SWE 4x	**Übung zu Softwareentwicklung mit modernen Plattformen 4**	**SS 2015, Übung 4**

Abgabetermin: SA in der KW 20

Gr. 1, E. Pitzer
Gr. 2, F. Gruber-Leitner

Name Roman Lumetsberger **Aufwand in h** 7

Punkte _____ **Kurzzeichen Tutor / Übungsleiter** _____ / _____

Verschiebe-Puzzle – A*-Algorithmus (24 Punkte)

Ein sehr bekanntes und beliebtes Rätsel ist das Verschiebe-Puzzle, das oft auch als 8- bzw. 15-Puzzle bezeichnet wird. Das Spiel besteht aus 8 (15) Kacheln, die von 1 bis 8 (15) durchnummeriert sind, die auf einem 3x3-Spielfeld (4x4-Spielfeld) angeordnet sind. Da ein Feld frei bleibt, können gewisse Kacheln verschoben werden Die Aufgabe besteht nun darin, ausgehend von einer beliebigen Anordnung der Kacheln, diese ausschließlich durch Verschiebungen in die richtige Reihenfolge zu bringen (siehe nebenstehende Abbildung).

Eine Möglichkeit, dieses Problem zu lösen, ist Backtracking. Allerdings wird bei Anwendung dieses Verfahrens der Suchraum sehr groß, was zu nicht vertretbaren Rechenzeiten führt. Ein effizienter Algorithmus zur Lösung dieses Problems ist der sogenannte A*-Algorithmus, der von Peter Hart, Nils Nilsson und Bertram Raphael bereits 1968 entwickelt wurde. Eine übersichtliche Darstellung des Algorithmus findet man beispielsweise auf der deutschen Wikipedia unter http://de.wikipedia.org/wiki/A*-Algorithmus. Der A*-Algorithus wird oft zur Wegsuche bei Routenplanern eingesetzt. Er ist aber auch auf die hier angeführte Problemstellung anwendbar.

Die Grund A*-Algorithmus enumeriert grundsätzlich auch alle möglichen Lösungsvarianten, allerdings versucht er, zunächst den erfolgsversprechendsten Weg zum Ziel zu verfolgen. Erst dann werden weitere Varianten untersucht. Findet der Algorithmus auf diese Weise bereits frühzeitig eine Lösung, müssen viele Lösungsvarianten erst gar nicht evaluiert werden. Damit der Algorithmus beim Durchwandern des Lösungsraums in die erfolgsversprechendste Richtung weitergehen kann, benötigt er eine Abschätzung der Kosten, die auf dem verbleibenden Weg zum Ziel anfallen werden. In unserer Problemstellung kann für diese Kostenfunktion h(x) die Summe der Manhatten-Distanzen (= Distanz in x-Richtung + Distanz in y-Richtung) aller Kacheln zu ihrer Zielposition herangezogen werden. Wenn g(x) die Kosten von der Ausgangskonfiguration bis zur Konfiguration x bezeichnet, stellt f(x) = g(x) + h(x) eine Abschätzung der Kosten von der Ausgangs- zur Zielkonfiguration dar, wobei der Weg zum Ziel über x verläuft.

Implementieren Sie die Lösung in folgenden Schritten:

a) Gehen Sie bei der Implementierung testgetrieben vor. Implementieren Sie die nachfolgend angeführten Klassen Methode für Methode und geben Sie für jede Methode zumindest einen einfachen Testfall an. Erstellen Sie zunächst nur den Methodenrumpf mit einer Standardimplementierung, die nur syntaktisch korrekt sein muss. Implementieren Sie dann für diese Methode die Unittests, deren Ausführung zunächst fehlschlagen wird. Erweitern Sie anschließend die Implementierung der Methode so lange, bis alle Unittests durchlaufen. Erst wenn die Methoden-bezogenen Tests funktionieren, können Sie komplexere Tests erstellen.

   Eine Testsuite mit einigen Tests wird Ihnen auf der E-Learning-Plattform zur Verfügung gestellt. Erweitern Sie diese Testsuite so wie beschrieben. Ihre Implementierung muss die vorgegebenen und die von Ihnen hinzugefügten bestehen.

