TU Wien
Institut für Logic and Computation
Forschungsbereich Algorithms and Complexity

186.866 Algorithmen und Datenstrukturen VU

Programmieraufgabe P6

PDF erstellt am: 1. Mai 2024

1 Vorbereitung

Um diese Programmieraufgabe erfolgreich durchführen zu können, müssen folgende Schritte umgesetzt werden:

1. Laden Sie das Framework `P6.zip` aus TUWEL herunter.
2. Entpacken Sie `P6.zip` und öffnen Sie das entstehende Verzeichnis als Projekt in IntelliJ (nicht importieren, sondern öffnen).
3. Öffnen Sie die nachfolgend angeführte Datei im Projekt in IntelliJ. In dieser Datei sind sämtliche Programmiertätigkeiten durchzuführen. Ändern Sie keine anderen Dateien im Framework und fügen Sie auch keine neuen hinzu.
   `src/main/java/exercise/StudentSolutionImplementation.java`
4. Füllen Sie Vorname, Nachname und Matrikelnummer in der Methode `StudentInformation provideStudentInformation()` aus.

2 Hinweise

Einige Hinweise, die Sie während der Umsetzung dieser Aufgabe beachten müssen:

- Lösen Sie die Aufgaben selbst und nutzen Sie keine Bibliotheken, die diese Aufgaben abnehmen.
- Sie dürfen beliebig viele Hilfsmethoden schreiben und benutzen. Beachten Sie aber, dass Sie nur die oben geöffnete Datei abgeben und diese Datei mit dem zur Verfügung gestellten Framework lauffähig sein muss.

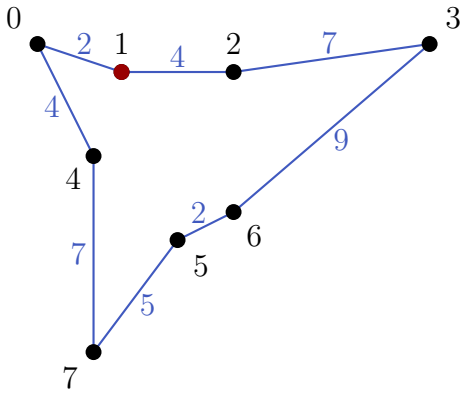
3 Übersicht

In dieser abschließenden Programmieraufgabe befassen Sie sich mit dem Traveling Salesperson Problem (TSP). Ziel ist es, drei verschiedene heuristische Verfahren sowie eine lokale Suche für dieses Problem zu implementieren, zu diskutieren und zu vergleichen.

4 Theorie

Gegeben ist ein vollständiger, ungerichteter Graph $G = (V, E)$ mit $|V| = n$ Knoten und Knotenmenge $V = \{0, \ldots, n-1\}$. Jeder Kante $e \in E$ sind Kosten zugeordnet, die mit $w(e) \in \mathbb{N}$ notiert werden. Die Kantengewichte sind also ganzzahlig. Beim Traveling Salesperson Problem (TSP) gilt es, eine Tour durch den Graphen mit minimalen Kosten zu bestimmen, wobei sich die Kosten einer Tour aus der Summe der Kosten aller Kanten ergeben.

Im Folgenden ist eine Beispielinstanz für ein TSP mitsamt einer möglichen Tour mit dem Startknoten 1 abgebildet:



Eine Tour wird als Array dargestellt, wobei an der Stelle i der Index jener Stadt steht, die in der Tour an i. Stelle besucht wird. Die Stelle 0 markiert den Startknoten der Tour. Für den Startknoten 1 gibt es im obigen Beispiel zwei mögliche Darstellungen (einmal im und einmal gegen den Uhrzeigersinn):

Index	0	1	2	3	4	5	6	7
Knoten	1	2	3	6	5	7	4	0

Index	0	1	2	3	4	5	6	7
Knoten	1	0	4	7	5	6	3	2

Um die Tour zu schließen, muss von der letzten Stadt (in obigen Beispielen von 0 bzw. 2) noch zur Startstadt 1 gegangen werden. In beiden Fällen ergeben sich die Tourkosten als Summe aller Kantengewichte der Tour gemäß $4 + 7 + 9 + 2 + 5 + 7 + 4 + 2 = 40$.

Die folgenden drei **heuristischen Verfahren** (allesamt sogenannte Konstruktionsheuristiken) sollen von Ihnen implementiert werden:

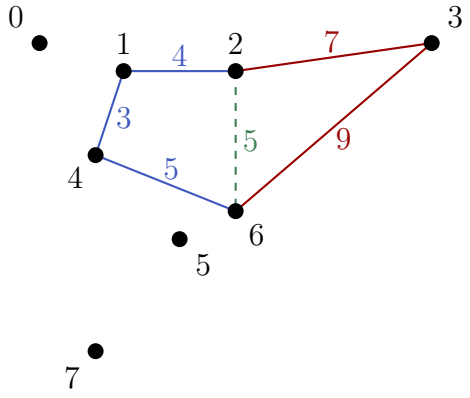
- **Nearest-Neighbor-Heuristik**: Starte bei einem beliebigen (vorgegebenen) Knoten $v_0 \in V$. Für $t = 0, 1, 2, \ldots$: Gehe ausgehend vom aktuellen Knoten $v_t \in V$ immer zu jenem noch nicht besuchten Knoten $v_{t+1} \in V$, der die geringste Distanz zum aktuellen Knoten v_t hat. Wiederhole dies, bis alle Knoten besucht sind.

