Algorithmen & Datenstrukturen

Übungsstunde 12

Michelle Honegger - CAB G59 - 08.12.2024

Miniquiz 8

- Passwort: **unionfind2**
- 10 Fragen
- 12 Minuten
- 0.1 Punkte pro richtige Antwort

Programm

- Miniquiz
- Besprechung Aufgaben
- Theory Recap
- Prüfungsaufgaben
- Semester-Recap Part 1

Besprechung Miniquiz

Sei d das berechnete Array nach $n-1$ Iterationen des Algorithmus von Bellmand und Ford (auf einem gerichteten Graphen G) und d' das nach n Iterationen. Wenn es einen negativen Kreis in G gibt, dann gilt $d'[v] < d[v]$ für jeden Knoten v.

- Wahr
- Falsch

Erinnern Sie sich daran, dass der Algorithmus von Boruvka in jeder Iteration eine Menge von Kanten zu seinem aktuellen Wald hinzufügt. Im Worst-Case, benötigt der Algorithmus von Boruvka eine lineare Anzahl solcher Iterationen.

- Wahr
- Falsch

Das Konzept eines minimalen Spannbaums kann auch für einen Graphen mit möglicherweise negativen Kosten definiert werden (wir summieren nach wie vor die Kosten und suchen den kleinstmöglichen Wert).

Wahr/Falsch: Der Algorithmus von Prim berechnet auch auf Graphen, in denen einige Kanten negative Kosten haben, einen minimalen Spannbaum.

- Wahr
- Falsch

Nehmen Sie an, dass wir einen Graphen $G = (V, E)$ mittels Adjazenzlisten speichern. Die Laufzeiten der Algorithmen von Prim, Boruvka and Dijkstra sind dann allesamt höchstens $O((|V| + |E|) \log |V|)$.

- Wahr
- Falsch

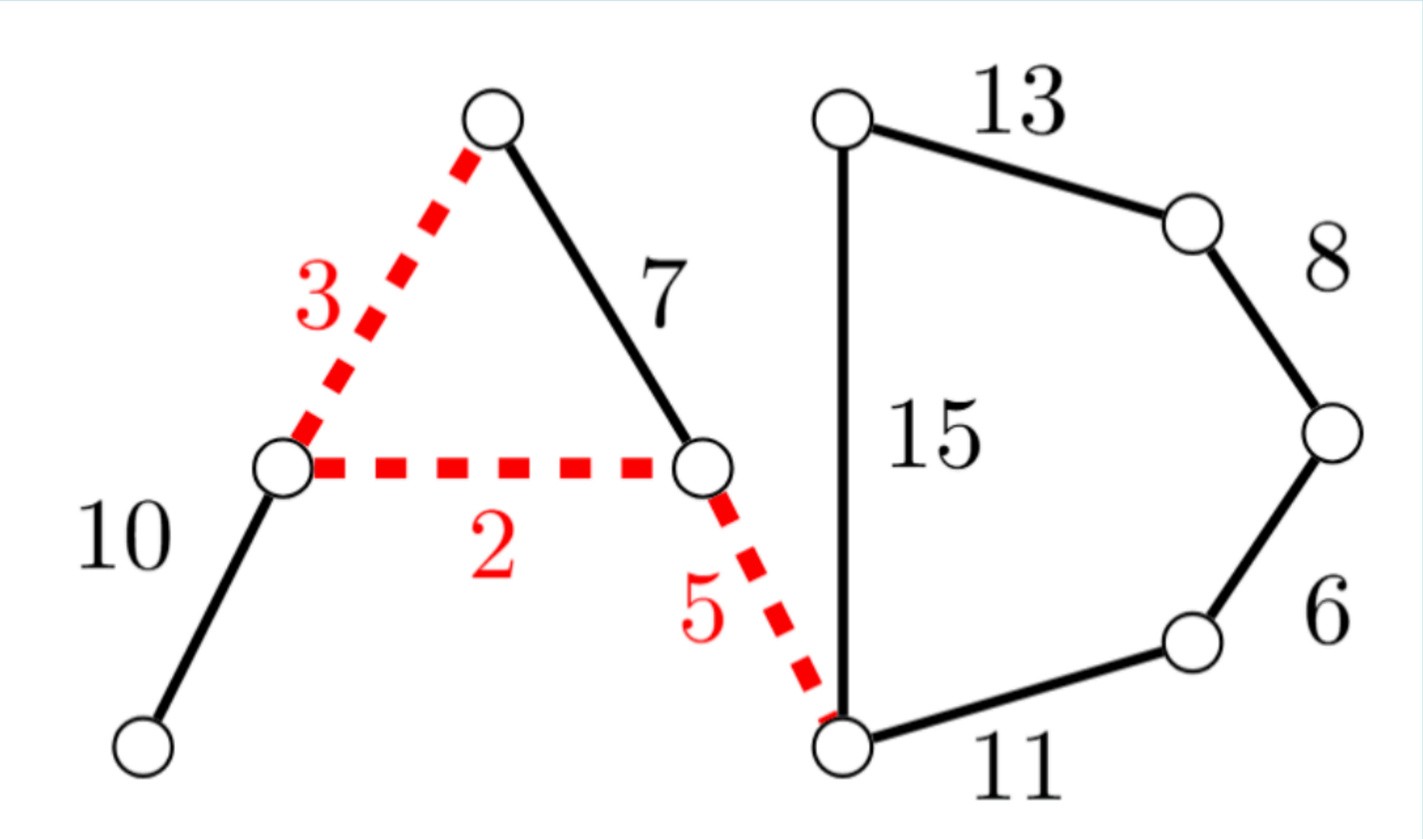
Sei $G = (V, E)$ ein ungerichteter, gewichteter Graph mit positiven Kosten. Nehmen Sie an, dass jeder kürzeste Pfad (hier im Sinne von günstigst) in diesem Graphen höchstens h Kanten verwendet (also ist die Länge in Kanten (und nicht Kosten) gezählt höchstens h). Wir können die kürzesten Wege (immer noch im Sinne von günstigt) von einem Startknoten v zu allen anderen Knoten (SSSP) in $O(h|E|)$ Zeit berechnen.