b) Implementieren Sie zunächst eine Klasse `Board`, die eine Board-Konfiguration repräsentieren kann und alle notwendigen Operationen auf einem Spielbrett unterstützt. `Board` soll folgende Schnittstelle aufweisen:

```
public class Board implements Comparable<Board> {
  // Board mit Zielkonfiguration initialisieren.
  public Board(int size);

  // Überprüfen, ob dieses Board und das Board other dieselbe Konfiguration aufweisen.
  public boolean equals(Object other);

  // <1, wenn dieses Board kleiner als other ist.
  // 0, wenn beide Boards gleich sind
  // >1, wenn dieses Board größer als other ist.
  public int compareTo(Board other);

  // Gibt die Nummer der Kachel an der Stelle (i,j) zurück, Indizes beginnen bei 1.
  // (1,1) ist somit die linke obere Ecke.
  // Wirft die Laufzeitausnahme InvalidBoardIndexException.
  public int getTile(int i, int j);

  // Setzt die Kachelnummer an der Stelle (i,j) zurück. Wirft die Laufzeitausnahmen
  //InvalidBoardIndexException und InvalidTileNumberException
  public void setTile(int i, int j, int number);

  // Setzt die Position der leeren Kachel auf (i,j)
  // Entsprechende Kachel wird auf 0 gesetzt.
  // Wirft InvalidBoardIndexException.
  public void setEmptyTile(int i, int j);

  // Zeilenindex der leeren Kachel
  public int getEmptyTileRow();

  // Gibt Spaltenindex der leeren Kachel zurück.
  public int getEmptyTileColumn();

  // Gibt Anzahl der Zeilen (= Anzahl der Spalten) des Boards zurück.
  public int size();

  // Überprüft, ob Position der Kacheln konsistent ist.
  public boolean isValid();

  // Macht eine tiefe Kopie des Boards.
  // Vorsicht: Referenztypen müssen neu allokiert und anschließend deren Inhalt kopiert werden.
  public Board copy();

  // Erzeugt eine zufällige lösbare Konfiguration des Boards, indem auf die bestehende
  // Konfiguration eine Reihe zufälliger Verschiebeoperationen angewandt wird.
  public void shuffle();

  // Verschiebt leere Kachel auf neue Position (row, col).
  // throws IllegalMoveException
  public void move(int row, int col);

  // Verschiebt leere Kachel nach links. Wirft Laufzeitausnahme IllegalMoveException.
  public void moveLeft();

  // Verschiebt leere Kachel nach rechts. Wirft IllegalMoveException.
  public void moveRight();

  // Verschiebt leere Kachel nach oben. Wirft IllegalMoveException.
  public void moveUp();
```

```
    // Verschiebt leere Kachel nach unten. Wirft IllegalMoveException.
    public void moveDown();

    // Führt eine Sequenz an Verschiebeoperationen durch. Wirft IllegalMoveException.
    public void makeMoves(List<Move> moves);
}
```

c) Zur Implementierung des A*-Algorithmus benötigt sie die Hilfsklasse `SearchNode`. Damit kann man den Weg von einem `SearchNode` zum Startknoten zurückverfolgen, da dieser mit seinem Vorgängerkonten verkettet ist. Ein `SearchNode` kennt die Kosten vom Startknoten bis zu ihm selbst. Ein `SearchNode` kann auch eine Schätzung für den Weg zum Zielknoten berechnen.

```
public class SearchNode implements Comparable<SearchNode> {

    // Suchknoten mit Board-Konfiguration initialisieren.
    public SearchNode(Board board);

    // Gibt Board-Konfiguration dieses Knotens zurück.
    public Board getBoard();

    // Gibt Referenz auf Vorgängerknoten zurück.
    public SearchNode getPredecessor();

    // Setzt den Verweis auf den Vorgängerknoten.
    public void setPredecessor(SearchNode predecessor);

    // Gibt Kosten (= Anzahl der Züge) vom Startknoten bis zu diesem Knoten zurück.
    public int costsFromStart():

    // Gibt geschätzte Kosten bis zum Zielknoten zurück.  Die Abschätzung
    // kann mit der Summe der Manhatten-Distanzen aller Kacheln erfolgen.
    public int estimatedCostsToTarget();

    // Setzt die Kosten vom Startknoten bis zu diesem Knoten.
    public void setCostsFromStart(int costsFromStart);

    // Gibt Schätzung der Wegkosten vom Startknoten über diesen Knoten bis zum Zielknoten zurück.
    public int estimatedTotalCosts();

    // Gibt zurück, ob dieser Knoten und der Knoten other dieselbe Board-Konfiguration darstellen.
    // Vorsicht: Knotenkonfiguration vergleichen, nicht die Referenzen.
    public boolean equals(Object other);

    // Vergleicht zwei Knoten auf Basis der geschätzten Gesamtkosten.
    // <1: Kosten dieses Knotens sind geringer als Kosten von other.
    //  0: Kosten dieses Knotens und other sind gleich.
    // >1: Kosten dieses Knotens sind höher als Kosten von other.
    public int compareTo(SearchNode other);

    // Konvertiert die Knotenliste, die bei diesem Knoten ihren Ausgang hat, in eine Liste von Zügen.
    // Da der Weg in umgekehrter Reihenfolge gespeichert ist, muss die Zugliste invertiert werden.
    public List<Move> toMoves();
}
```

d) Implementieren Sie schließlich den A*-Algorithmus in der Klasse `SlidingPuzzle`.

```
public class SlidingPuzzle {

   // Berechnet die Zugfolge, welche die gegebene Board-Konfiguration in die Ziel-Konfiguration
   // überführt. Wirft NoSolutionException (Checked Exception), falls es eine keine derartige
   // Zugfolge gibt.
   public List<Move> solve(Board board);

   // Gibt die Folge von Board-Konfigurationen auf der Konsole aus, die sich durch
   // Anwenden der Zugfolge moves auf die Ausgangskonfiguration board ergibt.
   public void printMoves(Board board, List<Move> moves);
}
```

Verwenden Sie bei Ihrer Lösung so weit wie möglich die Behälterklassen des JDK. Setzen Sie insbesondere bei der Implementierung des A*-Algorithmus (`SlidingPuzzle.solve()`) eine Prioritätswarteschlange (`PriorityQueue`) für die Speicherung Liste der offenen Knoten und eine sortierte Menge (`Set`) für die Verwaltung der Liste der geschlossenen Knoten ein.