Praktikumsaufgaben
Graphentheoretische Konzepte und Algorithmen

WiSe 23

6. Dezember 2023

J. Padberg

Aufgaben für alle Gruppen

Allgemeines zu allen Aufgaben

Die Aufgabenstellung ist aus folgenden Gründen nicht ganz genau spezifiziert:

- Sie sollen einen Entwurfsspielraum haben. Das heißt, Sie können relativ frei entscheiden, wie Sie die Aufgabe lösen, allerdings sollten Sie Ihre Entscheidungen bewusst fällen und begründen können. Insofern gibt es auch keine Musterlösung.
- Durch die unterschiedlichen Lösungen können Sie von Ihren Kommilitonen noch lernen und das *Structured Walk-Through* bleibt spannend.

Darüber hinaus dürfen Sie gerne unterschiedliche Quellen nutzen, aber nur wenn Sie diese auch angeben. Sie sollen auch nicht unbedingt alles selber programmieren, nutzen Sie gerne auch andere Libraries.
Für die Bearbeitung der Praktikumsaufgaben erhalten Sie im MS-Team **WiSe2023_BAI3-GKA**

- Beispielgraphen für das Praktikum
- Latex-Template

Erfolgreiche, *eigene* Bearbeitung der Aufgaben

Das Ergebnis der Bearbeitung einer Aufgabe wird von jeder Gruppe im Praktikum vorgestellt. Das heißt, die Aufgabe muss *rechtzeitig vor* dem Praktikumstermin fertig bearbeitet sein.
Über die Vorstellung hinaus wird für jede Aufgabe das Folgende erwartet:

1. die *selbstständige* Implementierung der gestellten Aufgabe, also
   - eine korrekte und möglichst effiziente Implementierung in Java, die der vorgegebenen Beschreibung entspricht,
   - die Kommentierung der zentralen Eigenschaften/Ereignisse etc. im Code

- die detaillierte Kennzeichnung des Codes, der nicht von Ihnen selber kommt (auf keinen Fall mehr als 15%) und
- hinreichende Testfälle in JUnit und ihre Kommentierung.

Hinweis: Wenn Sie in der Implementierung englische Fachbegriffe nutzen wollen, können Sie die im Index von Diestel nachschlagen.

2. die Lösungsdokumentation (schriftliche Erläuterung Ihrer Lösung) *etwa 4-7 Seiten*
   - **Template:** Benutzen Sie das Latex-Template
   - **Quellenangaben:** Angabe von wesentlichen Quellen, z.B. Web-Seiten/Bücher, von denen Quellcode/Algorithmen übernommen wurden
   - **Inhalte:**
     - eine kurze Beschreibung der Algorithmen und Datenstrukturen,
     - die wesentlichen Entwurfsentscheidungen Ihrer Implementierung,
     - die umfassende Dokumentation der JUnit-Tests und
     - die Diskussion der Abdeckung der Testfälle

Es gibt drei Praktikumsaufgaben, weitere Details werden in der jeweiligen Aufgabenstellung angegeben.

Eine Praktikumsaufgabe gilt als erledigt, wenn

1. die Abgabe der Lösungsdokumentation rechtzeitig in Ihrem Kanal erfolgt ist
2. Sie im Praktikum Ihre Implementierung vorgestellt haben und diese von mir (mindestens) als ausreichend anerkannt wurde.
3. Bewertung:
   - OK: mindestens ausreichend, nichts mehr zu tun
   - nacharbeiten: beim nächsten Praktikumstermin laufen die JUnit-Tests erfolgreich durch
   - nicht OK: keine PVL, keine Diskussion
   - Krankheit: Attest und Ihre Gruppe wuppt das ohne Sie

Qualität der Implementierung

Die Qualität der Implementierung spielt eine wesentliche Rolle, es genügt nicht, einen Algorithmus irgendwie zu implementieren. Es wird eine *professionelle* Lösung erwartet.
Beim Programmieren sollen als Ergänzung (wenn nicht ohnehin schon Bestandteil der PR-Vorlesung) zum in PR1 und PR2 Gelernten die Tipps von Joshua Bloch in 'Effective Java Third Edition Best practices for' befolgt werden, insbesondere:

