Algorithmen und Datenstrukturen 1

ALGO1 · SoSe-2023 · tcs.uni-frankfurt.de/algo1/ · 2023-07-06 · *ed6968a*

Darstellung von Graphen, Breitensuche, Tiefensuche (Woche 5)

Eigenständige Vorbereitung:
Lies 📖 CLRS Einleitung Teil VI, Kapitel 22.1–22.4, sowie Appendix B.4–B.5 und schau dir das 📺 Video der Woche an.

Zeichenlegende:

- ✏️ Schriftliche Aufgabe, die du fristgerecht in Moodle abgibst. In der Klausur wirst du alle Aufgaben schriftlich bearbeiten, daher ist das Feedback der Tutoren wichtig, damit du deine Schreibfähigkeiten verbessern kannst.
- 👍 Diese Art von Aufgabe musst du sicher können, um die Klausur zu bestehen.
- 🔑 Diese Art von Aufgabe musst du weitgehend können, um die Klausur zu bestehen.
- 🔭 Diese Art von Aufgabe musst du können, um eine gute Note zu erhalten.
- 🌈 Diese Aufgabe ist als Knobelspaß gedacht, der das algorithmische Verständnis vertieft.

Aufgabe 5.1 (Darstellung, Eigenschaften und Algorithmen 👍). Betrachte die Graphen in Abbildung 1. Löse die folgenden Teilaufgaben.

a) Gib die Adjazenzlisten und Adjazenzmatrizen für die Graphen 1 und 2 an.
b) Tiefensuche wird auf Graph 1, beginnend von Knoten 0, ausgeführt. Die Adjazenzlisten sind hierbei aufsteigend sortiert. Gib den Tiefensuchbaum, sowie die Entdeckungszeit und Endzeit an.
c) Breitensuche wird auf Graph 1, beginnend von Knoten 0, ausgeführt. Die Adjazenzlisten sind hierbei aufsteigend sortiert. Gib den Breitensuchbaum und die Distanz zum Startknoten für alle Knoten an.
d) Gib die Zusammenhangskomponenten der 3 Graphen an.
e) Welche der 3 Graphen sind bipartit?

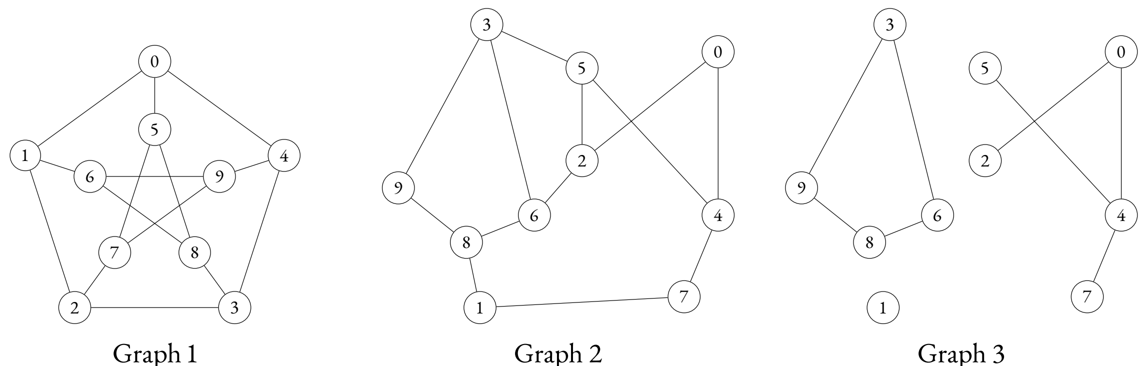


Abbildung 1: Graphen für Aufgabe 1. Graph 1 wird auch als *Petersen-Graph* bezeichnet.

