Michael Carlstrom
Professor Lewicki
CSDS 391
8th October 2022

Project 1 Writeup

1. Code Design

1.1. The first 100 lines of my Puzzle.java file is all Constructors and Constructors helpers. I chose to have one private generic constructor invoked by all static Puzzle methods to keep things simple.

```
// Creates a Puzzle based off a given string state
    public static Puzzle createFromString(String stringState) {

        String[] commandInput = stringState.split(" ");
        int[][] state = new int[commandInput.length][commandInput[1].length()];
        for (int i = 0; i < state.length; i++) {

            String[] stateLine = (commandInput[i].replace('b', '0')).split("");
            for (int j = 0; j < state[0].length; j++) {
                state[i][j] = Integer.parseInt(stateLine[j]);
            }
        }
        int length = state.length;
        int width = state[0].length;
        int g = 0;
        String moveMadeTo = "";
        long diffTime = Long.MIN_VALUE;
        int nodeCount = 0;
        return new Puzzle(width, length, state, moveMadeTo, g, diffTime,
nodeCount);
    }
```

This an example of Creating a Puzzle from a String.

createFromString(String stringState) takes a string state to create a Puzzle.
createFromDimension(int n, int m) n x m dimension and creates a solved n x m Puzzle
createFromDimension(int n) n x n dimension and creates a solved n x n Puzzle
move(Puzzle p, String direction) moves a puzzle a given direction
DivieAndConquer(Puzzle p) creates a smaller puzzle

All invoke this generic constructor

```
// Private generic constructor
    private Puzzle(int width, int length, int[][] data, String moveMadeTo, int g,
long diffTime, int nodeCount) {
        this.width = width;
        this.length = length;
        this.data = data;
        this.moveMadeTo = moveMadeTo;
        this.g = g;
        this.diffTime = diffTime;
        this.nodeCount = nodeCount;
    }
```

1.2. Next I have my getters, setters and toString() methods

1.3. Next we have all the moving tile logic

```
// moves a given direction if valid
    public void move(String direction) {
        int[] swapLocation = calcSwapLocation(direction);
        if (validSwap(swapLocation)) {
            swap(swapLocation);
        } else {
            System.out.println("Cannot move edge in the way.");
        }
    }
```

calcSwapLocation(String direction) returns an int[] of x,y coordinates of the swap
validSwap(swapLocation) returns a boolean if the swap is valid
swap(swapLocation) performs the swap
validMove(String direction) returns a boolean wether a move is valid
holeLocation() finds the in[] x,y coordinates of the blank tile

1.4. Randomization of Puzzle

Randomizes the puzzle

```
// Randomizes a state n times
    public void randomizeState(int n,long seed) {
        final Random r = new Random();
        r.setSeed(seed);
        for (int i = 0; i <= n; i++) {
            move(moveOptions[r.nextInt(moveOptions.length)]);
```

```
        }
    }
```

Makes n moves and decides seeding with a given seed.

1.5. Implementation of Heuristics

This entire sections goal was to reduce having to make a compare to for each individual heuristic because I originally planned on having more but ran out of time. In practice this was extremely overkill and copy and pasting comparators would have been easier.

First is the Heuristic interface with the heuristic method

Then we have all the math for each heuristic as a class which implements the Heuristic interface.

Finally I implemented my own Comparator so I could override the default compare method.

Finally we parse strings into heuristics which will be used by the priority queue

1.6. Algorithms

1.6.1. Breath First Search

It was originally written with a regular queue but, after extensive testing to ensure A* is correct it now calls aStar without a heuristic. This was implemented to confirm A* was finding optimal solutions after extensive testing it was no longer needed. Important to note using this implementation does hurt performance because a priority queue is more intensive than a regular queue.

1.6.2. A*

```
// aStar
    public Puzzle aStar(String heuristic) {

        setNodeCount(0);
        long startTime = System.nanoTime();
        HeuristicComparator comparator = heuristic(heuristic);
        HashSet<Puzzle> pastPuzzles = new HashSet<Puzzle>(stateSpaceSize());
        PriorityQueue<Puzzle> q = new PriorityQueue<Puzzle>(comparator);
        q.add(this);
        pastPuzzles.add(this);
        addNode();
        while (!q.isEmpty()) {
            Puzzle p = q.poll();
            if (p.solved()) {
                long endTime = System.nanoTime();
                long diffTime = endTime - startTime;
                p.diffTime = diffTime;
                p.nodeCount = this.nodeCount;
                return p;
```

```
            }
            Puzzle[] puzzles = p.childrenPuzzles();
            for (int i = 0; i < puzzles.length; i++) {
                if (pastPuzzles.add(puzzles[i])) {
                    addNode();
                    q.add(puzzles[i]);
                }
            }
        }
        System.out.println("Given an invalid starting state");
        return this;
    }
```

