

Gr. 1, E. Pitzer

Name Roman LumetsbergerAufwand in h 7

Gr. 2, F. Gruber-Leitner

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- (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](http://de.wikipedia.org/wiki/A*-Algorithmus). Der A\*-Algorithmus wird oft zur Wegsuche bei Routenplanern eingesetzt. Er ist aber auch auf die hier angeführte Problemstellung anwendbar.

Der 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 Manhattan-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:

- 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 Vorgängerknoten 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 Manhattan-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.

# 1 Verschiebe-Puzzle – A\*-Algorithmus

## 1.1 Lösungsidee

Als ersten Schritt der Lösung werden die, in der Angabe erwähnten, Klassen implementiert und die Testfälle erweitert. Dabei ist die Implementierung der meisten Methoden trivial. Im Folgenden werden nur mehr jede Lösungsideen angeführt, die mehr Überlegungen erfordern.

### 1.1.1 Board

Die Klasse *Board* verwendet als Datenspeicher eine *ArrayList*, wobei hier der Zeilen und Spalten-index auf den Index in der *ArrayList* abgebildet werden muss.

### 1.1.2 SearchNode - calcManhattanDistance

Die Manhattan Distance wird in der Klasse *SearchNode* berechnet und wird für den A\* Algorithmus benötigt.

Dabei wird das gesamte Board durchlaufen und für jedes Element die Abweichung zur Zielkonfiguration berechnet.

Die Zielposition kann folgendermaßen berechnet werden:

- Zeile: Nummer / Größe des Boards
- Spalte: Nummer % Größe des Boards

Die Manhattan Distance errechnet sich dann aus den Summen der Abweichungen der einzelnen Positionen.

### 1.1.3 SearchNode - toMoves

Um die Liste der Züge vom Start bis zur aktuellen Konfiguration zu berechnen, muss die verkettete Liste aufgelöst werden.

Das Ergebnis muss dann noch umgedreht werden, da wir ja die Züge vom Start bis zum Ziel benötigen.

### 1.1.4 SlidingPuzzle - solve

Um das Verschiebe-Puzzle zu lösen wird, wie in der Angabe erwähnt, der A\* Algorithmus angewendet.

Dieser benötigt eine **openQueue** und ein **closedSet**.

- openQueue: Enthält die noch zu prüfenden Pfade sortiert nach ihren Kosten zum Ziel. (Darum kann hier eine *PriorityQueue* verwendet werden.)
- closedSet: Enthält die bereits geprüften Pfade.

#### Ablauf:

- Zu Beginn wird die Startkonfiguration in die **openQueue** eingefügt.
- In einer Schleife wird dann das oberste Element der Queue herausgenommen.
- Ist das Element die Zielkonfiguration, wurde eine Lösung gefunden.

- Wenn nicht, wird das Element in das **closedSet** eingefügt.
- Dann werden alle Nachfolger berechnet(gültige Verschiebeoperationen anwenden).
- Jeder Nachfolger wird dann in die **openQueue** eingefügt, wenn er noch nicht betrachtet wurde oder seine Kosten geringer sind.

