Assignment Sliding Game

Object Orientation

Spring 2020

1 Goals

In this assignment you are implementing an algorithm that finds a solution of an N by N sliding game. After completing this exercise you should be able to:

- Use lists and queues to implement a search algorithm
- Traverse a search space using breadth-first search
- Use the best-first optimisation of breadth-first search
- Redefine and use the hash method
- Use priority queues from the Java standard library

2 Sliding Games

A standard sliding game has $N \times N$ squares with pieces numbered from 1 to $N^2 - 1$. The last square is left blank, so you can move the adjacent pieces into the resulting gap. The goal of the sliding game is to convert a random initial combination to a final configuration where all the pieces are in ascending order by sliding pieces around. Figure 1 shows a solved 4×4 game.

In every state of the game, there are at most four legal moves. We can indicate these moves by the direction the gap moves: *north*, *east*, *south* and *west*. The example configuration in fig. 1 has only two legal moves. The gap can be shifted *north* or *west*.

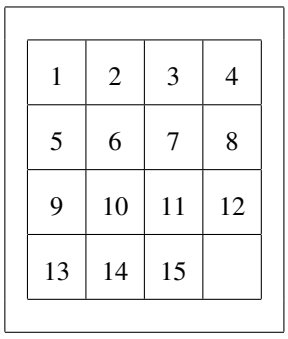

Figure 1: A solved 4×4 sliding game.

3 Instructions

Sketch a breadth-first search algorithm on paper before you actually write and test your code. It is often handy to use small examples during code development. Start with a 2×2 or 3×3 board with an almost completed set up.

4 Problem sketch

You need a *queue* to implement breadth-first search. This data type has the characteristic feature that the elements that stand in the queue the longest are released first – just like the queue at a supermarket check out. New candidates are placed at the end of the queue and need to wait their turn.

The queue in this assignment contains game configurations. To solve a sliding puzzle your algorithm should go on as follows. If the queue is empty, the puzzle is unsolvable. If the queue is not empty, take out the first element. We call this the *current configuration*. If the current configuration is identical to the solution, the puzzle is solved. Otherwise, generate all immediate successors of the current configuration and enqueue them.

This algorithm does not only work for sliding games, but for many other games that involve search strategies. Therefore we create an interface which our sliding puzzle implements.

```
public interface Configuration
  extends Comparable<Configuration> {
    public abstract Configuration getParent();

    public abstract Collection<Configuration> successors();

    public abstract boolean isSolution();

    public default List<Configuration> pathFromRoot(){
        throw new UnsupportedOperationException(
          "pathFromRoot: not supported yet." );
    }
}
```

We will discuss in section 5.6 why this interface implements `Comparable`. For now you should let NetBeans generate the default implementation. Do so by using Source / Insert Code / Implement Method / Comparable.compareTo. The code NetBeans generates just throws an UnsupportedOperationException, which is fine for now.

To represent the sliding game itself, make a class `SlidingGame`. The configuration of a game is represented by a two-dimensional array of integers. The position of the gap is stored explicitly so we do not have to search through the array every time.

The class we provided is far from being complete. You have to add a number of methods and attributes to get the program working.

```
public class SlidingGame implements Configuration {
    public static final int N = 3, SIZE = N*N, HOLE = SIZE;
    private int[][] board;
    private int holeX, holeY;
```

```
    public SlidingGame( int[] start ) {
        board = new int[N][N];

        assert start.length == N*N: "Length of specified board
            incorrect";

        for( int p = 0; p < start.length; p++ ) {
            board[p % N][p / N] = start[p];
            if ( start[p] == HOLE ) {
                holeX = p % N;
                holeY = p / N;
            }
        }
    }

    @Override
    public String toString() { ... }

    @Override
    public boolean equals(Object o) { ... }

    @Override
    public boolean isSolution () { ... }

    @Override
    public Collection<Configuration> successors () { ... }

    @Override
    public Configuration getParent() { ... }
}
```

The breadth-first solver should look as follows.

```
public class Solver {
    private Queue<Configuration> toExamine;
    private Collection<Configuration> encountered;

    public Solver( Configuration g ) {
        ...
    }

    public String solve() {
        while ( ! toExamine.isEmpty() ) {
            Configuration current = toExamine.remove();
            if ( current.isSolution() ) {
               return "Success!";
            } else {
               for(Configuration succ: current.successors())
               {
                   toExamine.add(succ);
               }
            }
```

```
        }
        return "Failure!";
    }

}
```

It does not make sense to revisit configurations that have already been visited. Adding such a configuration to the queue again will not give a shorter solution. It might even lead to an infinite loop. To prevent that, your program should store all solutions it has seen, and check before queueing successors.

5 Improving the Solver

Complete the program in the way described below.

5.1 The SlidingGame Class

Complete the class `SlidingGame`. Since this class is an extension of `Configuration`, you will need to implement at least the following methods.

- `isSolution`: indicates whether the configuration is solved.
- `getParent`: returns the predecessor of the current configuration. Hint: add an attribute to the class to keep track of the predecessor.
- `successors`: returns the direct successors of the current configuration. Store them in any datatype that implements the `Collection` interface, like a `LinkedList` or an `ArrayList`.

5.2 The Path to the Solution

The output of the solver above is rather disappointing. All we get to see is whether the game can be solved or not. Instead we want to know which pieces to move to arrive at the solved configuration! In order to present such a solution to the user, your program needs to trace back the direct predecessor of each intermediate configuration. Use the `getParent` of a `Configuration` for this purpose.

