8-Puzzle Solver Writeup/Testing

The code for this project can be found on my Github.

This document is a write-up explaining the A-Star and Local Beam Search for solving a 8 puzzle. Furthermore, for implementing A-Star we implemented two heuristics for solving the puzzle -- one that only accounts for misplaces and another that accounts for the Manhattan distance between the current puzzle state and goal puzzle state. We analyze the run time complexity and space efficiency of all 3 methods.

A-Star is an algorithm that is widely used in pathfinding and graph traversal. It has a worst case time complexity of $O(b^d)$ and a worst case runtime of $O(b^d)$. Mathematically, A-Start selects the path that minimizes $f(n) = g(n) + h(n)$ where $g(n)$ is the cost of the path from the state node to the current node (in our case the number of times we have displaced the 0-tile on the puzzle) while $h(n)$ is the heuristic that estimates the path to the goal state (in our case misplaced tiles or manhattan distance). A heuristic is admissible as long as it does not overestimate the path to the goal state. A heuristic is consistent if it obeys the triangle inequality leading to an average runtime of $O(bd)$. A-star essentially incorporates a greedy best-first component into Djikstra's by incorporation of heuristics into it.

Local beam search is an greedy algorithm that essentially explores the graph by choosing the most promising node in a limited set. One can view that doing a breadth first search on a pruned tree. As we increase the k-value in beam search, the algorithm approaches completeness. While the algorithm is not complete, it reduces memory complexity significantly.

Djikstra's algorithm can be seen as a variant of A-star with h(n) = 0

Few things that are noteworthy before diving into the experiments:

1) A-star search is a complete algorithm whereas local beam search is not. This means that the A-star algorithm is guaranteed to return the shortest path to the goal state as long as one exists. Local beam search on the other hand is not a complete algorithm meaning that a path to the goal state is not guaranteed even if one exists. Furthermore, there is no guarantee that the algorithm will return the shortest path to the goal state.
2) Next comes the optimality of A-Star which is satisfied by any heuristic that is consistent (obeys the triangle inequality). It can be mathematically shown that

only the Manhattan distance Heuristic obeys the equality (can be thought of as the L1 norm), while the Misplaced tiles heuristic does not, meaning that A-star with Manhattan heuristic can return the optimal path to the goal without processing any excess nodes.

a)

```
//Loop used for generating puzzle states
for(int i = 0;i<10;i++){
    P.randomizeState(60);
    P.solveAStar("h1");
}
```

I also set the initial state to “b12 345 678” before running the loop for the sake of consistency. I only use a sample set of 10 randomized States because Local Beam Search takes a while to run.

At MaxNodes=100:

A-Star(h1) - 1/10 solved

A-Star(h2) - 3/10 solved

SolveBeam(10) - 0/10 solved

At MaxNodes=1000:

A-Star(h1) - 6/10 solved

A-Star(h2) - 10/10 solved

SolveBeam(10) - 0/10 solved

At MaxNodes=10000:

A-Star(h1) - 10/10 solved

A-Star(h2) - 10/10 solved

SolveBeam(10) - 2/10 solved

b)

Well, going by our observations above, h2(Manhattan Distance Heuristic) is clearly better. It should also intuitively make sense that the Manhattan Distance Heuristic is doing a better job. While h1 gives an idea about only the missing tiles, the Manhattan Distance in a sense assigns weights to the missing tiles. For instance, a 7-tile at 1’s

position should be more heavily penalized than a 2-tile at 1's position. At the same time there is one drawback that I notice in this case - if there's a case where 0 and 1 are interchanged and another case where 0 and 3 are interchanged, the latter is penalized more heavily even though in both cases we are 1 move away from getting the tile to the correct positions but these cases do not dominate. In most of my test cases, h2 has also been giving faster results even though the path distance is remaining the same across both heuristics.

