Cmpe 160 - Introduction to Object Oriented Programming

Project III - N-Puzzle

Instructor: Tuna Tuğcu
TA: Yiğit Yıldırım
Contact: karahan.saritas@boun.edu.tr

Deadline: June 12, 2022 23.55

1 Introduction

N-puzzle is a popular puzzle game that consists of N tiles where N can have different values such as 8, 15, 24 and so on. The puzzle is divided into $\sqrt{N+1}$ rows and $\sqrt{N+1}$ columns. It consists of one empty space where the tiles can be moved and thus the puzzle is solved when a particular goal pattern is formed.
There are several algorithms developed for N-puzzle problems which can be categorized as *Uninformed Search Algorithms* and *Informed Search Algorithms*. *Uninformed Search Algorithms* include

- Breadth-first Search
- Depth-first Search
- Iterative Deepening Search

some of which, we are familiar from tree algorithms. *Informed Search Algorihms* include

- A* Search
- Greedy Search

To solve this problem, we will be using **A* Search** algorithm, which is a general artificial intelligence methodology to solve similar problems. *Informed Search Algorithms*, as the name suggest, must be informed about the current state by some parameter, which is called the **heuristic**. A **heuristic**, also called heuristic function, is a function that ranks alternatives in search algorithms at each branching step.
In the following sections, first we'll explain A* Search Algorithm and our Heuristic, Taxicab Distance. Then we'll be giving some details about the implementation.

2 A* Search Algorithm

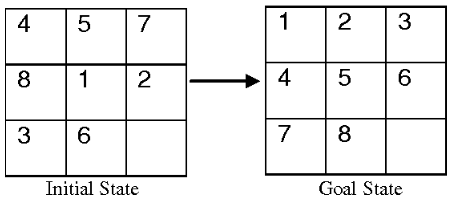
A* algorithm is a searching algorithm that searches for the shortest path between the *initial* and the *final* state. A* algorithm has 3 parameters:

- **g:** the cost of moving from initial cell to the current cell, which can be expressed as the *cost so far*.
- **h:** heuristic value, that is, the estimated cost of moving from current cell to the final cell.
- **f**: sum of **g** and **h**.

Ok, but what do they correspond to in our N-puzzle problem? Well, **g**, obviously, corresponds to the number of state transitions done so far for any state. **h** corresponds to the **Taxicab Distance** [2] between the current state and the final state (goal). **f**, again corresponds to $h + g$.

2.1 Taxicab Distance

Taxicab Distance between a state and the goal state is the sum of the vertical and horizontal distances from the tiles to their goal positions. For an example, examine the current state and goal state given below [1]:



Let's calculate the Taxicab distance between each tile and their goal positions:

for tile 4 $\rightarrow$ 1 (1 row)
for tile 5 $\rightarrow$ 1 (1 row)
for tile 7 $\rightarrow$ 4 (2 rows + 2 columns)
for tile 8 $\rightarrow$ 2 (1 row + 1 column)
for tile 1 $\rightarrow$ 2 (1 row + 1 column)
for tile 2 $\rightarrow$ 2 (1 row + 1 column)
for tile 3 $\rightarrow$ 4 (2 rows + 2 columns)
for tile 6 $\rightarrow$ 2 (1 row + 1 column)

So, in total, Taxicab distance between our current state and the goal state is 18. Let's assume we reached to the current state from the initial state by 5 state transitions. Therefore, for this state equation becomes: $f = g + h = 5 + 18 = 23$. Taxicab distance is an easy-to-use distance metric that is used in Artificial Intelligence algorithms along with other distance metrics such as Euclidean distance and Hamming distance.

3 Implementation

The project consists of two classes, namely Puzzle.java and Solver.java. In *Puzzle.java* we will be representing the current state of the game, and in *Solver.java* we will be solving the puzzle using A* search algorithm.

3.1 Puzzle.java

This class represents a particular state of the board. The following field should be defined for this class:

1. `int[][] tile`: In this field we store the state of the board in a 2D array. Initialize this field as *private* so that other classes cannot reach this field.

You can create additional *private* fields as you want. But make sure that they are *private*, not reachable from other classes.
Following *public* methods should be implemented. You are allowed to create additional helper *private* methods if you need.

