|  |  |  |
| --- | --- | --- |
| **SWE 4x** | **Übung zu Softwareentwicklung mit**  **mit modernen Plattformen 4** | **SS 2022, Übung 04** |

**Abgabe elektronisch bis Sa 8 Uhr in der KW 18**

* ~~Gr. 1,~~ Winkler, BSc Msc **Name: Roman Kofler-Hofer Aufwand in h: 14**
* **Gr. 2,** Dr. Pitzer

**Punkte: Kurzzeichen Tutor / Übungsleiter /**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Beispiel | L Lösungsidee | I Implementierung | T Tests | S = L+I+T | Multiplikator | S\*M |
| a | ⌧⌧⌧ | ⌧⌧⌧⌧ | ⌧⌧⌧ | 10 | 2 | 20 |
| b | ⌧⌧⌧ | ⌧⌧⌧⌧ | ⌧⌧⌧ | 10 | 3 | 30 |
| c | ⌧⌧⌧ | ⌧⌧⌧⌧ | ⌧⌧⌧ | 10 | 3 | 30 |
| d | ⌧⌧⌧ | ⌧⌧⌧⌧ | ⌧⌧⌧ | 10 | 2 | 20 |
|  |  |  |  |  | **Summe** | **100** |

[1. Verschiebepuzzle 2](#_Toc102500601)

[1.1. Board 2](#_Toc102500602)

[1.1.1. Lösungsidee 2](#_Toc102500603)

[1.1.2. Quellcode 3](#_Toc102500604)

[1.1.3. Tests 15](#_Toc102500605)

[1.2. SearchNode 24](#_Toc102500606)

[1.2.1. Lösungsidee 24](#_Toc102500607)

[1.2.2. Quellcode 25](#_Toc102500608)

[1.2.3. Tests 31](#_Toc102500609)

[1.3. SlidingPuzzle 35](#_Toc102500610)

[1.3.1. Lösungsidee 35](#_Toc102500611)

[1.3.2. Quellcode 36](#_Toc102500612)

[1.3.3. Tests 39](#_Toc102500613)

# Verschiebepuzzle

## Board

### Lösungsidee

Als Datenstruktur für das Board wurde eine ArrayList gewählt. Zusätzlich zum Array wird auch noch die Größe des Spielfeldes (3 oder 4) gespeichert. Um die zweidimensionalen Koordinaten in eindimensionale Koordinaten umzurechnen, gibt es eine Hilfsfunktion. In der Klasse biete ich zwei Konstruktoren an. Einer nimmt nur eine Size und initialisiert damit ein gelöstes Board. Der zweite Konstruktor kann eine Liste aufnehmen und daraus das Board generieren.

Die meisten weiteren geforderten Methoden sind eigentlich mit sehr einfachen Algorithmen zu lösen. Nur zu den folgenden möchte ich Kommentare machen.

**Copy**: um das Board zu kopieren rufe ich den Konstruktor (mit der size vom originalen Board) auf. Dadurch wird ein neues Board mit eigenem Array angelegt. Dann kopiere ich die Werte vom originalen Board in das neu erstellte Board (das ja zu Beginn als sortiertes Board initialisiert wird).

**IsSolveable:** Diese Funktion habe ich eingeführt, um zu überprüfen, ob ein Spielbrett lösbar ist oder nicht. Glücklicherweise kann man schon vor man das Puzzle löst feststellen, ob es lösbar ist. Dazu muss man sich die Anzahl der nötigen Vertauschungen ansehen. Bei einem Board mit ungerader Größe muss die Anzahl der Vertauschungen gerade sein. Bei einem Board mit gerade Größe muss man zusätzlich zur Anzahl der Vertauschungen auch noch die Reihennummer der leeren Zelle berücksichtigen. Wenn die Summe dieser beiden Werte ungerade ist, ist das Board lösbar.[[1]](#footnote-1)

### Quellcode

//file Board.java

package at**.**fhooe**.**swe4**.**slidingpuzzle**;**

**import** com**.**sun**.**xml**.**internal**.**ws**.**api**.**model**.**wsdl**.**WSDLOutput**;**

**import** java**.**util**.\*;**

**import** static java**.**lang**.**Math**.**abs**;**

public class Board **implements** Comparable**<**Board**>** **{**

private final List board**;**

private int size**;**

/\*\*

\* Calculcates the array index for a given row and column (row and column start with 1)

\* **@param** row = input row

\* **@param** col = input column

\* **@return** = array index for this row and column

\*/

private int getArrayIndex**(**int row**,** int col**)** **{**

**if(**row **<** 1 **||** row **>** size **||** col **<** 1 **||** col **>** size**)** **{**

**throw** **new** ArrayIndexOutOfBoundsException**(**"indices must be > 0 and <= size"**);**

**}**

int nRow **=** row**-**1**;**

int nCol **=** col**-**1**;**

**return** nRow**\***size**+**nCol**;**

**}**

/\*\*

\* Constructor for board. Initializes board 1 to n\*n-1. Last position of array (n|n) holds 0 cell.

\* Size must be 3 or 4. If other size is chosen board will be initialized with size 3

\* **@param** size = size of board (for a 3x3 board enter 3 and NOT 9)

\*/

public Board**(**int size**)** **{**

int saveSize **=** size**;**

**if((**size **!=** 3**)** **&&** **(**size **!=** 4**))** **{**

System**.**out**.**println**(**"Only size 3 or 4 allowed. Board was initialized with size = 3"**);**

saveSize **=** 3**;**

**}**

board **=** **new** ArrayList**<**Integer**>(**saveSize**\***saveSize**);**

**for(**int i **=** 0**;** i **<** saveSize**\***saveSize**;** i**++)** **{**

board**.**add**(**i**,**i**+**1**);**

**}**

board**.**set**(**saveSize**\***saveSize**-**1**,** 0**);**

**this.**size **=** saveSize**;**

**}**

/\*\*

\* Constructor taking a one dimensional list as start board

\* **@param** boardInput = list representing start baord

\* **@param** size = size of board (for a 3x3 board enter 3 and NOT 9)

\*/

public Board**(**List boardInput**,** int size**)** **{**

board **=** **new** ArrayList**<**Integer**>();**

board**.**addAll**(**boardInput**);**

**this.**size **=** size**;**

**}**

/\*\*

\* **@return** string representation of board

\*/

@Override

public String toString**()** **{**

StringBuilder sb **=** **new** StringBuilder**();**

**for(**int row **=** 1**;** row **<=** size**;** row**++)** **{**

**for(**int col **=** 1**;** col **<=** size**;** col**++)** **{**

sb**.**append**(**board**.**get**(**getArrayIndex**(**row**,** col**)));**

sb**.**append**(**"\t"**);**

**}**

sb**.**append**(**"\n"**);**

**}**

**return** sb**.**toString**();**

**}**

/\*\*

\* checks if boards have the same configuration (board and size are considered)

\* **@param** other = the other board

\* **@return** = true if this and other are the same, else false

\*/

public boolean equals**(**Object other**)** **{**

**if** **(this** **==** other**)** **return** **true;**

**if(**other **==** **null** **||** getClass**()** **!=** other**.**getClass**())** **return** **false;**

Board o **=** **(**Board**)**other**;**

**return** board**.**equals**(**o**.**board**)** **&&**

size **==** o**.**size**;**

**}**

/\*\*

\* **@return** hashCode for board (considering board and size)

\*/

@Override

public int hashCode**()** **{**

**return** Objects**.**hash**(**board**,** size**);**

**}**

/\*\*

\* **@param** other the object to be compared.

