Project 1: 8-Puzzle (Searching)
CSC 330

In "Computer Science as an Empirical Inquiry: Symbols and Search," Newell and Simon propose that physical symbol systems have "the necessary and sufficient means for general intelligent action," and that they "[exercise] their intelligence in problem-solving by search." In this assignment you will explore search-based problem-solving by implementing a general search-based problem-solver and testing the effect of knowledge on guiding the search process. This will be done in the domain of the 8-puzzle.

Detailed instructions are given below. Please follow them very carefully or risk making your life much more difficult and/or losing points.

Introduction to the 8-Puzzle

You can try an online version of the 8-puzzle here: http://mypuzzle.org/sliding. Please let me know if you have questions on what the 8-Puzzle is.

Some Code To Get You Started

As discussed in class, I've provided a lot of code for you to start with, available on the I: drive. Please base your solution to this project on that code.

Working in Pairs

Like most out-of-class work, this project is to be done in your pairs. I want to encourage you to always work on this together. Don't work in isolation and plan on "explaining" the code later – that always results in less learning for the one hearing the explanation. For similar reasons, don't split up the work, figuring "I'll do this, you do that".

Part 1 (Due Sunday 2/10 at 11:55 PM via Moodle – zip up your entire BlueJ project folder, with a name including both your last names)

Please note that this part doesn't require knowledge of search algorithms. It sets the stage for later work, and also gives a nice warm-up / review of some programming.

As described in class, please complete the following:

- PuzzleState – the posOf, getTileAt, and expand methods. Write whatever private helper methods you'd like to make your code modular and well-designed.
    - Just so that all of our expand methods behave in the same way, please return your results in the order [up, down, left, right], where "up" means to move the ***blank*** up, etc. Note that the results provided in the test code follow this ordering.
    - If you want to make a copy of a 1-dimensional array, use Arrays.copyOf. Plain assignment (=) doesn't really make a copy, just an alias.
- DisplacementHeuristic – complete the distance method
- ManhattanHeuristic – complete the distance method

Make sure you test your code using the appropriate methods in the PuzzleTester class. I strongly encourage you to write your own methods in the PuzzleTester class as well, to try things out as needed. But please do not change the methods that are already in the PuzzleTester class.

Part 2 (Due TBA via Moodle – zip up your entire BlueJ project folder)

General Description

For part 2, you'll first implement the solve method to work for BFS. Recall that BFS and DFS both use the generic search code. They differ only in what addToFrontier does. More specifically, BFS uses a FIFO queue, while DFS uses a LIFO stack.

After that, you'll implement A* search as well. As you'll see in class soon, it too differs from BFS and DFS only in how the frontier behaves. The frontier for A* will be a *priority queue*, which will enable us to grab the "most promising" path on the frontier, according to a heuristic and the cost so far. We'll talk more about that later. I'm mentioning it so that you know why I will insist now that your code is done in a very generic way, using the generic search pseudocode we've seen in class. Ultimately, your BFS and A* implementations will use the exact same code, just with a different kind of queue.

So, your task for part 2, again, is to implement the generic search code, and pass an argument in for the queue such that BFS is done. After all that, we'll just add a tiny bit more for A*.

SearchProblem Constructor

Check out the provided code in the SearchProblem class. Note the SearchProblem constructor, and read the comments about the meaning of the different parameters. Then observe how we're calling the constructor in PuzzleTester.testBFS(). Note how the use of it there corresponds to the instructions in the SearchProblem constructor comments.

Go ahead and check out how PuzzleTester.testAStarDisplacement() and PuzzleTester. testAStarManhattan() work as well, just as a preview, to further understand how the arguments to the SearchProblem constructor work. We'll deal with those methods more in a bit.

Complete the SearchProblem constructor to keep track of the goalCheckLimit, goalState, and heuristic. You'll need to declare some fields as well. For the frontier, verify with an if statement that queueType is "FIFO", and then initialize the frontier to a LinkedList of PuzzlePath objects (not PuzzleState objects). Add the starting "path" to the frontier (as indicated in the beginning of the generic search pseudocode).

For now, if queueType is anything other than "FIFO", feel free to just signal an error somehow. We'll do more with this in a bit.

SearchProblem solve method

Finally, implement the solve method in the SearchProblem class. This will be called on a constructed SearchProblem object, of course, so some things will already be initialized. It is essential that you follow the generic search pseudocode from class very closely. If you deviate from this, you'll make part 2 harder, and may introduce subtle errors.

