Algorithmen und Datenstrukturen SoSe25

-Assignment 11-

Moritz Ruge
Matrikelnummer: 5600961
Lennard Wittenberg
Matrikelnummer: —

Juli 2025

1 Problem: Tiefensuche

a. Geben Sie Pseudocode für einen Algorithmus, der das folgende Problem löst: Gegeben ein zusammenhängender ungerichteter Graphen G, finde eine Weg, der jede Kante in G genau einmal in jeder Richtung durchläuft.
Hinweis: Verwenden Sie Tiefensuche.

b. Sei G = (V, E) ein Graph. Gegeben ein Algorithmus, der in $O(|V|)$ Zeit überprüft, ob G genau einen Kreis enthält.
Hinweis: Der Graph G muss nicht zusammenhängend sein. Wie müssen die Komponenten von G aussehen, wenn G genau einen Kreis enthält? Was haben Sie in Diskrete Strukturen über Bäume gelernt?

1.1 Teilaufgabe a.

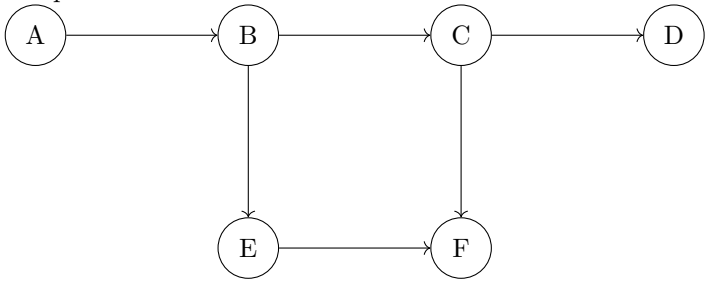
1.1.1 Pseudocode:

(In einem ungerichteten zusammenhängenden Graph findet der normaler DFS alle Knoten und durchläuft jede Kante zwei-mal (hin und zurück))

1. initialisiere ein leeres Set visited (speichert ob ein Knoten bereichts besucht wurde) und ein Dictionary edge_visits welches speichert wie oft eine Kante benutzt wurde.
   (a) initialisiere die Anzahl von Traversierungen / Durchläufe aller Kanten zunächst als 0
2. führe DFS mit dem Startknoten durch.
   (a) Ist der Aktuelle Knoten bereits besucht worden inkrementiere den Eintrag edge_visit[aktuller_Knoten][Vorgän um 1. (Backtracking gehe den Pfad den genommen wurde wieder zurück). Da der erste Knoten keinen Vorgänger hat kann von diesem auch nicht weiter gebacktracked werden.
   (b) wenn der aktuelle Knoten noch nicht besucht wurde füge ihn in das Set visited() ein.
   (c) Inkrementiere den Dictionary Eintrag edge_visits[aktueller_Knoten][Nachbar] um 1.
   (d) für jeden Nachbarn vom aktuellen Knoten führe den DFS aus.
3. in einem ungerichteten Knoten gilt: $v, u == u, v$, für v, u in V und v, u, u, v in E
4. Jetzt sollten alle Kanten 2-mal traversiert worden sein, mittels der Tiefensuche.

Jede Kante wurde 2-mal verwendet einmal hin und einmal zurück. das rückwärts traversieren passiert im normalen DFS normalerweise indirekt. Durch: besuche Nachbar→Nachbar wurde bereits besucht→starte keine weiter tiefensuche→gehe den Weg wieder zurück.

Beispiel:



In einem ungerichteten Graphen gilt: $\{v, u\} = \{u, v\}$ für $v, u \in V$ und $\{v, u\}, \{u, v\} \in E$.

