalität von B reicht es zu zeigen dass für alle $i \in \{0, \ldots, n\}$ ein minimaler Spannbaum von G mit Kantenmenge M_i existiert so dass $E_i \subseteq M_i$. Dann ist nämlich $E_n = M_n$ und damit B minimal. Wir gehen mit Induktion nach i vor. Der Fall $i = 0$ ist trivial. Für den Induktionsschritt definieren wir $M_{i+1} = M_i$ falls keine Kante hinzugefügt wird oder die hinzugefügte Kante $e_{i+1} \in M_i$ ist. Sei nun also $E_{i+1} = E_i \cup \{e_{i+1}\}$, (V, E_{i+1}) zyklenfrei und $e_{i+1} \notin M_i$. Dann enthält $(V, M_i \cup \{e_{i+1}\})$ einen Zyklus. Da aber (V, E_{i+1}) zyklenfrei ist, existiert ein e_j auf dem Zyklus mit $e_j \notin E_{i+1}$. Sei nun $M_{i+1} = (M_i \setminus \{e_j\}) \cup \{e_{i+1}\}$. Dann ist (V, M_{i+1}) zusammenhängend da (V, M_i) zusammenhängend ist und in jedem M_i-Pfad die Kante e_j durch e_{i+1} und den Rest des Zyklus ersetzt werden kann. Weiters ist $|M_{i+1}| = |M_i| = |V| - 1$ da ja $e_j \in M_i$ und $e_{i+1} \notin M_i$ ist und (V, M_{i+1}) mit Satz 2.3 also ein Spannbaum von G. Außerdem ist $c(e_j) \geq c(e_{i+1})$ denn $c(e_j) < c(e_{i+1})$ impliziert $j < i + 1$ sowie $(V, E_j \cup \{e_j\})$ zyklenfrei und damit $e_j \in E_i$. Also ist $c(M_{i+1}) \leq c(M_i)$ und, da M_i minimal ist, $c(M_{i+1}) = c(M_i)$ und M_{i+1} also ebenfalls ein minimaler Spannbaum. □

Um die Laufzeit des Algorithmus von Kruskal zu analysieren formulieren wir ihn zunächst im Detail aus. Im Prinzip wäre es möglich für jede Kante einen expliziten Test auf die Zyklenfreiheit des Graphen (z.B. mit Breiten- oder Tiefensuche) durchzuführen, um zu entscheiden, ob diese Kante zum Wald hinzugefügt werden soll. Allerdings ist es geschickter, explizite Tests auf Zyklenfreiheit zu unterlassen da diese viel Zeit benötigen. Stattdessen wollen wir die Zusammenhangskomponenten des Waldes mitführen und auf dem aktuellen Stand halten. Der Algorithmus von Kruskal erzeugt einen Spannbaum durch sukzessive Vereinigung zweier Bäume in einem Wald zu einem neuen Baum durch Hinzufügen einer Kante. Er benötigt also eine Datenstruktur zur Darstellung einer Partition einer endlichen Menge, die Partition der Knoten in Zusammenhangskomponenten. Eine solche Datenstruktur zur Darstellung einer Partition einer endlichen Menge wird auch als *union-find*-Datenstruktur bezeichnet. Sie stellt (zumindest) die folgenden Operationen zur Verfügung:

1. INITIALISIEREN(M) erzeugt Datenstruktur für die Partition von M die aus Singleton-Klassen besteht.

2. VEREINIGEN(P, x_1, x_2) für zwei Elemente x_1 und x_2 von M.

3. FINDEN(P, x) gibt für $x \in M$ die Klasse C zurück die x enthält.

Algorithmus 27 Algorithmus von Kruskal

```
Prozedur KRUSKAL(V, E, c)
    B := ∅
    P := INITIALISIEREN(V)
    Für alle (v, w) ∈ E in nichtfallender Reihenfolge
        Falls FINDEN(P, v) ≠ FINDEN(P, w) dann
            B := B ∪ {{v, w}}
            VEREINIGEN(v, w)
        Ende Falls
    Ende Für
    Antworte (V, B)
Ende Prozedur
```