We do have one little addition here. Make your main loop check not just if the frontier is empty, but also if the goalCheckLimit is reached. That is, each time you check if some state is the goal, increment a

counter, and if you go above the limit, just give up. This is a nice way to stop erroneous infinite loops as you're debugging, as well as giving up when a search is impossible to complete.

One other quick adjustment we'll make to what we've done with BFS in class: In class, we've imagined that expand only returns "new" nodes. More precisely, that it only returns nodes not already at the end of a path on the frontier, and also not already in the expanded set. To set ourselves up for A* search in a bit, we need to change that slightly in our code here. First, let's leave the expand that you wrote in part 1 alone. Don't try to change it here. Instead, once you have the results of an expand, only add to the frontier if the node is not already in the *expanded* set. Don't worry about whether it's in the frontier already or not. Just check expanded. So this is different from what we've done in class, where we checked both expanded and frontier. This won't mess up BFS here, and it sets things up nicely for A* Search. We'll see why very soon.

General Advice

This is not necessarily an easy task, but if done in the best way should not require a lot of code. In my solution, for example, I declare a few fields, have several lines in the constructor, and 20-30 lines in the solve method. If your solution is significantly longer, it probably means one or more of the following:

- You could make more effective use of the Java facilities from our class discussions. Everything we discussed is super useful here!
- You could make more effective use of the provided code (especially PuzzlePath). Don't implement it again if code I gave you already does it.

Another little hint along these lines: if I gave you some code, or talked about something in class, and you haven't yet applied that concept, that probably means there's something you're missing. You either have an error, or at least you're making things more complex than they need to be.

When you're done with all the above, you should be able to run PuzzleTester.testBFS() successfully. Correct paths should be found (see the provided testBFS results at the end of this document) and the number of states checked should be about like this:

	Extremely Easy	**Easy**	**Harder**
BFS	3	93	470

You may very likely have the exact same number of states checked as me. It's possible that some variations will occur, though. So let's say that your results are correct as long as:

- Your code comes up with the correct path (of course).
- The number of states your code checks for each scenario above is less than double the results I have.
- The relative amounts are the same. That is, extremely easy < easy < harder.

When I run testBFS(), I get the results pretty much instantaneously. So if your program doesn't halt quickly, there's a problem somewhere, probably in handling repeated states. (Check the expanded set before adding to the frontier.)

I encourage you to make up your own test code too, trying out things as you go. Just be aware that half of all possible 8-puzzle states have no solution, so check by hand to be sure that the tests you try actually can be solved. And again, please don't change the test code I've provided. Just write additional methods if you want.

A* Search

Once you've completed all the above, and we've talked a bit about how A* search works in class, you're ready to add on just a little bit more code to implement A* search. The solve method should already be a generic search implementation, applicable to any search algorithm. So you shouldn't need to make any changes to the solve method here.

Your task here is to set up the execution of A* search. This should only take a couple additional lines of code in the SearchProblem constructor, where you initialize the frontier to be a PriorityQueue if the queueType is "Ordered". After that, solve can be called as usual and should work just fine. That's the power of polymorphism!

It is interesting to note that we could make the object-oriented design of this code even better, by making more effective use of generics (the < > notation) to make the SearchProblem class totally domain-independent. This would make it quite easy to write search problem formulations for other domains too. There's one possible end-of-semester project idea (more on that later), for those who have that kind of experience and interest...

When I run the PuzzleTester.testAll() method, I get the results shown way down below, at the end of this file. Take note in particular of the number of states checked (to see if it's the goal state) for each search on each problem. Those results are summarized in the following table:

	Extremely Easy	**Easy**	**Harder**
BFS	3	93	470
A* Displacement	2	13	53
A* Manhattan	2	9	32

As above, we'll say that your results are correct as long as:

- Your code comes up with the correct path (of course).
- The number of states your code checks for each scenario above is less than double the results I have.
- The relative amounts are the same. That is, extremely easy < easy < harder, and A* Manhattan < A* Displacement < BFS.

Again, your code should also finish pretty much instantaneously.

