Fakultät IV - Elektrotechnik und Informatik
Algorithmen und Datenstrukturen SoSe 2016
Brock / Deimel / Foucard / Mabrouk / Sieverling

Aufgabenblatt 11

letzte Aktualisierung: 11. July, 10:14 Uhr

Ausgabe: 10.07.2016
Abgabe: 20.07.2016 23:59

Thema: *Heuristische Suche, A^**

Abgabe

Die folgenden Datei muss für eine erfolgreiche Abgabe im svn Ordner
`Tutorien/txx/Studierende/deinname@TU-BERLIN.DE/Abgaben/`
eingecheckt sein:

Geforderte Datei:	
`Blatt11/src/HeuristicManhattan.java`	Aufgabe 1.2
`Blatt11/src/HeuristicEuclidean.java`	Aufgabe 1.3
`Blatt11/src/GridGraph.java`	Aufgabe 1.4

Wichtige Ankündigungen

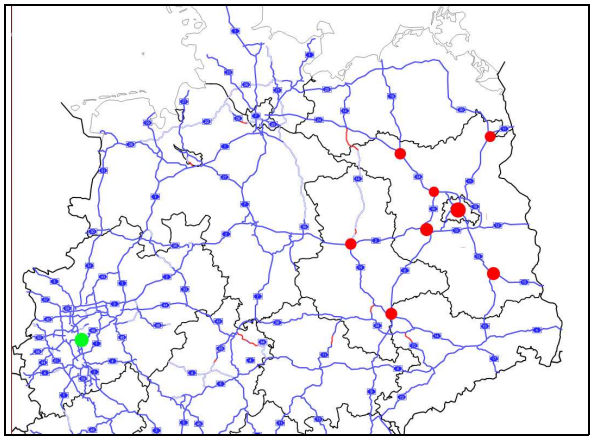
- Unit tests dürfen geteilt werden. Lösungen dürfen auf keinen Fall geteilt werden! Wir testen auf Plagiate.

1. Aufgabe: Kürzester Weg mit A^*

A^* ist ein Algorithmus zum Finden des kürzesten Pfades zwischen zwei Knoten. Der Algorithmus arbeitet wie Dijkstra's Algorithmus auf einer Priority-Queue, jedoch benutzt der Algorithmus zusätzliche Information in Form einer sogenannten Heuristik.
Die Heuristik hilft abzuschätzen, welche Knotenpfade (wahrscheinlich) schneller zum Ziel führen, und welche Teilpfade selbst bei besten Bedingungen nicht mehr zu Pfaden erweitert werden können, die besser als bereits gefundene Teilpfade sind (Branch and Bound).
A^* implementiert also eine durch die Heuristik informierte Graphsuche.

1.1. (Tut) Kürzeste Wege mit (A^*) - Intuition

Angenommen, wir möchten ein Navigationssystem programmieren, dass den schnellsten Weg über das Autobahnnetz von Berlin nach Köln findet. Wir modellieren jede Kreuzung als Knoten und jedes Verbindungsstück als Kante. Als Kantengewicht nehmen wir die tatsächliche Länge der Straße:



Wie löst der Dijkstra-Algorithmus das Problem?

Gibt es etwas besseres als Dijkstra's Algorithmus? Welche Information kann man für die Suche ausnutzen?

1.2. Implementierung der Manhattan-Distanz Heuristik für A* (30 Punkte)

Implementiert in der Klasse `HeuristicManhattan` die Methoden:

- `estimateDistanceToGoal(CellNode n)` Schätze die Distanz eines Knotens zum Ziel.
- `compare(CellNode a, CellNode b)`. Implementiere einen Vergleichsoperator, der zwei Knoten aufgrund ihrer geschätzten Pfadlängen vergleicht.

Dank der Methode `compare` implementiert die Klasse das `Comparator` interface. Dadurch kann sie dazu verwendet werden, Elemente einer `PriorityQueue` nach der Heuristik zu sortieren. Seht euch dazu die Dokumentation des Konstruktors `public PriorityQueue(int initialCapacity, Comparator<? super E> comparator)` [1] an!

Verwende als Heuristik die so genannte "Manhattan-Distanz" (L1 Norm):

$$d(n, ziel) = |x_n - x_{ziel}| + |y_n - y_{ziel}| \tag{1}$$

Wobei die Koordinaten den Zeilen und Spalten im Grid entsprechen.

