SWE 4x Übung zu Softwareentwicklung mit modernen Plattformen 4 — SS 2022, Übung 4

Abgabetermin: SA 8:00 in KW 18

☒ **Gr. 1**, M. Winkler, BSc MSc **Name** Dietmar Franzen **Aufwand in h** 25

☐ **Gr. 2**, Dr. E. Pitzer

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

Beispiel	L Lösungsidee	I Implementierung	T Tests	S = L+I+T	M Multiplikator	S*M
a	☐☒☐	☐☐☒	☐☐☒	9	2	18
b	☐☒☐	☐☐☒	☐☐☒	9	3	27
c	☐☒☐	☐☐☒	☐☐☒	27	3	20
d	☐☒☐	☐☐☒	☐☒☐	8	2	16
				Erfüllungsgrad ankreuzen bzw. Summen eintragen und auch online ausfüllen!		88

Verschiebe-Puzzle – A*-Algorithmus (SlidingPuzzle)

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 oder 15 Kacheln, die von 1 bis 8 bzw. 15 durchnummeriert sind, die auf einem 3x3-Spielfeld bzw. 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 Basisvariante des A*-Algorithmus enumeriert grundsätzlich auch alle möglichen Lösungsvarianten, allerdings wird versucht, zuerst 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 nicht mehr 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, sollten 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 benötigen 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 zu-rück.
    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(); }
```

Contents

1 Lösungsidee:

1.1 a)

Von den gegebenen Klassen wird nur der Rumpf erstellt. Die Tests dazu werden generiert und dann befüllt.

1.2 b)

Das Board verwendet zur Speicherung eine einfache ArrayList, das zweidimendsionale Feld wird über (row-1)*size+size angelegt.

1.3 c)

Die Distanz bereitet alles vor, damit man mit dem A*- Algorithmus das Puzzle lösen kann.

1.4 d)

Im SlidingPuzzle wird mittels A* Algorithmus das Puzzle gelöst. Dazu benötigt der Algorithmus eine PriorityQueue als openQueue und ein HashSet als closedSet. openQueue enthällt die noch zu prüfenden Pfade und closedSet die bereits geprüften.

2 Code:

Board.java

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class Board implements Comparable<Board> {
  private final int size;
  private final List<Integer> board;

  // Board mit Zielkonfiguration initialisieren.
  public Board(int size) {
    if (size < 0)
      throw new IllegalArgumentException("Size has to be bigger then 0");
    this.size = size;
    board = new ArrayList<>(size * size);
    for (int i = 0; i < size * size - 1; i++) {
      board.add(i + 1);
    }
    board.add(0);
  }

  // Überprüfen, ob dieses Board und das Board other dieselbe Konfiguration
  // aufweisen.
  public boolean equals(Board other) {
    if (this.size != other.size())
      return false;
    for (int i = 1; i < size+1; i++) {
      for (int j = 1; j < size+1; j++) {
        if (this.getTile(i, j) != other.getTile(i, j))
          return false;
      }
    }
    return true;
  }

  // <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) {
    return other.size() - this.size();
  }

  // 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) {
    if (checkCoordinates(i, j))
      throw new InvalidBoardIndexException("i: " + i + " or j: " + j + " is bigger then size: " + size)
    return this.board.get((i - 1) * size + (j - 1));
  }

  // Setzt die Kachelnummer an der Stelle (i,j) zurück. Wirft die
  // Laufzeitausnahmen
  // InvalidBoardIndexException und InvalidTileNumberException
  public void setTile(int i, int j, int number) {
    if (checkCoordinates(i, j))
      throw new InvalidBoardIndexException("i: " + i + " or j: " + j + " is bigger then size: " + size)
```

```
  this.board.set((i - 1) * size + (j - 1), number);
}

private boolean checkCoordinates(int i, int j) {
  return i < 1 || j < 1 || i > this.size || j > this.size;
}

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

// Zeilenindex der leeren Kachel
public int getEmptyTileRow() {
  return this.board.indexOf(0) / size + 1;
}

// Gibt Spaltenindex der leeren Kachel zurück.
public int getEmptyTileColumn() {