Item 9: Prefer 'try-with-resources' to 'try-finally'

Item 11: Always override 'hashCode' when you override 'equals'

Item 26: Don't use raw types

Kapitel 7 komplett; Lambdas und Streams benutzen; übersichtlicher

Item 49: Check Parameters for validity
Wichtig auch für die Doku: Sinnvolle Preconditions überlegen und testen

Item 54: Return empty collections or arrays, not nulls
Wenn kein Ergebnis gefunden wird, ist die Liste eben leer

Item 54: Return optionals judiciously

Item 57: Minimize the scope of local variables

Item 58: Prefer for-each loops to traditional for loops

Item 59: Know and use the libraries
Insbesondere RegExp

Item 60: Avoid 'float' and 'double' if exact answers are required
z.B. Kantengewichte

Item 61: Prefer primitive types to boxed primitives (Wrapperklassen)
Wir bauen Algorithmen; da ist Schnelligkeit gefragt

Unterstützung durch eine LLM

LLM steht für Large Language Model. Es ist ein KI-Algorithmus, der Deep Learning und große Datensätze nutzt, um neue Inhalte zu verstehen und zu generieren [1]. LLMs werden in verschiedenen Bereichen, einschließlich der Programmierung, immer beliebter. Hier sind einige Möglichkeiten, wie LLMs für die Programmierung verwendet werden können:

- Code-Generierung: LLMs können Code für bestimmte, vom Benutzer beschriebene Aufgaben generieren. Dies kann nützlich sein, um sich wiederholende Codierungsaufgaben zu automatisieren oder Codeschnipsel zu erzeugen [2].
- Problemlösung: LLMs können mit einem Programmierproblem konfrontiert werden und aufgefordert werden, eine Lösung zu generieren. Indem der LLM eine Problemstellung erhält, kann er Code generieren, der das Problem löst [3].
- Dokumentation und Lernen: LLMs können auf die Dokumentation von Programmiersprachen trainiert werden und dazu verwendet werden, Erklärungen, Beispiele und Tutorials zu erstellen. Dies kann für das Erlernen von Programmierkonzepten und das Verständnis komplexer Themen hilfreich sein [4].
- Unterstützung und Zusammenarbeit: LLMs können als virtuelle Assistenten oder Kollaborateure für Programmierer eingesetzt werden. Sie können Vorschläge machen, Fragen beantworten und beim Debuggen oder Optimieren von Code helfen [5].

Es ist wichtig, darauf hinzuweisen, dass LLMs zwar leistungsstarke Werkzeuge für die Programmierung sein können, aber auch Grenzen haben. Sie erzeugen möglicherweise nicht immer korrekten oder optimalen Code, und ihre Ausgabe sollte sorgfältig überprüft und getestet werden. Darüber hinaus sollten LLMs als Hilfsmittel eingesetzt werden und nicht ausschließlich für Programmieraufgaben verwendet werden, da sie die Entwicklung eines tiefen Verständnisses und einer Problemlösungskompetenz behindern können.[1]

Für die Praktikumsaufgabe 1 und 3 dürfen Sie die Unterstützung einer Large Language Maschine (LMM) Ihrer Wahl nutzen, wenn Sie Ihre Erfahrung damit kurz (etwa eine Seite zusätzlich) beschreiben.

[1] Dieser Textabschnitt wurde mit perplexity und DeepL erzeugt.

Aufgabe 1:
Visualisierung, Speicherung und Traversierung von Graphen

Die Graphen, mit denen Sie arbeiten, sollen gespeichert und gelesen werden können. Dabei ist das in der VL beschriebene Format zu verwenden. Beispieldateien finden Sie im MS-Team **WiSe2023_BAI3-GKA**.