## 1.2 Sourcecode

### Board.java

---

```
1 package at.lumetsnet.puzzle;
2
3 import java.util.ArrayList;
4 import java.util.List;
5 import java.util.Random;
6
7 import at.lumetsnet.puzzle.exceptions.IllegalMoveException;
8 import at.lumetsnet.puzzle.exceptions.InvalidBoardIndexException;
9 import at.lumetsnet.puzzle.exceptions.InvalidTileNumberException;
10
11 public class Board implements Comparable<Board> {
12
13     /**
14      * Number of shuffle operations
15      */
16     private static final int SHUFFLE_COUNT = 100;
17
18     private final int size;
19     private final List<Integer> container;
20
21     public Board(int size) {
22         if (size <= 0) {
23             throw new IllegalArgumentException(
24                 "Size has to be greater than zero.");
25         }
26         this.size = size;
27         container = new ArrayList<Integer>(size * size);
28         for (int i = 0; i < size * size - 1; i++) {
29             container.add(i + 1);
30         }
31         container.add(0);
32     }
33
34     /**
35      * Gets the tile index inside the container
36      *
37      * @param rowIdx
38      * @param colIdx
39      * @return
40      */
41     private int getTileIndex(int rowIdx, int colIdx) {
42         return ((rowIdx - 1) * size) + (colIdx - 1);
43     }
44
45     /**
```

```
46  * Checks for valid board indices
47  *
48  * @param rowIdx
49  * @param colIdx
50  * @throws InvalidBoardIndexException
51  */
52  private void checkBoardIndex(int rowIdx, int colIdx)
53      throws InvalidBoardIndexException {
54      if (rowIdx < 1 || rowIdx > size || colIdx < 1 || colIdx > size) {
55          throw new InvalidBoardIndexException(rowIdx + ", " + colIdx
56              + " are invalid indices");
57      }
58  }
59
60  /**
61   * gets the tile on the given position
62   *
63   * @param rowIdx
64   * @param colIdx
65   * @throws InvalidBoardIndexException
66   * @return
67   */
68  public int getTile(int rowIdx, int colIdx)
69      throws InvalidBoardIndexException {
70      checkBoardIndex(rowIdx, colIdx);
71      return container.get(getTileIndex(rowIdx, colIdx));
72  }
73
74  /**
75   * Sets the tile on the given index
76   *
77   * @param rowIdx
78   * @param colIdx
79   * @param number
80   * @throws InvalidBoardIndexException
81   * @throws InvalidTileNumberException
82   */
83  public void setTile(int rowIdx, int colIdx, int number)
84      throws InvalidBoardIndexException, InvalidTileNumberException {
85      checkBoardIndex(rowIdx, colIdx);
86      if (number < 0 || number > size * size) {
87          throw new InvalidTileNumberException("Tile number " + number
88              + " is not a valid tile number");
89      }
90      container.set(getTileIndex(rowIdx, colIdx), number);
91  }
92
93  /**
94   * Sets the empty tile on the given position
```



```
95  *
96  * @param rowIdx
97  * @param colIdx
98  * @throws InvalidBoardIndexException
99  */
100 public void setEmptyTile(int rowIdx, int colIdx)
101     throws InvalidBoardIndexException {
102     checkBoardIndex(rowIdx, colIdx);
103     setTile(rowIdx, colIdx, 0);
104 }
105
106 /**
107  * Gets the row index of the empty tile
108  *
109  * @return
110  */
111 public int getEmptyTileRow() {
112     return (container.indexOf(0) / size) + 1;
113 }
114
115 /**
116  * Gets the column index of the empty tile
117  *
118  * @return
119  */
120 public int getEmptyTileColumn() {
121     return (container.indexOf(0) - ((getEmptyTileRow() - 1) * size) + 1);
122 }
123
124 /**
125  * Gets the size of the board
126  *
127  * @return
128  */
129 public int size() {
130     return size;
131 }
132
133 /**
134  * Checks if the board is valid
135  *
136  * @return
137  */
138 public boolean isValid() {
139     if (container.stream().distinct().count() == size * size) {
140         for (int i = 0; i < (size * size) - 1; i++) {
141             if (!container.contains(i)) {
142                 return false;
143             }
144         }
145     }
```

```
144     }
145     return true;
146 }
147 return false;
148 }
149
150 /**
151  * Shuffle the board
152  */
153 public void shuffle() {
154     Random rnd = new Random(System.nanoTime());
155     // Do 100 random moves
156     for (int i = 0; i < SHUFFLE_COUNT; i++) {
157         int random = rnd.nextInt(4);
158         try {
159             switch (random) {
160                 case 0:
161                     moveUp();
162                     break;
163                 case 1:
164                     moveRight();
165                     break;
166                 case 2:
167                     moveDown();
168                     break;
169                 case 3:
170                     moveLeft();
171                     break;
172             }
173         } catch (IllegalMoveException ex) {
174             // Ignore illegal moves
175         }
176     }
177 }
178
179 /**
180  * Moves the empty tile to the new position
181  *
182  * @param rowIdx
183  * @param colIdx
184  */
185 public void move(int rowIdx, int colIdx) throws IllegalMoveException {
186     if (rowIdx < 1 || rowIdx > size || colIdx < 1 || colIdx > size) {
187         throw new IllegalMoveException("Cannot move outside the board!");
188     }
189
190     int curRowIdx = getEmptyTileRow();
191     int curColIdx = getEmptyTileColumn();
192 }
```

```
193     if ((Math.abs(curRowIdx - rowIdx) == 1 && (curColIdx - colIdx == 0))
194         || (curRowIdx - rowIdx == 0 && Math.abs(curColIdx - colIdx) == 1)) {
195
196         int tile = getTile(rowIdx, colIdx);
197         setEmptyTile(rowIdx, colIdx);
198         setTile(curRowIdx, curColIdx, tile);
199     } else {
200         throw new IllegalMoveException("Cannot perform move!");
201     }
202
203 }
204
205 /**
206  * Moves the empty tile left
207  *
208  * @throws IllegalMoveException
209  */
210 public void moveLeft() throws IllegalMoveException {
211     int curRowIdx = getEmptyTileRow();
212     int curColIdx = getEmptyTileColumn();
213     move(curRowIdx, curColIdx - 1);
214 }
215
216 /**
217  * Moves the empty tile right
218  *
219  * @throws IllegalMoveException
220  */
221 public void moveRight() throws IllegalMoveException {
222     int curRowIdx = getEmptyTileRow();
223     int curColIdx = getEmptyTileColumn();
224     move(curRowIdx, curColIdx + 1);
225 }
226
227 /**
228  * Moves the empty tile up
229  *
230  * @throws IllegalMoveException
231  */
232 public void moveUp() throws IllegalMoveException {
233     int curRowIdx = getEmptyTileRow();
234     int curColIdx = getEmptyTileColumn();
235     move(curRowIdx - 1, curColIdx);
236 }
237
238 /**
239  * Moves the empty tile down
240  *
241  * @throws IllegalMoveException
```

```
242     */
243     public void moveDown() throws IllegalMoveException {
244         int curRowIdx = getEmptyTileRow();
245         int curColIdx = getEmptyTileColumn();
246         move(curRowIdx + 1, curColIdx);
247     }
248
249     /**
250      * Copies the board and all data
251      *
252      * @return
253      */
254     public Board copy() {
255         Board result = new Board(size);
256         result.container.clear();
257         result.container.addAll(container);
258         return result;
259     }
260
261     /**
262      * Execute the series of moves
263      *
264      * @param moves
265      * @throws IllegalMoveException
266      */
267     public void makeMoves(List<Move> moves) throws IllegalMoveException {
268         moves.forEach((x) -> {
269             move(x.getRow(), x.getCol());
270         });
271     }
272
273     /**
274      * clones the board
275      */
276     @Override
277     public Object clone() {
278         return this.copy();
279     }
280
281     /**
282      * Compares the size of this and the other board
283      */
284     @Override
285     public int compareTo(Board o) {
286         return size - o.size();
287     }
288
289     /**
290      * checks if the boards are equal
```

```
291     */
292     @Override
293     public boolean equals(Object other) {
294         if (this == other) {
295             return true;
296         }
297
298         if (!(other instanceof Board)) {
299             return false;
300         }
301         Board otherBoard = (Board) other;
302         if (this.compareTo(otherBoard) != 0) {
303             return false;
304         }
305
306         return this.container.equals(otherBoard.container);
307     }
308
309     /**
310      * calculates the hashCode of the board
311      */
312     @Override
313     public int hashCode() {
314         final int prime = 31;
315         int result = 1;
316         result = prime * result
317             + ((container == null) ? 0 : container.hashCode());
318         return result;
319     }
320
321     /**
322      * prints the board
323      */
324     @Override
325     public String toString() {
326         StringBuilder builder = new StringBuilder();
327         for (int i = 1; i <= size; i++) {
328             for (int j = 1; j <= size; j++) {
329                 builder.append(String.format("%2d", getTile(i, j)));
330             }
331             builder.append("\n");
332         }
333         return builder.toString();
334     }
335 }
336 }
```

---

**Move.java**

---

```
1 package at.lumetsnet.puzzle;
2
3 public class Move {
4     private int row;
5     private int col;
6
7     /**
8      * @param row
9      * @param col
10     */
11     public Move(int row, int col) {
12         super();
13         this.row = row;
14         this.col = col;
15     }
16
17     /**
18      * @return the row
19     */
20     public int getRow() {
21         return row;
22     }
23
24     /**
25      * @param row
26      *          the row to set
27     */
28     public void setRow(int row) {
29         this.row = row;
30     }
31
32     /**
33      * @return the col
34     */
35     public int getCol() {
36         return col;
37     }
38
39     /**
40      * @param col
41      *          the col to set
42     */
43     public void setCol(int col) {
44         this.col = col;
45     }
46
47 }
```