  return this.board.indexOf(0) - ((getEmptyTileRow() - 1) * this.size) + 1;
}

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

// Überprüft, ob Position der Kacheln konsistent ist.
public boolean isValid() {
  for (int i = 0; i < (size * size) - 1; i++) {
    if (!this.board.contains(i)) {
      return false;
    }
  }
  return true;
}

// Macht eine tiefe Kopie des Boards.
// Vorsicht: Referenztypen müssen neu allokiert und anschließend deren Inhalt
// kopiert werden.
public Board copy() {
  Board result = new Board(this.size);
  result.board.clear();
  result.board.addAll(this.board);
  return result;
}

// Erzeugt eine zufällige lösbare Konfiguration des Boards, indem auf die
// bestehende
// Konfiguration eine Reihe zufälliger Verschiebeoperationen angewandt wird.
public void shuffle() {
  Random rnd = new Random(System.nanoTime());
  for (int i = 0; i < 100000; i++) {
    int rndNummer = rnd.nextInt(4);
    try {
      switch (rndNummer) {
        case 0:
          moveDown();
          break;
```

```
          case 1:
            moveUp();
            break;
          case 2:
            moveRight();
            break;
          case 3:
            moveLeft();
            break;
        }
        ;
      } catch (IllegalMoveException e) {
        // Nothing to do
      }
    }
  }

  // Verschiebt leere Kachel auf neue Position (row, col).
  // throws IllegalMoveException
  public void move(int row, int col) {
    if (checkCoordinates(row, col))
      throw new IllegalMoveException("Cannot move to (" + row + ", " + col + ")");
    int curRow = getEmptyTileRow();
    int curCol = getEmptyTileColumn();
    if (!((Math.abs(curRow - row) == 1 && (curCol - col == 0))
            || (curRow - row == 0 && Math.abs(curCol - col) == 1)))
      throw new IllegalMoveException("Move row: " + row + " col: " + col + " is illegal.");
    int tile = getTile(row, col);
    setEmptyTile(row, col);
    setTile(curRow, curCol, tile);
  }

  // Verschiebt leere Kachel nach links. Wirft Laufzeitausnahme
  // IllegalMoveException.
  public void moveLeft() {
    move(getEmptyTileRow(), getEmptyTileColumn() - 1);
  }

  // Verschiebt leere Kachel nach rechts. Wirft IllegalMoveException.
  public void moveRight() {
    move(getEmptyTileRow(), getEmptyTileColumn() + 1);
  }

  // Verschiebt leere Kachel nach oben. Wirft IllegalMoveException.
  public void moveUp() {
    move(getEmptyTileRow() - 1, getEmptyTileColumn());
  }

  // Verschiebt leere Kachel nach unten. Wirft IllegalMoveException.
  public void moveDown() {
    move(getEmptyTileRow() + 1, getEmptyTileColumn());
  }

  // Führt eine Sequenz an Verschiebeoperationen durch. Wirft
  // IllegalMoveException.
  public void makeMoves(List<Move> moves) {
    moves.forEach(m -> move(m.getRow(), m.getCol()));
  }

  private class InvalidBoardIndexException extends BoardException {
    public InvalidBoardIndexException(String message) {
```

```
      super(message);
    }
  }

  private class BoardException extends RuntimeException {
    public BoardException(String message) {
      super(message);
    }
  }

  private class IllegalMoveException extends BoardException {
    public IllegalMoveException(String message) {
      super(message);
    }
  }
}
```

Move.java

```
public class Move {
  private int row;
  private int col;

  public Move(int row, int col) {
    super();
    this.row = row;
    this.col = col;
  }

  public int getRow() {
    return row;
  }

  public void setRow(int row) {
    this.row = row;
  }

  public int getCol() {
    return col;
  }

  public void setCol(int col) {
    this.col = col;
  }
}
```

SearchNode.java

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

public class SearchNode implements Comparable<SearchNode> {

  private Board board;
  private SearchNode getPredecessor;
  private int sumCosts;
  private Move move;