Die Aufgabe umfasst:

- die Einarbeitung in die Bibliothek GraphStream https://graphstream-project.org/
- das Einlesen/ Speichern von ungerichteten sowie gerichteten Graphen und die Visualisierung der Graphen,
  (Hinweis: reguläre Ausdrücke in Java nutzen)
- die Implementierung des Dijkstra-Algorithmus[2];
  gefordert werden dabei
  - als Ergebnis der kürzeste Weg und dessen Länge und
  - einfache JUnit-Tests, die Ihre Methoden überprüfen und
  - JUnit-Tests, die das Lesen, das Speichern sowie den Dijkstra-Algorithmus überprüfen unter Benutzung
    * der gegeben *.grph*-Dateien (der geeigneten) sowie
    * selbst generierter *.grph*-Dateien.

Wenn Sie sich wegen dieser Aufgabe per email an uns wenden, bitte geben Sie im Betreff die Teamnummer und die Aufgabennummer an.

[2] ggf nur BFS für GKAP04

Aufgabe 2:
Berechnung des Minimalen Spannwaldes (MSF)

In der Vorlesung wurde der Algorithmus von Prim zur Berechnung des Spannbaums vorgestellt. Hier wollen wir uns mit der effizienten Implementierung des Kruskal-Algorithmus' zur Berechnung eines minimalen, spannenden Waldes befassen. Lesen Sie dazu bitte die Kapitel 6.1 bis 6.3 sowie den Anhang B.2 in [KN].

Es soll der Algorithmus so implementiert werden, dass der minimale Spannwald visualisiert und die Kantengewichtssumme des minimalen Spannwaldes berechnet werden kann.
Die Aufgabe umfasst folgende Teile:

1. Ihre eigene Implementierung des Kruskal-Algorithmus.
   Benutzen Sie nicht den Kruskal aus Graphstream!!
   - Implementieren und testen Sie den Algorithmus von Kruskal (Algorithmus 6.2 in [KN])
   - unter Nutzung der Datenstruktur aus Anhang B2 in [KN],

   so dass Sie folgendes liefern:
   - als Ergebnis den minimalen Spannwald und sein Gesamtgewicht,
     sowohl als Datenstruktur als auch visualisiert,
   - JUnit-Tests, die den Algorithmus überprüfen und
   - JUnit-Tests, die den Algorithmus unter Benutzung der zu konstruierenden randomisierten Graphen (s.u) gegen den Prim-Algorithmus in Graphstream prüfen
2. Überlegen Sie sich eine allgemeine Konstruktion eines ungerichteten Graphen für eine vorgegebene Anzahl von Knoten und Kanten mit beliebigen, aber unterschiedlichen, nicht-negativen Kantengewichten.
3. Erzeugen Sie mindestens 3 randomisierte, ungerichtete, gewichtete Graphen mit beliebigen, aber unterschiedlichen Kantenbewertungen. Die Graphen sollen **möglichst groß** sein, aber bei dem Kruskal-Algorithmus jeweils eine Laufzeit von unter **unter fünf Minuten** haben.
   Bitte geben Sie deren Größe (Knoten- und Kantenanzahl) in der Dokumentation an.

Wenn Sie sich wegen dieser Aufgabe per email an uns wenden, bitte geben Sie im Betreff die Teamnummer und die Aufgabennummer an.