- **Cheapest-Insertion-Heuristik**: Gegeben ist eine (zufällige) Reihenfolge der einzufügenden Knoten $v_0, v_1, \ldots, v_{n-1}$. Füge die ersten beiden Knoten v_0 und v_1 in die Tour ein. Für $t \geq 2$: Füge den aktuellen Knoten v_t an jener Stelle in der Tour ein, sodass die Einfügekosten minimal sind.

  Beispiel: Für das obige Beispiel sei bereits die folgende Teiltour konstruiert worden:

Index	0	1	2	3	4	5	6	7
Knoten	1	2	6	4	–	–	–	–

  Nun soll die Stadt **3** in die Tour eingefügt werden. Die folgende Grafik zeigt, wie die Stadt **3** zwischen den Städten 2 und 6 (**Stelle 2**) eingefügt wird:



  Nach dem Einfügen sieht das Array so aus:

Index	0	1	**2**	3	4	5	6	7
Knoten	1	2	**3**	6	4	–	–	–

Die Einfügekosten belaufen sich auf $7+9-5=11$. Es ist nicht möglich, den Knoten **3** an einer anderen Stelle günstiger einzufügen.

- **Spanning-Tree-Heuristik (STH)**: Die Grundidee basiert darauf, ausgehend von einem minimalen Spannbaum (MST) eine Tour zu konstruieren. Die Details zu dieser Heuristik finden Sie in den Vorlesungsfolien zu „Optimierung – Approximation“.

Für die Implementierung der **Spanning-Tree-Heuristik** sollen Sie außerdem einen minimalen Spannbaum mit Hilfe des **Algorithmus von Prim** berechnen. Die notwendige Theorie dazu finden Sie in den Vorlesungsfolien „Greedy-Algorithmen“.

Bei der **lokalen Suche** wird eine gegebene Startlösung solange verbessert, bis ein lokales Optimum bezüglich einer Nachbarschaftsstruktur erreicht ist. Hintergrundinformationen zur lokalen Suche finden Sie in den Vorlesungsfolien „Optimierung – Heuristische Verfahren“. In dieser Programmieraufgabe wird die **1-Insertion**-Nachbarschaftsstruktur betrachtet: Ausgehend von einer Tour x besteht die Nachbarschaft $N(x)$ aus allen Lösungen, bei denen eine beliebige Stadt aus x entfernt und an einer beliebigen anderen Stelle eingefügt wird.

Für die **lokale Suche** sollen Sie in dieser Programmieraufgabe folgende Nachbarschaftsstruktur implementieren:

- **1-Insertion**: Ist ein lokales Optimum erreicht, so ist es nicht mehr möglich, die Tour zu verbessern, indem man einen beliebigen Knoten der Tour an einer anderen Stelle einfügt.

5 Implementierung

Als Hilfestellung werden die Klassen `TSPInstance` und `TSPSolution` zur Verfügung gestellt.

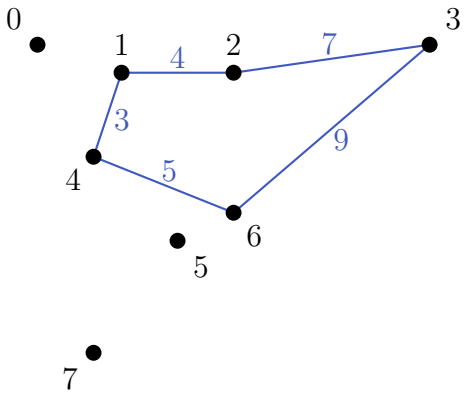
Die Klasse `TSPInstance` verwaltet eine Instanz des TSPs und stellt unter anderem folgende Methoden bereit, die Sie verwenden dürfen:

- `int getDistance(int from, int to)`: Gibt die Distanz zwischen den Knoten `from` und `to` zurück. Da alle betrachteten TSPs symmetrisch sind, sind die Rollen von `from` und `to` vertauschbar.

- `int getN()`: Gibt die Instanzgröße zurück.
- `String getName()`: Gibt den Namen der Instanz zurück.
- `int getOptValue()`: Gibt die Länge einer optimalen Tour zurück. Verwenden Sie diesen Wert, um die Qualität Ihrer gefundenen Lösungen mit dem Optimum vergleichen zu können.

Die Klasse `TSPSolution` verwaltet die Lösung eines TSPs. Die gleich folgenden Erklärbeispiele beziehen sich auf das Objekt `TSPSolution sol`, das sich auf folgende Teiltour bezieht:

Index	0	1	2	3	4	5	6	7
Knoten	1	2	3	6	4	–	–	–



Folgende Methoden stellt die Klasse `TSPSolution` unter anderem bereit:

- `int get(int index)`: Gibt den Knoten an der Stelle `index` zurück. Ist an der Stelle `index` kein Knoten angeordnet, so wird `-1` zurückgegeben. `sol.get(0)` würde `1` und `sol.get(6)` würde `-1` zurückgeben.
- `int getIndexOfCity(int city)`: Gibt die Stelle zurück, bei welcher der Knoten `city` eingefügt ist. Ist `city` (noch) nicht Teil der Tour, so wird `-1` zurückgegeben. `sol.getIndexOfCity(1)` würde `0` und `sol.getIndexOfCity(7)` würde `-1` zurückgeben.
- `int getSolutionLength()`: Gibt die Anzahl der aktuell in der Lösung befindlichen Knoten zurück. `sol.getSolutionLength()` würde `5` liefern.
- `int getObjective()`: gibt die aktuelle Tourlänge der Lösung zurück. `sol.getObjective()` würde `28` liefern ($4 + 7 + 9 + 5 + 3$).