- Wahr
- Falsch

Sei G ein gewichteter ungerichteter Graph mit verschiedenen Kantenkosten (es gibt keine zwei Kanten mit den gleichen Kosten). Sei $e_{\max}$ die Kante mit dem grössten Gewicht. Dann, ist $e_{\max}$ in keinem minimalen Spannbaum (MST) von G enthalten.

- Wahr
- Falsch

Wir sehen unten einen Graphen auf dem wir den Algorithmus von Kruskal teilweise ausgeführt haben. Die Kanten die bereits durch den Algorithmus hinzugefügt wurden sind gestrichelt und rot gefärbt. Was ist das Gewicht der Kante, die als nächstes hinzugefügt wird?



Antwort:

Die Worst-Case Laufzeit einer einzelnen union Operation auf der Union-find Datenstruktur, die wir in der Vorlesung gesehen haben, ist höchstens $O(\log n)$.

- Wahr
- Falsch

Wie wir in der Vorlesung gesehen haben, benutzt der Algorithmus von Kruskal ein Array repr in dem jeder Knoten seinen Repräsentanten speichert. Rufen Sie sich in Erinnerung, dass es in diesem Array am Anfang n verschiedene Werte (da $\mathrm{repr}[i] = i$ für alle i) und am Ende nur noch einen verschiedenen Wert gibt.

Wahr oder Falsch: Nehmen Sie an, dass der Algorithmus von Kruskal zu einem Zeitpunkt x Kanten zum Wald hinzugefügt hat. Dann enthält das Array repr genau $n - x$ verschiedene Werte.

- Wahr
- Falsch

Betrachten Sie das folgende Array von Repräsentanten im Kontext der Suche eines minimalen Spannbaums (MST) in einem Graphen mit Knoten $0, 1, 2, 3, 4, 5$ mittels einer Union-find Datenstruktur.

$$[1, 1, 1, 3, 3, 5].$$

Nehmen Sie jetzt an, dass wir die Zusammenhangskomponenten der Knoten 1 und 3 mit der optimierten union Operation vereinen, was zu einer Laufzeit von $O(|V| \log |V|)$ für den union Schritt im Algorithmus von Kruskal führt. Was ist das daraus resultierende Array?