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

```
    this.board = board;

  }

  // Gibt Board-Konfiguration dieses Knotens zurück.
  public Board getBoard() {
    return null;
  }

  // Gibt Referenz auf Vorgängerknoten zurück.
  public SearchNode getPredecessor() {
    return null;
  }

  // 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() {
    return 0;
  }

  // Gibt geschätzte Kosten bis zum Zielknoten zurück. Die Abschätzung
  // kann mit der Summe der Manhatten-Distanzen aller Kacheln erfolgen.
  public int estimatedCostsToTarget() {
    int result = 0;
    for (int x = 1; x <= board.size(); x++)
      for (int y = 1; y <= board.size(); y++) {
        int value = board.getTile(x, y);
        if (value != 0) {
          int targetX = (value - 1) / board.size();
          int targetY = (value - 1) % board.size();
          result += Math.abs(x - (targetX + 1)) + Math.abs(y - (targetY + 1));
        }
      }
    return result;
  }

  // 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 zu-rück.
  public int estimatedTotalCosts() {
    return 0;
  }

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

  // 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) {
    return other.sumCosts - this.sumCosts;
  }

  // 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() {
    List<Move> result = new ArrayList<Move>();
    SearchNode cur = this;
    while (cur != null) {
      if (cur.move != null) {
        result.add(cur.move);
      }
      cur = cur.getPredecessor();
    }
    // reverse the order of the collection
    // to get the moves from the start
    List<Move> tmp = new ArrayList<>();
    for (int i = 0; i < result.size(); i++) {
      tmp.add(result.get(result.size() - i - 1));
    }
    return result;
  }

  public void setMove(Move move) {
    this.move = move;
  }
}
```

SlidingPuzzle.java

```
import java.util.*;

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) throws NoSolutionException {
    Queue<SearchNode> openQueue = new PriorityQueue<SearchNode>();
    HashSet<SearchNode> closedSet = new HashSet<SearchNode>();

    // create search node from current board
    SearchNode current = new SearchNode(board);
    openQueue.add(current);

    while (!openQueue.isEmpty()) {
      // get next node
      current = openQueue.poll();

      // estimatedCostsToTarget = 0 means we found a solution
      if (current.estimatedCostsToTarget() == 0) {
        return current.toMoves();
      }

      closedSet.add(current);

      // calculate the successors
```

```
    final List<SearchNode> successors = getSuccessors(current);
    for (SearchNode successor : successors) {

      if (!closedSet.contains(successor)) {
        if (openQueue.contains(successor)
                && current.estimatedTotalCosts() >= successor
                .estimatedTotalCosts()) {
          // remove old node
          openQueue.remove(successor);
        }
        openQueue.add(successor);
      }
    }
  }
  return null;
}

// Gibt die Folge von Board-Konfigurationen auf der Konsole aus, die sich durch
// Anwenden der Zugfolge moves auf die Ausgangskonfiguration board ergibt.
public static void printMoves(Board board, List<Move> moves) {
  System.out.println("Starting board");
  System.out.println(board);
  moves.stream().forEach((x) -> {
    board.move(x.getRow(), x.getCol());
    System.out.println(board);
  });
}

private List<SearchNode> getSuccessors(SearchNode parent) {
  final List<SearchNode> result = new ArrayList<SearchNode>();
  for (int i = 0; i < 4; i++) {
    Board newBoard = parent.getBoard().copy();
    try {
      switch (i) {
        case 0:
          newBoard.moveLeft();
          break;
        case 1:
          newBoard.moveUp();
          break;
        case 2:
          newBoard.moveRight();
          break;
        case 3:
          newBoard.moveDown();
          break;
      }
      SearchNode node = new SearchNode(newBoard);
      node.setPredecessor(parent);
      node.setCostsFromStart(parent.costsFromStart() + 1);
              node.setMove(new Move(
              newBoard.getEmptyTileRow(),
              newBoard.getEmptyTileColumn()));
      result.add(node);
    } catch (IllegalMoveException ex) {
      // nothing to do here
    }
  }
  return result;
}
private class BoardException extends RuntimeException {
```

```
    public BoardException(String message) {
      super(message);
    }
  }
  private class IllegalMoveException extends BoardException {
    public IllegalMoveException(String message) {
      super(message);
    }
  }

  private class NoSolutionException extends Exception{
    public NoSolutionException(String message) {
      super(message);
    }
  }
}
```

