Liam McConlogue
CSDS 391 - Intro to AI
Programming Assignment 1
Thursday March 2, 2023

Code Design

Overview

Here is a broad overview of the structure of my Java code.

- **Main** class - For user interaction. Handles the file inputs and (most) outputs
- **Puzzle** class - Represents the puzzle board and simple actions like
    - Moving a tile, setting the state, setting the number of allowed nodes, randomizing the state, and other methods that help the search algorithms run.
- **AStarSearch** class - performs the A Star Search algorithm on a class with the method: `public static Move solveAStar(Puzzle puzzle, Heuristic heuristic)`
    - It performs the search on the puzzle such that the puzzle state becomes "b12 345 678" if the algorithm succeeds.
    - It returns the final **move** (see below) and prints the solution
- **LocalBeamSearch** class - performs the Local Beam Search algorithm using the evaluation function $f(n) = h(n)$ for each node *n* where *h(n)* is the heuristic "H2," or Manhattan Distance. When it reaches the goal state, it has a minimum cost of 0.
    - It uses the method `solveLocalBeam`, and also returns the final **move** along with printing the solution.
- **Move** class - represents one move or "node," that stores the **direction** of the move (up, down, …) and its parent move (prev).
    - A sequence of moves, taken by starting at the last move and working back through the parent moves gives the path of the solution
    - To aid in searching, the move class also stores the g(n) and h(n) values as well as the resulting state of the puzzle after the move ("1b2 345 678").

        - g(n) is the number of moves taken since beginning the search
- **Heuristic** is an enumerated class with values H1 and H2, used in the search algorithms
    - Also has the method `calculateHeuristic(String state, Heuristic heuristic)` which returns the value of $h(n)$ for a given puzzle state
- **Direction** is a simple enumerated class with values `UP`, `DOWN`, `LEFT`, and `RIGHT`, which are used in the Puzzle, Move, and search classes.

Design Decisions

Puzzle

In the puzzle class, I choose to store the current game board as a 2D array of integers: `private int [][] board;` to make operations on the board more intuitive than when working with the String representation `"b12 345 678"`. The blank 'b' is converted to a 0 so that it can be used in the board. The private method `private int [][] buildBoard(String state)` is used to convert the string to the int array.

The puzzle is initialized with a given `state` (the String representation), or defaults to the solved state if none is given. Other classes only interact with a Puzzle object using the String state - the conversion is hidden from view. Puzzle also stores the maximum number of nodes as `long maxNodes` because you may want to test very large search trees.

AStarSearch

The A Star Search class solves a puzzle with the following method:

```
public static Move solveAStar(Puzzle puzzle, Heuristic heuristic) {
        long nodesGenerated = 1;

        //Create Frontier and add initial node
        Move move = makeInitialMove(puzzle, heuristic);
        PriorityQueue<Move> frontier = new PriorityQueue<Move>();
        frontier.add(move);
        //Create Reached and add initial node
        Hashtable<String, Integer> reachedCost = new Hashtable<>();
        reachedCost.put(move.state(), move.cost());

        long maxNodes = puzzle.getMaxNodes();

        //Main loop
        while (!frontier.isEmpty()){
            //Pop from the frontier
            move = frontier.poll();

            //Solution was found
            if (move.state().equals(SOLVED)){
                Move.runAndPrintSolution(puzzle, move,
nodesGenerated, heuristic, "A Star Search");
                return move;
            }

            //Generate all successive nodes (next) with helper method
expand()
            for (Move next: expand(move, heuristic)){
                nodesGenerated++;
                if (nodesGenerated > maxNodes) throw new
IllegalStateException("Exceeded maximum allowed number of generated
nodes (" + nodesGenerated + ")");

                //if next is not in reached OR next.cost is less than
the cost for the equivalent state in reached
                if (!reachedCost.containsKey(next.state()) ||
next.cost() < reachedCost.get(next.state())){
                    //Add next to the frontier and reached set
                    frontier.add(next);
                    reachedCost.put(next.state(), next.cost());
                }
            }
        }
        //If no solution is found. Does not run under normal
circumstances
        return null;
```

```
    }
```

As seen above, the reached table is a `HashMap` that takes a `String state` as the key with a `Move node` as its value. This works well because the Strings can be quickly compared to each other to check if a state is already stored. It is mapped to a `Move` (or node), which represents the path taken to reach that state, as many Move paths can lead to the same board state.