In order to reconstruct the path from the initial configuration to the current state, implement the following method in `Configuration`.

```
public default List<Configuration> pathFromRoot()
```

5.3 Observed States

To prevent repetitions during the search, your algorithm need to keep track of the states already visited. The easiest way to do that is to store them in a list. To keep this part of your program flexible, use the `Collection` interface. Use the `contains` method to determine if an element is present in the list. Use `add` to add an element to the list.

Storing all states requires a lot of memory. If while running your program you get an error that the entire heap space is used, you might have made a mistake that led to an infinite search loop. If your program is correct and you still get a memory error, you

can assign more memory to your program by running it with the option `-Xmx1g` . In NetBeans you can add this option under File / Project Properties / Run / VM Options. With this flag, NetBeans starts your program with a maximum heap space of 1 GB.

5.4 A Basic Solution

Use the given ingredients to write a Java program that tries to solve a given puzzle. Once a solution is found, the program should display the path of all intermediate configurations that lead to the solution in the right order.

Not all games are solvable. By swapping two adjacent pieces, a configuration changes its *parity* and can never be recovered by moving pieces. In the case of an unsolvable game, your program should print a message accordingly.

5.5 Handling Intermediate Solutions More Efficiently

The number of solvable configurations for an $n \times n$ puzzle is $(n^2)!/2$. Even for a modest 3×3 puzzle, this adds up to a total of 181440 configurations. Before your program concludes that a given puzzle is unsolvable, it must create a list of all reachable configurations. For every new configuration, your program checks if it has already been visited. Checking if an element is in a list requires iteration over the entire list, which has complexity $O(n)$.

Checking for already visited configurations can be sped up by using a *hash table*. A hash table stores elements based on a *hash value*. The hash value is used to look up the configuration in the hash table. A hash table lookup has complexity $O(1)$.

It is no problem if a small number of different configurations have an identical hash value. The hash table takes care of this without significantly slowing down the lookup.

The Java standard library has different kinds of hash tables. You should use a `HashSet`. In order to store configurations in a `HashSet`, you have to implement the `hashCode` method. This method is part of the class `Object`, and hence every class inherits it.

The method `hashCode` has to return an `int`, subject to the following restriction. If two objects are equal according to the method `equals`, they must have the same hash code. The converse is not required: Two unequal objects may have the same hash code.

You should use the locations and the values of the pieces of a configuration to compute a hash value. You can use the following formula.

$$\text{hashCode} = \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} \text{board}[x][y] \cdot 31^{y+x \cdot N}$$

`HashSet` implements the `Collection` interface. Apart from using `HashSet` instead of List, and implementng `hashCode`, you will need to change almost nothing else in the structure of your program.

5.6 A More Effective Search Algorithm

If you implement the algorithm described above, you will notice that the capabilities of the solver are rather small. As soon as your program gets fed a bigger puzzle, it will use too much time and memory to find a solution.

A simple improvement would be to use best-first search that examines the best successor states first. We can easily achieve this by placing the best elements at the front of the queue.

However, we do not actually know the best successor states until we completely solved the puzzle. We can use a heuristic to estimate the goodness of a configuration. The *Manhattan distance* is the number of squares a piece has to cover vertically and horizontally to get to a desired location, in our case its place in the end configuration. By summing up the Manhattan distances of all pieces, you get an estimation for how far a configuration is away from the solution. You should store the Manhattan distance for every configuration, so that it does not have to be recalculated over and over.

To implement the best-first search strategy, you have to replace the generic queue by a `PriorityQueue`. This data structure places the best elements in front. To determine the position for a new element, `PriorityQueue` uses the method `compareTo` from the `Comparable<T>` interface. This explains why `Configuration` extends the `Comparable <Configuration>` interface. For the sliding game, you should make `compareTo` return the difference in Manhattan distance.

The following properties should always hold for an implementation of `compareTo`.

- `sgn(x.compareTo(y))== -sgn(y.compareTo(x))` for all `x` and `y`. The function `sgn` is the *sign function*. It returns 0 if the argument is zero, 1 if the argument is greater than zero, -1 if the argument is less than zero.
- The comparison should be transitive. This means for example that if `x.compareTo(y)>0` and `y.compareTo(z)>0` then `x.compareTo(z)>0`.
- If `x.equals(y)` then `x.compareTo(y)==0`.
  The converse is not required. For example two different configurations can have the same Manhattan distance.

6 Your Tasks

- Complete the `Solver` found in the project template in two steps.
  - It should skip configurations that it has encountered by storing them in a List.
  - Once a solution has been found, it should print all configurations from the start to the solution.
  - Implement `hashCode` in `SlidingGame` and let `Solver` store encountered configurations in a HashSet instead of a List.
- Improve `Solver` to use a `PriorityQueue` for `toExamine`.
  - In the constructor of `SlidingGame`, calculate the manhattan distance of this configuration to the solution.
  - Implement `compareTo` of `SlidingGame` by using the manhattan distance.
  - This allows you to store configurations in a `PriorityQueue`.
  - Hint: You only have to change one line in `Solver`!

7 Supplementary Files

On Brightspace you can find a project template to start this assignment. The project template comes with some test cases you can use to verify your implementation.

8 Submit Your Project

To submit your project, follow these steps.

1. Use the **project export** feature of NetBeans to create a zip file of your entire project: File → Export Project → To ZIP.
2. **Submit this zip file on Brightspace**. Do not submit individual Java files. Do not submit any other archive format. Only one person in your group has to submit it. Submit your project before the deadline, which can be found on Brightspace.