---

### SearchNode.java

```
1 package at.lumetsnet.puzzle;
2
3 import java.util.ArrayList;
4 import java.util.Collections;
5 import java.util.List;
6
7 public class SearchNode implements Comparable<SearchNode> {
8
9     private Board board;
10    private SearchNode predecessor;
11    private int costsFromStart;
12    private Move move;
13    private int manhattanDistance;
14
15    /**
16     * Constructor with board
17     *
18     * @param board
19     */
20    public SearchNode(Board board) {
21        this.board = board;
22        // calculate manhattan distance
23        manhattanDistance = calcManhattanDistance();
24    }
25
26    /**
27     * Constructor with all data
28     *
29     * @param board
30     * @param predecessor
31     * @param costsFromStart
32     * @param move
33     */
34    public SearchNode(Board board, SearchNode predecessor, int costsFromStart,
35        Move move) {
36        this(board);
37        this.predecessor = predecessor;
38        this.costsFromStart = costsFromStart;
39        this.move = move;
40    }
41
42    /**
43     * Gets the board
44     */
45    public Board getBoard() {
46        return board;
47    }
48
```

```
49  /**
50   * Gets the predecessor
51   *
52   @return
53   */
54  public SearchNode getPredecessor() {
55      return predecessor;
56  }
57
58  /**
59   * Sets the predecessor
60   *
61   @param predecessor
62   */
63  public void setPredecessor(SearchNode predecessor) {
64      this.predecessor = predecessor;
65  }
66
67  /**
68   * Gets the costs from start
69   *
70   @return
71   */
72  public int costsFromStart() {
73      return this.costsFromStart;
74  }
75
76  /**
77   * Sets the costs from start
78   *
79   @param costsFromStart
80   */
81  public void setCostsFromStart(int costsFromStart) {
82      this.costsFromStart = costsFromStart;
83  }
84
85  /**
86   * Gets the estimated costs to the goal
87   *
88   @return
89   */
90  public int estimatedCostsToTarget() {
91      return manhattanDistance;
92  }
93
94  /**
95   * Gets the estimated total costs
96   *
97   @return
```



```
98     */
99     public int estimatedTotalCosts() {
100         return costsFromStart + estimatedCostsToTarget();
101     }
102
103     /**
104      * Gets the move represented by this node
105      *
106      * @return
107      */
108     public Move getMove() {
109         return move;
110     }
111
112     /**
113      * Sets the move represented by this node
114      *
115      * @return
116      */
117     public void setMove(Move move) {
118         this.move = move;
119     }
120
121     /**
122      * Compares the node with another object
123      */
124     public boolean equals(Object other) {
125         if (other == null)
126             return false;
127         if (!(other instanceof SearchNode))
128             return false;
129
130         SearchNode otherNode = (SearchNode) other;
131         if (board == null && otherNode.board != null)
132             return false;
133         return board.equals(otherNode.board);
134     }
135
136     /**
137      * Compare the costs of the node against the other
138      *
139      * @return
140      */
141     @Override
142     public int compareTo(SearchNode other) {
143         return estimatedTotalCosts() - other.estimatedTotalCosts();
144     }
145
146     /**
```

```
147  * Converts the list into a list of moves from the start
148  *
149  * @return
150  */
151  public List<Move> toMoves() {
152      List<Move> result = new ArrayList<Move>();
153      SearchNode cur = this;
154      while (cur != null) {
155          if (cur.getMove() != null) {
156              result.add(cur.getMove());
157          }
158          cur = cur.getPredecessor();
159      }
160      // reverse the order of the collection
161      // to get the moves from the start
162      Collections.reverse(result);
163      return result;
164  }
165
166  /**
167   * Calculates the manhattan distance
168   *
169   * @return
170   */
171  private int calcManhattanDistance() {
172      int manhattanDistanceSum = 0;
173      for (int x = 1; x <= board.size(); x++)
174          for (int y = 1; y <= board.size(); y++) {
175              int value = board.getTile(x, y);
176              if (value != 0) {
177                  int targetX = (value - 1) / board.size();
178                  int targetY = (value - 1) % board.size();
179                  int dx = x - (targetX + 1);
180                  int dy = y - (targetY + 1);
181                  manhattanDistanceSum += Math.abs(dx) + Math.abs(dy);
182              }
183          }
184      return manhattanDistanceSum;
185  }
186
187  /**
188   * calculates the hashCode of the SearchNode
189   */
190  @Override
191  public int hashCode() {
192      final int prime = 17;
193      int result = 1;
194      result = prime * result + ((board == null) ? 0 : board.hashCode());
195      return result;
196  }
```

```
196     }  
197  
198 }
```

---

### SlidingPuzzle.java

---

```
1 package at.lumetsnet.puzzle;  
2  
3 import java.util.ArrayList;  
4 import java.util.HashSet;  
5 import java.util.List;  
6 import java.util.PriorityQueue;  
7 import java.util.Queue;  
8  
9 import at.lumetsnet.puzzle.exceptions.IllegalMoveException;  
10 import at.lumetsnet.puzzle.exceptions.NoSolutionException;  
11  
12 public class SlidingPuzzle {  
13  
14     /**  
15      * Solves the board  
16      *  
17      * @param board  
18      * @return  
19      */  
20     public List<Move> solve(Board board) {  
21         Queue<SearchNode> openQueue = new PriorityQueue<SearchNode>();  
22         HashSet<SearchNode> closedSet = new HashSet<SearchNode>();  
23  
24         // create search node from current board  
25         SearchNode current = new SearchNode(board);  
26         openQueue.add(current);  
27  
28         while (!openQueue.isEmpty()) {  
29             // get next node  
30             current = openQueue.poll();  
31  
32             // estimatedCostsToTarget = 0 means we found a solution  
33             if (current.estimatedCostsToTarget() == 0) {  
34                 return current.toMoves();  
35             }  
36  
37             closedSet.add(current);  
38  
39             // calculate the successors  
40             final List<SearchNode> successors = getSuccessors(current);  
41             for (SearchNode successor : successors) {  
42  
43                 if (!closedSet.contains(successor)) {
```

```
44         if (openQueue.contains(successor)
45             && current.estimatedTotalCosts() >= successor
46                 .estimatedTotalCosts()) {
47             // remove old node
48             openQueue.remove(successor);
49         }
50         openQueue.add(successor);
51     }
52 }
53 }
54 throw new NoSolutionException("Board has no solution");
55 }
56
57 /**
58  * returns the successors for the node
59  *
60  * @param parent
61  * @return
62  */
63 private List<SearchNode> getSuccessors(SearchNode parent) {
64     final List<SearchNode> result = new ArrayList<SearchNode>();
65     for (int i = 0; i < 4; i++) {
66         Board newBoard = parent.getBoard().copy();
67         try {
68             switch (i) {
69                 case 0:
70                     newBoard.moveLeft();
71                     break;
72                 case 1:
73                     newBoard.moveUp();
74                     break;
75                 case 2:
76                     newBoard.moveRight();
77                     break;
78                 case 3:
79                     newBoard.moveDown();
80                     break;
81             }
82             SearchNode node = new SearchNode(newBoard, parent,
83                 parent.costsFromStart() + 1, new Move(
84                     newBoard.getEmptyTileRow(),
85                     newBoard.getEmptyTileColumn()));
86             result.add(node);
87         } catch (IllegalMoveException ex) {
88             // ignore illegal moves
89         }
90     }
91     return result;
92 }
```

```
93
94  /**
95   * Prints the moves to the console
96   *
97   * @param board
98   * @param moves
99   */
100 public void printMoves(Board board, List<Move> moves) {
101     System.out.println("Starting board");
102     System.out.println(board);
103     moves.stream().forEach((x) -> {
104         board.move(x.getRow(), x.getCol());
105         System.out.println(board);
106     });
107 }
108 }
```