c)

```
> P.randomizeState(10000)
> P.solveAStar("h1")
[805 647 213, 845 607 213, 845 067 213, 845 267 013, 845 267 103, 845 207 163, 845 027 163, 045 827 163, 405 827 163, 425 807
163, 425 087 163, 425 187 063, 425 187 603, 425 107 683, 425 170 683, 425 173 680, 425 173 608, 425 103 678, 425 130 678, 420
135 678, 402 135 678, 042 135 678, 142 035 678, 142 305 678, 102 345 678, 012 345 678] moves: 25
> P.solveAStar("h2")
[805 647 213, 845 607 213, 845 067 213, 845 267 013, 845 267 103, 845 207 163, 845 027 163, 045 827 163, 405 827 163, 425 807
163, 425 087 163, 425 187 063, 425 187 603, 425 107 683, 425 170 683, 425 173 680, 425 173 608, 425 103 678, 425 130 678, 420
135 678, 402 135 678, 042 135 678, 142 035 678, 142 305 678, 102 345 678, 012 345 678] moves: 25

> P.randomizeState(100)
> P.solveAStar("h2")
[432 150 678, 432 105 678, 402 135 678, 042 135 678, 142 035 678, 142 305 678, 102 345 678, 012 345 678] moves: 7
> P.solveBeam(10)
[432 150 678, 432 105 678, 402 135 678, 042 135 678, 142 035 678, 142 305 678, 102 345 678, 012 345 678] moves: 7
```

Well h1 and h2 are giving the same length solutions across multiple input cases and this should be expected since both are admissible heuristics. Therefore we should be getting the one optimal solution that is guaranteed.

I only tested Local Beam Search on sets that were randomized to a smaller extent(P.randomizedStates(30-100)) and got solutions of the same length as AStar. This may be due to the fact that the evaluation function that I am using is the path cost + h1(from AStar).

So essentially the optimal path length remained the same for the 3 methods.

d)

Max Nodes was set to 100,000 for all 3 methods

```
for(int i = 0;i<10;i++){
    P.randomizeState(60);
    P.solveAStar("h1");
```

}

For AStar H1 and A Star H2, I got **100%** results meaning that all puzzle states were solved when MaxNodes were set to 100,000

For Local Beam Search however, I got **20%** results meaning that only 2/10 of the puzzle states were solved within the maxNodes constraint and these were the first 2 states randomized from “b12 345 678”. This means as we are randomizing the puzzles more and more, Local beam search is not able to perform as well as A-Star.

Discussions:

a)

Based on our experiments, clearly AStar(H2)-Manhattan Heuristic has been doing the best job. It gives the optimum in minimum time and also works well when MaxNodes has a higher value. 3b has more details on why Manhattan Heuristic does a better job compared to the H1. AStar search performs better than LocalBeamSearch when it comes to runtime. This is due to the fact that each iteration, we expand only 1 state in A-Star while in LocalBeamSearch, we have to do this for K-states. When it comes to memory, I feel LocalBeamSearch might be superior in general for well randomized data. In Local Beam Search, we cap the Open List at K states whereas in A-Star search our openList can have way more than K-states.

Again, all of these assume that both of these methods have been implemented with some consistency in terms of the data structures used for implementation.

I don’t think I had an algorithm in particular that got the shortest solution. As mentioned earlier, LocalBeamSearch seems to be running out of time when I use super randomized states(P.randomizeState(>1000)) which should be expected as this is a slower algorithm. AStar search does a much better job with runtime. With the states that were solved by all 3 algorithms, I got the same path length for all of them.

b)

I made a State object to make implementation easier for me. Abstraction always helps. The state object essentially stores a matrix, a parent and f value. The puzzle holds a field called currentState which is a State type that represents the current State of the puzzle. Moreover, I made sure I was using the appropriate data structures for both the search algorithms. For instance, I was maintaining both a Hashset and priority queue for the open List in AStar Search. This allowed me to search for the goalState in the open List in constant time. I also had to override the HashValues to make sure the hashing is done according to the matrix values. I kept track of the indices of the 0 tiles in the State object to make sure we don't have to loop over the matrix everytime we want to find this tile. All the methods in both State and Puzzle classes were tested by me thoroughly for correctness. I have a section below on testing to show how I tested the methods. I obviously just included a few examples to give you guys an idea about how to test my methods.

The currentState of the puzzle is initialized to "b12 345 678" and maxNodes to 100,000 as I feel that gave decent results for all 3 algorithms. The method setMaxNodes(int k) can be used to change this.