Starts by setting the PriorityQueue and HashSet via the String heuristic
Then adds starting Puzzle to pastPuzzles and the q
Checks if puzzle is solved and returns if it is
Then gets the children of the puzzle and adds them the the hashset and the queue
Then repeated checking if puzzle is solved

1.6.3. Beam

```
// beam search
    public Puzzle beam(String string) {

        setNodeCount(0);
        long startTime = System.nanoTime();
        // Normally start with k states
        int startingK = Integer.parseInt(string);
        int k = startingK;
        HashSet<Puzzle> pastPuzzles = new HashSet<Puzzle>(stateSpaceSize());
        PriorityQueue<Puzzle> q = new PriorityQueue<Puzzle>(h2);
        q.add(this);
        pastPuzzles.add(this);
        addNode();
        while (!q.isEmpty()) {

            // At start you only have 1 state so check all the states if there is
less than
            // k
            k = startingK < q.size() ? startingK : q.size();
```

```
            // Selecting the next k children for a solution
            Puzzle[] nextPuzzle = new Puzzle[k];
            for (int i = 0; i < k; i++) {
                nextPuzzle[i] = q.poll();
                if (nextPuzzle[i].solved()) {
                    long endTime = System.nanoTime();
                    long diffTime = endTime - startTime;
                    nextPuzzle[i].diffTime = diffTime;
                    nextPuzzle[i].nodeCount = this.nodeCount;
                    return nextPuzzle[i];
                }
            }

            for (int j = 0; j < k; j++) {
                Puzzle[] puzzles = nextPuzzle[j].childrenPuzzles();
                for (int i = 0; i < puzzles.length; i++) {
                    if (pastPuzzles.add(puzzles[i])) {
                        addNode();
                        q.add(puzzles[i]);
                    }
                }
            }
        }
        System.out.println("Given an invalid starting state");
        return this;
    }
```

Initializes hashset and comparator
Local beam search is supposed to k states ahead. When there isn't k states in the queue it look the size of the queue ahead.
If it finds the puzzle in the look ahead it returns.
If not found in look ahead adds all the children puzzles for the k states
Then repeated looking at the k best states
Uses h2 as the heuristic

1.6.4. A* Divide and Conquer

This is A* where when it solves a side it recurses and solves the smaller puzzle. More will be discussed later in the Extra Credit Section.

1.7. Algorithm helpers

This is a collection of various helpers used by the searching algorithms
stateSpaceSize() calculates the puzzle State space size or sets to Integer.MAX_VALUE if the its bigger than the maximum int.
factorial(int n) returns the factorial of n
hashCode() generates a hashcode based on the 2d array of the puzzle
equals(Object o) determines if Puzzles are equal by comparing the elements with the 2d array
solved() boolean wether the puzzle is solved
bottomSolved() boolean wether the bottom row of the puzzle is solved
rightSolved() boolean wether the right column of the puzzle is solved
optimal(Puzzle bfs) boolean wether a Puzzle was solve optimally given a Puzzle solved by bfs.

2. Code Correctness

Included in the Code_Correctness are a bunch of test files to show basic correctness. It shows each algorithm solving a puzzle. This is convenient for testing but, is not super definitive in its proof. For that I would recommend looking at the data in Section 3 which is much more verbose.