\* **@return** < 1 if this is smaller than other; 0 if they are the same, >1 if this is larger than oterh

\*/

public int compareTo**(**Board other**)** **{**

**if** **(**size **<** other**.**size**)** **{**

**return** **-**1**;**

**}**

**else** **if** **(**size **==** other**.**size**)** **{**

**return** 0**;**

**}** **else**

**return** 1**;**

**}**

/\*\*

\* Gets the value of a certain tile

\* **@param** i = row of tile

\* **@param** j = column of tile

\* **@return** = number of tile at position i|j

\* throws InvalidBoardIndexException if i or j are < 1 or > size

\*/

public int getTile**(**int i**,** int j**)** **{**

**if(**i **<** 1 **||** i **>** size **||** j **<** 1 **||** j **>** size**)** **{**

**throw** **new** InvalidBoardIndexException**(**i**,**j**,**size**);**

**}**

Integer val **=** **(**Integer**)** board**.**get**(**getArrayIndex**(**i**,**j**));**

**return** val**;**

**}**

/\*\*

\* Sets a tile to input number

\* **@param** i = row of tile to be set

\* **@param** j = column of tile to be set

\* **@param** number = value which should be set on position i|j

\* throws InvalidBoardIndexException if i or j are < 1 or > size

\* throws InvalidTileNumberException if number < 1 or > size\*size-1

\*/

public void setTile**(**int i**,** int j**,** int number**)** **{**

**if(**i **<** 1 **||** i **>** size **||** j **<** 1 **||** j **>** size**)** **{**

**throw** **new** InvalidBoardIndexException**(**i**,**j**,**size**);**

**}**

**if(**number **<** 0 **||** number **>** size**\***size**-**1**)** **{**

**throw** **new** InvalidTileNumberException**(**number**,** size**);**

**}**

int aIndex **=** getArrayIndex**(**i**,**j**);**

board**.**set**(**aIndex**,** number**);**

**}**

/\*\*

\* Sets a certain position to 0 (it could be possible to set several tiles to 0 with this method)

\* **@param** i = row of tile which should be set to 0

\* **@param** j = column of tile which should be set to 0

\* throws InvalidBoardIndexException if i or j are < 1 or > size

\*/

public void setEmptyTile**(**int i**,** int j**)** **{**

**if(**i **<** 1 **||** i **>** size **||** j **<** 1 **||** j **>** size**)** **{**

**throw** **new** InvalidBoardIndexException**(**i**,**j**,**size**);**

**}**

setTile**(**i**,**j**,**0**);**

**}**

/\*\*

\* **@return** row number of empty tile (rows start with 1)

\*/

public int getEmptyTileRow**()** **{**

int retVal **=** **-**1**;**

**for(**int row **=** 1**;** row **<=** size**;** row**++)** **{**

**for(**int col **=** 1**;** col **<=** size**;** col**++)** **{**

int curVal **=** getTile**(**row**,** col**);**

**if(**curVal **==** 0**)** **{**

retVal **=** row**;**

**}**

**}**

**}**

**return** retVal**;**

**}**

/\*\*

\* **@return** column number of empty tile (columns start with 1)

\*/

public int getEmptyTileColumn**()** **{**

int retVal **=** **-**1**;**

**for(**int row **=** 1**;** row **<=** size**;** row**++)** **{**

**for(**int col **=** 1**;** col **<=** size**;** col**++)** **{**

int curVal **=** getTile**(**row**,** col**);**

**if(**curVal **==** 0**)** **{**

retVal **=** col**;**

**}**

**}**

**}**

**return** retVal**;**

**}**

/\*\*

\* **@return** number of rows / columns

\*/

public int size**()** **{**

**return** size**;**

**}**

// Überprüft, ob Position der Kacheln konsistent ist.

/\*\*

\* checks if a board is valid.

\* **@return** true if all numbers from 0 to size-1 is once in board (irrespective of order)

\*/

public boolean isValid**()** **{**

ArrayList**<**Integer**>** copy **=** **new** ArrayList**<>();**

copy**.**addAll**(**board**);**

Collections**.**sort**(**copy**);**

**for(**int i **=** 0**;** i **<** size**\***size**;** i**++)** **{**

int curVal **=** **(**Integer**)** copy**.**get**(**i**);**

**if(**curVal **!=** i**)** **{**

**return** **false;**

**}**

**}**

**return** **true;**

**}**

/\*\*

\* Makes a deep copy of the board and returns the new board

\* **@return** the copied board

\*/

public Board copy**()** **{**

Board copy **=** **new** Board**(**size**);**

**for(**int i **=** 0**;** i **<** size**\***size**;** i**++)** **{**

copy**.**board**.**set**(**i**,** board**.**get**(**i**));**

**}**

**return** copy**;**

**}**

/\*\*

\* Checks if a board is solvable

\* **@return** true if it is solvable, otherwise false

\*/

public boolean isSolvable**()** **{**

int inversions **=** 0**;**

**for(**int i **=** 0**;** i **<** board**.**size**()** **-** 1**;** i**++)** **{**

**for(**int j **=** i **+** 1**;** j **<** board**.**size**();** j**++)**

**if((**Integer**)** board**.**get**(**i**)** **>** **(**Integer**)** board**.**get**(**j**))** inversions**++;**

**if((**Integer**)** board**.**get**(**i**)** **==** 0 **&&** i **%** 2 **==** 1**)** inversions**++;**

**}**

/\*

If size is odd, then number of inversions must be even for board to be solvable.

If size is even, then number of inversions+row\_number\_of\_empty\_cell (starting with 1) must be odd for board to be solvable.

\*/

**if** **(**size **%** 2 **==** 1**)** **{**

**return** **(**inversions **%** 2 **==** 0**);**

**}** **else** **{**

int blankRow **=** getEmptyTileRow**();**

int test **=** inversions**+**blankRow**-**1**;**

**return** **(**test **%** 2 **==** 1**);**

**}**

**}**

/\*\*

\* Shuffles a board but guarantees that it is still solvable

\*/

public void shuffle**()** **{**

Collections**.**shuffle**(**board**);**

**while(!this.**isSolvable**())** **{**

Collections**.**shuffle**(**board**);**

**}**

**}**

/\*\*

\* Checks if a tile is a neighbour of the tile containing 0

\* **@param** row = row to be checked

\* **@param** col = column to be checked

\* **@return** true if the cell is a neighbour of the empty cell otherwise false

\* throws InvalidBoardIndexException if row or colum < 1 or > size

\*/

private boolean isNeighbourOfEmptyField**(**int row**,** int col**)** **{**

**if(**row **<** 1 **||** row **>** size **||** col **<** 1 **||** col **>** size**)** **{**

**throw** **new** InvalidBoardIndexException**(**row**,**col**,**size**);**

**}**

int rowEmpty **=** getEmptyTileRow**();**

int colEmpty **=** getEmptyTileColumn**();**

int rowOffset **=** row**-**rowEmpty**;**

int colOffset **=** col**-**colEmpty**;**

**if((**abs**(**rowOffset**)** **==** 0 **&&** abs**(**colOffset**)** **==** 1**)** **||**

**(**abs**(**rowOffset**)** **==** 1 **&&** abs**(**colOffset**)** **==** 0**))** **{**

**return** **true;**

**}** **else** **{**

**return** **false;**

**}**

**}**

/\*\*

\* Moves empty tile to this position. In fact the empty tile and the chosen tile will be exchanged.