Aufgabe 5.2 (Buchstabenlabyrinth 🔑). Algolina und ihre kleine Schwester spielen *Buchstabenlabyrinth*. In diesem Spiel ist eine $N \times N$ Matrix gegeben, wo jeder Eintrag entweder A oder B ist. Zum Beispiel:

$$
\begin{array}{ccccc}
\mathbf{A} & \mathrm{A} & \mathbf{A} & \mathbf{B} & \mathbf{A} \\
\mathbf{B} & \mathrm{B} & \mathbf{B} & \mathrm{B} & \mathbf{B} \\
\mathbf{A} & \mathbf{B} & \mathbf{A} & \mathrm{A} & \mathbf{A} \\
\mathrm{A} & \mathrm{B} & \mathrm{B} & \mathrm{B} & \mathbf{B} \\
\mathrm{A} & \mathrm{A} & \mathrm{A} & \mathrm{A} & \mathbf{A}
\end{array}
$$

Die algorithmische Aufgabe ist es nun, einen kürzesten Pfad von oben links nach unten rechts zu finden. Die Knoten auf dem Pfad müssen dabei allerdings zwischen A und B alternieren, sprich die Knoten eines Pfades buchstabieren ABABABAB. . .. Der Pfad darf in jedem Schritt immer nur horizontal oder vertikal gehen, diagonale Bewegungen sind also nicht erlaubt. Im Beispiel sind die Buchstaben des kürzesten Pfads fett geschrieben.

Da die Schwestern sich nicht sicher sind, ob sie auch tatsächlich den kürzesten Weg gefunden haben, wollen sie ein Programm schreiben, das es für sie beantwortet. Entwirf einen Algorithmus, der für eine gegebene AB-Matrix die Länge eines kürzesten Weges findet. Implementiere den Algorithmus in einer Programmiersprache deiner Wahl.

Aufgabe 5.3 (Tiefensuche mittels eines Stapels 👍). Erkläre, wie Tiefensuche ohne Rekursion mit einem Stapel implementiert werden kann.

Aufgabe 5.4 (Wer nix weiß, sucht einen Kreis 👍). Entwirf einen Algorithmus, der feststellt, ob ein gegebener Graph einen Kreis enthält. Wie schnell ist dein Algorithmus?

Aufgabe 5.5 (Anzahl kürzester Wege 👍). Entwirf einen Algorithmus, der für einen Graphen G und zwei Knoten s, t die Anzahl der kürzesten Pfade zwischen s und t ausgibt.

Aufgabe 5.6 (Labyrinthe und Gittergraphen ✏️). Ein $k \times k$ Gittergraph ist ein Graph, in dem die Knoten, wie in einem Netz, in k Zeilen mit jeweils k Knoten angeordnet sind. Kanten dürfen sich hierbei nur zwischen Knoten befinden, die in horizontaler und vertikaler Richtung adjazent sind. Siehe Abbildung 2 (a). Löse die folgenden Teilaufgaben.

a) 👍 Seien n und m die Anzahl der Knoten und Kanten in einem $k \times k$ Gittergraph. Drücke obere Schranken für n und m in asymptotischer Notation als Funktion von k aus.

Ein $k \times k$ Labyrinth ist eine quadratische Struktur, die aus k Zeilen mit jeweils k Zellen besteht. Jede Zelle wird durch vier Seiten begrenzt, und jede Seite ist entweder frei oder eine Wand. Ein Pfad im Labyrinth ist eine Sequenz F von Zellen $f_1, \ldots, f_\ell$, sodass aufeinanderfolgende Zellen f_i, f_{i+1} mit $1 \leq i < \ell$ horizontal oder vertikal adjazent sind und sich keine Wand zwischen ihnen befindet. Eine spezielle Zelle ist als Start gekennzeichnet und eine weitere als Ziel.
Der Verein „Daten- und Gartenbau" bewertet ein Labyrinth als *schön*, falls die folgenden Voraussetzungen eingehalten werden:

- Es gibt genau einen Weg vom Start zum Ziel.
- Es gibt einen Weg vom Start zu jeder anderen Zelle des Labyrinths.
- Es gibt keinen Weg, der im Kreis führt.