BoardTests.java

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

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

class BoardTests {
  Board board;

  @BeforeEach
  void setUp() {
    /* Bord for testing
     * 6 4 3
     * - 1 2
     * 8 5 7
     *
     * solved in 25 moves
     * */
    board = new Board(3);
    board.setTile(1, 1, 6);
    board.setTile(1, 2, 4);
    board.setTile(1, 3, 3);

    board.setEmptyTile(2, 1);
    board.setTile(2, 2, 1);
    board.setTile(2, 3, 2);

    board.setTile(3, 1, 8);
    board.setTile(3, 2, 5);
    board.setTile(3, 3, 7);
  }

  @AfterEach
  void tearDown() {
    board = null;
  }

  @Test
  void testEquals() {
    Board otherBoard = new Board(3);
    otherBoard.setTile(1, 1, 6);
```

```
    otherBoard.setTile(1, 2, 4);
    otherBoard.setTile(1, 3, 3);

    otherBoard.setEmptyTile(2, 1);
    otherBoard.setTile(2, 2, 1);
    otherBoard.setTile(2, 3, 2);

    otherBoard.setTile(3, 1, 8);
    otherBoard.setTile(3, 2, 5);
    otherBoard.setTile(3, 3, 7);

    assertTrue(board.equals(otherBoard));

    otherBoard = null;
    /*
     * 6 4 3
     * 1 5 -
     * 8 7 2
     * */
    otherBoard = new Board(3);
    otherBoard.setTile(1, 1, 6);
    otherBoard.setTile(1, 2, 4);
    otherBoard.setTile(1, 3, 3);

    otherBoard.setTile(1, 2, 1);
    otherBoard.setTile(2, 2, 5);
    otherBoard.setEmptyTile(2, 3);

    otherBoard.setTile(3, 1, 8);
    otherBoard.setTile(3, 2, 7);
    otherBoard.setTile(3, 3, 2);

    assertFalse(board.equals(otherBoard));
  }

  @Test
  void compareTo() {
    assertEquals(0, board.compareTo(new Board(3)));
    assertTrue(board.compareTo(new Board(4)) > 0);
    assertTrue(new Board(4).compareTo(board) < 0);
  }

  @Test
  void getTile() {
    assertEquals(6, board.getTile(1, 1));
    assertEquals(1, board.getTile(2, 2));
    assertEquals(7, board.getTile(3, 3));
  }

  @Test
  void setTile() {
    board=null;
    board = new Board(3);
    board.setTile(1,1,1);
    assertEquals(1,board.getTile(1,1));
    board.setTile(3,3,8);
    assertEquals(8,board.getTile(3,3));
  }

  @Test
  void setEmptyTile() {
```

```
  board = null;
  board = new Board(3);
  board.setEmptyTile(1,1);
  assertEquals(0,board.getTile(1,1));
  board.setEmptyTile(2,3);
  assertEquals(0,board.getTile(2,3));
}

@Test
void getEmptyTileRow() {
  assertEquals(2,board.getEmptyTileRow());
}

@Test
void getEmptyTileColumn() {
  assertEquals(1,board.getEmptyTileColumn());
}

@Test
void size() {
  assertEquals(3, board.size());
  assertEquals(4,new Board(4).size());
  assertEquals(6,new Board(6).size());
}

@Test
void isValid() {
  assertTrue(board.isValid());
}

@Test
void copy() {
  Board otherBoard = board.copy();
  assertTrue(board.equals(otherBoard));
}

@Test
void shuffle() {
  Board originalBoad = board.copy();
  board.shuffle();
  assertFalse(originalBoad.equals(board));
}