Auf Basis einer solchen Datenstruktur für eine Partition kann der Algorithmus von Kruskal dann wie in Algorithmus 27 ausformuliert werden. Eine einfache Implementierung der Partitionsdatenstruktur kann wie folgt erreicht werden. Sei $M = \{1, \ldots, n\}$. Wir verwenden auch zur Bezeichnung der Äquivalenzklassen Zahlen von 1 bis n. Wir arbeiten mit einem Datenfeld *ElementInKlasse* der Länge n wobei *ElementInKlasse*$[i]$ die Klasse ist in der sich das Element i befindet. Wir benötigen weiters ein Datenfeld *Kardinalität* der Länge n wobei *Kardinalität*$[i]$ die Kardinalität der Klasse i ist. Und schließlich verwenden wir noch ein weiteres Datenfeld *Elemente* der Länge m wobei *Elemente*$[i]$ eine einfach verkettete Liste der Elemente der Klasse i ist.

Beispiel 7.2. Sei $M = \{1, \ldots, 8\}$. Nach Initialisierung und den Aufrufen VEREINIGEN$(P, 2, 4)$, VEREINIGEN$(P, 6, 8)$ und VEREINIGEN$(P, 1, 4)$ haben die Datenfelder den folgenden Inhalt:

ElementInKlasse:	$2, 2, 3, 2, 5, 6, 7, 6$
Kardinalität:	$0, 3, 1, 0, 1, 2, 1, 0$
Elemente:	NIL; 2, 4, 1; 3; NIL; 5; 6, 8; 7; NIL

Die Initialisierung benötigt Zeit $\Theta(n)$. Das Finden eines Elements geschieht direkt durch Nachsehen im Datenfeld *ElementInKlasse* und benötigt daher nur Zeit $\Theta(1)$. Für das Vereinigen der Klassen von x_1 und x_2 gehen wir wie folgt vor: zunächst seien C_1 die Klasse von x_1 und C_2 die Klasse von x_2. Dann kann über *Kardinalität* festgestellt werden welche der beiden Klassen die größere ist, sei o.B.d.A. C_1 die größere und C_2 die kleinere. Wir entfernen dann die Klasse C_2 und füge alle ihre Elemente der Klasse C_1 hinzu. Die dementsprechende Aktualisierung der Datenfelder benötigt Zeit $\Theta(|C_2|)$ da die zu ändernden Indices im Datenfeld *Elemente* zu finden sind.

Der Algorithmus von Kruskal führt $|V| - 1$ Vereinigungen aus. Jeder der Klassen hat höchstens die Größe $|V|$. Eine naive obere Schranke ist also $O(|V|^2)$ für die Laufzeit der Vereinigungsoperationen. Diese Schranke kann durch eine genauere Analyse allerdings verbessert werden. Sie ist nämlich insofern nicht realistisch als sie zwei miteinander inkompatible Annahmen über den schlechtesten Fall kombiniert, nämlich 1. dass viele Vereinigungen stattfinden und 2. dass diese Vereinigungen großer Mengen sind.

Lemma 7.1. *In der Datenfeld-Implementierung der Partitionsdatenstruktur benötigen k sukzessive Vereinigungsoperationen beginnend bei der Singleton-Partition höchstens $O(k \log k)$ Zeit.*

Beweis. In k (binären) Vereinigungsoperationen sind höchstens $2k$ Elemente von M involviert. Sei $i \in M$. Wir schreiben $C_j(i)$ für den Index der Klasse von i nach der j-ten Vereinigungs-

operation und $|C_j(i)|$ für die Kardinalität dieser Klasse. Dann ist $|C_k(i)| \leq 2k$.[1] Außerdem gilt $C_{j+1}(i) \neq C_j(i) \Rightarrow |C_{j+1}(i)| \geq 2|C_j(i)|$ da ja bei jeder Vereinigungsoperation der Index der größeren Klasse behalten wird. Deshalb kann sich also die Klassenzugehörigkeit von i höchstens $\log 2k$ mal ändern. Da in k Vereinigungen höchstens $2k$ Elemente involviert sind, erhalten wir also für die gesamte Laufzeit $O(2k \log 2k) = O(k \log k)$. □