Ein Labyrinth wird als *unschön* bewertet, wenn mindestens einer dieser Punkte verletzt wird. Siehe Abbildung 2 (b)–(d).

b) 👍 Beschreibe wie man ein $k \times k$ Labyrinth als $k \times k$ Gittergraph modelliert.

c) 👍 Zeichne Abbildung 2 (b) als Gittergraph.

d) 🔑 Mit dem Aufschwung der Gärtnerei im letzten Jahr wurden nun so viele Labyrinthe eingereicht, dass der Verein es nun nicht mehr stemmen kann, jedes Labyrinth von Hand zu bewerten. Entwirf einen Algorithmus, der als Eingabe einen $k \times k$ Gittergraph erhält, der ein $k \times k$ Labyrinth modelliert, und prüft, ob das Labyrinth schön ist. Zeige die Korrektheit des Algorithmus[1] und gib eine Laufzeitanalyse an, in der die asymptotische Laufzeit als Funktion von k beschreiben wird.

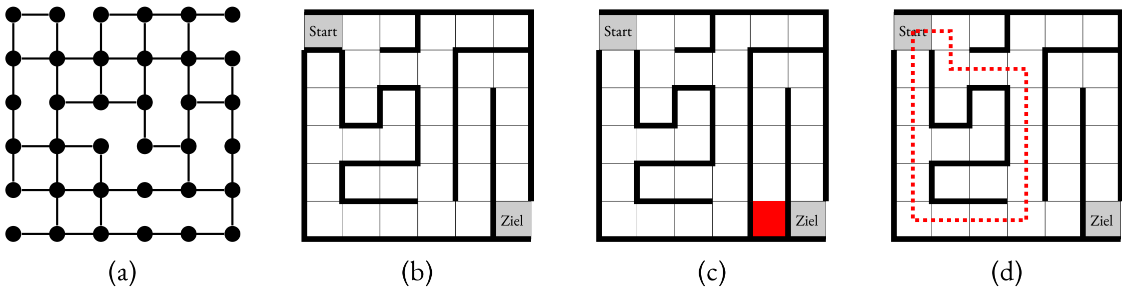


Abbildung 2: (a) ein 6×6 Gittergraph. (b), (c) und (d) sind 6×6 Labyrinthe. (b) ist schön, (c) und (d) sind unschön. In (c) ist das Ende nicht erreichbar und (d) enthält einen Weg der im Kreis führt.

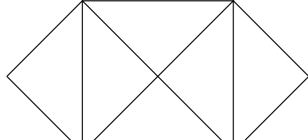
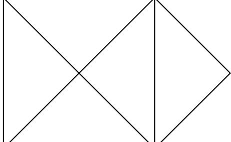
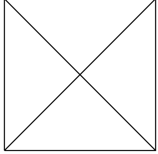
[1] Beweis, dass die Ausgabe deines Algorithmus auf allen möglichen Eingaben richtig ist.

Aufgabe 5.7 (Eulerkreis und Eulerpfad). Sei G ein zusammenhängender Graph mit n Knoten und m Kanten. Ein Eulerpfad in G ist ein Pfad, der alle Kanten genau ein mal enthält. Ein Eulerkreis ist ein Eulerpfad, der in demselben Knoten beginnt und endet. Löse die folgenden Teilaufgaben.

a) 🔭 Beweise, dass G einen Eulerkreis genau dann enthält, wenn alle Knoten einen geraden Knotengrad haben.

b) 🔭 Beweise, dass G einen Eulerpfad genau dann enthält, wenn 0 oder 2 Knoten einen ungeraden Knotengrad haben.

c) 👍 Welche dieser Zeichnungen können gezeichnet werden ohne den Stift abzusetzen? Kannst du den Stift am selben Punkt auf- und absetzen?

d) 👍 Entwirf einen Algorithmus, der in Zeit $O(n+m)$ ermittelt, ob G einen Eulerkreis enthält.

e) 🔭 Entwirf einen Algorithmus, der in Zeit $O(n+m)$ einen Eulerkreis ausgibt, falls G einen enthält.

Aufgabe 5.8 (Durchmesser von Bäumen). Sei T ein binärer Baum mit n Knoten. Der *Durchmesser von* T ist die Länge des längsten kürzesten Weges zwischen Knotenpaaren aus T.[2]

a) 🔑 Entwirf einen Algorithmus, der den Durchmesser von T in Zeit $O(n^2)$ ermittelt.

b) 🔭 (sehr schwer) Entwirf einen Algorithmus, der den Durchmesser von T in Zeit $O(n)$ ermittelt.