- a. $[1, 1, 1, 3, 3, 5]$
- b. $[3, 3, 3, 3, 3, 5]$
- c. $[1, 1, 1, 1, 1, 5]$
- d. $[1, 1, 1, 1, 3, 5]$

Theory Recap

Union Find Datenstruktur

Operation	Funktion	Implementierung	Laufzeit
make(V)	erstelle Datenstruktur für F=ø	rep[v] <- v für v in V	O(n)
same(u,v)	teste ob u,v in derselben ZHK von F	teste ob rep[u] = rep[v]	O(1)
union(u,v)	vereinige die ZHKs von u und v	for x in members[rep[u]] rep[x] <- rep[v]	O(min{|ZHK(u)|,|ZHK(v)|})

Kruskal

- Kanten nach Gewicht sortieren
- Dann Kanten hinzufügen, insofern Endpunkte in versch. ZHKs
- Laufzeit: $O(|E| \cdot log(|E|) + |V| \cdot log(|V|))$

Kruskal

Laufzeitherleitung

- $N_u :=$ #Änderungen von rep[u]
  - N_u wird nur verändert, wenn |ZHK(u)|≤|ZHK(v)|
  - |ZHK(u)| wird mindestens verdoppelt
  - $N_u \leq log_2(n)$

- $O(\underbrace{|E| \cdot log(|E|)}_{\text{sortieren}} + \underbrace{|V| \cdot log(|V|)}_{\text{Nu}})$