In einer Situation wie der des obigen Lemmas sagen wir auch dass eine einzelne von k Vereinigungsoperation *amortisierte Kosten* $\frac{1}{k}O(k \log k) = O(\log k)$ hat. Wir sprechen dann auch von *amortisierter Laufzeitanalyse*. Eine solche Vorgehensweise zur Abschätzung des schlechtesten Falls ist immer dann sinnvoll wenn mehrere Operationen durchgeführt werden von denen zwar eine einzelne recht teuer sein kann, die aber insgesamt im Durchschnitt nicht sehr teuer sind. In einer ähnlichen Situation waren wir auch bei der Abschätzung der Laufzeit der Erzeugung einer Halde aus einem Datenfeld im Kontext von Haldensortieren.

Die Gesamtlaufzeit des Algorithmus von Kruskal setzt sich also wie folgt zusammen: Wir benötigen $O(|V|)$ Zeit für die Initialisierung der Partitionsdatenstruktur, $O(|E| \log |E|)$ Zeit für das Sortieren der Kanten, $O(|E|)$ Schleifendurchläufe sowie $O(|V| \log |V|)$ für die $|V| - 1$ Vereinigungsoperationen. Da in einem zusammenhängenden Graphen $|E| \geq |V| - 1$ ist, erhalten wir insgesamt also $O(|E| \log |E|)$. Weiters ist ja $|E| \leq |V|^2$, also $\log |E| \leq 2 \log |V|$ und damit $\log |E| = O(\log |V|)$. Somit kann die Laufzeit des Algorithmus von Kruskal auch geschrieben werden als $O(|E| \log |V|)$.

7.2 Der Algorithmus von Prim

Wir betrachten jetzt noch einen zweiten, ebenfalls gierigen, Algorithmus zur Berechnung eines minimalen Spannbaums: den Algorithmus von Prim.

Definition 7.1. Sei $G = (V, E)$ und $c : E \to \mathbb{R}_{\geq 0}$, sei $G' = (V', E')$ ein Teilgraph von G und sei $v \in V \setminus V'$. Dann sind die *Anschlusskosten* von v an G' bezüglich c

$$\min\{c(\{u, v\}) \mid \{u, v\} \in E, u \in V'\}$$

bzw. $+\infty$ falls diese Menge leer ist.

Der Algorithmus von Prim geht wie folgt vor: gegeben einen Graphen $G = (V, E)$ mit $|V| = n$ und eine Kostenfunktion $c : E \to \mathbb{R}_{\geq 0}$ setzen wir zu Beginn $V_1 = \{s\}$, $E_1 = \emptyset$. Für $i = 2, \ldots, n$ sei $v_i \in V \setminus V_i$ der Knoten mit minimalen Anschlusskosten an (V_i, E_i) und $\{u_i, v_i\}$ eine Kante die diese Anschlusskosten realisiert. Dann setzen wir $V_{i+1} = V_i \cup \{v_i\}$ und $E_{i+1} = E_i \cup \{\{u_i, v_i\}\}$. Der Algorithmus antwortet mit (V_n, E_n). Auf diese Weise verwalten wir einen wachsenden Baum der Teilgraph von G ist.

Satz 7.3. *Der Algorithmus von Prim ist korrekt.*

Beweis. Zunächst ist leicht zu beobachten dass alle $B_i = (V_i, E_i)$ Bäume sind. Da $V_n = V$ ist B_n ein Spannbaum von G. Für die Minimalität von B_n reicht es zu zeigen dass für alle $i \in \{1, \ldots, n\}$ ein minimaler Spannbaum von G mit Kantenmenge M_i existiert so dass $E_i \subseteq M_i$. Dann ist nämlich $M_n = E_n$ und damit B_n minimal. Wir gehen mit Induktion nach i vor. Der Fall $i = 1$ ist trivial. Für den Induktionsschritt machen wir eine Fallunterscheidung: Falls $\{u_i, v_i\} \in M_i$, sei $M_{i+1} = M_i$. Falls $\{u_i, v_i\} \notin M_i$, dann muss M_i eine Kante $\{w, w'\}$ enthalten

