SWE4.4: Ue04_Heseltine_Jack

Inhalt

Erfüllungsgrad

Nr	L	I	T	Summe	Multiplikator	Punkte
a	3/3	4/4	3/3	10	2	20
b	3/3	4/4	3/3	10	3	30
c	0/3	0/4	0/3	0	3	0
d	0/3	0/4	0/3	0	2	0
					Summe Erfüllungsgrad	50

Lösungsidee

a)

Für SearchNodes, Member-Variablen state und targetState mit dem Datentyp des entsprechenden Interfaces, predecessor ist vom Typ SearchNode, transition ist Transition.

Getter und Setter werden wie zu erwarten angelegt, costsFromStart arbeitet mit Transition Interface: das Double, das costs() (auf getTransition()) liefert, wird in einer Schleife zum ersten Knoten zurück aufsummiert.

Offen bleibt für mich die Implementierung von setCostsFromStart(), weil Transition keinen Setter anbietet.

b)

Die-Solver-Lösungsidee folgt dem in der Angabe verlinkten Wikipedia-Artikel. Dabei wird die openList abgearbeitet, bis entweder eine Lösung gefunden wurde (targetState), oder bis die Liste leer ist, dann wird NoSolutionException geworfen.

closedList (eigentlich Set) dient dabei der Markierung der bereits überprüften SearchNodes.

Die berechneten Kosten werden immer mit tentative_g, einer Vorberechnung für die Nachbarknoten, verglichen:

