P1 Writeup

Code is run from outside of the package like so: `java P1.Parser P1/example1.txt`

Code Design

For this project I chose to work with Java because of its extensive built-in data structures, which allowed me to avoid external libraries entirely.

I segmented the project into three class files: `Parser.java`, `Puzzle.java`, and `PuzzleState.java`. All code is in a package called "`P1`".

`Parser.java` contains the main method for running the program. The `Parser` can be run with or without specifying a file containing the commands to be executed. If no filename is given, commands can be entered from a command line interface. Commands are tokenized by spaces and are generally checked for validity and proper usage. From the `Parser` class a different method is called depending on what command was given. The parser maintains a reference to the `Puzzle` object.

`Puzzle.java` corresponds to the 8-puzzle itself. It maintains a state and contains most methods necessary for manipulating the puzzle tiles. When movements are made, the state is updated. `Puzzle` objects contains the methods to move individual tiles, scramble the puzzle, generate a (valid) random state, and solve the puzzle in one of three different ways. Scrambling the puzzle makes a given number of random moves starting from the goal state. The process:

1. Create goal state
2. Create random number generator with the given number of moves as a seed
3. Generate successor states from the current state
4. Randomly pick a successor
5. Repeat previous two steps until the desired number of moves is made

Because a seed is used, `randomizeState n` will always result in the same final state for equal values of n.

A completely random single state can be generated, although it is not required for the assignment and was used for testing. A pair of methods is used for the generation: one method shuffles the tile string, another checks the validity of the shuffled tile string. If the string is found to be invalid, it is reshuffled.

The three available searching methods are local beam search (LBS), A* with the heuristic of a count of misplaced tiles (A*h1), and A* with the heuristic of a sum of Manhattan distances (A*h2). Because these searches all require finding the lowest value for $h(n)$ or $f(n)$, `PriorityQueue`s are used extensively for their low cost pop/poll/peek operations.

`PuzzleState.java` corresponds to a single state of the 8-puzzle. `PuzzleState` objects contain a tile ordering as well as a value for $h(n)$ and $g(n)$. No information about parent/child states is contained in the `PuzzleState`. `PuzzleState`s are ordered using a `compareTo` method which returns the difference between $f(n)$ values for two states. When states are generated as part of a LBS, a custom comparator is used which considers $h(n)$ as an evaluation function and does not consider differences between $g(n)$ values. `PuzzleState`s can generate children and calculate their own heuristic (predicted cost) values.

Code Correctness

To verify correctness, a number of examples are used. Each example has a corresponding `.txt` file.

Example 1 – Goal state base case

```
{ EECS-391 } java P1.Parser P1/example1.txt
$ setState "b12 345 678"
$ solve beam 10
Solved with 1 node(s), 0 ply(s), 0 step(s), 0 ms elapsed
$ printState
b12 345 678
$ setState "b12 345 678"
$ solve a-star h1
Solved with 1 node(s), 0 step(s), 0 ms elapsed
$ printState
b12 345 678
$ setState "b12 345 678"
$ solve a-star h2
Solved with 1 node(s), 0 step(s), 0 ms elapsed
$ printState
b12 345 678
```

Example 1 shows that all three algorithms successfully halt when the goal state is the intial state.

Example 2 – Low entropy

```
{ EECS-391 } java P1.Parser P1/example2.txt
$ randomizeState 5
$ printState
142 375 6b8
$ solve beam 10
Solved with 36 node(s), 3 ply(s), 3 step(s), 0 ms elapsed
$ printState
b12 345 678
$ randomizeState 5
$ printState
142 375 6b8
$ solve a-star h1
Solved with 11 node(s), 3 step(s), 0 ms elapsed
$ printState
b12 345 678
$ randomizeState 5
$ printState
142 375 6b8
$ solve a-star h2
Solved with 11 node(s), 3 step(s), 0 ms elapsed
$ printState
b12 345 678
```

Example 2 shows that all three algorithms successfully find the goal state from a state requiring only a small few moves. In this case, the LBS is different from both A* searches, but the A*

searches aren't differentiable when only three steps are needed. Additionally, we can see that the `randomizeState` command always generates the same state after five moves.

Example 3 – High entropy (textbook example)

```
{ EECS-391 } java P1.Parser P1/example3.txt
$ maxNodes 500000
$ setState "724 5b6 831"
$ solve beam 10
Solved with 499884 node(s), 18699 ply(s), 26 step(s), 565 ms elapsed
$ printState
b12 345 678
$ setState "724 5b6 831"
$ solve a-star h1
Solved with 117017 node(s), 26 step(s), 36513 ms elapsed
$ printState
b12 345 678
$ setState "724 5b6 831"
$ solve a-star h2
Solved with 8573 node(s), 26 step(s), 216 ms elapsed
$ printState
b12 345 678
```

Example 3 shows that while all three algorithms find the goal state eventually, A*h1 takes a entire 36 seconds to complete, and LBS needs to consider almost $500,000$ nodes. In this case A*h2 is by far the best choice. The problem state was used as an example in our textbook.