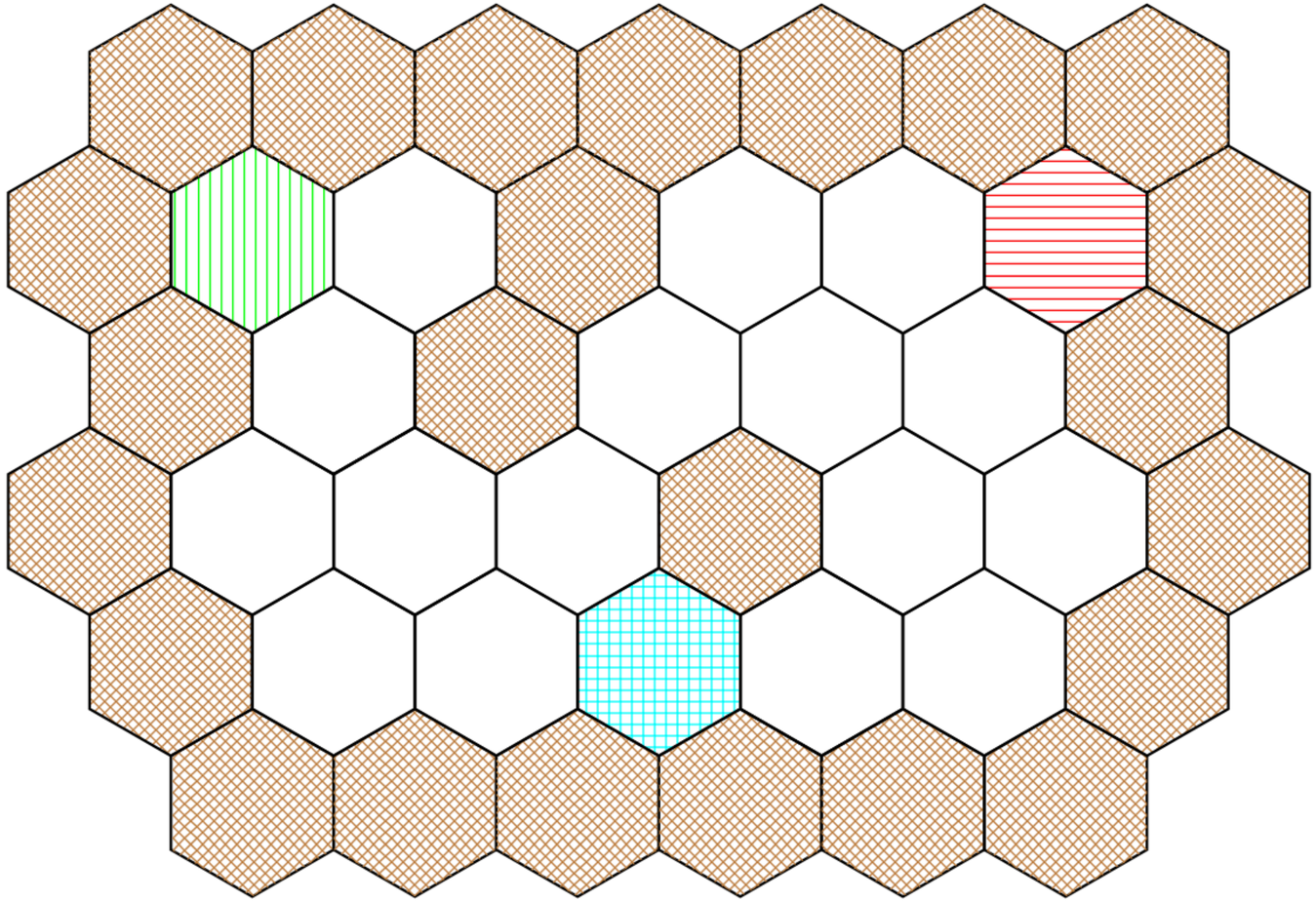
müssen alle Nu berücksichtigen

You are alone and stranded in a desert trying to reach the only city. The desert can be divided into hexagons and is surrounded by impassable cliffs. The desert terrain varies, and hexagons take differing amounts of time to traverse. There are oases in the desert.

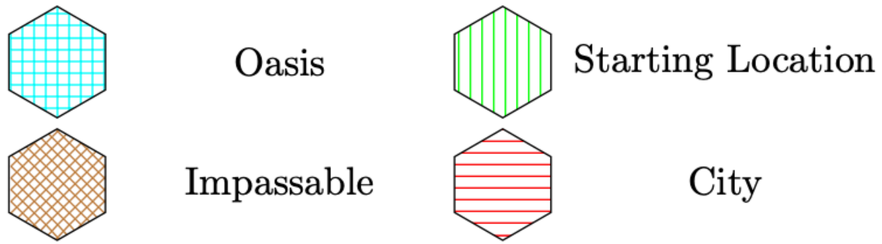
- In this problem, you move from the center of one hexagon to the center of an adjacent hexagon.
- The amount of time spent traveling from the center of one hexagon to another is equal to half the amount of time needed to cross the first hexagon plus half the amount of time needed to cross the second.

a) Your goal is to minimize the amount of time it takes to get from the starting location to the city. Model the problem with a graph in such a way that an algorithm from the lecture computes a solution to the problem. Describe precisely the set of vertices and edges, and describe what the vertices and edges represent. Name an, as efficient as possible, algorithm from the lecture to solve this problem, and state the running time of the algorithm in terms of the number of hexagons, N.

Example:



Legend:



b) We now modify the problem, adding the following rules:

- You begin a full water bottle that you can drink from D times, and you can refill it at each *oasis*.
- When you arrive at a hexagon that isn't an oasis or the city, you must drink water from your water bottle.
- If your water bottle is empty, you become dehydrated.

Your goal is to find the fastest route from the starting location to the city, without becoming dehydrated.

Model the modified problem with a graph in such a way that an algorithm from the lecture computes a solution to the problem. Describe precisely the set of vertices and edges, and describe what the vertices and edges represent. Name an, as efficient as possible, algorithm from the lecture to solve this problem, and state the running time of the algorithm in terms of the number of hexagons, N, and the size of your water bottle, D.

c) We now further modify the problem from (b) so that there are magic portals at the center of some hexagons in the desert. When you enter such a portal, it teleports you back in time, and you exit the portal in a different location in the desert. The teleportation does not affect your thirst, only the time passed and the location. For a fixed portal, the destination and time lapse are always the same, and they are known to you. Your goal is to reach the city at a time as early as possible. You still must avoid dehydration.

Model the modified problem with a graph in such a way that an algorithm from the lecture computes a solution to the problem. Describe precisely the set of vertices and edges, and describe what the vertices and edges represent. Name an, as efficient as possible, algorithm from the lecture to solve this problem, and state the running time of the algorithm in terms of the number of hexagons, N, and the size of your water bottle, D. It is part of your solution to decide whether you can go back arbitrarily far in time by repeatedly using the same portals.