- `boolean isVisited(int city)`: Gibt an, ob der Knoten `city` in der Tour enthalten ist (`true`) oder nicht (`false`). `sol.isVisited(1)` würde `true` und `sol.isVisited(7)` würde `false` zurückgeben.

- `boolean insert(int city, int index)`: Versucht, den Knoten `city` an der Stelle `index` einzufügen. Gelingt dies, so wird `true` zurückgegeben. Kann der Knoten `city` hingegen nicht eingefügt werden, so wird `false` zurückgegeben. `sol.insert(7, 4)` würde den Knoten `7` an der Stelle `4` zwischen den Städten `6` und `4` einfügen und `true` zurückliefern. `sol.insert(7, 6)` hingegen würde `false` liefern und die Lösung unverändert lassen, da eine Stadt in der aktuellen Tour von `sol` höchstens an Stelle 5 eingeordnet werden kann. `sol.insert(3, 1)` würde die Lösung ebenso nicht verändern und `false` zurückliefern, da die Stadt `3` bereits in der Tour enthalten ist.

- `boolean insertLast(int city)`: Versucht, den Knoten `city` hinten an die Lösung anzuhängen. Gelingt dies, so wird `true` zurückgegeben. Kann der Knoten `city` hingegen nicht eingefügt werden, so wird `false` zurückgegeben. Der Aufruf `sol.insertLast(7)` entspricht dem Aufruf `sol.insert(7, sol.getSolutionLength())`.

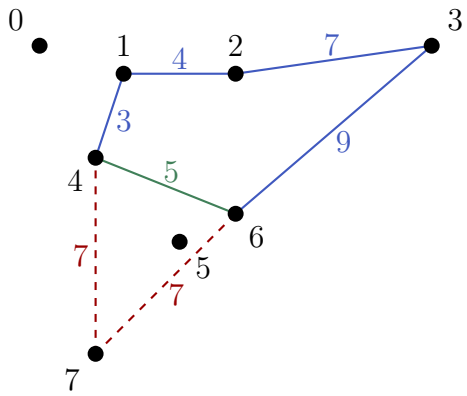
- `boolean insertFirst(int city)`: Versucht, den Knoten `city` vorne an die Lösung anzuhängen. Der Aufruf `sol.insertFirst(7)` entspricht dem Aufruf `sol.insert(7, 0)`.

- `int delete(int index)`: Löscht den Knoten an der Stelle `index` und gibt den Index jenes Knotens zurück, der sich an dieser Stelle befunden hat. Konnte an der gewünschten Stelle kein Knoten gefunden werden (und daher keine Löschung erfolgen), so wird `-1` zurückgegeben. `sol.delete(2)` würde die Stadt an der Stelle `2` löschen und `3` zurückgeben. `sol.delete(6)` würde die Lösung nicht verändern und `-1` zurückgeben.

- `int deleteCity(int city)`: Löscht den Knoten `city` aus der Tour und gibt die Stelle zurück, an der sich dieser Knoten befunden hat. Kann die Stadt in der Lösung nicht gefunden werden (und daher keine Löschung erfolgen), so wird `-1` retourniert. `sol.deleteCity(3)` würde den Knoten `3` löschen und `2` zurückliefern. `sol.deleteCity(7)` würde die Lösung unberührt lassen und `-1` zurückgeben.

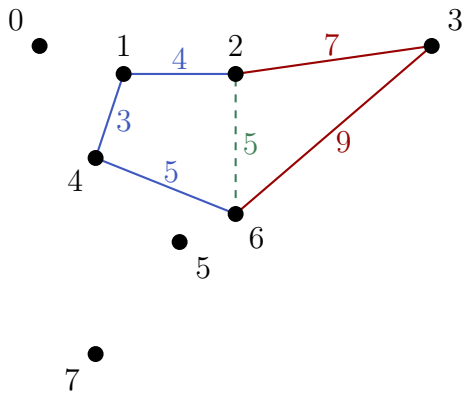
- `int deltaInsert(int city, int index)`: Berechnet, um wie viel sich die Tourlänge verändern würde, wenn der Knoten `city` an der Stelle `index` eingefügt werden würde. Es handelt sich dabei um eine

Zahl größer oder gleich 0. Kann der Knoten nicht (an der Stelle `index`) eingefügt werden, so wird `0` zurückgegeben. Die Tour bleibt jedenfalls unberührt. `sol.deltaInsert(7, 4)` würde `9` zurückgeben ($7 + 7 - 5$), wie der folgenden Skizze entnommen werden kann:



`sol.deltaInsert(7, 6)` hingegen würde `0` liefern, da eine Stadt in der aktuellen Tour von `sol` höchstens an Stelle 5 eingeordnet werden kann. `sol.deltaInsert(3, 1)` würde ebenso `0` zurückgeben, da `3` bereits Teil der Tour ist.