---

### BoardException.java

```
1 package at.lumetsnet.puzzle.exceptions;
2
3 public class BoardException extends RuntimeException {
4
5     /**
6      * Serial Id
7      */
8     private static final long serialVersionUID = 1L;
9
10    public BoardException(String message) {
11        super(message);
12    }
13 }
```

---

### IllegalMoveException.java

```
1 package at.lumetsnet.puzzle.exceptions;
2
3 public class IllegalMoveException extends BoardException {
4
5     /**
6      * Serial Id
7      */
8     private static final long serialVersionUID = 1L;
9
10    public IllegalMoveException(String message) {
11        super(message);
12    }
13 }
```

---

### InvalidBoardIndexException.java

---

```
1 package at.lumetsnet.puzzle.exceptions;
2
3 public class InvalidBoardIndexException extends BoardException {
4
5     /**
6      * Serial Id
7      */
8     private static final long serialVersionUID = 1L;
9
10    public InvalidBoardIndexException(String message) {
11        super(message);
12    }
13 }
```

---

### InvalidTileNumberException.java

---

```
1 package at.lumetsnet.puzzle.exceptions;
2
3 public class InvalidTileNumberException extends BoardException {
4
5     /**
6      * Serial Id
7      */
8     private static final long serialVersionUID = 1L;
9
10    public InvalidTileNumberException(String message) {
11        super(message);
12    }
13 }
```

---

### NoSolutionException.java

---

```
1 package at.lumetsnet.puzzle.exceptions;
2
3 public class NoSolutionException extends RuntimeException {
4
5     /**
6      * Serial Id
7      */
8     private static final long serialVersionUID = 1L;
9
10    public NoSolutionException(String message) {
11        super(message);
12    }
13 }
```

---

**AbstractTest.java**

---

```
1 package at.lumetsnet.puzzle.tests;
2
3 import static org.junit.Assert.assertTrue;
4 import at.lumetsnet.puzzle.Board;
5
6 public class AbstractTest {
7     /**
8      * Creates a 3x3 testboard with empty tile on 2x2
9      *
10     * @return board
11     */
12     protected Board getTestBoard() {
13         Board board = new Board(3);
14         board.setTile(1, 1, 1);
15         board.setTile(1, 2, 2);
16         board.setTile(1, 3, 3);
17         board.setTile(2, 1, 4);
18         board.setEmptyTile(2, 2);
19         board.setTile(2, 3, 6);
20         board.setTile(3, 1, 7);
21         board.setTile(3, 2, 8);
22         board.setTile(3, 3, 5);
23
24         assertTrue(board.isValid());
25         return board;
26     }
27 }
```