The expand method is:

```
private static Move[] expand(Move move, Heuristic heuristic) {
        //List all possible directions the blank tile can move in
        //Which become the generated successive states
        List<Direction> possibleDirections = Puzzle.getValidDirections(
move.state());

        //Create a new node (move) for each possible direction
        List<Move> moves = new ArrayList<>();
        possibleDirections.forEach(direction -> {
            String nextState = Puzzle.move(move.state(), direction);
            int newDistFromStart = move.distFromStart() + 1;
            Move nextMove = new Move (
                    direction,
                    move,
                    newDistFromStart,
                    Heuristic.calculateHeuristic(nextState, heuristic),
                    nextState
            );
            moves.add(nextMove);
        });

        //Return the generated nodes
        Move [] moveArray = new Move[moves.size()];
        moves.toArray(moveArray);
        return moveArray;
    }
```

Each move keeps track of its distance from the starting state by adding one to its parent node. The initial node created by the search has a `distFromStart` of 0.

LocalBeamSearch

The local beam search class works similarly to A Star Search, with the following method to solve the puzzle.

```
public static Move solveLocalBeam(Puzzle puzzle, long maxNodes, int
k) {
        int generatedNodes = 0;

        //Create the list nextMoves which stores generated nodes
        Move move = makeInitialMove(puzzle);
        PriorityQueue<Move> nextMoves = new PriorityQueue<>();
        nextMoves.add(move);
        //Create list currentMoves to indicate nodes to be expanded
        PriorityQueue<Move> currentMoves;

        while (true) {

            //Set currentMoves to nextMoves and clear nextMoves
            currentMoves = new PriorityQueue<>(nextMoves);
            nextMoves.clear();

            //generate next states for all K current states and add
them to nextStates
            for (Move m : currentMoves){
                generatedNodes++;
                if (generatedNodes > max_nodes) {
                    System.out.print("Failed at cost " +
currentMoves.remove().cost() + ". ");
                    throw new IllegalStateException("Exceeded maximum
allowed number of generated nodes (" + generatedNodes + ")");
                }
                if (m.state().equals(Puzzle.SOLVED)){
                    //Solution found
                    Move.runAndPrintSolution(puzzle, m, 
generatedNodes, heuristic, "Local Beam Search");
                    return m;
                }

                //Generate the states
                nextMoves.addAll(expand(m));
            }

            //select best K from the generated states AND current 
states and discard all else
            nextMoves.addAll(currentMoves);
            nextMoves = removePastK(nextMoves, k);
        }
    }
```

The line `System.out.print("Failed at cost " + currentMoves.remove( ).cost() + ". ");` is important in determining when the search algorithm gets stuck in local minima, as demonstrated in code correctness.

The expand method works similarly to in A Star Search, except that it sets the *g(n)* value, or the number of moves since the start of the search, to 0.

Move

In addition to representing nodes, the Move class contains a method to run the found solution on a given puzzle and print the resulting path with `runAndPrintSolution( … )`. This method loops backwards from the solution node to the starting node with

```
while(move.prev() != null){
            stack.add(move);
            move = move.prev();
}
```

Then carries out the moves on the original puzzle and prints the solution by popping from that stack:

```
while(!stack.isEmpty()){
            move = stack.pop();
            puzzle.move(move.direction);
            ...
}
```

Heuristic

An important part of the heuristics is to not count the blank tile. When you count the blank tile, the efficiency is dramatically reduced - I was unable to get Local Beam Search to find solutions to any fully randomized puzzles at first because of this.

The static method `calculateHeuristic(String state, Heuristic heuristic)` in Heuristic returns the value of *h(n)* for a given state with either H1 or H2. The heuristic H1, or number of misplaced tiles, is a simple calculation of which tiles are in the correct place.