\* Only works if empty tile and inputted tile are neighbours. Otherwise IllegalMoveException will be thrown

\* **@param** row = row which should be set to 0

\* **@param** col = column which should be set to 0

\*/

public void move**(**int row**,** int col**)** **{**

int rowEmpty **=** getEmptyTileRow**();**

int colEmpty **=** getEmptyTileColumn**();**

**if((**row **<** 1 **||** row **>** size **||** col **<** 1 **||** col **>** size**)**

**||** **(!**isNeighbourOfEmptyField**(**row**,** col**)))** **{**

**throw** **new** IllegalMoveException**(**row**,** col**,** rowEmpty**,** colEmpty**);**

**}**

int temp **=** getTile**(**row**,** col**);**

setTile**(**row**,** col**,** 0**);**

setTile**(**rowEmpty**,** colEmpty**,** temp**);**

**}**

/\*\*

\* Moves the empty tile to the left

\* throws IllegalMoveExcpetion if the column of the empty tile is already 1

\*/

public void moveLeft**()** **{**

int rowEmpty **=** getEmptyTileRow**();**

int colEmpty **=** getEmptyTileColumn**();**

**if(**colEmpty **<** 2**)** **{**

**throw** **new** IllegalMoveException**(**rowEmpty**,** colEmpty**-**1**,** rowEmpty**,** colEmpty**);**

**}** **else** **{**

move**(**rowEmpty**,** colEmpty**-**1**);**

**}**

**}**

/\*\*

\* Moves the empty tile to the right

\* throws IllegalMoveExcpetion if the column of the empty tile is already the same as size

\*/

public void moveRight**()** **{**

int rowEmpty **=** getEmptyTileRow**();**

int colEmpty **=** getEmptyTileColumn**();**

**if(**colEmpty **>** size**-**1**)** **{**

**throw** **new** IllegalMoveException**(**rowEmpty**,** colEmpty**+**1**,** rowEmpty**,** colEmpty**);**

**}** **else** **{**

move**(**rowEmpty**,** colEmpty**+**1**);**

**}**

**}**

/\*\*

\* Moves the empty tile up

\* throws IllegalMoveExcpetion if the row of the empty tile is already 1

\*/

public void moveUp**()** **{**

int rowEmpty **=** getEmptyTileRow**();**

int colEmpty **=** getEmptyTileColumn**();**

**if(**rowEmpty **<** 2**)** **{**

**throw** **new** IllegalMoveException**(**rowEmpty**-**1**,** colEmpty**,** rowEmpty**,** colEmpty**);**

**}** **else** **{**

move**(**rowEmpty**-**1**,** colEmpty**);**

**}**

**}**

/\*\*

\* Moves the empty tile down

\* throws IllegalMoveExcpetion if the row of the empty tile is already the same as size

\*/

public void moveDown**()** **{**

int rowEmpty **=** getEmptyTileRow**();**

int colEmpty **=** getEmptyTileColumn**();**

**if(**rowEmpty **>** size**-**1**)** **{**

**throw** **new** IllegalMoveException**(**rowEmpty**+**1**,** colEmpty**,** rowEmpty**,** colEmpty**);**

**}** **else** **{**

move**(**rowEmpty**+**1**,** colEmpty**);**

**}**

**}**

/\*\*

\* Peeks and returns potential new row number of empty tile if move would be made

\* **@param** m = the move to be made (up or down)

\* **@return** the row number of empty tile if move is made

\*/

int getNewRowAfterMove**(**Move m**)** **{**

int rowEmpty **=** getEmptyTileRow**();**

int rowTo **=** rowEmpty**;**

**switch(**m**)** **{**

**case** DOWN**:**

rowTo **+=** 1**;**

**break;**

**case** UP**:**

rowTo **-=** 1**;**

**break;**

**}**

**return** rowTo**;**

**}**

/\*\*

\* Peeks and returns potential new column number of empty tile if move would be made

\* **@param** m = the move to be made (left or right)

\* **@return** the column number of empty tile if move is made

\*/

int getNewColAfterMove**(**Move m**)** **{**

int colEmpty **=** getEmptyTileColumn**();**

int colTo **=** colEmpty**;**

**switch(**m**)** **{**

**case** LEFT**:**

colTo **-=** 1**;**

**break;**

**case** RIGHT**:**

colTo **+=** 1**;**

**break;**

**}**

**return** colTo**;**

**}**

/\*\*

\* Performs a sequence of move operations

\* **@param** moves = a list of moves

\* throw IllegalMOveException if the move can't be made because the empty tile cannot be moved in this direction anymore

\*/

public void makeMoves**(**List**<**Move**>** moves**)** **{**

**for(**int i **=** 0**;** i **<** moves**.**size**();** i**++)** **{**

Move m **=** moves**.**get**(**i**);**

int newRow **=** getNewRowAfterMove**(**m**);**

int newCol **=** getNewColAfterMove**(**m**);**

**if** **((**newRow **<** 1 **||** newRow **>** size **||** newCol **<** 1 **||** newCol **>** size**)** **||** **(!**isNeighbourOfEmptyField**(**newRow**,** newCol**)))** **{**

**throw** **new** IllegalMoveException**(**newRow**,** newCol**,** getEmptyTileRow**(),** getEmptyTileColumn**());**

**}**

**switch** **(**m**)** **{**

**case** UP**:**

moveUp**();**

**break;**

**case** DOWN**:**

moveDown**();**

**break;**

**case** LEFT**:**

moveLeft**();**

**break;**

**case** RIGHT**:**

moveRight**();**

**break;**

**}**

**}**

**}**

/\*\*

\* checks if a board is solved (means it is sorted but the 0 is on the last position)

\* **@return** true if board is solved

\*/

public boolean isSolved**()** **{**

**for(**int i **=** 0**,** shouldVal **=** 1**;** i **<** size**\***size**-**1**;** i**++,** shouldVal**++)** **{**

Integer isVal **=** **(**Integer**)**board**.**get**(**i**);**

**if(**shouldVal **!=** isVal**)** **{**

**return** **false;**

**}**

**}**

int last **=** **(**Integer**)** board**.**get**(**size**\***size**-**1**);**

**return** last **==** 0**;**

**}**

**}**

### Tests

//file BoardTest.java

package at**.**fhooe**.**swe4**.**slidingpuzzle**;**

**import** org**.**junit**.**jupiter**.**api**.**AfterEach**;**

**import** org**.**junit**.**jupiter**.**api**.**BeforeEach**;**

**import** org**.**junit**.**jupiter**.**api**.**Test**;**

**import** org**.**junit**.**jupiter**.**api**.**TestClassOrder**;**

**import** java**.**util**.**Arrays**;**

**import** java**.**util**.**List**;**

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

public class BoardTest **{**

private Board board**;**

@BeforeEach

void setUp**()** **{**

board **=** **new** Board**(**3**);**

**}**

@AfterEach

void tearDown**()** **{** board **=** **null;** **}**

@Test

public void simpleIsValidTest**()** **{**

Board board**;**

**try** **{**

board **=** **new** Board**(**3**);**

board**.**setTile**(**1**,** 1**,** 1**);**

board**.**setTile**(**1**,** 2**,** 2**);**

board**.**setTile**(**1**,** 3**,** 3**);**

board**.**setTile**(**2**,** 1**,** 4**);**

board**.**setTile**(**2**,** 2**,** 5**);**

board**.**setTile**(**2**,** 3**,** 6**);**

board**.**setTile**(**3**,** 1**,** 7**);**

board**.**setTile**(**3**,** 2**,** 8**);**

board**.**setTile**(**3**,** 3**,** 0**);**

assertTrue**(**board**.**isValid**());**

**}** **catch** **(**BoardException e**)** **{**

fail**(**"BoardException not expected."**);**

**}**

**}**

@Test

public void simpleIsNotValidTest**()** **{**

Board board**;**

**try** **{**

board **=** **new** Board**(**3**);**

board**.**setTile**(**1**,** 1**,** 1**);**

board**.**setTile**(**1**,** 2**,** 2**);**

board**.**setTile**(**1**,** 3**,** 3**);**

board**.**setTile**(**2**,** 1**,** 4**);**

board**.**setTile**(**2**,** 2**,** 5**);**

board**.**setTile**(**2**,** 3**,** 6**);**

board**.**setTile**(**3**,** 1**,** 7**);**

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

board**.**setTile**(**3**,** 3**,** 0**);**

assertTrue**(!** board**.**isValid**());**

**}** **catch** **(**BoardException e**)** **{**

fail**(**"BoardException not expected."**);**

**}**

**}**

@Test

public void simpleIsNotValidTest2**()** **{**

Board board**;**

**try** **{**

board **=** **new** Board**(**3**);**

board**.**setTile**(**1**,** 1**,** 8**);**

board**.**setTile**(**1**,** 2**,** 2**);**

board**.**setTile**(**1**,** 3**,** 0**);**

board**.**setTile**(**2**,** 1**,** 7**);**

board**.**setTile**(**2**,** 2**,** 5**);**

board**.**setTile**(**2**,** 3**,** 4**);**

board**.**setTile**(**3**,** 1**,** 3**);**

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

board**.**setTile**(**3**,** 3**,** 6**);**

assertTrue**(**board**.**isValid**());**

**}** **catch** **(**BoardException e**)** **{**

fail**(**"BoardException not expected."**);**

**}**

**}**

@Test

void testToString**()** **{**

assertEquals**(**"1\t2\t3\t\n4\t5\t6\t\n7\t8\t0\t\n"**,** board**.**toString**());**