Rechenregeln

Summen:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}, \quad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}, \quad \sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2$$

Logarithmen:

$$\log(ab) = \log a + \log b, \quad \log\left(\frac{a}{b}\right) = \log a - \log b,$$

$$\log(a^k) = k \log a, \quad \log_a b = \frac{\log b}{\log a}$$

O-Notation und asymptotisches Verhalten

Definitionen:

$$f(n) \in O(g(n)) \Leftrightarrow \exists c, n_0 : f(n) \leq c \cdot g(n) \; \forall n \geq n_0$$

$$f(n) \in \Omega(g(n)) \Leftrightarrow f(n) \geq c \cdot g(n)$$

$$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n)) \cap \Omega(g(n))$$

Typische Wachstumsraten:

$$1 \ll \log n \ll n \ll n \log n \ll n^2 \ll n^3 \ll 2^n \ll n!$$

Suchalgorithmen – Übersicht

Algorithmus	Worst Case	Best Case	Voraussetzung
Linear Search	$O(n)$	$O(1)$	Keine
Binary Search	$O(\log n)$	$O(1)$	Array sortiert

Ideen:

- **Linear Search:** Durchlaufe das Array von links nach rechts.
- **Binary Search:** Teile das sortierte Suchintervall halb; verwerfe jeweils eine Hälfte.

Sortieralgorithmen – Übersicht

Algorithmus	**Best**	**Worst**	**Bedingung**
Bubble Sort	$O(n)$	$O(n^2)$	Best: sortiert; Worst: reverse
Selection Sort	$O(n^2)$	$O(n^2)$	Unabhängig vom Input
Insertion Sort	$O(n)$	$O(n^2)$	Best: sortiert; Worst: reverse
Merge Sort	$O(n \log n)$	$O(n \log n)$	Immer gleich
Heap Sort	$O(n \log n)$	$O(n \log n)$	Immer gleich
Quick Sort	$O(n \log n)$	$O(n^2)$	Best: gute Pivot-Wahl; Worst: schlechtes Pivot

Kurzideen:

- **Bubble Sort:** Wiederholt benachbarte Elemente vertauschen, grösstes Element nach oben 'bubblen'.
- **Selection Sort:** Minimum finden und vorne platzieren.
- **Insertion Sort:** Sortiertes Präfix erweitern (Kartenspiel).
- **Merge Sort:** Teile und herrsche – rekursives halbieren, dann mergen.
- **Heap Sort:** Max-Heap aufbauen, größtes Element extrahieren.
- **Quick Sort:** Pivot wählen, partitionieren und rekursiv sortieren.

Datenstrukturen – Übersicht

Struktur	Operationen	Laufzeiten
Array	Zugriff, Setzen	$O(1)$
Linked List (Doubly)	Einfügen/Entfernen am Knoten	$O(1)$ (wenn Knoten bekannt), sonst $O(n)$
Stack	Push, Pop (LIFO)	$O(1)$
Queue	Enqueue, Dequeue (FIFO)	$O(1)$
Priority Queue (Heap)	Insert, Extract-max/min	$O(\log n)$, Peek $O(1)$
Heap (Binary)	Heapify, Insert, Extract	$O(\log n)$, Build-Heap $O(n)$
Binary Search Tree	Find, Insert, Delete	$O(\log n)$

Ideen:

- **Array:** Feste Größe, schneller Zugriff.
- **Linked List:** Dynamische Größe, schnelle Lokale Operationen.
- **Stack/Queue:** Abstrakte Datenstrukturen mit definiertem Zugriffsprinzip.
- **Heap:** Baumstruktur im Array; Eltern $\geq$ Kinder (Max-Heap).
- **Priority Queue:** Heap-basiert; immer schnellstes Extrahieren der höchsten Priorität.
- **Binary Search Tree:** Suchbaum; Linkes Kind $\leq$ Eltern, rechtes Kind $>$ Eltern.

Exercise Recommendations

Aufgabenblatt 5

- 12.1 Bonus
- 12.2 Gute Aufgabe, machen wenn Zeit
- 12.3 Bonus
- 12.4 Bonus

Peergrading

- Aufgabe 11.1
- Grading Scheme in Lösungen beachten