1. `Puzzle(int[][] tiles)`: Constructor of the *Puzzle* class. Fill up the `int[][] tile` field according to the given 2D array. Empty tile will ve represented with 0. **Do not** assign the given parameter to the field directly. Doing such is a bad practice because in general, we can't guarantee that given parameter won't be modified by some other classes in our project. Copy the content of the given array to `int[][] tile`. This practice makes our class Immutable. The immutable objects are objects whose value can not be changed after initialization.
2. `int h()`: This function is used to calculate **Heuristic value**, that is, Taxicab distance between this particular Puzzle object (current state) and the goal state. Within this function, calculate the vertical and horizontal distances between each tile and goal positions. Return the summation of distances. We will be using this information for our algorithm. In our goal state, numbers should be in ascending order from left to right and from up to bottom. Empty tile should be at the right-bottom of the puzzle.
3. `boolean isCompleted()`: Return *true* if current state is the same as the goal state, otherwise *false*.
4. `Iterable<Puzzle> getAdjacents()`: This function should return all adjacent states to the current state of the puzzle. An adjacent state is another

puzzle object we reach when the position of the empty tile is changed by one row or column. Examine the current `int[][] tile`, and create new *Puzzle* objects that are adjacent to the current state. Do not modify the `int[][] tile` of the current *Puzzle*. This function will be very useful when we explore each branching step in the search algorithm.

5. `String toString()`: Override the *toString()* method. It will be used to print out the *Puzzle* objects to the output file. You should append " " between the tiles and "\n" after each row including the last row.

3.2 Solver.java

In this class we will be solving the puzzle using the algorithm described above. We'll also have a *main* method to test our program. You can create any number of *private* fields and methods.

As we discussed earlier, we have to store $f = g + h$ information for each *Puzzle* objects. During the search algorithm, you can either use a PriorityQueue or a Stack to store each adjacent state in each step. However, storing the adjacents in a stack may become very inefficient for large input sizes and *PriorityQueue* implementation may reduce the execution time drastically. However, we have to somewhat "order" *Puzzle* objects we store in our *PriorityQueue* so that, `pq.peek()` returns "minimum" *Puzzle* according to this "order". In order to do so we'll create a private subclass within the *Solver* class named as *PriorityObject*, and this class will have another subclass that implements `Comparator<PriorityObject>`. This class will have the following fields. **Please keep in mind that following implementation is provided to guide you (everything is declared as *private*) - it's not compulsory to do so, you can come up with a different implementation to store the *Puzzle* objects in *PriorityQueue* with an order.**

1. `private Puzzle board`: Each *PriorityObject* actually corresponds to a state (Puzzle).

2. `private int f`: f will be used to order the puzzles. In our A* algorithm, we always want to prefer states with minimum f value.

3. `private PriorityObject prev`: We will store the previous *PriorityObject* in this field. Whenever we create a *PriorityObject*, we have to store its previous *PriorityObject*. This will be very useful to print out the states in the path from inital state to the goal state.

4. `private int g`: g denotes the cost from inital state to this state, that is, number of transitions between the states.

Our *PriorityObject* class will have the following methods:

1. `PriorityObject(Puzzle board, int g, PriorityObject prev)`:In the constructor, assign the fields with the given parameters. Calculate f as $f = g +$ Taxicab Distance. Did we implement a function to calculate taxicab distance for a particular state before?

2. `Comparator<PriorityObject> comparator()`: Return a new *CustomComparator()*, which is a subclass described below.

   Implement a *CustomComparator* subclass (subclass of *PriorityObject*) that **implements** `Comparator<PriorityObject>`. It will have one method:

   `int compare(PriorityObject o1, PriorityObject o2)`: Override the *compare* method of `Comparator<PriorityObject>`. As usual, you should return 1 if $o1.f > o2.f$. Return -1 if $o1.f < o2.f$ and return 0 for equality.

Having talked about *PriorityObject* inner class, now let's proceed to the methods that will be implemented for *Solver*:

1. `public static void main(String[] args)`: Within the *main* function, open a file named as input.txt. First, you have to read an integer n that represents the size of the board. For a 8-puzzle game, n will be 3. Then you'll read the tiles one by one to create 2D tile array. As stated above, 0 represents the empty tile. Create a *Puzzle* object and solve it using the function below. Then open another file named output.txt and print out number of minimum moves and each state that in the path to the final state. Functions described below will be helpful during the solution. You may use *PrintStream* to write into a file.

2. `Solver(Puzzle root)`: This is the constructor of the *Solver* class. Throw an *IllegalArgumentException* if root is null, otherwise call the *Solve(root)* function to solve the puzzle.

3. `void solve(Puzzle root)`: This function is where we solve the problem. As stated earlier, you can use a stack or a priority queue to store the states. However, priority queue is strongly recommended in order to reduce the execution time of the algorithm. Store each state in a priority queue and take states from priority queue with minimum $f = g + h$ heuristic values. If the recently popped state is final state, finish the execution. You may want to store final state (final *PriorityObject*) in a private field to use it in other functions.