- `int deltaDelete(int index)`: Berechnet, um wie viel sich die Tourlänge verändern würde, wenn der Knoten an der Stelle `index` aus der Tour gelöscht werden würde. Es handelt sich dabei um eine Zahl kleiner oder gleich 0. Ist an dieser Stelle keine Stadt angeordnet (wodurch keine Löschung erfolgen kann), so wird `0` zurückgegeben. Die Tour bleibt jedenfalls unberührt. `sol.deltaDelete(2)` würde `-11` zurückgeben ($-7 - 9 + 5$), siehe folgende Skizze:



`sol.deltaDelete(6)` würde `0` zurückliefern, da es an der Stelle 6 keine Stadt zum Entfernen gibt.

- `void print()`: Druckt die aktuelle Lösung auf die Console. Mit `sol.print()` kann die Lösung `sol` ausgedruckt werden.

Beachten Sie, dass sich die Klasse `TSPSolution` eigenständig um die Berechnung der Tourlänge kümmert, Sie brauchen also nicht die Tourlängen berechnen oder modifizieren. Während die Methoden `insert`, `insertLast` und `insertFirst` den übergebenen Knoten `city` tatsächlich in die Lösung einfügen, geschieht dies **nicht** in der Methode `deltaInsert`. Analog verhält es sich mit den Methoden `delete`, `deleteCity` und `deltaDelete`.

5.1 Nearest Neighbor

Implementieren Sie die Nearest-Neighbor-Heuristik in der Methode `public void nearestNeighbor(TSPInstance instance, TSPSolution solution, int startCity)`.

Das Objekt `instance` enthält die Instanz, auf welche die Heuristik angewendet werden soll. `startCity` ist der Index jener Stadt, von der aus die Tourkonstruktion starten soll.

Das Objekt `solution` ist ein vorinitialisiertes Objekt der Klasse `TSPSolution`, das eine anfangs leere Lösung enthält. Legen Sie Ihre gefundene Tour in dieses Objekt ab, indem Sie die Knoten der Reihe nach in die Tour einfügen.

Achtung: `startCity` muss der erste Knoten der Tour sein; am Ende muss also `solution.get(0)` den Wert `startCity` liefern!

5.2 Cheapest-Insertion-Heuristik

Implementierten Sie die Cheapest-Insertion-Heuristik in der Methode `public void cheapestInsertion(TSPInstance instance, TSPSolution solution, int[] order)`.

Das Objekt `instance` enthält die Instanz, auf welche die Heuristik angewendet werden soll. `order` gibt die Reihenfolge an, in der die Knoten in die Tour eingefügt werden sollen. `int city = order[2]` gibt also beispielsweise an, dass der Knoten `city` als dritter Knoten in die Tour eingefügt werden soll.

Die gefundene Tour soll wiederum im Objekt `solution` (ein vorinitialisiertes Objekt der Klasse `TSPSolution` mit einer anfangs leeren Lösung) hinterlegt werden.

5.3 Lokale Suche

Implementierten Sie die lokale Suche in der Methode `void localOptimum(TSPInstance instance, TSPSolution solution)`. Ziel dieser Methode ist es, ein lokales Optimum bezüglich der **1-Insertion**-Nachbarschaftsstruktur zu bestimmen.

Das Objekt `instance` enthält die zugrundeliegende Instanz und `solution` ist ein Objekt der Klasse `TSPSolution`, das bereits eine vollständige Tour enthält, die nun mit Hilfe einer lokalen Suche verbessert werden soll.

Nach Ausführung der Methode `localOptimum` soll es nicht mehr möglich sein, einen beliebigen Knoten der Tour an einer anderen Stelle einzufügen, sodass sich die Tourlänge verringert.

Hinweis: Sie können die Methode `localOptimum` mit den Testinstanzen der Datei `local-search.csv` testen (siehe Abschnitt 6). Sollte das Durchlaufen länger als eine Minute dauern, dann können Sie die Laufzeit drastisch reduzieren. Stellen Sie sich dazu folgende Fragen:

1. Ist es beim Durchprobieren von anderen Einfügepositionen notwendig, die Stadt wirklich jedes Mal einzufügen?
2. Wenn zum Beispiel die Stadt von der Position `i` an einer anderen Stelle eingefügt wurde und eine Verbesserung erzielt wurde: Ist es notwendig bzw. sinnvoll, im nächsten Durchlauf wieder von vorne (also beim Index `0`) mit der Suche anzufangen? Oder könnte man evtl. die lokale Suche im nächsten Durchlauf bei der Position `i` fortsetzen?

5.4 Minimaler Spannbaum – Algorithmus von Prim

Implementieren Sie den Algorithmus von Prim in der Methode `int prim(TSPInstance instance, PriorityQueue q, Graph mst)`. Ziel dieser Methode ist es, einen minimalen Spannbaum zu bestimmen, der als Grundlage für die in Abschnitt 5.5 folgende Spanning-Tree-Heuristik dient.