DFS Start bei Knoten 1:

- visited: $\{A\}$ edges_num_visits: $\{\text{A–B} : 0, \text{B–C} : 0, \text{C–D} : 0, \text{B–E} : 0, \text{E–F} : 0, \text{C–F} : 0\}$
- A–B besucht: visited: $\{A, B\}$ edges_num_visits: $\{\text{A–B} : 1, \text{B–C} : 0, \text{C–D} : 0, \text{B–E} : 0, \text{E–F} : 0, \text{C–F} : 0\}$
- A–B–C besucht: visited: $\{A, B, C\}$ edges_num_visits: $\{\text{A–B} : 1, \text{B–C} : 1, \text{C–D} : 0, \text{B–E} : 0, \text{E–F} : 0, \text{C–F} : 0\}$
- A–B–C–D besucht: visited: $\{A, B, C, D\}$ edges_num_visits: $\{\text{A–B} : 1, \text{B–C} : 1, \text{C–D} : 1, \text{B–E} : 0, \text{E–F} : 0, \text{C–F} : 0\}$
- zurück zu C: visited: $\{A, B, C, D\}$ edges_num_visits: $\{\text{A–B} : 1, \text{B–C} : 1, \text{C–D} : 2, \text{B–E} : 0, \text{E–F} : 0, \text{C–F} : 0\}$
- zu F: visited: $\{A, B, C, D, F\}$ edges_num_visits: $\{\text{A–B} : 1, \text{B–C} : 1, \text{C–D} : 2, \text{B–E} : 0, \text{E–F} : 0, \text{C–F} : 1\}$
- zu E: visited: $\{A, B, C, D, F, E\}$ edges_num_visits: $\{\text{A–B} : 1, \text{B–C} : 1, \text{C–D} : 2, \text{B–E} : 0, \text{E–F} : 1, \text{C–F} : 1\}$
- zurück zu B: visited: $\{A, B, C, D, F, E\}$ edges_num_visits: $\{\text{A–B} : 1, \text{B–C} : 1, \text{C–D} : 2, \text{B–E} : 1, \text{E–F} : 1, \text{C–F} : 1\}$
- zurück zu E: visited: $\{A, B, C, D, F, E\}$ edges_num_visits: $\{\text{A–B} : 1, \text{B–C} : 1, \text{C–D} : 2, \text{B–E} : 2, \text{E–F} : 1, \text{C–F} : 1\}$
- zurück zu F: visited: $\{A, B, C, D, F, E\}$ edges_num_visits: $\{\text{A–B} : 1, \text{B–C} : 1, \text{C–D} : 2, \text{B–E} : 2, \text{E–F} : 2, \text{C–F} : 1\}$
- zurück zu C: visited: $\{A, B, C, D, F, E\}$ edges_num_visits: $\{\text{A–B} : 1, \text{B–C} : 1, \text{C–D} : 2, \text{B–E} : 2, \text{E–F} : 2, \text{C–F} : 2\}$
- zurück zu B: visited: $\{A, B, C, D, F, E\}$ edges_num_visits: $\{\text{A–B} : 1, \text{B–C} : 2, \text{C–D} : 2, \text{B–E} : 2, \text{E–F} : 2, \text{C–F} : 2\}$
- zurück zu A: visited: $\{A, B, C, D, F, E\}$ edges_num_visits: $\{\text{A–B} : 2, \text{B–C} : 2, \text{C–D} : 2, \text{B–E} : 2, \text{E–F} : 2, \text{C–F} : 2\}$

1.2 Teilaufgabe b.

1.2.1 Gesucht:

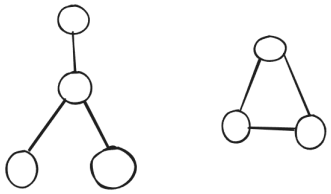
Wie sehen die Komponenten des Graphen aus, damit genau ein Kreis im gesamten Graphen enthalten ist.

1.2.2 Lösung/Ansatz:

Kreis: Was genau ist ein Kreis in einem Graph?
Ein Kreis in einem Graphen ist ein geschlossenen Pfad zwischen Start und Ziel, dabei gibt es keine Wiederholung von Knoten außer Start und Ziel.
Da wir nach **"genau ein Kreis"** suchen sollen, bedeutet das:

- es darf **keinen** weiteren Kreis im Graphen geben
- und es **muss aber genau ein** Kreis vorhanden sein

Der Graph G muss nicht zusammenhänged sein: Das bedeutet der Graph kann aus mehreren Teilkomponenten bestehen, also z.B.: aus zwei einzelne Graphen, die nebeneinander stehen, ohne Verbindung zueinander. D.h. der Kreis kann/darf (für diese Aufgabe) nur innerhalb einer der Zusammenhängenden Komponente entstehen, da ein Kreis ja ein geschlossener Pfad ist, müssen alle beteiligten Knoten und Kanten Verbunden sein.
Da es nicht mehr als ein Kreis geben darf, wären das andere Teilkomponent also Kreisfrei und damit also Baume.