Testing:

1) Testing the initialization, printState(), setState(input), move(direction), and randomizeState()

```
> Puzzle P = new Puzzle();
> P.getCurrentState()
012 345 678
> P.printState()
012 345 678
> P.move("down")
> P.printState()
312 045 678
> P.setState("125 34b 678")
> P.printState()
125 340 678
> P.move("up")
> P.printState()
120 345 678
> P.randomizeState(150)
> P.printState()
843 251 067
```

Traced out the moves on papers and confirmed that the commands were giving the correct outputs.

2) Testing A-Star using both the heuristics.

```
Welcome to DrJava.  Working directory is C:\Users\Param
> Puzzle P = new Puzzle();
> P.randomizeState(150)
> P.printState()
318 564 207
> P.solveAStar("h1")
[318 564 207, 318 504 267, 318 054 267, 318 254 067, 318 254 607, 318 204 657, 318 024 657, 018 324 657, 108 324 657, 128 304
657, 128 340 657, 120 348 657, 102 348 657, 142 308 657, 142 358 607, 142 358 670, 142 350 678, 142 305 678, 102 345 678, 012
345 678] moves: 20
> P.solveAStar("h2")
[318 564 207, 318 504 267, 318 054 267, 318 254 067, 318 254 607, 318 204 657, 318 024 657, 018 324 657, 108 324 657, 128 304
657, 128 340 657, 120 348 657, 102 348 657, 142 308 657, 142 358 607, 142 358 670, 142 350 678, 142 305 678, 102 345 678, 012
345 678] moves: 20
> P.randomizeState(15)
> P.randomizeState(154)
> P.solveAStar("h2")
[301 268 547, 361 208 547, 361 028 547, 361 528 047, 361 528 407, 361 528 470, 361 520 478, 361 502 478, 301 562 478, 310 562
478, 312 560 478, 312 568 470, 312 568 407, 312 508 467, 312 058 467, 312 458 067, 312 458 607, 312 458 670, 312 450 678, 312
405 678, 312 045 678, 012 345 678] moves: 22
> P.solveAStar("h1")
[301 268 547, 361 208 547, 361 028 547, 361 528 047, 361 528 407, 361 528 470, 361 520 478, 361 502 478, 301 562 478, 310 562
478, 312 560 478, 312 568 470, 312 568 407, 312 508 467, 312 058 467, 312 458 067, 312 458 607, 312 458 670, 312 450 678, 312
405 678, 312 045 678, 012 345 678] moves: 22
```

Used A-Star Search to solve randomized puzzles using both the heuristics and they give the correct output

3) Testing the solveBeam using random inputs.

```
> Puzzle P = new Puzzle();
> P.randomizeState(54)
> P.solveBeam(10)
[245 187 360, 245 180 367, 245 108 367, 205 148 367, 025 148 367, 125 048 367, 125 348 067, 125 348 607, 125 348 670, 125 340
678, 120 345 678, 102 345 678, 012 345 678] moves: 13
> P.solveAStar("h1")
[245 187 360, 245 180 367, 245 108 367, 205 148 367, 025 148 367, 125 048 367, 125 348 067, 125 348 607, 125 348 670, 125 340
678, 120 345 678, 102 345 678, 012 345 678] moves: 13
> P.solveAStar("h2")
[245 187 360, 245 180 367, 245 108 367, 205 148 367, 025 148 367, 125 048 367, 125 348 067, 125 348 607, 125 348 670, 125 340
678, 120 345 678, 102 345 678, 012 345 678] moves: 13
```

.

More testing

```
Welcome to DrJava.  Working directory is C:\Users\Param
> Puzzle P = new Puzzle()
> P.randomizeState(100)
> P.solveAStar("h2")
[432 150 678, 432 105 678, 402 135 678, 042 135 678, 142 035 678, 142 305 678, 102 345 678, 012 345 678] moves: 7
> P.solveBeam(10)
[432 150 678, 432 105 678, 402 135 678, 042 135 678, 142 035 678, 142 305 678, 102 345 678, 012 345 678] moves: 7
> P.randomizeState(100)
> P.solveBeam(10)
MaxNodes exceeded
[] moves: -1
>
```