[1] Tatsächlich könnte man das noch genauer abschätzen. Das ist aber für dieses Resultat nicht nötig.

mit $w \in V_i$ und $w' \notin V_i$ da M_i zusammenhängend ist. Dann ist $\{w, w'\} \neq \{u_i, v_i\}$ und da v_i minimale Anschlusskosten hat ist $c(\{u_i, v_i\}) \leq c(\{w, w'\})$. Wir definieren $M_{i+1} = (M_i \setminus \{\{w, w'\}\}) \cup \{\{u_i, v_i\}\}$. Nun ist (V_i, E_i) und G zusammenhängend, also gibt es einen Pfad p von w nach u_i und einen Pfad q von v_i nach w'. Somit kann also in jedem M_i-Pfad die Kante $\{w, w'\}$ durch $p, \{u_i, v_i\}, q$ ersetzt werden um daraus einen M_{i+1}-Pfad zu erhalten. M_{i+1} ist also zusammenhängend und da $|M_{i+1}| = |M_i| = |V| - 1$ ist M_{i+1} mit Satz 2.3 ein Spannbaum von G. Außerdem ist $c(M_{i+1}) \leq c(M_i)$ und, da M_i minimal ist, ist $c(M_{i+1}) = c(M_i)$ und damit (V, M_{i+1}) ein minimaler Spannbaum von G. □

Im Prinzip wäre es hier möglich, den Knoten mit minimalen Anschlusskosten in jeden Schritt neu zu bestimmen. Diese Vorgehensweise kostet aber unnötig Laufzeit. Geschickter ist es zur Implementierung des Algorithmus von Prim die Knoten in einer nach Minimum sortierten Prioritätswarteschlange abzulegen. Für einen Knoten v ist $v.x$ die Priorität, also die minimalen Anschlusskosten bezüglich des aktuellen Baums. Wir führen ein Datenfeld B mit so dass $B[v] =$ **wahr** ist genau dann wenn der Knoten v im aktuellen Baum enthalten ist. Die Kanten des Baums werden implizit konstruiert, indem wir mit jedem Knoten v das Feld $v.\mathit{Vorgänger}$ mitführen das auf jenen Knoten zeigt mit dem die minimalen Anschlusskosten erreicht werden können. Bei Bedarf kann der Baum dann aus diesen Feldern explizit gemacht werden. Die Laufzeit des Algorithmus von Prim setzt sich wie folgt zusammen: Die Initialisierung in-

Algorithmus 28 Algorithmus von Prim

```
Prozedur PRIM(V, E, c, s)
    Sei B ein neues Datenfeld der Länge |V|, überall mit falsch initialisiert
    Für alle v ∈ V
        v.x := ∞
        v.Vorgänger := NIL
    Ende Für
    s.x := 0
    Q := MIN-PRIORITÄTSWARTESCHLANGE(V)
    Solange Q nicht leer
        v := EXTRAHIEREMINIMUM(Q)
        B[v] := wahr
        Für alle (v, w) ∈ E
            Falls c(v, w) < w.x und B[w] = falsch dann
                w.Vorgänger := v
                REDUZIEREPRIORITÄT(Q, w, c(v, w))
            Ende Falls
        Ende Für
    Ende Solange
    Antworte (V, {{u, v} | u = v.Vorgänger})
Ende Prozedur
```

klusive Aufbau der Prioritätswarteschlange benötigt Zeit $O(|V|)$, ebenso die Berechnung der Antwort. Die äußere Schleife wird $O(|V|)$ mal durchlaufen wobei die Extraktion des Minimums jeweils $O(\log |V|)$ Zeit benötigt. Die innere Schleife wird $|E|$ mal durchlaufen wobei bei jedem Durchlauf durch die Änderung der Priorität Kosten von $O(\log |V|)$ anfallen können. In einem zusammenhängenden Graphen ist $|E| = \Omega(|V|)$, insgesamt erhalten wir also eine Laufzeit von $O(|E| \log |V|)$.