Die Struktur von G bei genau einem Kreis:

- Der Graph G besteht aus mehreren Teilkomponenten
- Nur einer dieser Teilkomponenten ist ein Baum mit zusätzlichen Kanten, die genau einen Kreis bilden
- Alle anderen Komponenten sind Bäume und Kreisfrei

Bäume aus Diskrete Strukturen:

1. Ein Baum ist ein zusammenhängender, kreisfreier Graph
2. In einem Baum gilt immer: $|E| = |V| - 1$
3. Wenn ein Kreis existiert, dann ist der Graph kein Baum mehr

2 Problem: Topologisches Sortieren

Sei $G = (V, E)$ ein gewichteter gerichteter azyklischer Graph. Seien $s, t \in V$. Geben Sie einen Algorithmus, der einen kürzesten Weg von s nach t in Zeit $O(|V| + |E|)$ berechnet. Beweisen Sie die Korrektheit und die Laufzeit Ihres Algorithmus. Hinweis: Beachten Sie Aufgabe 2(b) und verfahren Sie ähnlich zum Algorithmus von Dijkstra.

a. Führen Sie den Algorithmus auf dem untigen Graphen aus. Gehen Sie dabei davon aus, dass die Knoten in vertices und outgoingEdges gemäß der Knotennummern angeordnet sind. Zeigen Sie die einzelnen Schritte. Wie erhält man im Anschluss die topologische Sortierung?

b. Beweisen Sie, dass der Algorithmus aus (a) funktioniert.
Hinweis: Führen Sie einen Widerspruchsbeweis. Nehmen Sie an, es gäbe eine Kante e von einem späteren zu einem früheren Knoten in topoSort, und überlegen Sie sich, was passiert, wenn dfs die Kante e untersucht. Beachten Sie dabei, dass es einen Unterschied macht, ob die Tiefensuche für den Endpunkt von e noch aktiv ist.

2a)

Schritte der Topologischen Sortierung (DFS-basiert):

1. visited: {}, topoorder: {}, Start bei 1
   - visited: {1}, topoorder: {}
   - 1 hat Kante zu 3 → `dfs(3)`
2. visited: {1, 3}
   - 3 hat Kanten zu 2 und 5 → 2 kommt zuerst → `dfs(2)`
3. visited: {1, 3, 2}
   - 2 hat Kante zu 5 → `dfs(5)`
4. visited: {1, 3, 2, 5}
   - 5 hat keine Nachfolger → push(5), zurück zu 2 → push(2), zurück zu 3 → push(3), zurück zu 1 → push(1)
5. visited: {1, 3, 2, 5}, topoorder bisher: {1, 3, 2, 5}
6. 2, 3 bereits besucht → nichts weiter tun
7. `dfs(4)`:
   - visited: {1, 3, 2, 5, 4}, 4 hat Kante zu 5 (bereits besucht) → push(4)
8. `dfs(6)`:
   - visited: {1, 3, 2, 5, 4, 6}, 6 hat Kante zu 5 (bereits besucht) → push(6)

Topologische Sortierung (Stack-LIFO):

$$\text{topoorder} = [6, 4, 1, 3, 2, 5]$$

Alle Knoten sind besucht. Die Reihenfolge ergibt sich durch Rückgabe vom Stack (LIFO).

2b)

Beweis durch Widerspruch:

Angenommen, es existiert ein Knoten w mit einer ausgehenden Kante zu einem Knoten v, wobei v in der TopoSort **vor** w liegt. Dies würde bedeuten, dass die Kante (w, v) **gegen** die topologische Ordnung verläuft.