Das Objekt `instance` enthält die zugrundeliegende Instanz. Der zweite Parameter `PriorityQueue q` ist eine vorinitialisierte und anfangs leere Priority Queue. Diese Klasse stellt einige hilfreiche Methoden zur Verfügung, die Sie verwenden dürfen:

- `void add(int weight, int vertexId)`: Ein neuer Knoten kann der Priority Queue hinzugefügt werden, indem eine Gewichtung `weight` und eine Knoten-ID `vertexId` angegeben wird. Sollte eine Knoten-ID bereits in der Priority Queue vorhanden sein, hat der Aufruf dieser Methode keine Folgen.
- `boolean isEmpty()`: Gibt Ihnen Auskunft darüber, ob sich keine Knoten mehr in der Priority Queue befinden.
- `int removeFirst()`: Mit `int city = q.removeFirst()` können Sie die ID jener Stadt mit der niedrigsten Gewichtung aus der Priority Queue auslesen und gleichzeitig aus dieser entfernen. Ist die Priority Queue leer, wird -1 zurückgegeben.
- `void decreaseWeight(int weight, int vertexId)`: Um die Gewichtung eines Knotens innerhalb der Priority Queue zu senken, kann diese Methode aufgerufen werden. Einerseits muss das neue, verringerte Gewicht angegeben werden (`weight`) und andererseits die Knoten-ID des gewünschten Knotens, also die ID der gewünschten Stadt (`vertexId`). Änderungen passieren nur, wenn ein Gewicht angegeben wird, das kleiner ist, als das bisherige, und wenn ein Knoten mit der Knoten-ID in der Priority Queue vorhanden ist.

Der dritte Parameter `mst` enthält ein Objekt der Klasse `Graph`. Diese Klasse dient der Verwaltung von ungerichteten Graphen und ist mit der Knotenmenge V und einer leeren Kantenmenge vorinitialisiert. Die Klasse `Graph` stellt einige Methoden zur Verfügung, die sich für die Implementierung des Algorithmus von Prim als nützlich erweisen könnten:

- `int numberOfVertices()`: Gibt die Anzahl der Knoten im Graph zurück. Knoten werden mittels aufsteigender IDs gekennzeichnet. Angenommen es gibt insgesamt n Knoten, dann gibt es jeweils Knoten mit Knoten-ID $0, 1, \ldots, n-1$.
- `int numberOfEdges()`: Gibt die Anzahl der Kanten zurück.
- `boolean containsEdge(int vertexStart, int vertexEnd)`: Gibt an, ob eine Kante zwischen den Knoten `vertexStart` und `vertexEnd` existiert (`true`) oder nicht (`false`).
- `int[] getNeighbors(int vertexId)`: Gibt die Nachbarn des Knotens `vertexId` zurück als Array von Knoten-IDs. Bei der Übergabe einer ungültigen Knoten-ID wird `null` zurückgegeben.

- `boolean addEdge(int vertexIdStart, int vertexIdEnd, int weight)`: Fügt dem Graphen eine Kante zwischen den Knoten `vertexIdStart` und `vertexIdEnd` hinzu. Das Gewicht dieser Kante wird dem Parameter `weight` übergeben. Kann die Kante eingefügt werden, so wird `true` zurückgegeben. Kann die Kante nicht eingefügt werden (weil mindestens einer der beiden Knoten nicht existiert oder die Kante bereits existiert), so bleibt der Graph unverändert und es wird `false` zurückgegeben.

- `boolean addEdge(int vertexIdStart, int vertexIdEnd)`: Entspricht der Methode `boolean addEdge(int vertexIdStart, int vertexIdEnd, int weight)`, wobei `weight` auf den Wert `0` gesetzt wird.

- `int getEdgeWeight(int vertexIdStart, int vertexIdEnd)`: Gibt das Gewicht der Kante zwischen den Knoten `vertexIdStart` und `vertexIdEnd` zurück. Ist diese Kante nicht im Graphen enthalten, so wird `-1` zurückgegeben.

- `void setEdgeWeight(int vertexIdStart, int vertexIdEnd, int weight)`: So die Kante zwischen `vertexIdStart` und `vertexIdEnd` existiert, wird das Kantengewicht dieser Kante auf den Wert `weight` gesetzt.

- `int[][] getEdges()`: Gibt die Kanten des Graphen in Gestalt eines $m \times 2$ Arrays zurück, wobei m die Anzahl der enthaltenen Kanten ist. Für `Graph g` und `int[][] edges = g.getEdges()` kann mittels `int[] edge = edges[i]` auf die i. Kante ($i = 0, 1, \ldots, m-1$) zugegriffen werden. `edge[0]` und `edge[1]` enthält sodann die beiden Knoten. Mit `edges[i][0]` und `edges[i][1]` kann auch direkt auf die beiden Knoten der i. Kante zugegriffen werden.

- `void printEdges()`: Druckt die Kanten des Graphen auf die Console.

Am Ende sollen im Objekt `mst` die Kanten des minimalen Spannbaums abgelegt sein. Es ist dabei nicht notwendig, dass Sie die Kanten mit Kantengewichten versehen; die Kantengewichte werden bei der Überprüfung Ihrer Lösung ignoriert. Als Rückgabewert von `int prim(TSPInstance instance, PriorityQueue q, Graph mst)` wird das korrekte Gesamtgewicht des minimalen Spannbaumes erwartet.

5.5 Spanning-Tree-Heuristik

Implementieren Sie die Spanning-Tree-Heuristik in der Methode `void spanningTreeHeuristic(TSPInstance instance, TSPSolution solution, Graph mst, int startCity)`.