**}**

@Test

void testEquals**()** **{**

Board b2 **=** **new** Board**(**3**);**

assertEquals**(**board**,**b2**);**

b2**.**setTile**(**2**,**1**,**7**);**

assertNotEquals**(**board**,**b2**);**

**}**

@Test

void compareTo**()** **{**

Board b2 **=** **new** Board**(**3**);**

Board b3 **=** **new** Board**(**4**);**

Board b4 **=** **new** Board**(**8**);**

assertEquals**(**0**,** board**.**compareTo**(**b2**));**

assertEquals**(-**1**,** board**.**compareTo**(**b3**));**

assertEquals**(**1**,** b3**.**compareTo**(**board**));**

assertEquals**(**0**,** board**.**compareTo**(**b4**));**

**}**

@Test

void getTile**()** **{**

assertThrows**(**InvalidBoardIndexException**.**class**,** **()** **->** board**.**getTile**(**2**,**7**));**

assertEquals**(**0**,** board**.**getTile**(**3**,**3**));**

assertEquals**(**1**,** board**.**getTile**(**1**,**1**));**

assertEquals**(**5**,** board**.**getTile**(**2**,**2**));**

**}**

@Test

void setTile**()** **{**

assertEquals**(**6**,** board**.**getTile**(**2**,**3**));**

board**.**setTile**(**2**,**3**,**7**);**

assertEquals**(**7**,** board**.**getTile**(**2**,**3**));**

assertThrows**(**InvalidBoardIndexException**.**class**,** **()** **->** board**.**setTile**(**4**,**5**,** 4**));**

assertThrows**(**InvalidTileNumberException**.**class**,** **()** **->** board**.**setTile**(**1**,**3**,** 99**));**

**}**

@Test

void setEmptyTile**()** **{**

assertEquals**(**0**,** board**.**getTile**(**3**,**3**));**

assertNotEquals**(**0**,** board**.**getTile**(**1**,**2**));**

board**.**setEmptyTile**(**1**,**2**);**

assertEquals**(**0**,** board**.**getTile**(**1**,**2**));**

**}**

@Test

void getEmptyTileRow**()** **{**

assertEquals**(**3**,** board**.**getEmptyTileRow**());**

board**.**setEmptyTile**(**1**,**2**);**

board**.**setTile**(**3**,**3**,** 2**);**

assertEquals**(**1**,** board**.**getEmptyTileRow**());**

**}**

@Test

void getEmptyTileColumn**()** **{**

assertEquals**(**3**,** board**.**getEmptyTileColumn**());**

board**.**setEmptyTile**(**1**,**2**);**

board**.**setTile**(**3**,**3**,** 2**);**

assertEquals**(**2**,** board**.**getEmptyTileColumn**());**

**}**

@Test

void size**()** **{**

assertEquals**(**3**,** board**.**size**());**

assertFalse**(**board**.**size**()** **!=** 3**);**

**}**

@Test

void copy**()** **{**

Board b2 **=** board**.**copy**();**

assertEquals**(**b2**,** board**);**

b2**.**setTile**(**1**,**3**,** 6**);**

assertNotEquals**(**b2**,** board**);**

assertTrue**(**board**.**getTile**(**1**,**3**)** **==** 3**);**

assertTrue**(**b2**.**getTile**(**1**,**3**)** **==** 6**);**

assertFalse**(**board**.**getTile**(**1**,**3**)** **==** 6**);**

assertFalse**(**b2**.**getTile**(**1**,**3**)** **==** 3**);**

**}**

@Test

void isSolvable**()** **{**

List**<**Integer**>** a1 **=** Arrays**.**asList**(**0**,**5**,**2**,**1**,**8**,**3**,**4**,**7**,**6**);**

List**<**Integer**>** a2 **=** Arrays**.**asList**(**4**,**1**,**2**,**5**,**8**,**3**,**7**,**0**,**6**);**

Board b2 **=** **new** Board**(**a1**,**3**);**

Board b3 **=** **new** Board**(**a2**,**3**);**

assertTrue**(**b2**.**isSolvable**());**

assertTrue**(**b3**.**isSolvable**());**

List**<**Integer**>** a3 **=** Arrays**.**asList**(**1**,**2**,**3**,**4**,**5**,**6**,**8**,**7**,**0**);**

List**<**Integer**>** a4 **=** Arrays**.**asList**(**1**,**5**,**0**,**3**,**2**,**8**,**4**,**6**,**7**);**

Board b4 **=** **new** Board**(**a3**,**3**);**

Board b5 **=** **new** Board**(**a4**,**3**);**

assertFalse**(**b4**.**isSolvable**());**

assertFalse**(**b5**.**isSolvable**());**

**}**

@Test

void shuffle**()** **{**

assertEquals**(**"1\t2\t3\t\n4\t5\t6\t\n7\t8\t0\t\n"**,** board**.**toString**());**

board**.**shuffle**();**

assertNotEquals**(**"1\t2\t3\t\n4\t5\t6\t\n7\t8\t0\t\n"**,** board**.**toString**(),**board**.**toString**());**

assertTrue**(**board**.**isSolvable**());**

assertTrue**(**board**.**isValid**());**

**}**

@Test

void move**()** **{**

board**.**move**(**3**,**2**);**

assertEquals**(**0**,** board**.**getTile**(**3**,**2**));**

assertEquals**(**8**,** board**.**getTile**(**3**,**3**));**

//index must be valid

assertThrows**(**IllegalMoveException**.**class**,** **()** **->** board**.**move**(**2**,**7**));**

//index must be neighbour of empty field (currently 3|2)

assertThrows**(**IllegalMoveException**.**class**,** **()** **->** board**.**move**(**1**,**2**));**

board**.**move**(**2**,**2**);**

assertEquals**(**0**,** board**.**getTile**(**2**,**2**));**

assertEquals**(**5**,** board**.**getTile**(**3**,**2**));**

**}**

@Test

void moveLeft**()** **{**

board**.**moveLeft**();**

assertEquals**(**0**,** board**.**getTile**(**3**,**2**));**

assertEquals**(**8**,** board**.**getTile**(**3**,**3**));**

board**.**moveLeft**();**

assertEquals**(**0**,** board**.**getTile**(**3**,**1**));**

assertEquals**(**7**,** board**.**getTile**(**3**,**2**));**

assertEquals**(**8**,** board**.**getTile**(**3**,**3**));**

//now can't move any further

assertThrows**(**IllegalMoveException**.**class**,** **()** **->** board**.**moveLeft**());**

**}**

@Test

void moveRight**()** **{**

//set start position of empty field

board**.**moveLeft**();**

board**.**moveLeft**();**

board**.**moveUp**();**

assertEquals**(**0**,** board**.**getTile**(**2**,**1**));**

assertEquals**(**8**,** board**.**getTile**(**3**,**3**));**

board**.**moveRight**();**

assertEquals**(**5**,** board**.**getTile**(**2**,**1**));**

assertEquals**(**0**,** board**.**getTile**(**2**,**2**));**

board**.**moveRight**();**

assertEquals**(**5**,** board**.**getTile**(**2**,**1**));**

assertEquals**(**6**,** board**.**getTile**(**2**,**2**));**

assertEquals**(**0**,** board**.**getTile**(**2**,**3**));**

//now can't move any further

assertThrows**(**IllegalMoveException**.**class**,** **()** **->** board**.**moveRight**());**