The heuristic H2, or Manhattan distance, is more complicated to calculate. To do this, I started with a HashMap that maps a given tile number to its goal location (stored in a Coordinate record to represent the row and column):

```
    static final Map<Integer, Coordinate> goalCoordinate;
    // Instantiating the static goalCoordinate
    static {
        goalCoordinate = new HashMap<>();
        goalCoordinate.put(0, rowCol(0,0));
        goalCoordinate.put(1, rowCol(0,1));
        goalCoordinate.put(2, rowCol(0,2));
        goalCoordinate.put(3, rowCol(1,0));
        …
}
```

I calculate the “score” of any given tile in a puzzle with the method: `tileScoreH2( int[][] board, int row, int col )`

Which essentially adds the vertical distance between the tile’s current row and its goal row to the horizontal distance between the tile’s current column and its goal column:

```
int goalRow = goalCoordinate.get(tile).row();
int goalCol = goalCoordinate.get(tile).col();
int score = 0;
score += Math.abs(goalRow - row);
score += Math.abs(goalCol - col);
return score;
```

Code Correctness

To demonstrate the code correctness, I will be running the input file “sample-input.txt” included in my submission. I have included this file below. Note that lines starting with ‘#’ are comments to be printed in the output. I will break this up into 3 parts.

Part 1

In part 1, I test the simple and easily verifiable starting state of “312 645 7b8” on all three algorithms. Input is:

```
setState 312 645 7b8
#Solve state "312 645 7b8" for all three algorithms
solveAStar h1

setState 312 645 7b8
solveAStar h2

setState 312 645 7b8
solveBeam 100
#For this simple starting state, all three algorithms solved it the same way.
```

And the resulting output is:

```
#Solve state "312 645 7b8" for all three algorithms
Solving puzzle 312 645 7b8 with AStarSearch and 1000 allowed nodes.
Starting state: 312 645 7b8
Move 1: LEFT. State: 312 645 b78
Move 2: UP. State: 312 b45 678
Move 3: UP. State: b12 345 678
Solved using A Star Search with heuristic 'H1' with 9 nodes generated.

Solving puzzle 312 645 7b8 with AStarSearch and 1000 allowed nodes.
Starting state: 312 645 7b8
Move 1: LEFT. State: 312 645 b78
Move 2: UP. State: 312 b45 678
Move 3: UP. State: b12 345 678
Solved using A Star Search with heuristic 'H2' with 9 nodes generated.

Solving puzzle 312 645 7b8 with Local Beam Search using k = 100 and
1000 allowed nodes.
Starting state: 312 645 7b8
Move 1: LEFT. State: 312 645 b78
Move 2: UP. State: 312 b45 678
Move 3: UP. State: b12 345 678
Solved using Local Beam Search with heuristic 'H2' with 21 nodes
generated.
```

For this simple starting state, all three algorithms solved it the same way. Both heuristics for the A* algorithm generated 9 nodes while local beam search generated 21. Local beam search tends to generate exponentially more nodes in all examples.

Part 2

For part 2, I test the randomized starting state of “561 832 74b,” obtained by making 500 (seeded) random moves on the starting state. The input is:

```
#Fully randomize state with 500 moves
randomizeState 500
maxNodes 100000
#Solve A* with H1 and H2 on the same starting state 500 moves out,
starting with A* using H1
solveAStar h1

#Randomize the solved puzzle by the same amount then run A* with H2
```

```
randomizeState 500
printState
solveAStar h2
```