---

**BoardTest.java**

---

```
1 package at.lumetsnet.puzzle.tests;
2
3 import static org.junit.Assert.assertEquals;
4 import static org.junit.Assert.assertFalse;
5 import static org.junit.Assert.assertTrue;
6 import static org.junit.Assert.fail;
7
8 import java.util.ArrayList;
9 import java.util.List;
10
11 import org.junit.Test;
12
13 import at.lumetsnet.puzzle.Board;
14 import at.lumetsnet.puzzle.Move;
15 import at.lumetsnet.puzzle.exceptions.BoardException;
16 import at.lumetsnet.puzzle.exceptions.IllegalMoveException;
```

```
17
18 public class BoardTest extends AbstractTest {
19
20     @Test
21     public void getTileTest() {
22         Board board = getTestBoard();
23         assertEquals(0, board.getTile(2, 2));
24         assertEquals(1, board.getTile(1, 1));
25     }
26
27     @Test
28     public void setTileTest() {
29         Board board = new Board(3);
30         // default empty tile
31         assertEquals(1, board.getTile(1, 1));
32         board.setTile(1, 1, 2);
33         assertEquals(2, board.getTile(1, 1));
34     }
35
36     @Test
37     public void getEmptyTileTest() {
38         Board board = getTestBoard();
39         assertEquals(2, board.getEmptyTileColumn());
40         assertEquals(2, board.getEmptyTileRow());
41     }
42
43     public void sizeTest() {
44         Board board = getTestBoard();
45         assertEquals(2, board.size());
46         board = new Board(9);
47         assertEquals(9, board.size());
48     }
49
50     @Test
51     public void simpleIsValidTest() {
52         Board board;
53         try {
54             board = getTestBoard();
55             assertTrue(board.isValid());
56         } catch (BoardException e) {
57             fail("BoardException not expected.");
58         }
59     }
60
61     @Test
62     public void simpleIsNotValidTest() {
63         Board board;
64         try {
65             board = new Board(3);
```



```
66     board.setTile(1, 1, 1);
67     board.setTile(1, 2, 2);
68     board.setTile(1, 3, 3);
69     board.setTile(2, 1, 4);
70     board.setTile(2, 2, 5);
71     board.setTile(2, 3, 6);
72     board.setTile(3, 1, 7);
73     board.setTile(3, 2, 1);
74     board.setTile(3, 3, 0);
75
76     assertTrue(!board.isValid());
77 } catch (BoardException e) {
78     fail("BoardException not expected.");
79 }
80 }
81
82 @Test
83 public void simpleIsNotValidTest2() {
84     Board board;
85     try {
86         board = new Board(3);
87         board.setTile(1, 1, 8);
88         board.setTile(1, 2, 2);
89         board.setTile(1, 3, 0);
90         board.setTile(2, 1, 7);
91         board.setTile(2, 2, 5);
92         board.setTile(2, 3, 4);
93         board.setTile(3, 1, 3);
94         board.setTile(3, 2, 1);
95         board.setTile(3, 3, 6);
96
97         assertTrue(board.isValid());
98     } catch (BoardException e) {
99         fail("BoardException not expected.");
100    }
101 }
102
103 @Test
104 public void simpleIsNotValidTest3() {
105     Board board;
106     try {
107         board = new Board(3);
108         board.setTile(1, 1, 8);
109         board.setTile(1, 2, 2);
110         // not all tiles set
111         assertFalse(board.isValid());
112     } catch (BoardException e) {
113         fail("BoardException not expected.");
114     }
```

```
115     }
116
117     @Test
118     public void simpleIsNotValidTest4() {
119         Board board;
120         try {
121             board = new Board(3);
122             board.setTile(1, 1, 8);
123             board.setTile(1, 2, 2);
124             board.setTile(1, 3, 0);
125             board.setTile(2, 1, 7);
126             board.setTile(2, 2, 5);
127             board.setTile(2, 3, 4);
128             board.setTile(3, 1, 3);
129             board.setTile(3, 2, 1);
130             board.setTile(3, 3, 8); // invalid entry
131
132             assertFalse(board.isValid());
133         } catch (BoardException e) {
134             fail("BoardException not expected.");
135         }
136     }
137
138     @Test
139     public void moveOutsideTest() {
140         Board board = getTestBoard();
141         boolean hadException = false;
142         try {
143             board.move(1, 0);
144         } catch (IllegalMoveException ex) {
145             hadException = true;
146         }
147         assertTrue(hadException);
148         hadException = false;
149         try {
150             board.move(0, 1);
151         } catch (IllegalMoveException ex) {
152             hadException = true;
153         }
154         assertTrue(hadException);
155         hadException = false;
156         try {
157             board.move(4, 1);
158         } catch (IllegalMoveException ex) {
159             hadException = true;
160         }
161         assertTrue(hadException);
162         hadException = false;
163         try {
```

```
164     board.move(1, 4);
165 } catch (IllegalMoveException ex) {
166     hadException = true;
167 }
168 assertTrue(hadException);
169 hadException = false;
170
171 }
172
173 @Test
174 public void illegalMoveTest() {
175     Board board = getTestBoard();
176     boolean hadException = false;
177     try {
178         board.move(1, 3);
179     } catch (IllegalMoveException ex) {
180         hadException = true;
181     }
182     assertTrue(hadException);
183     hadException = false;
184
185     try {
186         board.move(3, 1);
187     } catch (IllegalMoveException ex) {
188         hadException = true;
189     }
190     assertTrue(hadException);
191     hadException = false;
192     try {
193         board.move(2, 2);
194     } catch (IllegalMoveException ex) {
195         hadException = true;
196     }
197     assertTrue(hadException);
198     hadException = false;
199 }
200
201 @Test
202 public void allowdMovesTest() {
203     Board board = getTestBoard();
204     board.move(2, 3);
205     assertTrue(board.isValid());
206     assertEquals(2, board.getEmptyTileRow());
207     assertEquals(3, board.getEmptyTileColumn());
208
209     board = getTestBoard();
210     board.move(3, 2);
211     assertTrue(board.isValid());
212     assertEquals(3, board.getEmptyTileRow());
```

```
213     assertEquals(2, board.getEmptyTileColumn());
214
215     board = getTestBoard();
216     board.move(2, 1);
217     assertTrue(board.isValid());
218     assertEquals(2, board.getEmptyTileRow());
219     assertEquals(1, board.getEmptyTileColumn());
220
221     board = getTestBoard();
222     board.move(2, 3);
223     assertTrue(board.isValid());
224     assertEquals(2, board.getEmptyTileRow());
225     assertEquals(3, board.getEmptyTileColumn());
226 }
227
228 @Test
229 public void moveLeftTest() {
230     Board board = getTestBoard();
231     board.moveLeft();
232     assertTrue(board.isValid());
233     assertEquals(2, board.getEmptyTileRow());
234     assertEquals(1, board.getEmptyTileColumn());
235 }
236
237 @Test(expected = IllegalMoveException.class)
238 public void moveLeftExceptionTest() {
239     Board board = getTestBoard();
240     board.moveLeft();
241     board.moveLeft();
242 }
243
244 @Test
245 public void moveRightTest() {
246     Board board = getTestBoard();
247     board.moveRight();
248     assertTrue(board.isValid());
249     assertEquals(2, board.getEmptyTileRow());
250     assertEquals(3, board.getEmptyTileColumn());
251 }
252
253 @Test(expected = IllegalMoveException.class)
254 public void moveRightExceptionTest() {
255     Board board = getTestBoard();
256     board.moveRight();
257     board.moveRight();
258 }
259
260 @Test
261 public void moveUpTest() {
```

```
262     Board board = getTestBoard();
263     board.moveUp();
264     assertTrue(board.isValid());
265     assertEquals(1, board.getEmptyTileRow());
266     assertEquals(2, board.getEmptyTileColumn());
267 }
268
269 @Test(expected = IllegalMoveException.class)
270 public void moveUpExceptionTest() {
271     Board board = getTestBoard();
272     board.moveUp();
273     board.moveUp();
274 }
275
276 @Test
277 public void moveDownTest() {
278     Board board = getTestBoard();
279     board.moveDown();
280     assertTrue(board.isValid());
281     assertEquals(3, board.getEmptyTileRow());
282     assertEquals(2, board.getEmptyTileColumn());
283 }
284
285 @Test(expected = IllegalMoveException.class)
286 public void moveDownExceptionTest() {
287     Board board = getTestBoard();
288     board.moveDown();
289     board.moveDown();
290 }
291
292 @Test
293 public void equalsTest() {
294     Board testBoard = getTestBoard();
295     Board sameBoard = getTestBoard();
296     Board otherBoard = getTestBoard();
297     otherBoard.moveDown();
298
299     assertTrue(testBoard.equals(sameBoard));
300     assertFalse(testBoard.equals(otherBoard));
301     assertFalse(testBoard.equals(1));
302 }
303
304 @Test
305 public void compareToTest() {
306     Board testBoard = new Board(3);
307     Board sameSizeBoard = new Board(3);
308     Board biggerBoard = new Board(4);
309     Board smallerBoard = new Board(2);
310     assertTrue(testBoard.compareTo(sameSizeBoard) == 0);
```

```
311     assertTrue(testBoard.compareTo(smallerBoard) == 1);
312     assertTrue(testBoard.compareTo(biggerBoard) == -1);
313 }
314
315 @Test
316 public void copyTest() {
317     Board board = getTestBoard();
318     Board copyBoard = board.copy();
319     // change original board to check copy of references
320     board.moveDown();
321     Board originalBoard = getTestBoard();
322     // check against original board
323     assertTrue(copyBoard.equals(originalBoard));
324     // check against changed board
325     assertFalse(copyBoard.equals(board));
326 }
327
328 @Test
329 public void shuffleTest() {
330     Board board = getTestBoard();
331     board.shuffle();
332     assertTrue(board.isValid());
333     // check that the board is not the same as the
334     // original board
335     assertFalse(board.equals(getTestBoard()));
336 }
337
338
339 @Test
340 public void makeMovesTest() {
341     Board board = getTestBoard();
342     List<Move> moveList = new ArrayList<Move>();
343     moveList.add(new Move(1, 2));
344     moveList.add(new Move(1, 3));
345     moveList.add(new Move(2, 3));
346     moveList.add(new Move(3, 3));
347     moveList.add(new Move(3, 2));
348     board.makeMoves(moveList);
349     assertTrue(board.isValid());
350     assertEquals(3, board.getEmptyTileRow());
351     assertEquals(2, board.getEmptyTileColumn());
352 }
353 }
```