**}**

@Test

void moveUp**()** **{**

board**.**moveUp**();**

assertEquals**(**0**,** board**.**getTile**(**2**,**3**));**

assertEquals**(**6**,** board**.**getTile**(**3**,**3**));**

board**.**moveUp**();**

assertEquals**(**6**,** board**.**getTile**(**3**,**3**));**

assertEquals**(**3**,** board**.**getTile**(**2**,**3**));**

assertEquals**(**0**,** board**.**getTile**(**1**,**3**));**

//now can't move any further

assertThrows**(**IllegalMoveException**.**class**,** **()** **->** board**.**moveUp**());**

**}**

@Test

void moveDown**()** **{**

//set start position of empty field

board**.**moveUp**();**

board**.**moveUp**();**

board**.**moveLeft**();**

assertEquals**(**0**,** board**.**getTile**(**1**,**2**));**

assertEquals**(**6**,** board**.**getTile**(**3**,**3**));**

board**.**moveDown**();**

assertEquals**(**5**,** board**.**getTile**(**1**,**2**));**

assertEquals**(**0**,** board**.**getTile**(**2**,**2**));**

board**.**moveDown**();**

assertEquals**(**5**,** board**.**getTile**(**1**,**2**));**

assertEquals**(**8**,** board**.**getTile**(**2**,**2**));**

assertEquals**(**0**,** board**.**getTile**(**3**,**2**));**

//now can't move any further

assertThrows**(**IllegalMoveException**.**class**,** **()** **->** board**.**moveDown**());**

**}**

@Test

void makeMoves**()** **{**

board**.**moveLeft**();**

board**.**moveLeft**();**

board**.**moveUp**();**

board**.**moveUp**();**

board**.**moveRight**();**

board**.**moveRight**();**

Board b1 **=** **new** Board**(**3**);**

List**<**Move**>** m **=** Arrays**.**asList**(**Move**.**LEFT**,** Move**.**LEFT**,** Move**.**UP**,** Move**.**UP**,** Move**.**RIGHT**,** Move**.**RIGHT**);**

b1**.**makeMoves**(**m**);**

assertEquals**(**0**,** b1**.**getTile**(**1**,**3**));**

assertEquals**(**board**,** b1**);**

**}**

@Test

void isSolved**()** **{**

assertTrue**(**board**.**isSolved**());**

board**.**moveLeft**();**

board**.**moveUp**();**

assertFalse**(**board**.**isSolved**());**

**}**

**}**

**Ein Bild, das Text enthält.

Automatisch generierte Beschreibung**

## SearchNode

### Lösungsidee

Die Klasse SearchNode enthält eine Datenkomponente, die ein Board speichert. Darüber hinaus wird eine Referenz auf den predecessor SearchNode gespeichert sowie die Kosten bis zu dem Knoten und die geschätzten Kosten bis zum Zielzustand.

Im Konstruktor erhält die Klasse ein Board. Hier wird auch gleich die Methode calculateEstimatedCostsToTarget aufgerufen, welche die Kosten bis zur Zielposition berechnet und die Datenkomponente dementsprechend initialisiert. Die Komponenten für den Predecessor und die Kosten vom Start müssen durch Setter gesetzt werden.

Die Kosten bis zum Ziel werden wie beschrieben mit der Manhatten-Distanz berechnet. Dazu gibt es Hilfsfunktionen, die für eine Nummer als Input deren Zielreihe und Spalte berechnen. Durch die Unterschiede zur aktuellen Position lässt sich dadurch die Distanz abschätzen.

In der Methode „toMoves“ hantle ich mich über den Predecessor so lange die Liste nach vorne, bis es keinen Predecessor mehr gibt. In jedem Schritt berechne ich den gemachten Move (entsprechend der Veränderung von Zeile bzw. Spalte). Der Move wird zu einer Liste geaddet. Zum Schluss muss die Liste noch invertiert werden (da ich die Moves ja in verkehrter Reihenfolge geaddet habe).

Die restlichen Methoden dieser Klasse sind hauptsächlich Getter und Setter.

### Quellcode

//file: SearchNode.java

package at**.**fhooe**.**swe4**.**slidingpuzzle**;**

**import** java**.**util**.**ArrayList**;**

**import** java**.**util**.**Collections**;**

**import** java**.**util**.**List**;**

**import** java**.**util**.**Objects**;**

**import** static java**.**lang**.**Math**.**abs**;**

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

private Board boardConfig**;**

private SearchNode predecessor **=** **null;**

//costs=steps from start position to this current state

private int costsFromStart**;**

//cost from current position to target position

private int estimatedCostsToTarget**;**

/\*\*

\* Constructor taking a board configuration

\* The estimatedCostsToTarget will be calculated automatically

\* (as it can be calculated from the board configuration)

\* predecessor and costsFromStart must be "manually" set

\* **@param** board = input board

\*/

// Suchknoten mit Board-Konfiguration initialisieren.

public SearchNode**(**Board board**)** **{**

boardConfig **=** board**;**

estimatedCostsToTarget **=** calculateEstimatedCostsToTarget**();**

**}**

/\*\*

\* Getter for board

\* **@return** board

\*/

public Board getBoard**()** **{**

**return** boardConfig**;**

**}**

/\*\*

\* Getter for predecessor

\* **@return** predecessor

\*/

public SearchNode getPredecessor**()** **{**

**return** predecessor**;**

**}**

/\*\*

\* Setter for predecessor. Sets predecessor to input value

\* **@param** predecessor

\*/

public void setPredecessor**(**SearchNode predecessor**)** **{**

**this.**predecessor **=** predecessor**;**

**}**

/\*\*

\* Getter for costsFromStart

\* **@return** costsFromStart

\*/

public int costsFromStart**()** **{**

**return** costsFromStart**;**

**}**

/\*\*

\* Getter for estimatedCostsToTarget

\* **@return** estimatedCostsToTarget

\*/

public int estimatedCostsToTarget**()** **{**

**return** estimatedCostsToTarget**;**

**}**

/\*\*

\* Calculates the target row for a certain number/tile

\* **@param** number = number for which target row is calculated

\* **@return** the target row of the input number

\*/

private int getTargetRow**(**int number**)** **{**

**if(**number **%** boardConfig**.**size**()** **==** 0**)** **{**

**return** number **/** boardConfig**.**size**();**

**}** **else** **{**

**return** **(**number **/** boardConfig**.**size**())** **+** 1**;**

**}**

**}**

/\*\*

\* Calculates the target column for a certain number/tile

\* **@param** number = number for which target column is calculated

\* **@return** the target column of the input number

\*/

private int getTargetCol**(**int number**)** **{**

**if(**number **%** boardConfig**.**size**()** **==** 0**)** **{**

**return** boardConfig**.**size**();**

**}** **else** **{**

**return** number **%** boardConfig**.**size**();**

**}**

**}**

/\*\*

\* Calculates the estimatedCostsToTarget using the Manhatten distance

\* **@return** estimated costs to target

\*/

private int calculateEstimatedCostsToTarget**()** **{**

int heuristic **=** 0**;**

**for(**int row **=** 1**;** row **<=** boardConfig**.**size**();** row**++)** **{**

**for** **(**int col **=** 1**;** col **<=** boardConfig**.**size**();** col**++)** **{**

int curTile **=** boardConfig**.**getTile**(**row**,** col**);**

**if** **(**curTile **!=** 0**)** **{**

heuristic **+=** calculateManhatten**(**curTile**,** row**,** col**);**

**}**

**}**

**}**

**return** heuristic**;**

**}**

/\*\*

\* Calculates for a number and it's current row and column the Manhatten distance to it's target position

