Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

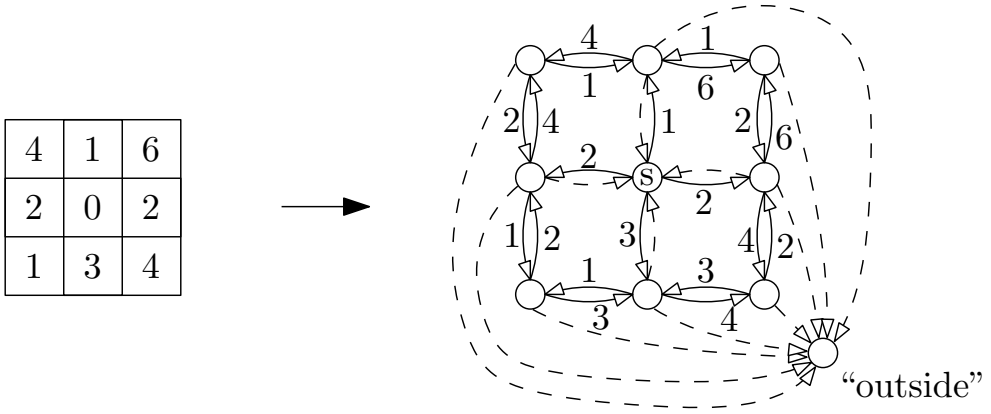
Departement Informatik
Markus Püschel
Peter Widmayer
Thomas Tschager
Tobias Pröger
Tomáš Gavenčiak

22. Dezember 2016

Algorithmen & Datenstrukturen Lösungen zu Blatt P13 HS 16

Lösung P13.1 *Dschungel.*

Die Aufgabe kann aufgefasst werden als die Berechnung eines Kürzesten Weges in einem *Gittergraphen* von einer gegebenen Startposition zu einem speziellen Knoten, dem sog. "äusseren Knoten". Der neue Graph hat Kanten in beide Richtungen, und die Kosten einer Kante $(x, y) \to (x', y')$ entsprechen den Kosten, um (x', y') zu betreten. Die Abbildung unten zeigt ein Beispiel, gestrichelte Kanten haben Kosten 0.



Dijkstras Algorithmus ist eine ideale Lösung für dieses Problem, und er kann leicht auf diesem Graphen implementiert werden. Es ist nicht einmal notwendig, den Graphen explizit zu konstruieren – wir können jeden Knoten durch seine Koordinaten (x, y) repräsentieren, und wenn die Nachbarknoten besucht werden müssen, wissen wir dass diese genau $(x + 1, y)$, $(x - 1, y)$, $(x, y + 1)$ und $(x, y - 1)$ sind (einige dieser Knoten könnten der "äussere Knoten" sein, falls (x, y) am Rand des Dschungels liegt). Ausserdem stoppen wir die Berechnung sofort, sobald der "äussere Knoten" erreicht wird, denn dann wurde ein kürzester Pfad bereits berechnet.

Dijkstras Algorithmus ist auf dieser Art von Graph nicht schwierig zu implementieren, aber wir benötigen eine Datenstruktur für einen Min-Heap, um die zu besuchenden Knoten zu verwalten. Wir können einen Heap selbst implementieren oder aber die durch die Java-Bibliothek zur Verfügung gestellte Struktur `java.util.PriorityQueue` benutzen. Diese garantiert, dass die Operationen `add()` und `remove()` (angewendet auf das Minimum) in Zeit $\mathcal{O}(\log n)$ ausgeführt werden, nicht aber die Entfernung eines beliebigen Elements, da die Gewichte der Elemente im Heap nicht verändert werden können. Wir können das Problem lösen, indem wir jeden Besuch eines jeden Knotens in den Heap aufnehmen, was dazu führt dass ein einzelner Knoten u.U. mehrfach im Heap vorhanden ist. Wenn wir ein minimales Element aus dem Heap entfernen, prüfen wir ob es bereits zuvor entfernt wurde, und falls ja, ignorieren wir es.

Dies ändert nicht die Komplexität des Algorithmus, da die Heap-Grösse höchstens m beträgt, $\log m = \mathcal{O}(\log n)$ gilt und jeder Knoten höchstens einmal pro gerichteter Kante auftritt. Wir müssen auch eine eigene Klasse `SquareInHeap` implementieren, um das Auftreten eines Knotens im Heap zu repräsentieren, und einen Comparator für diese Klasse implementieren, um sie in der `PriorityQueue` zu verwenden. Eine eigene Heap-Implementierung kann viele der zuvorgenannten Probleme vermeiden, allerdings erhöht dies natürlich die Grösse des Codes.

Lösungen Auf der Vorlesungswebseite finden Sie eine Lösung, welche weitere Kommentare zur Implementierung enthält.

Daten Analog zu den Fällen in `test*` enthalten die Testdaten auf dem Judge Fälle in denen die Startposition am Rand des Dschungels lagen, grosse zufällige Quadrate und verschiedene Dschungel mit einer teuren "Mauer" um den Dschungel. Diese Mauer stellt sicher, dass zunächst der gesamte, "kostengünstige" innere Dschungel untersucht wird, bevor eine teure Lösung gefunden wird, die die Mauer kreuzt.