Here are the full testAll() results:

```
================================================
Test expand

---------------- a
[
```

```
1 _ 3
4 2 5
6 7 8
,
1 2 3
4 7 5
6 _ 8
,
1 2 3
_ 4 5
6 7 8
,
1 2 3
4 5 _
6 7 8
]

----------------- b
[
_ 2 3
1 4 5
6 7 8
,
1 2 3
6 4 5
_ 7 8
,
1 2 3
4 _ 5
6 7 8
]

----------------- c
[
3 1 2
_ 4 5
6 7 8
,
1 _ 2
3 4 5
6 7 8
]
================================================
Test displacement
Expected answers are provided.
0: 0
1: 1
1: 1
7: 7
================================================
Test Manhattan
Expected answers are provided.
```

```
0: 0
1: 1
1: 1
12: 12
================================================
Test BFS
------------------------------
States checked: 3
Solution length: 2
Solution:
(1: [
1 2 3
4 5 _
7 8 6
,
1 2 3
4 5 6
7 8 _
])
------------------------------
States checked: 93
Solution length: 7
Solution:
(6: [
1 2 3
7 _ 5
8 4 6
,
1 2 3
7 4 5
8 _ 6
,
1 2 3
7 4 5
_ 8 6
,
1 2 3
_ 4 5
7 8 6
,
1 2 3
4 _ 5
7 8 6
,
1 2 3
4 5 _
7 8 6
,
1 2 3
4 5 6
7 8 _
])
```

```
------------------------------
States checked: 470
Solution length: 11
Solution:
(10: [
4 2 3
5 1 6
7 8 _
,
4 2 3
5 1 _
7 8 6
,
4 2 _
5 1 3
7 8 6
,
4 _ 2
5 1 3
7 8 6
,
4 1 2
5 _ 3
7 8 6
,
4 1 2
_ 5 3
7 8 6
,
_ 1 2
4 5 3
7 8 6
,
1 _ 2
4 5 3
7 8 6
,
1 2 _
4 5 3
7 8 6
,
1 2 3
4 5 _
7 8 6
,
1 2 3
4 5 6
7 8 _
])
================================================
Test A* with displacement heuristic
------------------------------
```

```
States checked: 2
Solution length: 2
Solution:
(1: [
1 2 3
4 5 _
7 8 6
,
1 2 3
4 5 6
7 8 _
])
------------------------------
States checked: 13
Solution length: 7
Solution:
(6: [
1 2 3
7 _ 5
8 4 6
,
1 2 3
7 4 5
8 _ 6
,
1 2 3
7 4 5
_ 8 6
,
1 2 3
_ 4 5
7 8 6
,
1 2 3
4 _ 5
7 8 6
,
1 2 3
4 5 _
7 8 6
,
1 2 3
4 5 6
7 8 _
])
------------------------------
States checked: 53
Solution length: 11
Solution:
(10: [
4 2 3
5 1 6
```

```
7 8 _
,
4 2 3
5 1 _
7 8 6
,
4 2 _
5 1 3
7 8 6
,
4 _ 2
5 1 3
7 8 6
,
4 1 2
5 _ 3
7 8 6
,
4 1 2
_ 5 3
7 8 6
,
_ 1 2
4 5 3
7 8 6
,
1 _ 2
4 5 3
7 8 6
,
1 2 _
4 5 3
7 8 6
,
1 2 3
4 5 _
7 8 6
,
1 2 3
4 5 6
7 8 _
])
================================================
Test A* with Manhattan heuristic
------------------------------
States checked: 2
Solution length: 2
Solution:
(1: [
1 2 3
4 5 _
7 8 6
```

```
,
1 2 3
4 5 6
7 8 _
])
------------------------------
States checked: 9
Solution length: 7
Solution:
(6: [
1 2 3
7 _ 5
8 4 6
,
1 2 3
7 4 5
8 _ 6
,
1 2 3
7 4 5
_ 8 6
,
1 2 3
_ 4 5
7 8 6
,
1 2 3
4 _ 5
7 8 6
,
1 2 3
4 5 _
7 8 6
,
1 2 3
4 5 6
7 8 _
])
------------------------------
States checked: 32
Solution length: 11
Solution:
(10: [
4 2 3
5 1 6
7 8 _
,
4 2 3
5 1 _
7 8 6
,
4 2 _
```

```
5 1 3
7 8 6
,
4 _ 2
5 1 3
7 8 6
,
4 1 2
5 _ 3
7 8 6
,
4 1 2
_ 5 3
7 8 6
,
_ 1 2
4 5 3
7 8 6
,
1 _ 2
4 5 3
7 8 6
,
1 2 _
4 5 3
7 8 6
,
1 2 3
4 5 _
7 8 6
,
1 2 3
4 5 6
7 8 _
])
```