TU Kaiserslautern – FB Informatik
Dr. Patrick Michel

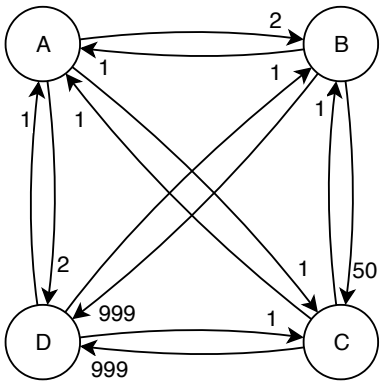
28.01.2020

Algorithmen und Datenstrukturen (WS 2019)

Aufgabenblatt 11

zu bearbeiten bis: 03.02.20 - 05.02.20

Aufgabe 11.1 (TSP - Theorie)
Wir haben nun einige abstrakte Datentypen gesehen und wollen diese zur Implementierung eines Algorithmus benutzen. Dazu betrachten wir wieder das TSP (*Travelling Salesman Problem*), dass wir unter anderem auf Blatt 06 behandelt haben. Betrachten Sie zunächst folgende Instanz des Problems:



Eine *Lösung* des TSP ist eine Folge (oder Liste) von Städten, die besucht werden sollen, zum Beispiel: $[C, D, A, B]$. Dabei muss jeder Stadt genau einmal vorkommen. Die *Güte* einer Lösung ist deren Gesamtkosten für die Rundreise. Da wir eine Rundreise suchen spielt es keine Rolle bei welcher Stadt man startet; im folgenden werden wir immer bei Stadt A starten.

- Welche Güte hat die oben angegebene Lösung?
- Wie lautet die äquivalente Rundreise, welche bei A startet?
- Wie viele wirklich verschiedene Lösungen gibt es für ein TSP mit n Knoten?
- Bestimmen Sie eine TSP Lösung nach dem **Greedy** Verfahren (fangen Sie bei Knoten A an). Geben Sie auch die Güte der Lösung an!
- Macht es für das Greedy Verfahren einen Unterschied wo wir beginnen?
- Bestimmen Sie eine **optimale** TSP Lösung, also eine mit den niedrigsten Kosten die möglich sind. Begründen Sie warum Ihre Lösung optimal ist.
- Gibt es eine andere optimale Lösung? Ist das für jedes TSP so?

Aufgabe 11.2 (TSP - Praxis)
Wir wollen den Algorithmus nun implementieren. Implementieren Sie dazu die Schnittstelle `TSPGraph`, welche Methoden anbietet um Knoten und Kanten (Hier: Städte und Distanzen) entgegen zu nehmen und drei Methoden um Lösungen für das aktuelle Problem zu berechnen:

- `void addCity(String name)`: Fügt eine neue Stadt hinzu (also einen Knoten), die Methode soll prüfen, ob die Stadt schon definiert wurde (damit kein Name doppelt vorkommt).
- `void addDistance(String from, String to, int cost)`: Fügt die Kosten (`cost`) für die Strecke von einer Stadt (`from`) zu einer anderen Stadt (`to`) hinzu. Beide Städte müssen natürlich schon existieren und die Kosten dürfen nicht negativ sein.
- `void reset()`: Entfernt alle Städte und Distanzen wieder.
- `Solution<?> solveAny()`: Gibt irgendeine Lösung zurück; Einfachste Option ist die Liste der Städte zurück zu liefern, so wie sie gegeben wurden...
- `Solution<?> solveGreedy()`: Gibt eine durch eine Implementierung des **Greedy** Algorithmus ermittelte Lösung zurück.
- `Solution<?> solveOptimal()`: Gibt eine *optimale* Lösung zurück, ermittelt durch **Backtracking**, zum Beispiel Ihres von Blatt 06.

Erweitern Sie dazu die vorgegebene Klasse `TSP`, die Ihnen (vorimplementiert) die folgenden Möglichkeiten zum Testen bietet:

- Die Methode `createExample()` ruft die Methoden `addCity` und `addDistance` so auf, dass das Beispiel von diesem Blatt entsteht.
- Die Methode `createRandomPlane(int size, String seed)` konstruiert eine zufällige Instanz mit `size` Städten, bei der jede Stadt zufällige Koordinaten auf einer virtuellen Karte bekommt und die Distanz zwischen Städten ungefähr die geometrische Entfernung ist, mit einer Abweichung von $\pm 20\%$.
- Die Methode `createTotallyRandom(int size, String seed)` konstruiert eine Instanz mit `size` Städten, bei der alle Distanzen komplett zufällig sind.
- Schließlich gibt es schon eine Klasse `Solution`, die Sie benutzen können um eine Lösung zu "speichern" (die übergebene Liste wird kopiert!) und später mit `print` auszugeben.

Überlegen Sie sich, wie Sie die Städte und Distanzen **speichern**, sprich welche *Datenstrukturen* Sie dazu definieren und/oder von Ihren ADT Implementierungen direkt wiederverwenden können. In Ihrern `solve`-Implementierungen werden Sie immer wieder auf Distanzen zwischen Städten zugreifen müssen, etc.

- Welche Komplexität haben Ihre `addCity` und `addDistance` Methoden?
- Welche Zugriffe auf Städte und Distanzen machen Sie in Ihren Algorithmen? Welche Komplexitäten haben diese Zugriffe?