Implementierung

a)

```
package swe4.astar;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

// SearchNode ist eine Hilfsklasse, die zur Implementierung des A*-Algorithmus
// benötigt wird. Damit kann man den Weg von einem SearchNode zum Startknoten
// zurückverfolgen, da dieser mit seinem Vorgängerknoten verkettet ist. Ein
// SearchNode kennt die Kosten vom Startknoten bis zu ihm selbst. SearchNode
// kann auch eine Schätzung für den Weg zum Zielknoten berechnen.
public class SearchNode implements Comparable<SearchNode> {


  private State state, targetState;
  private SearchNode predecessor;
  private Transition transition;
  private Double costs;


  // Suchknoten mit dem Zustand state und dem Zielzustand target initialisieren.
  // Es existiert kein Vorgängerknoten.
  public SearchNode(State state, State targetState) {
    this.state = state;
    this.targetState = targetState;
    this.costs = 0.0;
  }

  // Suchknoten mit dem Zustand state und dem Zielzustand target initialisieren.
  // predecessor ist der Vorgängerknoten, von dessen Zustand man mit transition
  // zum Zustand dieses Knotens kommt.
  public SearchNode(State state, State targetState, SearchNode predecessor, Transition
transition) {
    this.state = state;
    this.targetState = targetState;
    this.predecessor = predecessor;
    this.transition = transition;
    this.costs = predecessor.costsFromStart() + transition.costs();
  }

  // Gibt den Zustand dieses Knotens zurück.
  public State getState() {
    return this.state;
  }

  // Gibt den Vorgängerknoten zurück.
  public SearchNode getPredecessor() {
    return this.predecessor;
  }
```

```
// Setzt den Vorgängerknoten.
public SearchNode setPredecessor(SearchNode predecessor) {
  this.predecessor = predecessor;
  return this.predecessor;
}

// Gibt Transition zurück, die Vorgängerknoten in diesen Knoten überführt.
public Transition getTransition() {
  return this.transition;
}

// Setzt Transition, die Vorgängerknoten in diesen Knoten überführt.
public void setTransition(Transition transition) {
  this.transition = transition;
}

// Gibt Kosten vom Startknoten bis zu diesem Knoten zurück.
public double costsFromStart() {
  /*double sum = 0;
  if (this.transition != null) {
    // e.g. first SN has no transition
    sum += this.transition.costs();
  }
  SearchNode currentSN = this;
  // go back to first searchNode, summing the costs backwards
  while (currentSN.getPredecessor() != null) {
    currentSN = currentSN.getPredecessor();
    if (currentSN.getTransition() != null) {
      // e.g. first SN has no transition
      sum += currentSN.getTransition().costs();
    }
  }
  return sum;*/
  return this.costs;
}

// Setzt die Kosten vom Startknoten bis zu diesem Knoten.
public void setCostsFromStart(double costsFromStart) {
  /*SearchNode currentSN = this;
  // go back to first searchNode, setting the costs to new value
  while (this.predecessor != null) {
    currentSN = this.predecessor;
    // TODO
  }*/
  this.costs = costsFromStart;
}

// Gibt geschätzte Kosten von diesem Knoten bis Zum Zielknoten zurück.
public double estimatedCostsToTarget() {
  return this.state.estimatedCostsToTarget(this.targetState);
```

```
  }

  // Gibt Schätzung der Kosten vom Startknoten über diesen Knoten bis zum
  // Zielknoten zurück (= Kosten bis zu diesem Knoten + geschätzte Kosten
  // bis zum Zielknoten).
  public double estimatedTotalCosts() {
    return this.costsFromStart() + this.estimatedCostsToTarget();
  }

  // Gibt zurück, ob dieser Knoten und der Knoten other denselben Zustand
  // repräsentieren.
  // Vorsicht: Enthaltenen Zustand per Wert, vergleichen, nicht die Referenzen.