Aufgabe 5.9 (3-Farben Algorithmus). *For an English version of this exercise, see [Erickson, page 210].*
Eine Klasse von Suchalgorithmen auf Graphen, die Breitensuche und Tiefensuche verallgemeinern, wurde 1975 von Edsger Dijkstra, Leslie Lamport, Alain Martin, Carel Scholten und Elisabeth Steffens beschrieben. Die Autor:innen haben diese Algorithmen untersucht, um damit einen automatischen *garbage collector* zu entwerfen (siehe wikipedia). Anstatt markierte und unmarkierte Knoten zu verwalten, verwaltet ihr Algorithmus eine Farbe für jeden Knoten, entweder weiß, grau oder schwarz. Im Folgenden stellen wir uns einen als Adjazenzliste gegebenen, ungerichteten Graphen G vor.

```
procedure ThreeColorSearch(s)
    färbe alle Knoten weiß
    färbe s grau
    while mindestens ein Knoten ist grau do
        ThreeColorStep
```

```
procedure ThreeColorStep
    v ← irgendein grauer Knoten
    if v hat keine weißen Nachbarn then
        färbe v schwarz
    else
        w ← irgendein weißer Nachbar von v
        w.π ← v            ▷ v ist der Elternknoten von w.
        färbe w grau
```

[2] Sei $dist_G(u,v)$ die Länge des kürzesten Weges von Knoten u nach Knoten v im Graph G, dann ist der Durchmesser $\max_{u,v\in G}(dist_G(u,v))$.

a) 👍 Beweise, dass ThreeColorSearch zu jedem Zeitpunkt die folgende Invariante erhält: Kein schwarzer Knoten ist zu einem weißen Knoten benachbart. *(Hinweis: Das sollte einfach sein.)*

b) 🔑 Beweise Folgendes: Wenn ThreeColorSearch(s) terminiert, dann sind alle von s erreichbaren Knoten schwarz, alle nicht von s erreichbaren Knoten weiß, und die zu den Eltern zeigenden Kanten $(v, v.\pi)$ definieren einen Baum, der alle Knoten der Zusammenhangskomponente von s aufspannt. *Hinweis: Wenn man den Algorithmus mit DFS/BFS vergleicht, kann man sich intuitiv vorstellen, dass die schwarzen Knoten „markiert" sind und die grauen Knoten „auf dem Stapel/in der Warteschlange". Ein Unterschied ist, dass ThreeColorStep nicht im selben Aufruf* alle *Kanten abarbeiten muss, die aus einem Knoten v rausgehen.*

c) 🔑 Die folgende Variante von ThreeColorSearch verwaltet die grauen Knoten auf einem Stapel. Beweise, dass diese Variante äquivalent zu DFS ist, das heißt, die Knoten werden in genau derselben Reihenfolge entdeckt und die Elternbeziehungen sind identisch. *Hinweis: Die Reihenfolge der letzten zwei Zeilen von ThreeColorStackStep ist wichtig!*

```
procedure ThreeColorStackSearch(s)
    färbe alle Knoten weiß
    färbe s grau
    lege s auf den Stapel
    while mindestens ein Knoten ist grau do
        ThreeColorStackStep
```

```
procedure ThreeColorStackStep
    nimm v vom Stapel
    if v hat keine weißen Nachbarn then
        färbe v schwarz
    else
        w ←irgendein weißer Nachbar von v
        w.π ← v
        färbe w grau
        lege v auf den Stapel
        lege w auf den Stapel
```

d) 🔑 Die folgende Variante von ThreeColorSearch verwaltet die grauen Knoten in einer Warteschlange. Beweise, dass diese Variante ***nicht*** äquivalent zu BFS ist. *Hinweis: Die Reihenfolge der letzten zwei Zeilen von ThreeColorQueueStep ist nicht wichtig!*

```
procedure ThreeColorQueueSearch(s)
    färbe alle Knoten weiß
    färbe s grau
    schiebe s in die Warteschlange
    while mindestens ein Knoten ist grau do
        ThreeColorQueueStep
```

```
procedure ThreeColorQueueStep
    ziehe v aus der Warteschlange
    if v hat keine weißen Nachbarn then
        färbe v schwarz
    else
        w ←irgendein weißer Nachbar von v
        w.π ← v
        färbe w grau
        schiebe v in die Warteschlange
        schiebe w in die Warteschlange
```