Verhalten bei DFS:

Während der Tiefensuche wird ein Knoten w nur dann rekursiv besucht, wenn er noch nicht als "found" markiert wurde, also:

```
if !w.found then
dfs(w)
```

Ebenso für die Nachfolger v:

```
if !v.found then
dfs(v)
```

Wurde ein Knoten bereits gefunden, wird er **nicht erneut besucht**, weder direkt noch indirekt. → Eine Kante (w, v), bei der v bereits vollständig bearbeitet und aus dem DFS-Call zurückgekehrt ist (also bereits im Stack liegt), darf **nicht von einem späteren Knoten** w **ausgehen**, da dies sonst auf einen Zyklus hindeuten würde.

Zwei Fälle:

1. w **und** v **liegen im gleichen Pfad:** Dann wird w im DFS vor v besucht. In diesem Fall ist v automatisch nach w in der topoSort, wie es sein soll.
2. **Es gibt einen Zyklus:** Falls v bereits abgeschlossen ist und w eine ausgehende Kante auf v hat, entsteht ein Rückwärtsbezug im Pfad → Zyklus! Dies widerspricht der Voraussetzung, dass G ein DAG ist.

Fazit: Eine Kante (w, v), bei der v vor w in der TopoSort steht, ist in einem DAG **nicht möglich**. → Die topologische Sortierung funktioniert korrekt, da DFS nur Knoten verarbeitet, deren Nachfolger noch nicht vollständig besucht wurden.

3 Problem: Kürzeste Weg in DAGs

Sei Sei $G = (V, E)$ ein gewichteter gerichteter azyklischer Graph. Seien $s, t \in V$. Geben Sie einen Algorithmus, der einen kürzesten Weg von s nach t in Zeit $O(|V| + |E|)$ berechnet. Beweisen Sie die Korrektheit und die Laufzeit Ihres Algorithmus.
Hinweis: Beachten Sie Aufgabe 2(b) und verfahren Sie ähnlich zum Algorithmus von Dijkstra.

3.1 Gegeben/Gesucht:

Gegeben: Ein gewichteter, gerichteter, azyklischer Graph (DAG) $G = (V, E)$, und zwei Knoten $s, t \in V$

Gesucht: Ein Algorithmus, der den kürzesten Pfad von s nach t in $O(|V| + |E|)$ Zeit berechnet.

Zusätzlich: Beweis für Korrektheit und Laufzeit.

3.2 Lösung

3.2.1 Algorithmus:

Wir benutzen Topologisches Sortieren, um den kürzesten Pfad einfach durch einmaliges Durchgehen und Relaxieren zu berechnen. Das geht, weil DAGs keine Zyklen haben.[1]

```
#1. Fuehre eine topologische Sortierung von G durch -> Liste topoOrder

#2. Initialisiere:
  dist[v] = infinty fuer alle v in V
  dist[s] = 0
  prev[v] = None fuer alle v in V

#3. Fuer jeden Knoten u in topoOrder:
  Fuer jede ausgehende Kante (u, v) in E mit Gewicht w:
    Wenn dist[u] + w < dist[v]:
      dist[v] = dist[u] + w
      prev[v] = u

4. Rueckgabe:
  - dist[t] ist die Laenge des kuerzesten Pfads
  - Der Pfad selbst kann ueber prev[] rekonstruiert werden
```

3.2.2 Beweis:

Korrektheit: Die topologische Sortierung stellt sicher, dass jeder Knoten v erst bearbeitet wird, nachdem alle Vorgänger u betrachtet wurden. Da jede Kante nur einmal betrachtet wird und alle kürzesten Distanzen korrekt propagiert werden, liefert der Algorithmus den korrekten kürzesten Pfad.

Laufzeit:

- Topologische Sortierung: $O(|V| + |E|)$
- Initialisierung: $O(|V|)$
- Relaxierung jeder Kante: $O(|E|)$

→ **Gesamt:** $O(|V| + |E|)$

References

[1] Sarah Chen. *Topologische Sortierung: Python, C++ Algorithmusbeispiel*. URL: https://www.guru99.com/de/topological-sort-algorithm.html. (accessed: 10.07.2025).