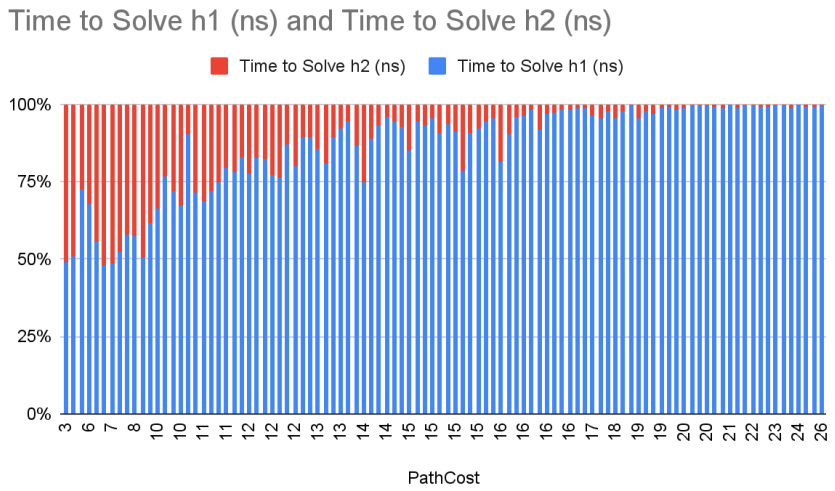
3. Experiments

Link to Data

3.1. How does fraction of Solvable puzzles from random initial state vary with the maxNodes limit?

% Solvable h1	% Solvable h2	% Solvable beam	% Solvable D&C
0.78	0.98	0.97	0.99

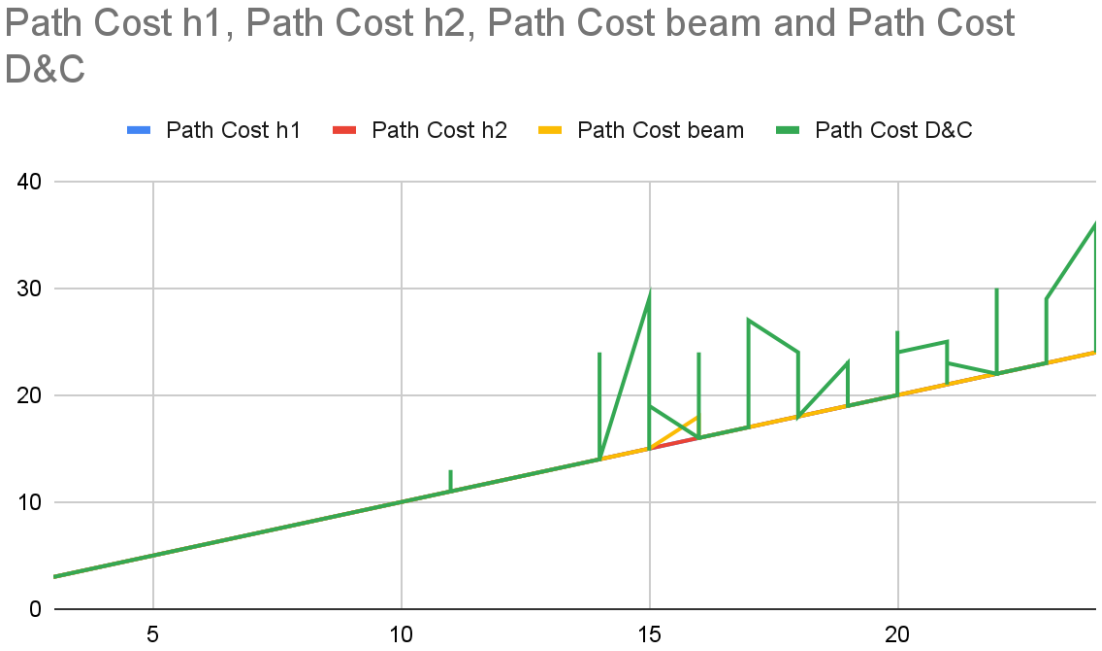
3.2. For A* search which heuristic is better?



AVGTime to Solve h1 (ns)	AVGTime to Solve h2 (ns)
7045045	11801470

H2 is vastly superior for solving the puzzle. For puzzles less than around 10 they are similar solve times.

3.3. How does the solution path vary across the search methods?



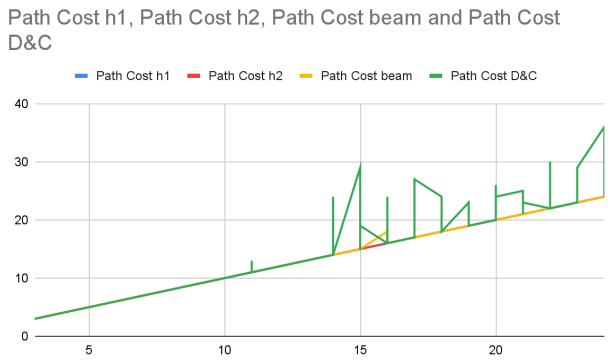
Because h1 and h2 are admissible heuristics they are guaranteed to be optimal and thus the same. Beam search was mostly optimal but occasionally nonoptimal. Divide and conquer search was extremely nonoptimal.