Das Objekt `instance` enthält die zugrundeliegende Instanz und `solution` ist ein vorinitialisiertes Objekt der Klasse `TSPSolution`, das anfangs leer ist und in das Sie Ihre berechnete Tour ablegen sollen. Beginnen Sie die Tour dabei beim Knoten `startCity`. Der minimale Spannbaum, der für die Tourkonstruktion herangezogen werden soll, ist im Objekt `mst` hinterlegt. In Abschnitt 5.4 finden Sie Informationen und Methoden der Klasse `Graph`.

Beachten Sie, dass der Algorithmus von Prim aus Abschnitt 5.4 bereits implementiert sein muss, da der minimale Spannbaum vor dem Aufruf der Methode `spanningTreeHeuristic` mit der Methode `prim` berechnet wird.

Achtung: `startCity` muss der erste Knoten der Tour sein; am Ende muss also `solution.get(0)` den Wert `startCity` liefern!

6 Testen

Führen Sie zunächst die `main`-Methode in der Datei `src/main/java/framework/Exercise.java` aus.

Anschließend wird Ihnen in der Konsole eine Auswahl an Testinstanzen angeboten, darunter befindet sich zumindest `abgabe.csv`:

```
Select an instance set or exit:
[1] abgabe.csv
[0] Exit
```

Durch die Eingabe der entsprechenden Ziffer kann entweder eine Testinstanz ausgewählt werden oder das Programm (mittels der Eingabe von `0`) verlassen werden. Wird eine Testinstanz gewählt, dann wird der von Ihnen implementierte Programmcode ausgeführt. Kommt es dabei zu einem Fehler, wird ein Hinweis in der Konsole ausgegeben.

Relevant für die Abgabe ist das Ausführen der Testinstanz `abgabe.csv`.

Die Testinstanzen `nearest-neighbor.csv`, `cheapest-insertion.csv`, `local-search.csv`, `prim.csv` sowie `spanning-tree-heuristic.csv` enthalten Testinstanzen, mit denen Sie die Methoden `nearestNeighbor` (Abschnitt 5.1), `cheapestInsertion` (Abschnitt 5.2), `localOptimum` (Abschnitt 5.3), `prim` (Abschnitt 5.4) sowie `spanningTreeHeuristic` (Abschnitt 5.5) testen können. Beachten Sie, dass Sie die Methode `spanningTreeHeuristic` erst testen können, wenn Sie die Methode `prim` implementiert haben! Die anderen vier genannten Methoden können Sie unabhängig voneinander testen.

Die Testinstanzen `nearest-neighbor-ls.csv`, `cheapest-insertion-ls.csv` und `spanning-tree-heuristic-ls.csv` führen die drei Heuristiken mit einer subsequenten lokalen Suche durch. Beachten Sie, dass sie bei diesen drei Testinstanzen die Methode `localOptimum` implementiert haben müssen!

7 Evaluierung

Wenn der von Ihnen implementierte Programmcode mit der Testinstanz `abgabe.csv` ohne Fehler ausgeführt werden kann, dann wird nach dem Beenden des Programms im Ordner `results` eine Ergebnis-Datei mit dem Namen `solution-abgabe.csv` erzeugt.

Die Datei `solution-abgabe.csv` beinhaltet die erstellten Touren, deren Lösungsqualitäten (Tourlängen) und Zeitmessungen der Ausführung der Testinstanz `abgabe.csv`, welche in einem Web-Browser visualisiert werden können. (Auch Ergebnis-Dateien anderer Testinstanzen können zu Testzwecken visualisiert werden.) Öffnen Sie dazu die Datei `visualization.html` in Ihrem Web-Browser und klicken Sie rechts oben auf den Knopf *Ergebnis-Datei auswählen*, um `solution-abgabe.csv` auszuwählen.

Beantworten Sie basierend auf der Visualisierung die Fragestellungen aus dem folgenden Abschnitt.

8 Fragestellungen

Öffnen Sie die Datei `solution-abgabe.csv`, die Ergebnisse zu Ihren implementierten Methoden enthält. Es wurden folgende sechs Lösungstechniken auf diverse TSP-Instanzen angewendet:

- *Ins* bzw. *Ins+LS*: **Cheapest Insertion** ohne bzw. mit subsequenter lokaler Suche.
- *NN* bzw. *NN+LS*: **Nearest Neighbor** ohne bzw. mit subsequenter lokaler Suche.
- *STH* bzw. *STH+LS*: **Spanning Tree Heuristic** ohne bzw. mit subsequenter lokaler Suche. Dabei fließt auch die die Berechnung eines minimalen Spannbaumes mit ein.

Jede dieser sechs Lösungsmethoden wurde jeweils 10 Mal mit unterschiedlichen Startstädten (bei *NN* und *STH*) oder Einfügereihenfolgen (bei *Ins*) auf folgende Instanzen angewendet: *test8*, *eil51*, *eil76*, *eil101*, *lin105*, *tsp225*, *a280*, *pcb442*, *pr1002* und *pr2392*. Mit Ausnahme von *test8* stammen alle Instanzen von der **TSPLIB**[1]. Die Zahl am Ende des Namens gibt die Instanzgröße an, so besteht etwa die Instanz *eil51* aus 51 Knoten.