And the output is:

```
#Fully randomize state with 500 moves
#Solve A* with H1 and H2 on the same starting state 500 moves out,
starting with A* using H1
Solving puzzle 561 832 74b with AStarSearch and 100000 allowed nodes.
Starting state: 561 832 74b
Move 1: LEFT. State: 561 832 7b4
Move 2: LEFT. State: 561 832 b74
Move 3: UP. State: 561 b32 874
Move 4: UP. State: b61 532 874
Move 5: RIGHT. State: 6b1 532 874
Move 6: DOWN. State: 631 5b2 874
Move 7: LEFT. State: 631 b52 874
Move 8: DOWN. State: 631 852 b74
Move 9: RIGHT. State: 631 852 7b4
Move 10: UP. State: 631 8b2 754
Move 11: LEFT. State: 631 b82 754
Move 12: UP. State: b31 682 754
Move 13: RIGHT. State: 3b1 682 754
Move 14: RIGHT. State: 31b 682 754
Move 15: DOWN. State: 312 68b 754
Move 16: DOWN. State: 312 684 75b
Move 17: LEFT. State: 312 684 7b5
Move 18: UP. State: 312 6b4 785
Move 19: RIGHT. State: 312 64b 785
Move 20: DOWN. State: 312 645 78b
Move 21: LEFT. State: 312 645 7b8
Move 22: LEFT. State: 312 645 b78
Move 23: UP. State: 312 b45 678
Move 24: UP. State: b12 345 678
Solved using A Star Search with heuristic 'H1' with 44000 nodes
generated.

#Randomize the solved puzzle by the same amount then run A* with H2
```

```
Current puzzle state: 561 832 74b
Solving puzzle 561 832 74b with AStarSearch and 100000 allowed nodes.
Starting state: 561 832 74b
Move 1: UP. State: 561 83b 742
Move 2: LEFT. State: 561 8b3 742
Move 3: LEFT. State: 561 b83 742
Move 4: DOWN. State: 561 783 b42
Move 5: RIGHT. State: 561 783 4b2
Move 6: UP. State: 561 7b3 482
Move 7: RIGHT. State: 561 73b 482
Move 8: DOWN. State: 561 732 48b
Move 9: LEFT. State: 561 732 4b8
Move 10: LEFT. State: 561 732 b48
Move 11: UP. State: 561 b32 748
Move 12: RIGHT. State: 561 3b2 748
Move 13: UP. State: 5b1 362 748
Move 14: LEFT. State: b51 362 748
Move 15: DOWN. State: 351 b62 748
Move 16: RIGHT. State: 351 6b2 748
Move 17: UP. State: 3b1 652 748
Move 18: RIGHT. State: 31b 652 748
Move 19: DOWN. State: 312 65b 748
Move 20: LEFT. State: 312 6b5 748
Move 21: DOWN. State: 312 645 7b8
Move 22: LEFT. State: 312 645 b78
Move 23: UP. State: 312 b45 678
Move 24: UP. State: b12 345 678
Solved using A Star Search with heuristic 'H2' with 4896 nodes
generated.
```

Both search attempts succeed with 24 moves but take different paths. Because both heuristics are admissible, they generate an optimal solution, meaning that they always return a solution in the fewest possible steps, even if they take different paths. H1 generates 44000 nodes while H2 generates 4896 nodes. I will discuss efficiency under **Experiments** but already it is clear that H2 is much more efficient than H1.

Part 3

For part 3, I test Local Beam Search with a the starting state “b51 342 678” generated with 100 random moves, then increase the random moves to 150. The **input** is:

```
#Randomize the state by only 100 moves for Beam search
randomizeState 100
maxNodes 100000
solveBeam 1000
#Succeeds with 8 steps for the (easier) starting state of b51 342 678

#Randomize the state a little more (by 200 moves) for Beam search
randomizeState 200
solveBeam 10000
#Fails and ends with the best state having a heuristic cost of 3.

#Try again and increase the max nodes and K value of beam search by
ten times
setState b12 345 678
randomizeState 200
maxNodes 1000000
solveBeam 100000
```