  // Muss konsistent mit compareTo implementiert werden:
  // a.equals(b) <==> a.comparesTo(b) == 0.
  @Override
  public boolean equals(Object obj) {
    return this.hashCode() == obj.hashCode();
  }

  // Notwendig für Verwendung in Hash-basierten Behältern. Muss konsistent
  // mit equals implementiert werden: a.equals(b) => a.hashCode() == b.hashCode.
  @Override
  public int hashCode() {
    return this.state.hashCode();
  }

  // Vergleicht zwei Knoten auf Basis der geschätzten Gesamtkosten.
  // <0: Kosten dieses Knotens sind kleiner oder gleich als die Kosten von other.
  //  0: Dieser Knoten und other sind gleich.
  // >0: Kosten dieses Knotens sind höher als Kosten von other.
  @Override
  public int compareTo(SearchNode other) {
    //return (int) (this.transition.costs() - other.getTransition().costs());
    return (int) (this.costs - other.costsFromStart());
  }

  // Konvertiert die Knotenliste, die bei diesem Knoten ihren Ausgang hat,
  // in eine Liste von Transitionen. Da der Weg in umgekehrter Reihenfolge
  // gespeichert ist, muss die Transitionsliste invertiert werden.
  public List<? extends Transition> getTransitionsFromStart() {
    /*List<Transition> list = new ArrayList<>();
    list.add(this.transition); // current
    SearchNode currentSN = this;
    while (currentSN.getPredecessor() != null) { // previous
      currentSN = this.predecessor;
      list.add(currentSN.getTransition());
    }

    Collections.reverse(list);
    return list;*/
```

```
    // with recursion
    List<Transition> list = new ArrayList<>();
    if(predecessor != null) {
      for(Transition edge : predecessor.getTransitionsFromStart()) {
        list.add(edge);
      }
    }
    if(transition != null) {
      list.add(transition);
    }
    return list;
  }

  @Override
  public String toString() {
    System.out.println("..");
    return null;
  }
}
```

b)

```
package swe4.astar;

import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Set;

public class AStarSolver {

  private PriorityQueue<SearchNode> openList = new PriorityQueue<>();
  private Set<SearchNode> closedList = new HashSet<>(); // cameFrom

  // Berechnet eine kostenoptimale Folge von Transitionen, welche den gegebenen
  // Anfangszustand initialState in den Zielzustand targetState überführt.
  // Wirft NoSolutionException (Checked Exception), falls es keine derartige
  // Folge von Transitionen gibt.
  public List<? extends Transition> solve(State initialState, State targetState)
throws NoSolutionException {
    openList.add(new SearchNode(initialState, targetState));
    SearchNode currNode;

    while (!openList.isEmpty()) {
      currNode = openList.poll();

      if (currNode.getState().equals(targetState)) {
        return currNode.getTransitionsFromStart();
      }
```

```
      closedList.add(currNode);

      for (Transition transition : currNode.getState().transitions()) {
        State n = currNode.getState().apply(transition);
        SearchNode successor = new SearchNode(n, targetState, currNode, transition);

        if (closedList.contains(successor)) { // if already in closed list: nothing to
do
          continue;       // closedList keeps the nodes already checked
        }

        double tentative_g = currNode.costsFromStart(); // g value for new path

        if (tentative_g < successor.costsFromStart()) {
          successor.setPredecessor(currNode); // connect nodes

          if(openList.contains(successor)) { // recalibrate
            openList.remove(successor);
            successor.setCostsFromStart(tentative_g);
          }

          openList.add(successor);
        }

      }
    }

    throw new NoSolutionException();
  }

}
```