Aufgabe 3:
Eulerkreise

Der Algorithmus von Hierholzer ist auch ein Algorithmus, mit dem man in einem ungerichteten Graphen Eulerkreise bestimmt. Er geht auf Ideen von Carl Hierholzer zurück.

```
Algorithmus

Voraussetzung:
Sei G = (V,E) ein zusammenhängender Graph, der nur
Knoten mit geradem Grad aufweist.

  i. Wähle einen beliebigen Knoten v0 des Graphen
     und konstruiere von v0 ausgehend einen
     Unterkreis K in G, der alle Eigenschaften
     eines Eulerkreises besitzt.

 ii. Vernachlässige nun alle Kanten dieses
     Unterkreises.

iii. Am ersten Eckpunkt des ersten Unterkreises,
     dessen Grad größer 0 ist, lässt man nun einen
     weiteren Unterkreis entstehen, der wiederum
     ein Eulerkreis ist.

 iv. Erstelle so viele Unterkreise, bis alle Kanten
     von einem Unterkreis durchlaufen wurden.

  v. Nun erhält man den Eulerkreis, indem man mit
     dem ersten Unterkreis beginnt und bei jedem
     Schnittpunkt mit einem anderen Unterkreis,
     den letzteren einfügt, und danach den ersten
     Unterkreis wieder bis zu einem weiteren
     Schnittpunkt oder dem Endpunkt fortsetzt.
```

Die Aufgabe umfasst folgende Teile:

1. Bitte implementieren und testen Sie die den Hierholzer-Algorithmus zur Eulerkreissuche:
   a) Entwerfen Sie bitte Junit-Tests erst für kleine, gespeicherte Graphen (sowohl Eulergraphen als auch andere). Wie prüfen Sie, ob eine gegebene Kantenfolge ein Eulerkreis ist?
   b) Erzeugen Sie dann randomisierte, ungerichtete Eulergraphen. Sie finden weiter unten den Pseudocode zur Erzeugung von Eulergraphen a) als Multigraphen und b) als schlichte Graphen.
   Bitte entwerfen Sie weitere Tests damit.

2. Laufzeit- und Energieverbrauchsmessung
   Für die Messung erzeugen Sie bitte 3 Graphen
   mit 100 Knoten & 3.000 Kanten,
   mit 1000 Knoten & 400.000 Kanten und
   mit 10.000 Knoten & 1.000.000 Kanten.
   Dann lassen Sie Ihre Algorithmen mehrfach darauf laufen.

   Für jeden der drei Graphen erstellen Sie bitte eine CVS-Datei (entsprechend dem Template), in der zunächst Ihr Team, Graphtyp (Simple oder Multi), die Kantenanzahl, Ihre Hardware und Ihr Messungswerkzeug geschrieben werden. Danach werden 13 Durchläufe, von denen nur die letzten 10 ausgewertet werden, gestartet und dann für jeden Durchlauf (ab dem 4.) die Zeit in Millisekunden und der Energieverbrauch in Joule (oder was auch immer Ihr Messwerkzeug raus gibt) in die CSV-Datei geschrieben. Laden Sie diese CSV-Dateien bitte mit der Doku zusammen hoch, aber als einzelne Dateien mit Ihrem Team-Kürzel zB TeamA_G04.csv.

   Gehen Sie bitte in der Lösungsdokumentation ausführlich auf die Messung ein.

   Wenn Sie sich wegen dieser Aufgabe per email an mich oder Herrn Berding wenden, bitte geben Sie im Betreff die Teamnummer und die Aufgabennummer an.

Eulergraphen generieren

Der folgende Algorithmus erzeugt einen zusammenhängenden Multi-Graphen $G = (V, E)$, der eulersch ist. Das Ergebnis ist die Kantenmenge $E \subseteq V \times V$ mit genau m Kanten.

```
createEul_MultiGraph(n:int, m:int)                           %% n ≥ 1

Gegeben n Knoten
ungerichteten Multigraphen G = (V, E) initialisieren mit:
V = {1, ..., n}                                         %% n Knoten
E = {}                                                  %% Kanten
P = {2, ..., m}                      %% Positionen im Eulerkreis
pos : V → P %% injektive Abb der Knoten auf Position im Eulerkreis

for 1 ≤ i ≤ n{                   %% jeden Knoten einmal in der Liste,
                                             %% also zusammenhängend
      select_randomly k ∈ P
      P := P − k                         %% Mengen, also P := P \ {k}
      pos(i) := k
      }
select_randomly start ∈ V              %% erster und letzter Knoten
cur := start

for 2 ≤ i ≤ m{    %% Position nicht belegt, dann an dieser Position
                                %% einen beliebigen Knoten einfügen
      if i ∉ pos(V) then {
            select_randomly nxt ∈ V
            }
      else {
            nxt := pos^{-1}(i)
            }
      E := E + (cur, nxt)
      cur := nxt
      }
E := E + (cur, start)
```