@Test
void move() {
  board.move(2,2);
  assertEquals(1,board.getTile(2,1));
}

@Test
void moveLeft() {
  board = null;
  board = new Board(3);
  board.setTile(1,1,6);
  board.setTile(1,2,4);
  board.setTile(1,3,3);

  board.setTile(2,1,1);
  board.setEmptyTile(2,2);
  board.setTile(2,3,2);
```

```
    board.setTile(3,1,8);
    board.setTile(3,2,5);
    board.setTile(3,3,7);
    board.moveLeft();
    assertEquals(1,board.getTile(2,2));
  }

  @Test
  void moveRight() {
    board.moveRight();
    assertEquals(1,board.getTile(2,1));
  }

  @Test
  void moveUp() {
    board.moveUp();
    assertEquals(6,board.getTile(2,1));
  }

  @Test
  void moveDown() {
    board.moveDown();
    assertEquals(8,board.getTile(2,1));
  }
}
```

SearchNodeTests.java

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.util.List;

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

class SearchNodeTests {
  Board board;
  SearchNode node;

  @BeforeEach
  void setUp() {
    /* Bord for testing
     * 6 4 3
     * - 1 2
     * 8 5 7
     *
     * solved in 25 moves
     * */
    board = new Board(3);
    board.setTile(1, 1, 6);
    board.setTile(1, 2, 4);
    board.setTile(1, 3, 3);

    board.setEmptyTile(2, 1);
    board.setTile(2, 2, 1);
    board.setTile(2, 3, 2);

    board.setTile(3, 1, 8);
    board.setTile(3, 2, 5);
```

```
    board.setTile(3, 3, 7);

    node = new SearchNode(board);
  }

  @AfterEach
  void tearDown() {
    board = null;
    node = null;
  }

  @Test
  void costsFromStart() {
    System.out.println(node.costsFromStart());
  }

  @Test
  void estimatedCostsToTarget() {
    System.out.println(node.estimatedCostsToTarget());
  }

  @Test
  void estimatedTotalCosts() {
    System.out.println(node.estimatedTotalCosts());
  }

  @Test
  void toMoves() {
    SearchNode node = new SearchNode(board);
    node.setCostsFromStart(0);

    board = board.copy();
    board.move(2, 2);
    SearchNode newNode =new SearchNode(board);
    newNode.setPredecessor(node);
    newNode.setCostsFromStart(1);
    newNode.setMove(new Move(1,2));
    node = newNode;


    List<Move> moves = node.toMoves();
    assertEquals(1, moves.size());
    assertEquals(1, moves.get(0).getRow());
    assertEquals(2, moves.get(0).getCol());
  }
}
```

SlidingPuzzleTest.java

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.util.List;

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

class SearchNodeTests {
  Board board;
  SearchNode node;
```

```
@BeforeEach
void setUp() {
  /* Bord for testing
   * 6 4 3
   * - 1 2
   * 8 5 7
   *
   * solved in 25 moves
   * */
  board = new Board(3);
  board.setTile(1, 1, 6);
  board.setTile(1, 2, 4);
  board.setTile(1, 3, 3);

  board.setEmptyTile(2, 1);
  board.setTile(2, 2, 1);
  board.setTile(2, 3, 2);

  board.setTile(3, 1, 8);
  board.setTile(3, 2, 5);
  board.setTile(3, 3, 7);

  node = new SearchNode(board);
}

@AfterEach
void tearDown() {
  board = null;
  node = null;
}

@Test
void costsFromStart() {
  System.out.println(node.costsFromStart());
}

@Test
void estimatedCostsToTarget() {
  System.out.println(node.estimatedCostsToTarget());
}

@Test
void estimatedTotalCosts() {
  System.out.println(node.estimatedTotalCosts());
}

@Test
void toMoves() {
  SearchNode node = new SearchNode(board);
  node.setCostsFromStart(0);

  board = board.copy();
  board.move(2, 2);
  SearchNode newNode =new SearchNode(board);
  newNode.setPredecessor(node);
  newNode.setCostsFromStart(1);
  newNode.setMove(new Move(1,2));
  node = newNode;
```

```
    List<Move> moves = node.toMoves();
    assertEquals(1, moves.size());
    assertEquals(1, moves.get(0).getRow());
    assertEquals(2, moves.get(0).getCol());
  }
}
```