Testung

a)

V.a. compareTo() wurde erweitert, um alle Fälle abzudecken.

```
package swe4.astar.test;

import org.junit.jupiter.api.Test;
import swe4.astar.MathUtil;
import swe4.astar.SearchNode;
import swe4.astar.State;
import swe4.astar.Transition;

import java.util.ArrayList;
import java.util.List;

import static org.junit.jupiter.api.Assertions.*;
```

```
public class SearchNodeTest {


  @Test
  public void oneParamConstructorInitializesCorrectly() {
    Vertex     v  = new Vertex(1, 10.0);
    SearchNode sn = new SearchNode(v, v);
    assertNotNull(sn.getState());
    assertNull(sn.getPredecessor());
    assertNull(sn.getTransition());
  }

  @Test
  public void threeParamConstructorInitializesCorrectly() {
    Vertex v1  = new Vertex(1, 10.0);
    Vertex v2  = new Vertex(2, 10.0);
    Edge   e12 = new Edge(v2, 15.0);
    v1.addEdge(e12);

    SearchNode sn1 = new SearchNode(v1, v2);
    SearchNode sn2 = new SearchNode(v2, v2, sn1, e12);

    assertSame(v2, sn2.getState());
    assertSame(sn1, sn2.getPredecessor());
    assertSame(e12, sn2.getTransition());
  }

  @Test
  public void constructorInitializesCostsCorrectly() {
    Vertex v1  = new Vertex(1, 10.0);
    Vertex v2  = new Vertex(2, 10.0);
    Edge   e12 = new Edge(v2, 15.0);
    v1.addEdge(e12);

    SearchNode sn1 = new SearchNode(v1, v2);
    SearchNode sn2 = new SearchNode(v2, v2, sn1, e12);

    assertTrue(MathUtil
                  .isDoubleEqual(v2.estimatedCostsToTarget(v2), sn2
.estimatedCostsToTarget()));
    assertTrue(MathUtil
                  .isDoubleEqual(sn1.costsFromStart() + e12.costs(), sn2.
costsFromStart()));
  }

  @Test
  public void costsCalculatedCorrectly() {
    Vertex v1  = new Vertex(1, 10.0);
    Vertex v2  = new Vertex(2, 10.0);
    Edge   e12 = new Edge(v2, 15.0);
```

```
    v1.addEdge(e12);

    SearchNode sn1 = new SearchNode(v1, v2);
    SearchNode sn2 = new SearchNode(v2, v2, sn1, e12);
    //System.out.println("sn2.costsFromStart() " + sn2.costsFromStart());
    //System.out.println("sn2.estimatedCostsToTarget() " +
sn2.estimatedCostsToTarget());
    //System.out.println("sn2.estimatedTotalCosts() " + sn2.estimatedTotalCosts());
    assertTrue(MathUtil.isDoubleEqual(sn2.costsFromStart() + sn2
.estimatedCostsToTarget(),
                                      sn2.estimatedTotalCosts()));
  }

  @Test
  public void compareSearchNodes() {
    Vertex v1  = new Vertex(1, 20.0);
    Vertex v2  = new Vertex(2, 10.0);
    Vertex v3  = new Vertex(3, 10.0);
    Edge   e12 = new Edge(v2, 10.0);
    Edge   e13 = new Edge(v3, 15.0);
    v1.addEdge(e12);
    v1.addEdge(e13);

    SearchNode sn1 = new SearchNode(v1, v3);
    SearchNode sn2 = new SearchNode(v2, v3, sn1, e12);
    SearchNode sn3 = new SearchNode(v3, v3, sn1, e13);

    assertTrue(sn2.compareTo(sn3) < 0);

    assertTrue(sn2.compareTo(sn2) == 0);
    assertTrue(sn3.compareTo(sn2) > 0);
  }

  @Test
  public void hashCodeConsistent() {
    Vertex v1 = new Vertex(1, 10.0);
    Vertex v2 = new Vertex(2, 0.0);

    SearchNode sn1 = new SearchNode(v1, v2);
    SearchNode sn2 = new SearchNode(v1, v2);

    assertTrue(sn1.hashCode() == sn2.hashCode());
  }

  @Test
  public void equalsConsistent() {
    Vertex v1 = new Vertex(1, 10.0);
    Vertex v2 = new Vertex(2, 0.0);

    SearchNode sn1 = new SearchNode(v1, v2);
    SearchNode sn2 = new SearchNode(v1, v2);
```

```
    assertTrue(sn1.equals(sn2));
  }
}
```