---

### SearchNodeTest.java

---

```
1 package at.lumetsnet.puzzle.tests;
2
3 import static org.junit.Assert.*;
```

```
4
5 import java.util.List;
6
7 import org.junit.Test;
8
9 import at.lumetsnet.puzzle.Board;
10 import at.lumetsnet.puzzle.Move;
11 import at.lumetsnet.puzzle.SearchNode;
12 import at.lumetsnet.puzzle.exceptions.BoardException;
13
14 public class SearchNodeTest extends AbstractTest {
15
16     @Test
17     public void estimatedCostsToTargetTest() {
18         Board board = getTestBoard();
19         SearchNode node = new SearchNode(board);
20         assertEquals(2, node.estimatedCostsToTarget());
21     }
22
23     @Test
24     public void equalsNullTest() {
25         Board board = getTestBoard();
26         SearchNode node = new SearchNode(board);
27         assertFalse(node.equals(null));
28     }
29
30     @Test
31     public void equalsOtherObjectTest() {
32         Board board = getTestBoard();
33         SearchNode node = new SearchNode(board);
34         assertFalse(node.equals("1"));
35     }
36
37     @Test
38     public void equalsTest() {
39         Board board = getTestBoard();
40         SearchNode node = new SearchNode(board);
41
42         Board copy = getTestBoard();
43         SearchNode newNode = new SearchNode(copy);
44         assertTrue(node.equals(newNode));
45     }
46
47     @Test
48     public void compareToTest() {
49         Board board = getTestBoard();
50         SearchNode node = new SearchNode(board);
51
52         Board copy = getTestBoard();
```

```
53     SearchNode newNode = new SearchNode(copy);
54     assertEquals(0, newNode.compareTo(node));
55 }
56
57 @Test
58 public void toMovesTest() {
59     Board board = getTestBoard();
60     SearchNode node = new SearchNode(board, null, 0, null);
61
62     board = (Board) board.clone();
63     board.move(1, 2);
64     SearchNode newNode = new SearchNode(board, node, 1, new Move(1,2));
65     node = newNode;
66
67     board = (Board) board.clone();
68     board.move(1, 1);
69     newNode = new SearchNode(board, node, 1, new Move(1,1));
70     node = newNode;
71
72     List<Move> moves = node.toMoves();
73     assertEquals(2, moves.size());
74     assertEquals(1, moves.get(0).getRow());
75     assertEquals(2, moves.get(0).getCol());
76     assertEquals(1, moves.get(1).getRow());
77     assertEquals(1, moves.get(1).getCol());
78 }
79
80 @Test
81 public void simpleNodeTest() {
82     try {
83         Board board = new Board(3);
84         board.setTile(1, 1, 1);
85         board.setTile(1, 2, 2);
86         board.setTile(1, 3, 3);
87         board.setTile(2, 1, 4);
88         board.setTile(2, 2, 5);
89         board.setTile(2, 3, 6);
90         board.setTile(3, 1, 7);
91         board.setTile(3, 2, 8);
92         board.setTile(3, 3, 0);
93         SearchNode node = new SearchNode(board);
94         assertEquals(0, node.estimatedCostsToTarget());
95
96         board = new Board(3);
97         board.setTile(1, 1, 1);
98         board.setTile(1, 2, 2);
99         board.setTile(1, 3, 3);
100        board.setTile(2, 1, 4);
101        board.setTile(2, 2, 0);
```



```
102     board.setTile(2, 3, 6);
103     board.setTile(3, 1, 7);
104     board.setTile(3, 2, 8);
105     board.setTile(3, 3, 5);
106     node = new SearchNode(board);
107     assertEquals(2, node.estimatedCostsToTarget());
108
109     board = new Board(3);
110     board.setTile(1, 1, 1);
111     board.setTile(1, 2, 0);
112     board.setTile(1, 3, 3);
113     board.setTile(2, 1, 4);
114     board.setTile(2, 2, 5);
115     board.setTile(2, 3, 6);
116     board.setTile(3, 1, 7);
117     board.setTile(3, 2, 8);
118     board.setTile(3, 3, 2);
119     node = new SearchNode(board);
120     assertEquals(3, node.estimatedCostsToTarget());
121 } catch (BoardException e) {
122     fail("Unexpected BoardException.");
123 }
124 }
125
126 }
```