Die Visualisierung enthält drei Teile:

- **Laufzeitgrafiken**: Hier werden die Laufzeiten der sechs Lösungsmethoden in Abhängigkeit von der Instanzgröße dargestellt. Jeder Punkt entspricht dem Mittelwert über die Laufzeiten all jener 10 Durchläufe, die auf der entsprechenden Instanz mit der entsprechenden Methode gemessen wurden.
- **Tourlängen**: Hier können Sie eine Instanz auswählen und die Tourlängen, die von den sechs Methoden in jeweils 10 Durchläufen erzielt wurden, im dargestellten Boxplot miteinander vergleichen.
- **Routen**: Hier können Sie für die im vorherigen Punkt selektierte Instanz eine Route auswählen und visualisieren.

[1] `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`

Bearbeiten Sie folgende Aufgaben- und Fragestellungen:

1. **Laufzeiten vergleichen**: Durch Klicken auf den Gruppennamen in der Legende neben der Plots lassen sich einzelne Gruppen aus- bzw. einblenden.

   (a) Blenden Sie in der Laufzeitgrafik alle Kategorien bis auf *Ins*, *NN* und *STH* aus. Es sollen also die Laufzeiten der drei Heuristiken ohne anschließende lokale Suche angezeigt werden. Wie schneiden die drei Verfahren laufzeittechnisch ab? Was beobachten Sie?

   (b) Blenden Sie in der Laufzeitgrafik alle Kategorien bis auf *Ins+LS*, *NN+LS* und *STH+LS* aus. Es sollen also die Laufzeiten der drei Heuristiken mit anschließender lokaler Suche angezeigt werden. Wie schneiden die drei Verfahren ab? Was beobachten Sie?

   (c) Blenden Sie in der Laufzeitgrafik alle Kategorien ein. Wie viel Zeit nimmt die lokale Suche bei den drei Heuristiken jeweils relativ ein?

   Drücken Sie jeweils in der Menüleiste rechts über dem Plot auf den Fotoapparat, um die drei erzeugten Plots als Bild zu speichern.

   Hinweis: Sie können beliebig in die Grafik hineinzoomen, indem Sie auf die Lupe (Zoom) klicken und den gewünschten Bereich per Rechtecksauswahl selektieren. Durch Klick auf das Haussymbol (Reset axes) können Sie die Grafik wieder zurücksetzen.

2. **Tourlängen vergleichen**: Klicken Sie sich zunächst im Abschnitt Tourlängen durch alle Instanzen durch und verschaffen Sie sich einen Überblick über die erzeugten Tourlängen der sechs Verfahren.

   (a) Wie schneiden die Verfahren gütemäßig ab? Welche Verfahren erzeugen tendenziell kürzere bzw. längere Touren?

   (b) Wie viel bringt die anschließende lokale Suche bei jedem Verfahren? Erkennen Sie einen Zusammenhang zwischen der Laufzeit und dem Gütegewinn der lokalen Suche?

   (c) Wie stark streuen die Längen der 10 erzeugten Touren jeweils?

   (d) Wie nahe kommen die Verfahren jeweils an das Optimum heran? Die Tourlänge der Optimallösung wird Ihnen im Dropdownfeld der Instanzauswahl angezeigt.

   Speichern Sie die Boxplots für die Instanzen *eil51*, *tsp225* und *pr2392* als Bild ab, indem Sie auf den Fotoapparat rechts über den Plot klicken.

3. **Routencharakteristika studieren**: Wählen Sie bei den Tourlängen die Instanz *eil51* aus und klicken Sie sich anschließend im Abschnitt **Routen für eil51** durch einige Lösungen um sich einen Überblick zu verschaffen.

   (a) Wie würden Sie die generierten Touren charakterisieren (etwa hinsichtlich der Längen der gewählten Kanten, etwaig auftretende Kreuzungen etc.)? Welche Gemeinsamkeiten bzw. Unterschiede sehen Sie dabei zwischen den drei Verfahren (ohne lokale Suche)?

   (b) Wie stark ändert sich die Charakteristik der Lösungen nach Anwendung der lokalen Suche? *Hinweis*: Die Touren der drei Heuristiken *Ins*, *NN* und *STH* sind mit Durchlauf 1 bis Durchlauf 10 nummeriert und dienen als Startlösungen für die Touren, die durch die subsequente lokale Suche verbessert wurden. So dient etwa die Tour von *Ins*, Durchlauf 4 als Startlösung für *Ins+LS*, Durchlauf 4.

   (c) Gegeben Ihrer bisherigen Beobachtungen: Welches der studierten Lösungsverfahren würden Sie für die Lösung des TSPs empfehlen und warum?

   Machen Sie für die Instanz *eil51* für jede der sechs Methoden jeweils einen Screenshot mit einer repräsentativen Tour.

   Sehr gerne dürfen Sie sich freiwillig auch die Lösungen für andere Instanzen ansehen oder/und mit der Optimallösung (ganz unten in der Auswahlbox) vergleichen! Die Instanz *tsp225* beispielsweise ist besonders geeignet, um abzuschätzen, wie nahe die Verfahren ans Optimum herankommen.

4. **Nachbarschaftsstruktur**: Bei euklidischen TSPs – das sind TSPs, bei denen alle Punkte in der Ebene eingebettet sind und die Distanzen sich aus der euklidischen Distanz ergeben – können Touren, die Kreuzungen enthalten, nicht optimal sein. Offensichtlich sind nicht alle generierten Lösungen kreuzungsfrei bzw. kann die **1-Insertion**-Nachbarschaftsstruktur offenbar nicht alle Kreuzungen auflösen. Beschreiben Sie eine Nachbarschaftsstruktur für die lokale Suche, die in der Lage ist, Kreuzungen zuverlässig zu entfernen.