Der folgende Algorithmus erzeugt einen zusammenhängenden, schlichten Graphen $G = (V, E)$, der eulersch ist. Das Ergebnis ist die Kantenmenge $E \subseteq V \times V$ mit etwa $|E| = m$ Kanten.

```
createEul_SimpleGraph(n:int,m:int)                    %% 3 ≤ n ≤ m ≤ n(n−1)/2

Gegeben n Knoten
schlichten, ungerichteten G = (V,E) initialisieren mit:
U := {1,...,n}                                         %% unverbundene Knoten
V = {}                                                 %% Knoten
Odd = {}                          %% Knoten mit ungeradem Knotengrad
Even = {}                           %% Knoten mit geradem Knotengrad
E = {}                                                 %% Kanten
A = (V × V) − {(i,i)|1 ≤ i ≤ n}                        %% alle möglichen,
%% noch nicht benutzten Kanten

select_randomly cur ∈ U
V := V + cur
U := U − cur

for 1 ≤ i ≤ n − 1{                                     %% Spannbaum erzeugen
      addnode(U,V, Odd, Even, E,A)
      }

while |E| ≤ ⌈0.85m⌉{            %% 85% der restlichen Kanten einfügen
      addedge(Odd, Even, E,A)
      }

makeAllEven( Odd, Even, E, A)                  %% Konten mit ungeradem
                                               %% Knotengrad begradigen
```

```
addnode( U,V, Odd, Even, E,A)

select_randomly cur ∈ V                        %% Knoten schon im Baum
select_randomly nxt ∈ U                   %% neuer unverbundener Knoten
V := V + nxt
U := U − nxt
E := E + (cur,nxt)                                     %% neue Kante
A := A − {(cur,nxt),(nxt,cur)}          %% mögliche Kanten reduzieren,
                                                       %% Richtung egal
Odd := Odd + nxt                               %% Blatt mit Knotengrad 1
toggle(cur,Odd, Even)                          %% alter Knoten wechselt
```

```
addedge(Odd, Even, E,A)


select_randomly (cur,nxt) ∈ A          %% bisher ungenutzte Kante wählen
E := E + (cur,nxt)                                        %% neue Kante
A := A − {(cur,nxt),(nxt,cur)}          %% mögliche Kanten reduzieren,
                                                      %% Richtung egal
toggle(cur,Odd, Even)                          %% beide Knoten wechseln
toggle(nxt,Odd, Even)
```

```
makeAllEven(Odd, Even, E,A)

M := A ∩ Odd × Odd                       %% alle noch möglichen Kanten
                                          %% zwischen ungeraden Knoten

while |M| > 0{                             %% alle noch übrigen Kanten
                                                %% als Kanten einfügen
      select_randomly (cur,nxt) ∈ M
      E := E + (cur,nxt)
      A := A − {(cur,nxt),(nxt,cur)}
      toggle(cur,Odd, Even)       %% beide Knoten wechseln
      toggle(nxt,Odd, Even)
      M := A ∩ Odd × Odd                  %% neu berechnen der
                                  %% noch möglichen Kanten

}

while |Odd| > 0{       %% sind dann immer noch ungerade Knoten übrig
      select_randomly (cur) ∈ Odd
      toggle(cur,Odd, Even)
      select_randomly (nxt) ∈ Odd
      toggle(nxt,Odd, Even)
      E := E − (cur,nxt)   %% wird die Kante dazwischen gelöscht
      A := A + {(cur,nxt),(nxt,cur)}

}
```

```
toggle(n:int,O:Set,E:Set)

if n ∈ O then
      {O := O − n
      E := E + n}
else
      {O := O + n
      E := E − n}
```