1.3. Implementierung der Euklidische-Distanz Heuristik für A* (30 Punkte)

Implementiert analog zu `HeuristicManhattan` die Klasse `HeuristicEuclidean`.

Verwendet allerdings dabei anstatt der Manhattan-Distanz die Euklidische Distanz (L2 Norm) als Heuristik:

$$d(n, ziel) = \sqrt{(x_n - x_{ziel})^2 + (y_n - y_{ziel})^2} \tag{2}$$

[1] https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html#PriorityQueue(int,java.

Die Koordinaten entsprechen den Zeilen- und Spaltenindizes im Grid.

Überlegt euch warum in unserer Welt die euklidische Distanz (L2 Norm) oft eine bessere Heuristik ist. Welche Heuristiken steht uns auf Kugeloberflächen zur Verfügung?

1.4. Implementierung Kürzeste Wege mit A^* (40 Punkte)

Implementiert in der Klasse `DiGraph` die Methode `populateAStar(Node startNode)`, die den kürzesten Weg von Startknoten zum Zielknoten berechnet. Nach dem Aufruf sollen alle besuchten Knoten grau oder schwarz gefärbt sein, und unbesuchte Knoten weiss.

Implementiert ebenfalls die Methode `getShortestPathAStar(Node startNode, Node ta` die `populateAStar(Node startNode)` aufruft und den kürzesten Pfad als Knotenliste zurückgibt.

Nach dem Aufruf des Algorithmus sollen unbesuchte Knoten weiss gefärbt sein und besuchte Knoten grau oder schwarz.

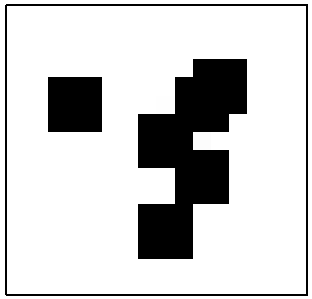
Hinweis: A^* ist sehr sehr ähnlich zum Dijkstra-Algorithmus und ihr dürft euren Dijkstra-Code aus der vorhergehenden Übung oder der `DiGraph` Klasse wiederverwenden.

Hinweis: Wir stellen euch eine Implementierung für die Pfadsuche in einem 2d-Labyrinth zur Verfügung. Die Welt is dabei als $m \times n$-Raster dargestellt und jede Zelle im Raster kann entweder frei oder blockiert sein. In diese Welt kann man sich nach oben, unten, links, oder rechts bewegen, jeweils mit Kosten 1 pro Schritt.

Ein derartiges Raster wird auch “Occupancy grid” genannt.

Aus diesem Raster erstellen wir einen Graphen, in dem jede freie Zelle ein Knoten ist und jede erlaubte Bewegung (nach oben, nach unten, nach links, nach rechts) mit einer Kante dargestellt wird.

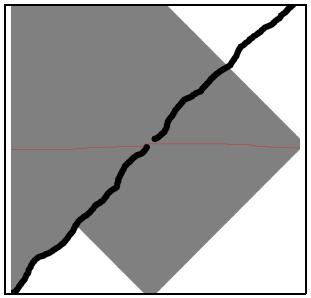
Um den Graphen bequem zu spezifizieren, verwenden wir Bilder. Weiße Pixel stellen freie Zellen dar, alle anderen Farbwerte stellen besetzte Zellen dar. Hier ein Beispiel:



Ihr könnt diese Bilder in einem Bildbearbeitungprogramm editieren um einfach komplexe Testfälle zu generieren.

Für die Visualisierung eurer Ergebnisse steht euch die Methode `toPicture(List<Node> path)` der Klasse `GridGraph` zur Verfügung!. Diese wandelt euren Graphen in ein Bild um. Die Pixelfarben haben folgende Bedeutung:

- Für Schwarze Pixel existiert kein entsprechender Knoten im Graph
- Ist ein Knoten weiss gefärbt, ist das entsprechende Pixel ebenfalls weiss, ansonsten grau (soll besuchte und unbesuchte Knoten kennzeichnen)
- Für alle Knoten in der übergebenen Knotenliste `path` werden die korrespondierenden Pixel rot gefärbt



Vergleicht A* mit beiden von euch implementierten Heuristiken (`HeuristicManhattan` und `HeuristicEuclidean`). Wählt eine der beiden Heuristiken für die Abgabe aus. Welche bevorzugt ihr und warum? Schreibt euer Argument als Kommentar in der Methode `getShortestPathAStar()`.