   Demonstrieren Sie diese Nachbarschaftsstruktur anhand einer Lösung für die Instanz *eil51*, die zumindest eine Kreuzung enthält.

5. **TSPSolution**: Öffnen Sie die Datei `TSPSolution.java`, die sich im selben Ordner wie `StudentSolutionImplementation.java` befindet und studieren Sie die Implementierung der Klasse `TSPSolution`.

   (a) Welche Datenstruktur wird für die Speicherung der Tour verwendet? Welche Hilfsdatenstrukturen bzw. Hilfsvariablen werden verwendet und welchen Zweck erfüllen sie?

   (b) Was passiert, wenn eine Stadt eingefügt wird? Ist es von der Laufzeit her günstiger, eine Stadt vorne oder hinten einzufügen?

   (c) Wie gut eignet sich die Implementierung für die Umsetzung der drei Heuristiken Nearest Neighbor, Cheapest Insertion und Spanning Tree Heuristik?

   (d) Welche Auswirkungen auf die Laufzeit hätte es, wenn bei der lokalen Suche **nicht** auf die delta-Methoden zurückgegriffen werden würde, also weder auf `deltaInsert` noch `deltaDelete`?

   Freiwillige Zusatzfrage: Wie könnte die Implementierung von `TSPSolution` aussehen, damit sie besser für die lokale Suche mit der **1-Insertion**-Nachbarschaftsstruktur geeignet ist?

Fügen Sie Ihre Antworten in einem Bericht gemeinsam mit den erstellten Bildern der Plots zusammen.

9 Abgabe

Laden Sie die Datei `src/main/java/exercise/StudentSolutionImplementation.java` in der TUWEL-Aktivität *Hochladen Source-Code P6* hoch. Fassen Sie diesen Bericht mit den anderen für das zugehörige Abgabegespräch relevanten Berichten in einem PDF zusammen und geben Sie dieses in der TUWEL-Aktivität *Hochladen Bericht Abgabegespräch 2* ab.

10 Nachwort

Das TSP ist vermutlich das bekannteste NP-schwierige Problem, zu dem es eine riesige Anzahl an Publikationen gibt. Dies liegt vermutlich daran, dass

dieses Problem sehr einfach definiert ist und verstanden werden kann, gleichzeitig es aber eben auch in der Praxis nicht leicht zu lösen ist. Vor allem hat sich das TSP auch als Benchmarkproblem etabliert wenn es darum geht neue Varianten von allgemeineren Optimierungsverfahren zu präsentieren bzw. zu vergleichen – oder das Prinzip unterschiedlicher Optimierungsverfahren in der Lehre zu veranschaulichen.

Interessant dabei ist, dass das klassische TSP gar nicht viele echte unmittelbare Anwendungen besitzt. Offensichtlich ist die Planung einer „Rundreise über Städte“ ja ein allzu stark vereinfachte Beispielanwendung, da bei einer wirklichen Reiseplanung sehr viele andere Aspekte eine allzu große Rolle spielen. Eine wirklich unmittelbare Anwendung, wo auch große TSP-Instanzen auftreten, ist beispielsweise das in der Vorlesung erwähnte automatisierte Bohren von Löchern in Platinen für elektronische Schaltkreise. Unabhängig davon stellt das TSP aber in gewissem Sinn „den kleinsten gemeinsamen Nenner“ einer Vielzahl von in der Praxis äußerst bedeutenden Transportproblemen dar, wie den vielen Varianten des Vehicle Routing Problems, Orienteering Problemen, Dial-and-Ride Problemen usw., was wohl auch die Prominenz des TSP erklärt.

Wir hoffen, dass Sie den in dieser LVA gegebenen Überblick über die sehr unterschiedlichen algorithmischen Herangehensweisen zum exakten oder näherungsweisen Lösen des TSP und anderer schwieriger kombinatorischen Optimierungsprobleme spannend fanden. - Jedenfalls hat das bestmögliche Lösen derartiger Probleme eine enorme praktische Bedeutung. Bessere Lösungen führen häufig unmittelbar zu geringeren Kosten, höherer Zeiteffektivität, Materialeinsparung, Energieeinsparung und/oder auch mehr Umweltverträglichkeit. Mit besseren Algorithmen kann somit sehr viel bewirkt werden!

Sollten Sie Algorithmen zum Lösen kombinatorischer Optimierungsaufgaben interessant finden würden wir uns freuen Sie in der einen oder anderen unsere Spezial-LVAs in Bezug auf diese Thematik begrüßen zu dürfen:

192.137 Heuristic Optimization Techniques, 186.835 Mathematical Programming, , 186.856 Structural Decompositions and Algorithms, 186.861 VU Modeling and Solving Constrained Optimization Problems, 186.102 Approximation Algorithms

Da die kombinatorische Optimierung auch in unseren Forschungsschwerpunkten stark vertreten ist, bieten wir hierzu auch immer wieder Themen für Bachelor-, Projekt- und Masterarbeiten an. Bei Interesse kontaktieren Sie uns bitte einfach! Weiteres hierzu finden Sie auf unserer Homepage `https://www.ac.tuwien.ac.at/courses`.