7. Praktikum "Algorithmen und Programmierung II"

SS 2014

Vorbemerkungen. Auf der Web-Seite `http://www.gm.fh-koeln.de/ehses/ap` finden Sie alle nötigen Hilfsdateien.

Bei den Aufgaben geht es darum, fehlende Klassen zu schreiben oder zu ergänzen. Diese Klassen sind Teil eines kleinen Programms zur Demonstration unterschiedlicher Graph-Suchverfahren. **Sie brauchen das Beispielprogramm zur Lösung der Aufgabe nicht im Detail zu verstehen!**

Die unterschiedlichen Suchstrategien verwenden alle eine Datenstruktur, in der die zu untersuchenden Wege vorgemerkt sind. Sie unterscheiden sich dadurch, nach welchen Kriterien der nächste Weg ausgewählt wird. Die Algorithmen sind.:

1. **Zufallssuche**. Der Algorithmus wählt aus einer Folge von gespeicherten Knoten (Städten) zufällig einen aus, dessen Weg zum Ziel weiter untersucht wird. Das Verfahren liefert eine zufällige, aber meist gar nicht mal so schlechte Lösung.
2. **Tiefensuche**. Der Algorithmus verwendet einen LIFO-Stack. Er entspricht dem Verfahren der rekursiven Baumtraversierung. Häufig wird anstelle der hier verwendeten Implementierung das rekursive Backtracking-Verfahren verwendet, das weniger Speicher benötigt.
3. **Breitensuche**. Der Algorithmus verwendet eine FIFO-Queue. Er entspricht der ebenenweisen Baumtraversierung. Es findet den Weg mit minimaler Anzahl von Kanten.
4. **Algorithmen mit Prioritätswarteschlange**. Hier wird der als nächstes zu untersuchende Graphknoten aufgrund eines Ordnungskriteriums ausgewählt. Das Ordnungskriterium wird durch ein Comparator-Objekt ausgedrückt. Unterschiedliche Reihenfolgen ergeben unterschiedliche Algorithmen. In dem Beispiel sind die folgenden drei Verfahren enthalten:
    a. **Dijkstra's Algorithmus**. Unter den vorgemerkten Wegen wird derjenige mit der kürzesten geometrischen Weglänge ausgesucht.
    b. **Bergsteigen.** Unter den vorgemerkten Wegen wird derjenige mit der kürzesten Luftlinienentfernung zum Ziel ausgesucht.
    c. **A-Star.** Es wird der Weg ausgesucht, bei dem die 1,27Summe von bisheriger Weglänge und verbleibender Luftlinienentfernung minimal ist.

    Die Verfahren a) und c) liefern garantiert den kürzesten Weg; sie heißen auch A*-Algorithmen. Bei dem Beispiel-Problem ist c) der optimale A*-Algorithmus. Der Algorithmus b) ist das schnellste Verfahren, liefert aber nicht immer den kürzesten Weg.

Setzen Sie sich neben der Lösung der eigentlichen Aufgaben auch etwas mit dem Eigenschaften dieser Algorithmen auseinander. Sie können den Algorithmus, Start- und Zielort und die Verzögerungszeit für die Ablaufverfolgung einstellen. Wenn die Verzögerungszeit größer als 0 ist, sind in der Darstellung in grüner Farbe auch Wege markiert, die erfolglos versucht wurden (bei kurzen Verzögerungen sind evtl. nicht alle grünen Wege gezeichnet).

In der vorgegebenen Fassung ist zunächst nur die Zufallssuche korrekt implementiert. Bei der Auswahl eines der anderen Verfahren erfolgt eine Fehlermeldung (Tiefensuche, Breitensuche) oder ein Ergebnis das nicht dem angegebenen Algorithmus entspricht (Dijkstra, Bergsteigen, A-Star)..

Testen Sie Aufgabe 1 und Aufgabe 2 zunächst mit JUnit!

Aufgabe 1) Schreiben Sie die folgenden Klassen, die die Schnittstelle `util.IQueue` implementieren.

1. Die Klasse `search.util.LIFOQueue` soll einen Stack nach dem LIFO-Prinzip realisieren. Implementieren Sie die Klasse mittels einer einfach verketteten Liste. Grundstrukturen sind schon vorbereitet.
2. Die Klasse `search.util.FIFOQueue` soll eine (normale) Queue nach dem FIFO-Prinzip realisieren. Benutzen Sie für die Implentierung die Klasse `java.util.LinkedList`. Eventuel benötigen Sie die Methoden `addFirst`, `addLast`, `removeFirst`, `removeLast`, `clear` und `size`. Hinweis: Für den Test müssen Sie in `FIFOQueueTest` einen Kommentar entfernen!

Sie müssen ansonsten nur wissen, dass LIFO = *last in first out* ist und FIFO = *first in first out*.Integrieren Sie die neuen Klassen in das Suchprogramm, indem Sie die Klasse `search.graph.SearchStrategy` geeignet abändern (Kommentare entfernen!).

Beachten Sie auch die Kommentare in dem Interface `IQueue`!

Aufgabe 2) Vervollständigen Sie die Klasse `search.util.PriorityQueue.` Die Daten sollen so in dem Array `data` gespeichert sein, dass die Heap-Bedingung erfüllt ist. Diese wird nach dem Einfügen eines Elements durch die Methode `swim` hergestellt. Wenn dies korrekt geschieht, steht in `data[0]` immer das kleines Element, d.h. `cmp.compare(data[0], data[i]) <= 0` für alle gültigen Indizes i.

Bei `get` wird daher `data[0]` zurückgegeben. Der frei werdende Platz wird durch das bisher letzte Element `(data[size-1])` gefüllt. Um die Heap-Bedingung wieder herzustellen, wird die Methode `sink` aufgerufen.

Sie sollen `swim` und `sink` vervollständigen. Für die Implementierung sollen Sie den auf *folie6.pdf* dargestellten Heap-Algorithmus verwenden (in den Folien ist er als Bestandteil eines Sortieralgorithmus bespochen). Zusätzlich zu den Folien finden sich im Quelltext weitere Hinweise. Verwenden Sie die Hilfsfunktionen am Ende der Klasse (`precedes` ...)

Anmerkung: Es gibt auch andere effiziente Algorithmen für Prioritätswarteschlangen.

Lernziele: Algorithmen