---

### SlidingPuzzleSolverTest.java

---

```
1 package at.lumetsnet.puzzle.tests;
2
3 import static org.junit.Assert.*;
4
5 import java.util.List;
6
7 import org.junit.Test;
8
9 import at.lumetsnet.puzzle.Board;
10 import at.lumetsnet.puzzle.Move;
11 import at.lumetsnet.puzzle.SlidingPuzzle;
12 import at.lumetsnet.puzzle.exceptions.NoSolutionException;
13
14 public class SlidingPuzzleSolverTest {
15
16     @Test
17     public void solveSimplePuzzleTest1() {
18         try {
19             SlidingPuzzle solver = new SlidingPuzzle();
20             Board board = new Board(3);
21             board.setTile(1, 1, 1);
```

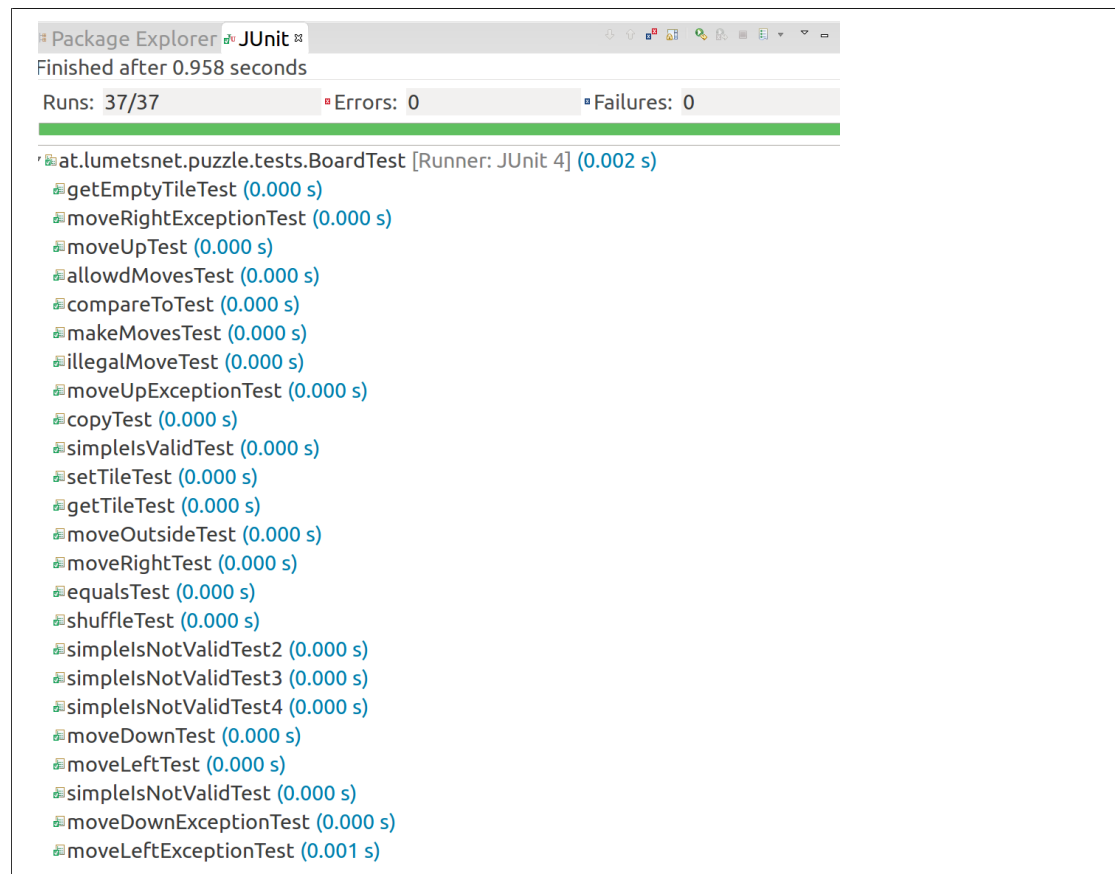
```
22     board.setTile(1, 2, 2);
23     board.setTile(1, 3, 3);
24     board.setTile(2, 1, 4);
25     board.setTile(2, 2, 5);
26     board.setTile(2, 3, 6);
27     board.setTile(3, 1, 7);
28     board.setTile(3, 2, 0);
29     board.setTile(3, 3, 8);
30
31     List<Move> moves = solver.solve(board);
32     assertEquals(1, moves.size());
33     assertTrue(moves.get(0).getRow() == 3 && moves.get(0).getCol() == 3);
34 } catch (NoSolutionException nse) {
35     fail("NoSolutionException is not expected.");
36 }
37 }
38
39 @Test
40 public void solveSimplePuzzleTest2() {
41     try {
42         SlidingPuzzle solver = new SlidingPuzzle();
43         Board board = new Board(3);
44         board.setTile(1, 1, 1);
45         board.setTile(1, 2, 2);
46         board.setTile(1, 3, 3);
47         board.setTile(2, 1, 4);
48         board.setTile(2, 2, 5);
49         board.setTile(2, 3, 6);
50         board.setTile(3, 1, 0);
51         board.setTile(3, 2, 7);
52         board.setTile(3, 3, 8);
53
54         List<Move> moves = solver.solve(board);
55         assertEquals(2, moves.size());
56         assertTrue(moves.get(0).getRow() == 3 && moves.get(0).getCol() == 2);
57         assertTrue(moves.get(1).getRow() == 3 && moves.get(1).getCol() == 3);
58     } catch (NoSolutionException nse) {
59         fail("NoSolutionException is not expected.");
60     }
61 }
62
63 @Test
64 public void solveComplexPuzzleTest1() {
65
66     try {
67         SlidingPuzzle solver = new SlidingPuzzle();
68
69         // 8 2 7
70         // 1 4 6
```

```
71     // 3 5 X
72     Board board = new Board(3);
73     board.setTile(1, 1, 8);
74     board.setTile(1, 2, 2);
75     board.setTile(1, 3, 7);
76     board.setTile(2, 1, 1);
77     board.setTile(2, 2, 4);
78     board.setTile(2, 3, 6);
79     board.setTile(3, 1, 3);
80     board.setTile(3, 2, 5);
81     board.setTile(3, 3, 0);
82
83     List<Move> moves = solver.solve(board);
84     board.makeMoves(moves);
85     assertEquals(new Board(3), board);
86     assertEquals(26, moves.size());
87 } catch (NoSolutionException nse) {
88     fail("NoSolutionException is not expected.");
89 }
90 }
91
92 @Test
93 public void solveRandomPuzzlesTest() {
94     SlidingPuzzle solver = new SlidingPuzzle();
95
96     for (int k = 0; k < 50; k++) {
97         try {
98             Board board = new Board(3);
99             int n = 1;
100             int maxN = board.size() * board.size();
101             for (int i = 1; i <= board.size(); i++)
102                 for (int j = 1; j <= board.size(); j++)
103                     board.setTile(i, j, (n++) % maxN);
104
105             board.shuffle();
106
107             List<Move> moves = solver.solve(board);
108             board.makeMoves(moves);
109             assertEquals(new Board(3), board);
110         } catch (NoSolutionException nse) {
111             fail("NoSolutionException is not expected.");
112         }
113     }
114 }
115
116 @Test
117 public void solveSimplePuzzleTest_4x4() {
118     try {
119         SlidingPuzzle solver = new SlidingPuzzle();
```

```
120     Board board = new Board(4);
121
122     board.moveLeft();
123
124     List<Move> moves = solver.solve(board);
125     assertEquals(1, moves.size());
126     assertTrue(moves.get(0).getRow() == 4 && moves.get(0).getCol() == 4);
127 } catch (NoSolutionException nse) {
128     fail("NoSolutionException is not expected.");
129 }
130 }
131
132 @Test
133 public void solveComplexPuzzleTest_4x4() {
134     try {
135         SlidingPuzzle solver = new SlidingPuzzle();
136         Board board = new Board(4);
137
138         board.moveLeft();
139         board.moveLeft();
140         board.moveUp();
141         board.moveLeft();
142         board.moveUp();
143         board.moveUp();
144         board.moveRight();
145         board.moveDown();
146         board.moveLeft();
147
148         List<Move> moves = solver.solve(board);
149         board.makeMoves(moves);
150         assertEquals(new Board(4), board);
151     } catch (NoSolutionException nse) {
152         fail("NoSolutionException is not expected.");
153     }
154 }
155
156 }
```