b)

Erweiterungen:

- Ich passe die quadrangle()-Tests an, einmal mit weniger möglichen Lösungen
- und einmal mit gar keinen Lösungen

Sonst finde ich die Testabdeckung gut, d.h. m.E. kann man nur noch die Komplexität steigern. Anmerkung: ich komme nicht darauf, warum triangleOneSideShorter() nicht durchläuft.

```
package swe4.astar.test;

import org.junit.jupiter.api.Test;
import swe4.astar.AStarSolver;
import swe4.astar.NoSolutionException;
import swe4.astar.Transition;

import java.util.List;

import static org.junit.jupiter.api.Assertions.*;

public class AStarSolverTest {

  @Test
  void singleStateSolutionExists() {
    AStarSolver solver     = new AStarSolver();
    Vertex      finalState = new Vertex(1, 0.0);
    try {
      List<? extends Transition> transitions = solver.solve(finalState, finalState);
      assertTrue(transitions.isEmpty());
    }
    catch (NoSolutionException e) {
      fail("NoSolutionException not expected");
    }
  }

  @Test
  void oneTransition() {
    AStarSolver solver  = new AStarSolver();
    Vertex      initial = new Vertex(1, 1.0);
    Vertex      target  = new Vertex(2, 0.0);
    initial.addEdge(new Edge(target, 1.0));

    try {
```

```
      List<? extends Transition> transitions = solver.solve(initial, target);
      assertEquals(1, transitions.size());
      assertEquals(target, initial.apply(transitions.get(0)));
    }
    catch (NoSolutionException e) {
      fail("NoSolutionException not expected");
    }
  }

  @Test
  void twoIsolatedNodesNoSolution() {
    AStarSolver solver = new AStarSolver();

    Vertex initial = new Vertex(1, 1.0);
    Vertex target  = new Vertex(2, 0.0);

    assertThrows(NoSolutionException.class, () -> solver.solve(initial, target));
  }

  @Test
  void twoNodesTwoTransition() {
    AStarSolver solver = new AStarSolver();

    Vertex initial = new Vertex(1, 1.0);
    Vertex target  = new Vertex(2, 0.0);
    Edge   e1      = new Edge(target, 3.0);
    Edge   e2      = new Edge(target, 2.0);
    initial.addEdge(e1);
    initial.addEdge(e2);

    try {
      List<? extends Transition> transitions = solver.solve(initial, target);
      assertEquals(1, transitions.size());
      assertSame(e2, transitions.get(0));
      assertEquals(target, initial.apply(transitions.get(0)));
    }
    catch (NoSolutionException e) {
      fail("NoSolutionException not expected");
    }
  }

  @Test
  void triangleSumTwoSidesShorter() {
    AStarSolver solver = new AStarSolver();

    Vertex v1 = new Vertex(1, 8.0);
    Vertex v2 = new Vertex(2, 5);
    Vertex v3 = new Vertex(3, 0.0);

    Edge e13 = new Edge(v3, 10.0);
    v1.addEdge(e13);
```

```
    Edge e12 = new Edge(v2, 3.0);
    v1.addEdge(e12);
    Edge e23 = new Edge(v3, 4.0);
    v2.addEdge(e23);

    try {
      List<? extends Transition> transitions = solver.solve(v1, v3);
      assertEquals(2, transitions.size());
      assertSame(e12, transitions.get(0));
      assertSame(e23, transitions.get(1));
    }
    catch (NoSolutionException e) {
      fail("NoSolutionException not expected");
    }
  }

  @Test
  void triangleOneSideShorter() {
    AStarSolver solver = new AStarSolver();

    Vertex v1 = new Vertex(1, 8.0);
    Vertex v2 = new Vertex(2, 5);
    Vertex v3 = new Vertex(3, 0.0);

    Edge e13 = new Edge(v3, 10.0);
    v1.addEdge(e13);
    Edge e12 = new Edge(v2, 6.0);
    v1.addEdge(e12);
    Edge e23 = new Edge(v3, 7.0);
    v2.addEdge(e23);

    try {
      List<? extends Transition> transitions = solver.solve(v1, v3);
      assertEquals(1, transitions.size());
      assertSame(e13, transitions.get(0));
    }
    catch (NoSolutionException e) {
      fail("NoSolutionException not expected");
    }
  }

  @Test
  void quadrangle() {
    AStarSolver solver = new AStarSolver();

    Vertex v1 = new Vertex(1, 8.0);
    Vertex v2 = new Vertex(2, 4.0);
    Vertex v3 = new Vertex(3, 5.0);
    Vertex v4 = new Vertex(4, 0.0);

    Edge e12 = new Edge(v2, 6.0);
```

```
    v1.addEdge(e12);
    Edge e13 = new Edge(v3, 4.0);
    v1.addEdge(e13);
    Edge e32 = new Edge(v2, 1.0);
    v3.addEdge(e32);
    Edge e24 = new Edge(v4, 5.0);
    v2.addEdge(e24);
    Edge e34 = new Edge(v4, 8.0);
    v3.addEdge(e34);

    try {
      List<? extends Transition> transitions = solver.solve(v1, v4);
      assertEquals(3, transitions.size());
      assertSame(e13, transitions.get(0));
      assertSame(e32, transitions.get(1));
      assertSame(e24, transitions.get(2));
    }
    catch (NoSolutionException e) {
      fail("NoSolutionException not expected");
    }
  }

  @Test
  void quadrangleFewerSolutions() {
    AStarSolver solver = new AStarSolver();

    Vertex v1 = new Vertex(1, 8.0);
    Vertex v2 = new Vertex(2, 4.0);
    Vertex v3 = new Vertex(3, 5.0);
    Vertex v4 = new Vertex(4, 0.0);

    Edge e12 = new Edge(v2, 6.0);
    v1.addEdge(e12);
    Edge e13 = new Edge(v3, 4.0);
    v1.addEdge(e13);
    Edge e32 = new Edge(v2, 1.0);
    v3.addEdge(e32);
    Edge e34 = new Edge(v4, 8.0);
    v3.addEdge(e34);

    try {
      List<? extends Transition> transitions = solver.solve(v1, v4);
      assertEquals(2, transitions.size());
      assertSame(e13, transitions.get(0));
      assertSame(e34, transitions.get(1));
    }
    catch (NoSolutionException e) {
      fail("NoSolutionException not expected");
    }
  }
```

```
  @Test
  void quadrangleNoSolutions() {
    AStarSolver solver = new AStarSolver();

    Vertex v1 = new Vertex(1, 8.0);
    Vertex v2 = new Vertex(2, 4.0);
    Vertex v3 = new Vertex(3, 5.0);
    Vertex v4 = new Vertex(4, 0.0);

    Edge e12 = new Edge(v2, 6.0);
    v1.addEdge(e12);
    Edge e13 = new Edge(v3, 4.0);
    v1.addEdge(e13);
    Edge e32 = new Edge(v2, 1.0);
    v3.addEdge(e32);

    assertThrows(NoSolutionException.class, () -> solver.solve(v1, v4));
  }
}
```