\* **@param** number = number for which distance should be calculated

\* **@param** currRow = the current row position of the number

\* **@param** currCol = the current column position of the number

\* **@return**

\*/

private int calculateManhatten**(**int number**,** int currRow**,** int currCol**)** **{**

int targetRow **=** getTargetRow**(**number**);**

int targetCol **=** getTargetCol**(**number**);**

int rowOffset **=** abs**(**currRow **-** targetRow**);**

int colOffset **=** abs**(**currCol **-** targetCol**);**

**return** rowOffset**+**colOffset**;**

**}**

/\*\*

\* Setter for costsFromStart

\* **@param** costsFromStart

\*/

public void setCostsFromStart**(**int costsFromStart**)** **{**

**this.**costsFromStart **=** costsFromStart**;**

**}**

/\*\*

\* **@return** sum of costsFromStart + estimatedCostsToTarget

\*/

public int estimatedTotalCosts**()** **{**

**return** costsFromStart **+** estimatedCostsToTarget**;**

**}**

/\*\*

\* Compares two nodes based on the boardConfiguration only (doesn't consider cost components of node)

\* **@param** other = the node with which this node is returned

\* **@return** true if nodes are the same, othewise false

\*/

public boolean equals**(**Object other**)** **{**

**if** **(this** **==** other**)** **return** **true;**

**if(**other **==** **null** **||** getClass**()** **!=** other**.**getClass**())** **return** **false;**

SearchNode o **=** **(**SearchNode**)** other**;**

**return** Objects**.**equals**(**boardConfig**,** o**.**boardConfig**);**

**}**

/\*\*

\* Compares nodes based on estimated total costs

\* **@param** other the object to be compared.

\* **@return** <1 if costs of this node are smaller than other, 0 if costs are the same; 1 if costs of this node are larger

\*/

public int compareTo**(**SearchNode other**)** **{**

**return** **this.**estimatedTotalCosts**()** **-** other**.**estimatedTotalCosts**();**

**}**

/\*\*

\* Gets a hashcode for a node based on the baordConfiguration (doesn't consider cost components)

\* **@return** hashcode

\*/

@Override

public int hashCode**()** **{**

**return** Objects**.**hash**(**boardConfig**);**

**}**

/\*\*

\* Returns a list of moves that were made from the start board to this current node

\* **@return** list of moves from start configuration to current configuration

\*/

public List**<**Move**>** toMoves**()** **{**

ArrayList**<**Move**>** movesUntilHere **=** **new** ArrayList**<>();**

SearchNode current **=** **this;**

**while(**current**.**predecessor **!=** **null)** **{**

Board curBoard **=** current**.**boardConfig**;**

Board prevBoard **=** current**.**predecessor**.**getBoard**();**

Move madeMove **=** calculateMove**(**curBoard**,** prevBoard**);**

movesUntilHere**.**add**(**madeMove**);**

current **=** current**.**predecessor**;**

**}**

Collections**.**reverse**(**movesUntilHere**);**

**return** movesUntilHere**;**

**}**

/\*\*

\* Calculates the move that was made between two board configurations

\* **@param** curBoard the current board configuration

\* **@param** prevBoard the previous board configuration (starting board before making the move)

\* **@return** the respective move (left, right, up, down) of null if the boards differ be more than one move

\*/

private Move calculateMove**(**Board curBoard**,** Board prevBoard**)** **{**

Move result **=** **null;**

int curBoardRow **=** curBoard**.**getEmptyTileRow**();**

int curBoardCol **=** curBoard**.**getEmptyTileColumn**();**

int prevBoardRow **=** prevBoard**.**getEmptyTileRow**();**

int prevBoardCol **=** prevBoard**.**getEmptyTileColumn**();**

int verticalMove **=** curBoardRow **-** prevBoardRow**;**

int horizontalMove **=** curBoardCol **-** prevBoardCol**;**

**if(**horizontalMove **!=** 0 **&&** verticalMove **!=** 0**)** **{**

**throw** **new** IllegalMoveException**(**curBoardRow**,** curBoardCol**,** prevBoardRow**,** prevBoardCol**);**

**}**

//must be left or right move

**if(**horizontalMove **!=** 0**)** **{**

**if(**horizontalMove **==** 1**)** **{**

result **=** Move**.**RIGHT**;**

**}** **else** **if(**horizontalMove **==** **-**1**)** **{**

result **=** Move**.**LEFT**;**

**}**

**}**

//must be up or down move

**if(**verticalMove **!=** 0**)** **{**

**if(**verticalMove **==** 1**)** **{**

result **=** Move**.**DOWN**;**

**}** **else** **if** **(**verticalMove **==** **-**1**)** **{**

result **=** Move**.**UP**;**

**}**

**}**

**return** result**;**

**}**

@Override

public String toString**()** **{**

**return** "SearchNode:\n" **+** boardConfig **+** "\n" **+**

"costs from start: " **+** costsFromStart **+** " | " **+**

"costs to target: " **+** estimatedCostsToTarget **+** "\n"**;**

**}**

**}**

### Tests

//file: SearchNodeTest.java

package at**.**fhooe**.**swe4**.**slidingpuzzle**;**

**import** org**.**junit**.**jupiter**.**api**.**AfterEach**;**

**import** org**.**junit**.**jupiter**.**api**.**BeforeEach**;**

**import** org**.**junit**.**jupiter**.**api**.**Test**;**

**import** java**.**util**.**ArrayList**;**

**import** java**.**util**.**Arrays**;**

**import** java**.**util**.**List**;**

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

public class SearchNodeTest **{**

private SearchNode node**;**

private Board board **=** **new** Board**(**3**);**

@BeforeEach

void setUp**()** **{**

node **=** **new** SearchNode**(**board**);**

**}**

@AfterEach

void tearDown**()** **{** node **=** **null;** **}**

@Test

public void simpleNodeTest**()** **{**

**try** **{**

Board board **=** **new** Board**(**3**);**

board**.**setTile**(**1**,** 1**,** 1**);**

board**.**setTile**(**1**,** 2**,** 2**);**

board**.**setTile**(**1**,** 3**,** 3**);**

board**.**setTile**(**2**,** 1**,** 4**);**

board**.**setTile**(**2**,** 2**,** 5**);**

board**.**setTile**(**2**,** 3**,** 6**);**

board**.**setTile**(**3**,** 1**,** 7**);**

board**.**setTile**(**3**,** 2**,** 8**);**

board**.**setTile**(**3**,** 3**,** 0**);**

SearchNode node **=** **new** SearchNode**(**board**);**

assertEquals**(**0**,** node**.**estimatedCostsToTarget**());**

board **=** **new** Board**(**3**);**

board**.**setTile**(**1**,** 1**,** 1**);**

board**.**setTile**(**1**,** 2**,** 2**);**

board**.**setTile**(**1**,** 3**,** 3**);**

board**.**setTile**(**2**,** 1**,** 4**);**

board**.**setTile**(**2**,** 2**,** 0**);**

board**.**setTile**(**2**,** 3**,** 6**);**

board**.**setTile**(**3**,** 1**,** 7**);**

board**.**setTile**(**3**,** 2**,** 8**);**

board**.**setTile**(**3**,** 3**,** 5**);**

node **=** **new** SearchNode**(**board**);**

assertEquals**(**2**,** node**.**estimatedCostsToTarget**());**

board **=** **new** Board**(**3**);**

board**.**setTile**(**1**,** 1**,** 1**);**

board**.**setTile**(**1**,** 2**,** 0**);**

board**.**setTile**(**1**,** 3**,** 3**);**

board**.**setTile**(**2**,** 1**,** 4**);**

board**.**setTile**(**2**,** 2**,** 5**);**

board**.**setTile**(**2**,** 3**,** 6**);**

board**.**setTile**(**3**,** 1**,** 7**);**

board**.**setTile**(**3**,** 2**,** 8**);**

board**.**setTile**(**3**,** 3**,** 2**);**

node **=** **new** SearchNode**(**board**);**

assertEquals**(**3**,** node**.**estimatedCostsToTarget**());**

**}**

**catch** **(**BoardException e**)** **{**

fail**(**"Unexpeced BoardException."**);**

**}**

**}**

@Test

void testEquals**()** **{**

Board b2 **=** **new** Board**(**3**);**

SearchNode n2 **=** **new** SearchNode**(**b2**);**

assertEquals**(**node**,** n2**);**

n2**.**getBoard**().**setTile**(**1**,**1**,**4**);**

assertNotEquals**(**node**,** n2**);**

**}**

@Test

void compareTo**()** **{**

Board b2 **=** **new** Board**(**3**);**

b2**.**moveLeft**();**

b2**.**moveUp**();**

b2**.**moveUp**();**

SearchNode n2 **=** **new** SearchNode**(**b2**);**

n2**.**setCostsFromStart**(**20**);**

node**.**setCostsFromStart**(**20**);**

//in n2 changes were made so more changes are needed. The cost of n2 should be higher (considering that the costs from start are the same