And the resulting **output** is:

```
#Randomize the state by only 100 moves for Beam search
Solving puzzle b51 342 678 with Local Beam Search using k = 1000 and
100000 allowed nodes.
Starting state: b51 342 678
Move 1: DOWN. State: 351 b42 678
Move 2: RIGHT. State: 351 4b2 678
Move 3: UP. State: 3b1 452 678
Move 4: RIGHT. State: 31b 452 678
Move 5: DOWN. State: 312 45b 678
Move 6: LEFT. State: 312 4b5 678
Move 7: LEFT. State: 312 b45 678
Move 8: UP. State: b12 345 678
Solved using Local Beam Search with heuristic 'H2' with 2813 nodes
generated.
```

```
#Succeeds with 8 steps for the (easier) starting state of b51 342 678

#Randomize the state a little more (by 150 moves) for Beam search
Solving puzzle 512 638 47b with Local Beam Search using k = 10000 and
100000 allowed nodes.
Failed at cost 3. Local Beam exceeded maximum allowed nodes.
#Fails and ends with the best state having a heuristic cost of 3.

#Try again and increase the max nodes and K value of beam search by
ten times
Solving puzzle 512 638 47b with Local Beam Search using k = 100000
and 1000000 allowed nodes.
Failed at cost 3. Local Beam exceeded maximum allowed nodes.
```

With the first medium-difficulty puzzle (100 moves) “b51 342 678,” local beam search successfully finds the solution in 8 moves. Testing the same starting state with A* search with heuristic ‘h2’ also results in an 8 step solution, meaning that this was an optimal solution, which makes sense as they are using the same evaluation function in my implementation.

When I increase the number of random moves from the solved state to 200 moves (“512 348 b67”), the search fails with the best state having a heuristic of 3. Even after increasing by tenfold the k value (to 10,000) and the maximum allowed nodes (to 1,000,000) the search still fails with the best state again having a heuristic of 3. Despite exploring exponentially more nodes, the algorithm did not improve. This indicates that it was stuck in local minima, which often happens when the starting state is fully randomized.

Experiments

To see the code for my experiments, go to `src/test/Experiments.java`

Experiment 1 - Fraction of solvable puzzles

Setup

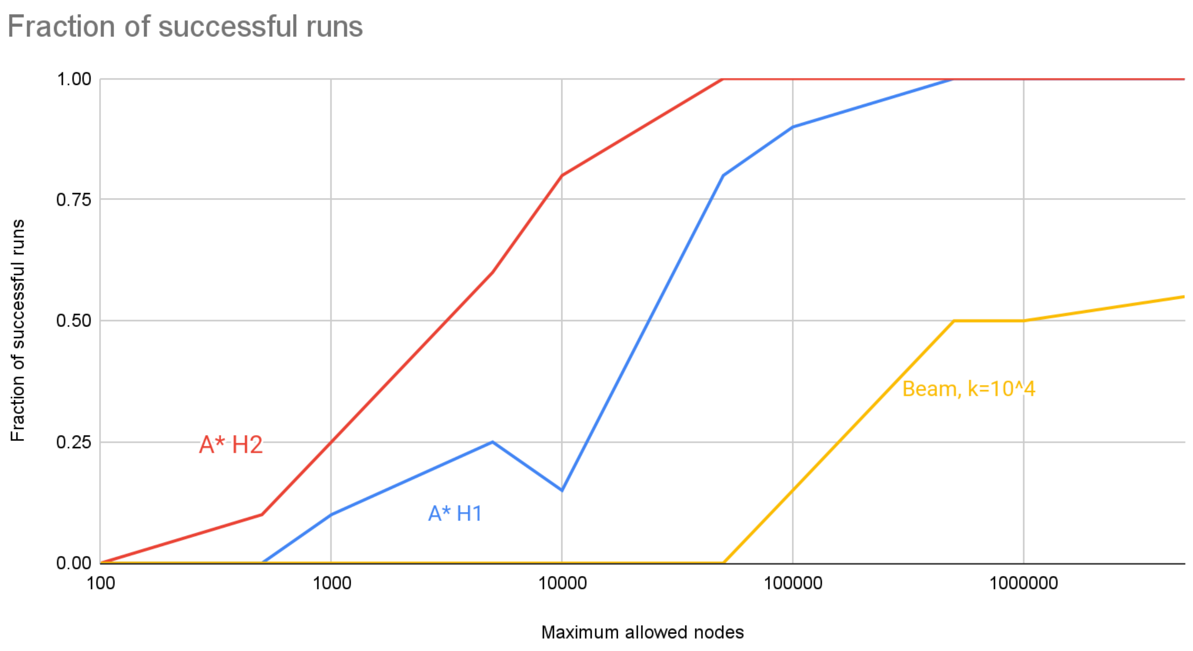
I start with a max nodes of 100 for the first trial and 500 for the next trial, and multiply by ten for each next trial, until 100,000 or 500,000:

```
for (int maxNodes = 100; maxNodes <= 100000; maxNodes *=10) //OR
for (int maxNodes = 500; maxNodes <= 500000; maxNodes *=10)
```