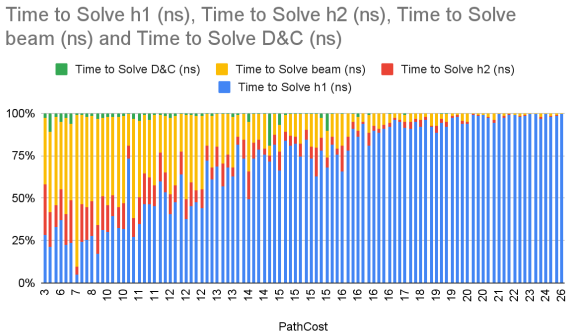
3.4. For each of the searches, what fraction of your generated problems were solvable?

For the 100 randomly generated puzzles for this experiment 100% were solvable.

4. Discussion

4.1. Best Algorithm





AVGTime to Solve h1 (ns)	AVGTime to Solve h2 (ns)	AVGTime to Solve beam (ns)	AVGTime to Solve D&C (ns)
1555898067	7045045	11801470	110399

The best optimal solution is A* with h2 heuristic. The best non-optimal solution is A* Divide and Conquer explained in the extra credit section. Divide and Conquer is also the most space efficient.

4.2. Observations

I think testing the functions asked in the project 1 paper was difficult. Different TA's had different expectations of what should be done when problems arose. Should things throw errors or just print errors. Having the input.txt file from slack was useful but, did not call every function nor did we have an expected outcome file to compare it to. In the future having more verbose input and output could be useful.

5. Extra Credit

5.1. Divide and Conquer A*

As a sacrifice of optimally we can greatly increase performance and lower memory required. To solve the larger puzzles the state space we need to shrink the statespace. The Divide and Conquere solves a side which narrows the statespace. A 5x4 compared to a 5x5 state space 43 thousand times smaller. This also happens on average to beat regular A* at 3x3.

```
    public Puzzle aStarDivideConquer(String heuristic) {

        setNodeCount(0);
        long startTime = System.nanoTime();
        HeuristicComparator comparator = heuristic(heuristic);
        HashSet<Puzzle> pastPuzzles = new HashSet<Puzzle>(stateSpaceSize());
        PriorityQueue<Puzzle> q = new PriorityQueue<Puzzle>(comparator);
        q.add(this);
        pastPuzzles.add(this);
        addNode();
```

```
        while (!q.isEmpty()) {
            Puzzle p = q.poll();
            if (p.solved()) {
                long endTime = System.nanoTime();
                long diffTime = endTime - startTime;
                p.diffTime = diffTime;
                p.nodeCount = this.nodeCount;
                return Puzzle.resetDimenstion(p);
            }

            // Stops at 3x2 State
            if ((p.width + p.length > 5)) {

                // If bottom solved shrink the problem or if bottom has been
solved and the
                // right has been solved
                if ((p.length >= p.width && p.bottomSolved()) || (p.width >
p.length && p.rightSolved())) {
                    Puzzle small = Puzzle.DivideAndConquer(p);
                    pastPuzzles = null;
                    return small.aStarDivideConquer(heuristic);
                }
            }

            Puzzle[] puzzles = p.childrenPuzzles();
            for (int i = 0; i < puzzles.length; i++) {
                if (pastPuzzles.add(puzzles[i])) {
                    addNode();
                    q.add(puzzles[i]);
                }
            }
        }
        System.out.println("Given an invalid starting state");
        return this;
    }
```

The code is basically A* except if you have solved a side it recurses on the smaller puzzle by simpling shrinking the width and length values. Importantly it sets the hashSet to null to avoid keeping multiple hashSets in memory.

Originally wanted to implement a custom heuristic that exceeded at solving sides over solving the whole puzzle but, couldn't get one that exceeded h2.

5.2. N x N Solving

Parametizes generating Puzzle to nxn. Had to add a setState = n because parsing double digit numbers was not possible. setState n only generates solved puzzle. Also had to add a check to make sure my hashSet size did not exceed maximum value for an int but, otherwise implementing an nxn puzzle was simple. Given enough time I was able to solve up to a 5x5. I did some research about better heuristics than h2 but, felt it would be against the spirit of the project if I stole those to solve the problem.