//20-23;

assertEquals**(-**3**,** node**.**compareTo**(**n2**));**

node**.**setCostsFromStart**(**100**);**

//100-23

assertEquals**(**77**,** node**.**compareTo**(**n2**));**

**}**

@Test

void toMoves**()** **{**

List**<**Move**>** moves **=** Arrays**.**asList**(**Move**.**LEFT**,** Move**.**UP**,** Move**.**UP**,** Move**.**RIGHT**,** Move**.**DOWN**,** Move**.**DOWN**);**

//creating data structure for test

//allocating some nodes and linking them via predecessor component

SearchNode current **=** node**;**

SearchNode theNewNode **=** **null;**

**for(**Move m **:** moves**)** **{**

Board copiedBoard **=** current**.**getBoard**().**copy**();**

**if(**m **==** Move**.**LEFT**)** **{**

copiedBoard**.**moveLeft**();**

**}** **else** **if(**m **==** Move**.**RIGHT**)** **{**

copiedBoard**.**moveRight**();**

**}** **else** **if** **(**m **==** Move**.**UP**)** **{**

copiedBoard**.**moveUp**();**

**}** **else** **{**

copiedBoard**.**moveDown**();**

**}**

theNewNode **=** **new** SearchNode**(**copiedBoard**);**

theNewNode**.**setPredecessor**(**current**);**

current **=** theNewNode**;**

**}**

//I'm standing now at "theNewNode". That's the last one in the list (that is linked via predecessors)

//For this node I'm now calling toMoves to get all previous moves that led to this state

//The result should be the same as the List "moves" defined in the beginning

List**<**Move**>** madeMoves **=** theNewNode**.**toMoves**();**

assertEquals**(**madeMoves**,** moves**);**

**}**

/\*

remaining methods are just getter and setters. So I'm not testing them in detail

\*/

**}**

**Ein Bild, das Text enthält.

Automatisch generierte Beschreibung**

## SlidingPuzzle

### Lösungsidee

Für die closedList verwende ich ein HashSet und kein TreeSet. Ich habe erstens nicht ganz verstanden, warum das sortiert sein müsste. Außerdem hatte ich ziemliche Laufzeitprobleme beim HashSet. Da hat der Test mit den Random 40 Puzzeln bei mir eine Minute gedauert. Eventuell fällt euch (@Tutor/Reviewer) auf, woran das liegen hätte können.

Nachdem die beiden Listen static Klassenkomponenten sind müssen sie bei jedem Aufruf von solve zuerst bereinigt werden. Dann wird ein Knoten erstellt und mit dem übergebenen Board initialisiert. Die Kosten vom Start setze ich für diesen Knoten auf 0. Die estimated costs to target werden im Konstruktor für den Node berechnet. Der Knoten wird der openList hinzugefügt.

Nun kann die Schleife gestartet werden. Aus der Priority Queue (openList) wird immer der Knoten mit den geringsten Kosten entnommen. Falls dieser Knoten die Lösung darstellt, ist der Algo fertig. Falls nicht, wird der Knoten zur closedList hinzugefügt. Im Anschluss werden die bis zu vier Folgezustände berechnet. Jene Folgezustände, die noch nicht auf der closedList sind, werden zur openList hinzugefügt, um dann in den weiteren Schleifendurchgängen abgearbeitet werden zu können. Ein Update der Knoten in der openList ist in unserem Fall nicht notwendig, da die Kanten keine Gewichte haben (sondern konstant 1 beträgt).

### Quellcode

package at**.**fhooe**.**swe4**.**slidingpuzzle**;**

**import** java**.**util**.\*;**

public class SlidingPuzzle **{**

private static Queue**<**SearchNode**>** openList **=** **new** PriorityQueue**<>();**

private static Set**<**SearchNode**>** closedList **=** **new** HashSet**<>();**

/\*\*

\* Solves a puzzle and returns a list of moves necessary to solve it

\* **@param** board the board to be solved

\* **@return** the list of moves to be made to solve the board

\* throws NoSolutionException

\*/

public static List**<**Move**>** solve**(**Board board**)** **{**

openList**.**clear**();**

closedList**.**clear**();**

**if(!**board**.**isSolvable**())** **{**

**throw** **new** NoSolutionException**();**

**}**

Queue**<**SearchNode**>** openList **=** **new** PriorityQueue**<>();**

Set**<**SearchNode**>** closedList **=** **new** HashSet**<>();**

SearchNode startNode **=** **new** SearchNode**(**board**);**

startNode**.**setPredecessor**(null);**

startNode**.**setCostsFromStart**(**0**);**

openList**.**add**(**startNode**);**

**while(!**openList**.**isEmpty**())** **{**

SearchNode currentNode **=** openList**.**poll**();**

**if(**currentNode**.**getBoard**().**isSolved**())** **{**

**return** currentNode**.**toMoves**();**

**}**

closedList**.**add**(**currentNode**);**

List**<**SearchNode**>** successors **=** calculateSuccessors**(**currentNode**);**

**for(**SearchNode succ**:**successors**)** **{**

**if(!**closedList**.**contains**(**succ**))** **{**

openList**.**add**(**succ**);**

**}**

**}**

**}**

**return** **null;**

**}**

/\*\*

\* Makes up to four possible moves from a certain starting point (left, right, up, down)

\* For each new state a node is created and linked to the predecessor

\* **@param** currentNode = the starting state from which the next possible states are created

\* **@return** a list of the next states

\*/

private static List**<**SearchNode**>** calculateSuccessors**(**SearchNode currentNode**)** **{**

List**<**Move**>** possibleMoves **=** Arrays**.**asList**(**Move**.**LEFT**,** Move**.**RIGHT**,** Move**.**UP**,** Move**.**DOWN**);**

ArrayList**<**SearchNode**>** successors **=** **new** ArrayList**<>();**

**for(**Move m**:**possibleMoves**)** **{**

SearchNode nextSuccessor **=** calculateSuccessor**(**currentNode**,** m**);**

**if(**nextSuccessor **!=** **null)** **{**

successors**.**add**(**nextSuccessor**);**

**}**

**}**

**return** successors**;**

**}**

/\*\*

\* Creates a new node for one particular move (m) based on the current search node

\* **@param** currentNode = the current state

\* **@param** m = the move to be made

\* **@return** a new searchNode with the new board configuration and costs or null if the move can't be made

\*/

private static SearchNode calculateSuccessor**(**SearchNode currentNode**,** Move m**)** **{**

Board nextBoard **=** currentNode**.**getBoard**().**copy**();**

**try** **{**

**switch** **(**m**)** **{**

**case** RIGHT**:**

nextBoard**.**moveRight**();**

**break;**

**case** LEFT**:**

nextBoard**.**moveLeft**();**

**break;**

**case** UP**:**

nextBoard**.**moveUp**();**

**break;**

**case** DOWN**:**

nextBoard**.**moveDown**();**

**break;**

**}**

//estimatedCostsToTarget are set automatically

SearchNode successorNode **=** **new** SearchNode**(**nextBoard**);**

successorNode**.**setPredecessor**(**currentNode**);**

successorNode**.**setCostsFromStart**(**currentNode**.**costsFromStart**()+**1**);**

