

Name: _____

Vorname: _____

Matrikelnummer: _____ Testat: _____

6. Praktikum "Algorithmen und Programmierung II"

SS 2011

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 (verkettete Listen und Bäume) zu schreiben. 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 Summe 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 in der Regel nicht den kürzesten Weg.

Setzen Sie sich neben der Lösung der eigentlichen Aufgaben auch etwas mit dem Suchverhalten 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 implementiert. Bei der Auswahl eines der anderen Verfahren erfolgt eine Fehlermeldung.

Aufgabe 1) Schreiben Sie alle noch fehlenden Klassen zur Implementierung der Schnittstelle `search.util.IQueue`. Sie sollen dabei jeweils eine geeignete Implementierung nach dem Prinzip der verketteten Liste vornehmen. **Die Listenimplementierung soll jeweils von Grund auf geschrieben werden (keine fertigen Listenklassen benutzen)!**

1. Die Klasse `search.util.LIFOQueue` soll einen Stack nach dem LIFO-Prinzip realisieren.
2. Die Klasse `search.util.FIFOQueue` soll eine (normale) Queue nach dem FIFO-Prinzip realisieren.

3. Die Klasse `search.util.PriorityQueue` soll so aufgebaut sein, dass nach jeder `put`-Operation die Reihenfolge der Daten mit dem durch den Konstruktor angegebenen `Comparator`-Objekt verträglich ist. Wenn `cmp` das `Comparator`-Objekt, `p.obj` ein beliebiges Datenelement ist und `p.next.obj` das nachfolgende Element ist, muss also gelten: `cmp.compare(p.obj, p.next.obj) <= 0`.

Testen Sie die drei Klassen mit den im Paket `search.util.test` enthaltenen JUnit-Testprogrammen (Kommentare entfernen!).

Lernziele: Datenstrukturen, Algorithmenentwicklung.

Aufgabe 2) Integrieren Sie die drei neuen Klassen in das Suchprogramm, indem Sie die Klasse `search.graph.SearchStrategy` geeignet abändern (Kommentare entfernen!).

Lernziele: Zurechtfinden in fremder Software.

Aufgabe 3) Vervollständigen Sie die Methoden der Klasse `search.util.GenericTreeAlgorithms`. Die Methode `numberOfNodes` wird mit der Referenz auf die Wurzel eines Baums aufgerufen (oder `null`). Der Rückgabewert ist gleich der Anzahl der Knoten. Testen Sie die Klasse mit dem entsprechenden JUnit-Test.

Eine weitere Aufgabe besteht darin, die Methode `GenericTreeAlgorithms.getPathToGoal` fertig zu programmieren, so dass am Ende der gefundene Weg zum Ziel als Liste von Strings zurückgegeben wird. Zunächst wird der Baum durchlaufen um den Zielknoten zu finden. Auf dem „Rückweg“ der Rekursion kann dann die Liste der Knotennamen aufgebaut werden. Den eigentlichen Pfadknoten erhält man aus dem Baumknoten mittels `value`. Den Knotennamen bekommt man daraus durch `toString`.

Die Anzahl der Knoten wird in der GUI unter „besuchte Orte“ dargestellt. Da damit die Anzahl aller näher betrachteten Orte im Suchbaum ist, ist die Zahl ein Maß für den Suchaufwand. Die Orte im gefundenen Weg sind in „Weg“ aufgelistet..

Hinweis: Durch Ändern des letzten Konstruktorparameters von `DataPanel` in der Klasse `search.gui.Main` lässt sich erreichen, dass der gefundene Suchbaum komplett ausgedruckt wird. Das kann eine Hilfe bei der Fehlersuche sein.

Lernziel: Algorithmen auf Bäumen, Beschäftigung mit Suchverfahren..

Aufgabe 4) Die Klasse `search.util.RawTreeAlgorithms` implementiert die gleichen Algorithmen wie die Klasse `GenericTreeAlgorithms` der Aufgabe 3. Der Unterschied besteht darin, dass in `RawTreeAlgorithms` keine Typparameter benutzt werden sollen. Füllen Sie also auch hier die fehlenden Implementierungen aus und testen Sie dies mit dem JUnit-Test. Sie können in der Klasse `search.gui.Main` einstellen (wo?), welche Baumalgorithmen Sie dann in der Suchanwendung verwenden.