   There is an important optimization to observe here. Let's say we are at state X and we pushed all adjacent states to the Priority Queue. After popping one of the adjacents, we will again push X to the priority queue, since obviously X is an adjacent of that node. In order to avoid such duplicates, you can check if previous state was equal to the adjacent. If so, do not push it to the priority queue. This is one of the points where storing previous states as a field in *PriorityObject* becomes helpful.

4. `Iterable<Puzzle> getSolution()`: This function will be called from the *main* after *Solve(root)* function is called. `Iterable<Puzzle> getSolution()` must return an *Iterable* that consists of all *Puzzle* objects from initial state to final state. We have stored previous states in our *PriorityObject* objects, now our approach will be very useful to us. We can collect all states easily by back traversing from final state to the initial state using `private PriorityObject prev` field. First element of the *Iterable* must be the the *Puzzle* object that corresponds to the initial state.

5. `int getMoves()`: This function returns minimum number of moves for the solution. You may want to store minimum number of moves in a private field after solving the puzzle so that this function takes constant time.

4 Input & Output Format

At the very beginning of the input file, you will be provided with the value N. Then following lines will consist of he initial state of the board. An example input format can be found below:

file

```
3
1 2 3
0 4 6
7 5 8
```

In the first line, 3 is provided which corresponds to the N for our NxN puzzle. Then in the following lines, initial state of the board is provided. In each line there exists three tiles since our puzzle is 3x3. After reading the input, we have to solve the puzzle using our algorithm explained above. For the output file, at the very beginning of the file we have to print out the *minimum number of moves* to solve the puzzle, which is in this case equal to 3. Then you have to list all the states of the board that goes to the solution one by one. You have to print out the number of rows/columns in the board (which is N) following with the content of the board. States are separated by a newline. Corresponding output file can be found below:

`file`

```
Minimum number of moves = 3
3
 1 2 3
 0 4 6
 7 5 8

3
 1 2 3
 4 0 6
 7 5 8

3
 1 2 3
 4 5 6
 7 0 8

3
 1 2 3
 4 5 6
 7 8 0
```

In the first line, 3 corresponds to the minimum number of moves. Then in the following lines, 3 corresponds to the N value of our NxN board.

5 Important Remarks

- You can use appropriate access modifiers, keywords (super, final, static etc.) whenever you want. However, do not declare anything as *public* except those listed in the **Implementation** section.
- Code documentation is important. Please provide comments for your fields, classes and methods. Brief descriptions are enough.
- You can implement additional private helper methods of your choice.
- 1. What is a *Comparator*? What's the point of creating a custom comparator that implements a *Comparator<T>*?
     A *Comparator* helps us to impose a *total ordering* on some collection of objects. Let's say you want to sort a *Collection* consisting of Integers. Integers have natural total order among them, therefore you can call the function `Collections.sort(myList)` directly, and Java knows how to sort Integers. But what about a data structure you created? How does Java know how to sort them? At this point, we can create our own *total ordering* and give it as a parameter. For example, let's say we have *Book* objects and we want to sort them by number of pages. We can implement a *CustomComparator* for that *Book* class and give it as a parameter to the sort function:
     `Collections.sort(myList, myList.get(0).pageOrder())`. I have prepared a small pdf explaining *Comparable* and *Comparators*. You

can check it from here. To implement `PriorityObject` subclass, you will do something similar to what is shown in last page of the pdf. This pdf is just one of my personal notes, and the content is not related to this project. Of course, you are strongly encouraged to use detailed sources online to become familiar with the concept.

2. What is an immutable object?
   An immutable object is the one that cannot be changed after creation. For example, Strings are immutable in Java, meaning that you cannot modify a string after creation. You can ask this point: But how can I do something like this?:
   ```
   String str = "hello";
   str = "modified";
   ```
   Actually, what you do in this scenario is not modifying the string but creating a new one in the memory and changing where *str* points to. You can think of like this: *str* used to point to a memory location in which "hello" was stored. Now, it points to a different position in which "modified" is stored. So what happened to "hello"? Well, since there is no *reference* pointing to the memory location in which "hello" is stored, it will be deleted by the Garbage Collector [3] of Java. Remember that strings were immutable in Python as well. In this project, we want to create *Puzzle* class as immutable.

References

[1] `https://tristanpenman.com/demos/n-puzzle/`.

[2] `https://en.wikipedia.org/wiki/Taxicab_geometry`

[3] `https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)`