In this loop, I attempt to solve each search algorithm twenty times, fully randomizing the board each time. To randomize the board I make 500 random (unseeded) moves from the solved state. For each algorithm, I record how many runs were successful out of the twenty. Here are the results, with the data being the fraction of successful runs.

Fraction of successful runs for the 3 search algorithms at different max nodes

		Max Nodes									
		10^2	$5*10^2$	10^3	$5*10^3$	10^4	$5*10^4$	10^5	$5*10^5$	10^6	$5*10^6$
Search Alg	A* H1	0.0	0.0	0.1	0.25	0.15	0.8	0.9	1.0	1.0	1.0
	A* H2	0.0	0.1	0.25	0.6	0.8	1.0	1.0	1.0	1.0	1.0
	Beam $k=10^4$	0.0	0.0	0.0	0.0	0.0	0.0	0.15	0.5	0.5	0.55



The fraction of solvable puzzles for the A* search algorithms seem to increase proportional to $log(maxNodes)$. As it approaches 1.0, the fraction slows but does reach it and stays there. The “Manhattan distance” heuristic H2 always solved a higher fraction than the “# of misplaced tiles” heuristic H1, and reaches 1.0 sooner. Local Beam Search seems to perform the worst, only reaching 11/20 successful runs.

I chose the range of the max nodes as 100 to 500,000 because it holds the greatest variation in successful runs. Below 100 max nodes, no runs in my tests were successful. At 500,000 max nodes, both A* algorithms always succeeded while Local Beam seemed to max out at around 0.5, at least for k=10,000. An important future test would be to run local beam search with a fixed high max-node value with increasing k values.

2 - Comparing heuristics

To test the difference between the efficiency of H1 and H2, I ran an experiment where I collect the number of nodes generated by H1 and H2 on a randomized puzzle state (500 random moves from the solved state) for N=50 runs. To see if the difference was statistically significant, I performed a One Way Anova test because this data set involves one categorical independent variable, the heuristic H1 or H2, and one quantitative dependent variable, the number of generated nodes. The summary of the data is:

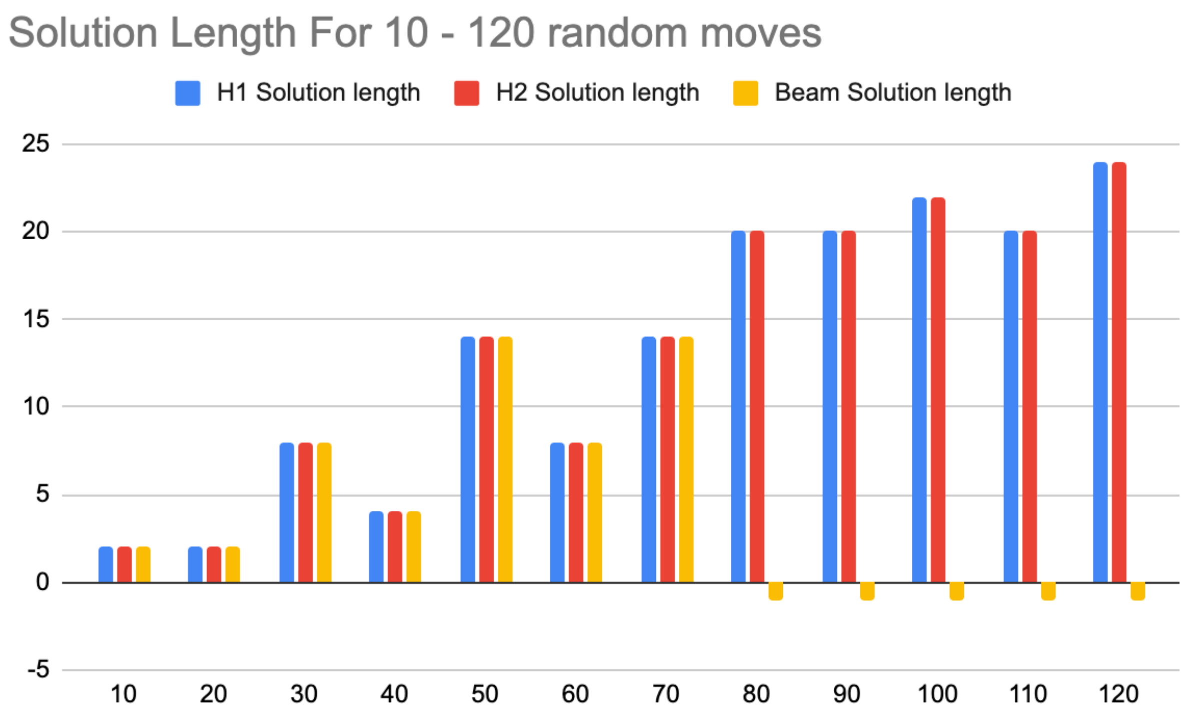
	N	Mean	St Dev
H1	50	41628	38826
H2	50	3637.6	4230.4