**return** successorNode**;**

**}** **catch** **(**IllegalMoveException e**){**

**return** **null;**

**}**

**}**

/\*\*

\* Prints the sequence of boards that result in executing all the moves

\* **@param** board = the start board

\* **@param** moves = the list of moves that will be applied to the start board

\*/

public static void printMoves**(**Board board**,** List**<**Move**>** moves**)** **{**

System**.**out**.**println**(**board**.**toString**());**

**for** **(**Move m**:**moves**)** **{**

**switch(**m**)** **{**

**case** DOWN**:**

board**.**moveDown**();**

**break;**

**case** UP**:**

board**.**moveUp**();**

**break;**

**case** LEFT**:**

board**.**moveLeft**();**

**break;**

**case** RIGHT**:**

board**.**moveRight**();**

**break;**

**}**

System**.**out**.**println**(**board**.**toString**());**

**}**

**}**

**}**

### Tests

package at**.**fhooe**.**swe4**.**slidingpuzzle**;**

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

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

**import** java**.**util**.\*;**

**import** org**.**junit**.**jupiter**.**api**.**Test**;**

public class SlidingPuzzleSolverTest **{**

@Test

public void solveSimplePuzzleTest1**()** **{**

**try** **{**

Board board **=** **new** Board**(**3**);**

board**.**setTile**(**1**,** 1**,** 1**);**

board**.**setTile**(**1**,** 2**,** 2**);**

board**.**setTile**(**1**,** 3**,** 3**);**

board**.**setTile**(**2**,** 1**,** 4**);**

board**.**setTile**(**2**,** 2**,** 5**);**

board**.**setTile**(**2**,** 3**,** 6**);**

board**.**setTile**(**3**,** 1**,** 7**);**

board**.**setEmptyTile**(**3**,** 2**);**

board**.**setTile**(**3**,** 3**,** 8**);**

List**<**Move**>** moves **=** SlidingPuzzle**.**solve**(**board**);**

assertEquals**(**1**,** moves**.**size**());**

assertEquals**(**Move**.**RIGHT**,** moves**.**get**(**0**));**

**}** **catch** **(**NoSolutionException nse**)** **{**

fail**(**"NoSolutionException is not expected."**);**

**}**

**}**

@Test

public void solveSimplePuzzleTest2**()** **{**

**try** **{**

Board board **=** **new** Board**(**3**);**

board**.**setTile**(**1**,** 1**,** 1**);**

board**.**setTile**(**1**,** 2**,** 2**);**

board**.**setTile**(**1**,** 3**,** 3**);**

board**.**setTile**(**2**,** 1**,** 4**);**

board**.**setTile**(**2**,** 2**,** 5**);**

board**.**setTile**(**2**,** 3**,** 6**);**

board**.**setEmptyTile**(**3**,** 1**);**

board**.**setTile**(**3**,** 2**,** 7**);**

board**.**setTile**(**3**,** 3**,** 8**);**

List**<**Move**>** moves **=** SlidingPuzzle**.**solve**(**board**);**

assertEquals**(**2**,** moves**.**size**());**

assertEquals**(**Move**.**RIGHT**,** moves**.**get**(**0**));**

assertEquals**(**Move**.**RIGHT**,** moves**.**get**(**1**));**

**}** **catch** **(**NoSolutionException nse**)** **{**

fail**(**"NoSolutionException is not expected."**);**

**}**

**}**

@Test

public void solveComplexPuzzleTest1**()** **{**

**try** **{**

// 8 2 7

// 1 4 6

// 3 5 X

Board board **=** **new** Board**(**3**);**

board**.**setTile**(**1**,** 1**,** 8**);**

board**.**setTile**(**1**,** 2**,** 2**);**

board**.**setTile**(**1**,** 3**,** 7**);**

board**.**setTile**(**2**,** 1**,** 1**);**

board**.**setTile**(**2**,** 2**,** 4**);**

board**.**setTile**(**2**,** 3**,** 6**);**

board**.**setTile**(**3**,** 1**,** 3**);**

board**.**setTile**(**3**,** 2**,** 5**);**

board**.**setEmptyTile**(**3**,** 3**);**

List**<**Move**>** moves **=** SlidingPuzzle**.**solve**(**board**);**

board**.**makeMoves**(**moves**);**

assertEquals**(new** Board**(**3**),** board**);**

**}**

**catch** **(**NoSolutionException nse**)** **{**

fail**(**"NoSolutionException is not expected."**);**

**}**

**}**

@Test

public void solveRandomPuzzlesTest**()** **{**

**for** **(**int k **=** 0**;** k **<** 40**;** k**++)** **{**

**try** **{**

Board board **=** **new** Board**(**3**);**

int n **=** 1**;**

int maxN **=** board**.**size**()** **\*** board**.**size**();**

**for** **(**int i **=** 1**;** i **<=** board**.**size**();** i**++)**

**for** **(**int j **=** 1**;** j **<=** board**.**size**();** j**++)**

board**.**setTile**(**i**,** j**,** **(**n**++)** **%** maxN**);**

board**.**shuffle**();**

Board shuffledBoard **=** board**.**copy**();**

List**<**Move**>** moves **=** SlidingPuzzle**.**solve**(**board**);**

board**.**makeMoves**(**moves**);**

assertEquals**(new** Board**(**3**),** board**);**

Board freshBoard **=** **new** Board**(**3**);**

**if(**freshBoard **!=** board**)** **{**

shuffledBoard**.**toString**();**

**}**

**}** **catch** **(**NoSolutionException nse**)** **{**

fail**(**"NoSolutionException is not expected."**);**

**}**

**}**

**}**

@Test

public void solveSimplePuzzleTest\_4x4**()** **{**

**try** **{**

Board board **=** **new** Board**(**4**);**

board**.**moveLeft**();**

List**<**Move**>** moves **=** SlidingPuzzle**.**solve**(**board**);**

assertEquals**(**1**,** moves**.**size**());**

assertEquals**(**Move**.**RIGHT**,** moves**.**get**(**0**));**

**}**

**catch** **(**NoSolutionException nse**)** **{**

fail**(**"NoSolutionException is not expected."**);**

**}**

**}**

@Test

public void solveComplexPuzzleTest\_4x4**()** **{**

**try** **{**

Board board **=** **new** Board**(**4**);**

board**.**moveLeft**();**

board**.**moveLeft**();**

board**.**moveUp**();**

board**.**moveLeft**();**

board**.**moveUp**();**

board**.**moveUp**();**

board**.**moveRight**();**

board**.**moveDown**();**

board**.**moveLeft**();**

List**<**Move**>** moves **=** SlidingPuzzle**.**solve**(**board**);**

board**.**makeMoves**(**moves**);**

assertEquals**(new** Board**(**4**),** board**);**

**}**

**catch** **(**NoSolutionException nse**)** **{**

fail**(**"NoSolutionException is not expected."**);**

**}**

**}**

@Test

void printMoves**()** **{**

Board b **=** **new** Board**(**3**);**

List**<**Move**>** movesIn **=** Arrays**.**asList**(**Move**.**LEFT**,** Move**.**UP**,** Move**.**UP**,** Move**.**RIGHT**,** Move**.**DOWN**,** Move**.**DOWN**);**

b**.**makeMoves**(**movesIn**);**

List**<**Move**>** movesOut **=** SlidingPuzzle**.**solve**(**b**);**

SlidingPuzzle**.**printMoves**(**b**,**movesOut**);**

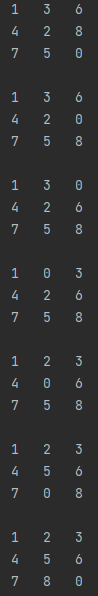
**}**

**}**

**Ein Bild, das Text enthält.

Automatisch generierte Beschreibung**

**Output of printMoves() Test**

****

1. <https://www.cs.princeton.edu/courses/archive/spring19/cos226/assignments/8puzzle/specification.php> [↑](#footnote-ref-1)