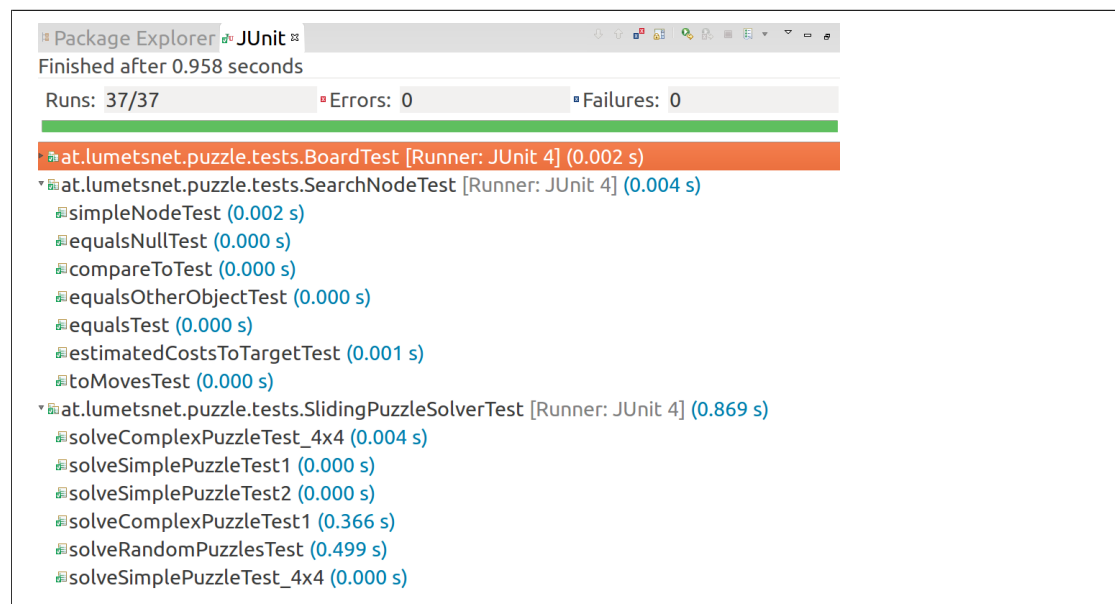
---

## 1.3 Testfälle



```
Package Explorer JUnit
Finished after 0.958 seconds
Runs: 37/37 Errors: 0 Failures: 0

at.lumetsnet.puzzle.tests.BoardTest [Runner: JUnit 4] (0.002 s)
  getEmptyTileTest (0.000 s)
  moveRightExceptionTest (0.000 s)
  moveUpTest (0.000 s)
  allowdMovesTest (0.000 s)
  compareToTest (0.000 s)
  makeMovesTest (0.000 s)
  illegalMoveTest (0.000 s)
  moveUpExceptionTest (0.000 s)
  copyTest (0.000 s)
  simpleIsValidTest (0.000 s)
  setTileTest (0.000 s)
  getTileTest (0.000 s)
  moveOutsideTest (0.000 s)
  moveRightTest (0.000 s)
  equalsTest (0.000 s)
  shuffleTest (0.000 s)
  simpleIsNotValidTest2 (0.000 s)
  simpleIsNotValidTest3 (0.000 s)
  simpleIsNotValidTest4 (0.000 s)
  moveDownTest (0.000 s)
  moveLeftTest (0.000 s)
  simpleIsNotValidTest (0.000 s)
  moveDownExceptionTest (0.000 s)
  moveLeftExceptionTest (0.001 s)
```



```
Package Explorer JUnit
Finished after 0.958 seconds
Runs: 37/37 Errors: 0 Failures: 0

at.lumetsnet.puzzle.tests.BoardTest [Runner: JUnit 4] (0.002 s)
at.lumetsnet.puzzle.tests.SearchNodeTest [Runner: JUnit 4] (0.004 s)
  simpleNodeTest (0.002 s)
  equalsNullTest (0.000 s)
  compareToTest (0.000 s)
  equalsOtherObjectTest (0.000 s)
  equalsTest (0.000 s)
  estimatedCostsToTargetTest (0.001 s)
  toMovesTest (0.000 s)
at.lumetsnet.puzzle.tests.SlidingPuzzleSolverTest [Runner: JUnit 4] (0.869 s)
  solveComplexPuzzleTest_4x4 (0.004 s)
  solveSimplePuzzleTest1 (0.000 s)
  solveSimplePuzzleTest2 (0.000 s)
  solveComplexPuzzleTest1 (0.366 s)
  solveRandomPuzzlesTest (0.499 s)
  solveSimplePuzzleTest_4x4 (0.000 s)
```