And the test returned an *f-ratio* value of 47.30918 with $p < .00001$. The high *f-ratio* indicates a highly significant difference between the means, so, H2 generates much less nodes than H1. In this trial, the mean number of nodes generated by H1 was about 11.4 times that of H2.

3 - Solution length by Search

To test the variance in solution length, I recorded the number of moves required for the same puzzle on puzzles with varying difficulty (varying number of random steps taken from the solved state). I started with an easy puzzle (10 random moves) and increased to more difficult puzzles (up to 500 random moves) to see if the difficulty changed variance among the solution lengths as well as to give Local Beam Search easier puzzles that it can actually solve.

In the resulting data, all three algorithms generated the same solution length for the same puzzle, so long as they did not run out of allowed nodes. I'll highlight one range of the data, the solution length for puzzles generated with between 10 and 120 random moves. Failed runs were recorded as "-1."



From 10 to 20 random moves, all three algorithms found the same length solution, after which H1 and H2 continued to find the same length solution while Local Beam ran out of nodes (increasing the maxNodes could have helped to an extent, as demonstrated in experiment 1).

To further test if they always return the same length of solution, I made another test (`experiment3B`) in which 7,000 puzzles were created all with an easy 30 random moves from the solved state. For each, I ran the three search methods and recorded the length of the solution. Then for all 1883 puzzles where all three found a solution I recorded whether the solutions were of the same length. The result obtained from the experiment output was:

```
Number of puzzles where all solutions were found: 1883
All solved puzzles match length: true
```

4 - Fraction of solvable puzzles

To find the fraction of solvable puzzles, I will take the average of what I found in experiment 1.

- The average for A* with H1 was 0.52,
- The average for A* with H2 was 0.675
- And the average for Local Beam was 0.17.

Again, A* with H2 performs the best, followed by A* with H1, then local beam.

Discussion

Based on my experiments, A Star Search with the heuristic of H2 is best suited for this problem because it is the most efficient and thus finds solutions the fastest and most often. Experiments 1 and 4 show that A* with H2 consistently solved a greater fraction of the puzzles, while A* with H1 and local beam ran out of nodes.

Experiment 2 demonstrates that compared to H1, H2 generates much less nodes. This also means that it requires less time to run, as it does not need to explore as many paths.

One interesting observation (based on Experiment 3) was that the three algorithms did not differ in terms of optimality, as long as a solution was found, meaning the solutions found were always equal in length for the three algorithms. The factor that differentiated the algorithms was instead whether they did find a solution and how quickly (how many nodes generated).

Local Beam Search consistently performed the worst, but because it does not need to store a reached set like A* does, it uses less memory. However, I was more limited in my situation by time than I was by space - while I never ran out of space on my machine, I was not able to run experiments for hours on end within the scope of this project.