e) 🔭 (sehr schwer) Hier sei G ein gerichteter Graph. Wir nehmen nun an, dass ein zweiter Prozess Kanten zu G hinzufügt, während ThreeColorSearch noch läuft. Diese neuen Kanten könnten die Farbinvariante zerstören, die in Teil a) beschrieben ist. Daher könnte es jetzt sein, dass ThreeColorSearch nicht mehr korrekt ist. Das heißt, es könnte sein, dass der Algorithmus zwar terminiert, aber es trotzdem noch Knoten gibt, die von s erreichbar sind und weiß sind. Wenn wir einen *garbage collector* implementieren wollen, wäre das fatal, denn hier würden wir „weiß" mit „unerreichbar und daher löschbar" gleichsetzen wollen.

Wenn der andere Prozess auf die Farbinvariante Rücksicht nimmt und diese explizit wiederherstellt, wann immer sie verletzt würde, dann können wir den ThreeColorSearch Algorithmus trotzdem sicher verwenden. Das möchten wir jetzt zeigen. Um die zwei parallelen Algorithmen zu modellieren, verwenden wir die *either/or* Syntax in GarbageCollect; wie bei *if/then/else* verzweigt das Programm hier, aber welcher Zweig verfolgt wird, wird nicht vom Programm selbst entschieden, sondern vom Betriebssystem.[3]

```
procedure GarbageCollect(s)
    färbe alle Knoten weiß
    färbe s grau
    while mindestens ein Knoten ist grau do
        either
            CollectStep
        or
            Mutate
```

```
procedure CollectStep
    v ← irgendein grauer Knoten
    if v hat keine weißen Nachbarn then
        färbe v schwarz
    else
        w ←irgendein weißer Nachbar von v
        färbe w grau
```

```
procedure Mutate
    u ← irgendein Knoten
    w ← irgendein Knoten
    if (u, w) ist keine Kante then
        füge (u, w) als Kante hinzu
        if u ist schwarz und w ist weiß then
            färbe u grau
        if u ist weiß und w ist schwarz then
            färbe w grau
```

Beweise, dass GarbageCollect irgendwann terminiert, und dass dann jeder von s erreichbare Knoten schwarz gefärbt ist und jeder von s nicht erreichbare Knoten weiß gefärbt ist.

f) (sehr schwer) Hier sei G wieder ein gerichteter Graph. Anstatt schwarze Knoten grau zu färben, soll Mutate jetzt die Farbinvariante aufrechterhalten, indem manche *weißen* Knoten grau gefärbt werden:

```
procedure Mutate
    u ← irgendein Knoten
    w ← irgendein Knoten
    if (u, w) ist keine Kante then
        füge (u, w) als Kante hinzu
        if u ist schwarz und w ist weiß then
            färbe w grau
        if u ist weiß und w ist schwarz then
            färbe u grau
```

Beweise, dass GarbageCollect irgendwann terminiert, und dass dann s schwarz gefärbt ist, jeder von einem schwarzen Knoten erreichbare Knoten wiederum schwarz ist, und jeder nicht von einem schwarzen Knoten erreichbare Knoten weiß ist.

[3] Das ist eine dramatische Vereinfachung, sowohl von paralleler Programmierung als auch von *garbage collection*. Mehrfädige Programmiersprachen wie Lua und Go benutzen einen viel komplexeren *mark and sweep* Algorithmus als *garbage collector*. Mathematisch wichtig für *either/or* ist, dass das Betriebssystem zwar nicht garantiert, wie oft hintereinander und in welcher Reihenfolge die zwei Programmzweige gewählt werden. Aber jeder Zweig wird immer wieder *irgendwann* gewählt, das heißt, nach endlicher Zeit. — Das Betriebsystem darf also ***nicht*** für immer den einen Zweig wählen und den anderen „vergessen“.