Example 4 – Beam width comparison

```
{ EECS-391 } master  java P1.Parser P1/example4.txt
$ setState "724 5b6 831"
$ solve beam 1
Solved with 551225 node(s), 205014 ply(s), 26 step(s), 609 ms elapsed
$ printState
b12 345 678
$ setState "724 5b6 831"
$ solve beam 10
Solved with 499884 node(s), 18699 ply(s), 26 step(s), 500 ms elapsed
$ printState
b12 345 678
$ setState "724 5b6 831"
$ solve beam 100
Solved with 687710 node(s), 2591 ply(s), 26 step(s), 586 ms elapsed
$ printState
b12 345 678
```

Example 4 uses the same starting state as Example 3 and shows the that the beam width makes on the LBS. The number of plys and the beam width are directly proportional, although the time elapsed is roughly the same in all cases.

Experiments

(a) The fraction of solvable puzzles is affected by the maxNodes limit. Using `example5.txt` to test,

maxNodes	LBS	A*h1	A*h2
1	Failed	Failed	Failed
10	Failed	Failed	Failed
100	Failed	Failed	Failed
1000	Failed	Failed	Failed
10000	Failed	Failed	**Solved**
100000	Failed	Failed	**Solved**
1000000	**Solved**	**Solved**	**Solved**

It takes A*h2 only **8,573** nodes. It takes A*h1 **117,017** nodes. It takes LBS **499,884** nodes.

(b) With A* search, $h2$ is much, much better in all difficult cases. In easy cases they become more similar. For a problem requiring 26 moves, A*h1 will solve with

```
117017 node(s), 26 step(s), 36513 ms elapsed
```

For the same problem, A*h2 will solve with

```
8573 node(s), 26 step(s), 216 ms elapsed
```

We can see that the search space is much smaller and the time needed is orders of magnitude shorter for $h2$ compared to $h1$. $h1$ expands more nodes because the range of $h(n)$ is much lower, with only 1-9 possible values.

(c) The solution

(d) Running `example6.txt` will test 15 *completely random* valid states. The results of one run are as follows (trimmed).

```
$ solve beam 10
Solved with 1265 node(s), 49 ply(s), 18 step(s), 12 ms elapsed
$ solve a-star h1
Solved with 121126 node(s), 27 step(s), 24726 ms elapsed
$ solve a-star h2
Solved with 378 node(s), 19 step(s), 0 ms elapsed
$ solve beam 10
Solved with 592864 node(s), 22433 ply(s), 24 step(s), 435 ms elapsed
$ solve a-star h1
Solved with 47192 node(s), 24 step(s), 4376 ms elapsed
$ solve a-star h2
Solved with 1277 node(s), 23 step(s), 3 ms elapsed
$ solve beam 10
Solved with 129959 node(s), 4949 ply(s), 23 step(s), 91 ms elapsed
$ solve a-star h1
Solved with 103065 node(s), 26 step(s), 16139 ms elapsed
$ solve a-star h2
Solved with 3201 node(s), 24 step(s), 21 ms elapsed
$ solve beam 10
Solved with 209985 node(s), 7759 ply(s), 22 step(s), 154 ms elapsed
$ solve a-star h1
Solved with 8562 node(s), 20 step(s), 168 ms elapsed
$ solve a-star h2
Solved with 1221 node(s), 23 step(s), 3 ms elapsed
```

```
$ solve beam 10
Solved with 156851 node(s), 5956 ply(s), 23 step(s), 105 ms elapsed
$ solve a-star h1
Solved with 32999 node(s), 23 step(s), 3189 ms elapsed
$ solve a-star h2
Solved with 2783 node(s), 23 step(s), 15 ms elapsed
```

If we count solutions with elapsed times over 5 seconds as failures, then above there are two failures out of 45 solutions ($2/45 = 4.44\%$). However, every search finished eventually, and so with the explicit definition, all cases (100%) are solvable. A*h2 seems to pass every test.

Discussion

(a) The best suited algorithm seems to be A*h2.

Time complexity:
`A*h1 < LBS < A*h2`

Space complexity:
`LBS < A*h1 < A*h2`

It seems A* works best depending on how accurate your heuristic function is. The $h1$ heuristic illustrates how the algorithm can fail if the heuristic is valid but not quite precise enough at determining the value of states.

LBS has the flexibility to become more or less space efficient depending on the beam width, which is an interesting to compare which A* which is popular but might not be viable on a system with high space constraints.

(b) Beam search has a surprisingly intuitive design and was not very difficult for me to implement. It is interesting how the two algorithms are similar; for LBS I used two containers: a beam and a nextPly queue, and for A* I also used two containers: a frontier and an explored queue.