CS 3200 Final Project - Solving sliding puzzles with A*

Introduction

This project aims to solve `NxN` sliding puzzles (also known as 8-puzzle for a 3x3 configuration) using the `A*` Graph Search algorithm. The GUI for this project would allow one to manually play the game or solve it using various search heuristics. The GUI supports the following board configurations: 3x3, 4x4, 5x5, and 6x6.

The motivation behind doing this stemmed from simple curiosity. We already did an assignment using the same algorithm, but in the context of a different path-finding problem. We had a pretty good foundation to begin with and build upon. Also, the sliding puzzles seem very simple on the surface, but there is a lot more to them than meets the eye, which we'll elaborate on in the sections that follow. One of the reasons that we think this problem was important for us to work on is that it allowed us to learn that even the simplest problems can have subtle limitations that affect their solutions (or lack thereof)!

Group Members

- Brooke Snow (201540267, brooke.snow@mun.ca)
- Eeshan Garg (201520319, eg3800@mun.ca)
- Greg Berezny (201421963, gb5426@mun.ca)

Instructions

To run and test our program, simply open the `index.html` file in a web browser of your choice.

To test the search, select the `AI` mode, shuffle the board as many times as you like (you will start off with the goal configuration), pick a heuristic, and then click `Solve` ! The number of iterations, path length, and the compute time should be displayed for a given solution.

Problem Description

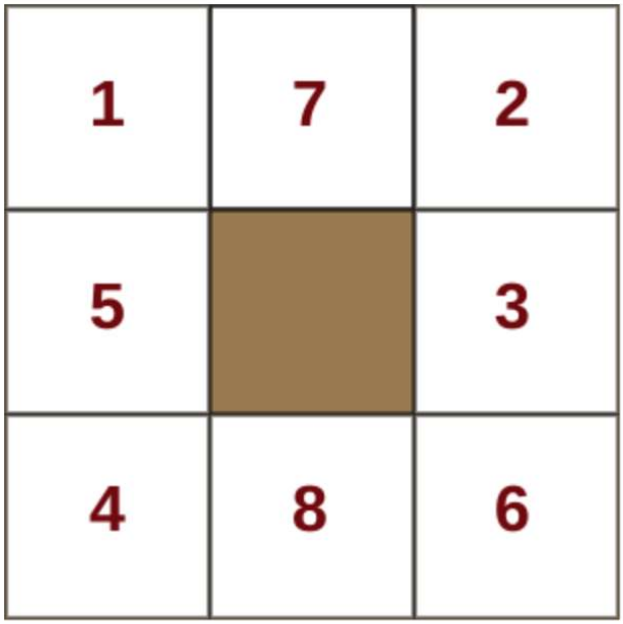
Overview

The main goal of this project is to solve numbered sliding puzzles using various search heuristics and see which ones lead to a quicker and optimal solution. Another goal of this project is to see if scrambling the puzzles using a certain number of random moves leads to any significant changes in the solution. Other things we explore are: unsolvable boards, how far a particular board configuration can be scrambled before a solution is unfeasible using the heuristics implemented, JavaScript hacks to quickly check if two states are equal and so on.

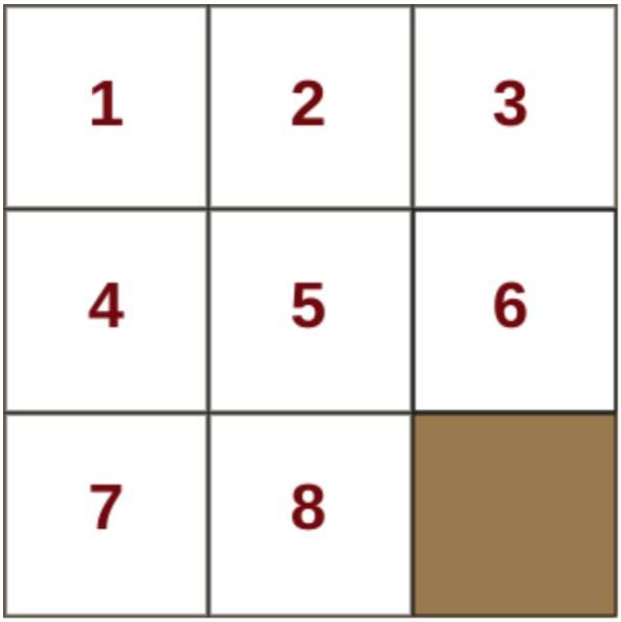
Environment and States

The environment is fully observable, deterministic, sequential, static, discrete, single-agent and has complete information.

In our implementation, we chose to model the states as 2D arrays.
Here are some possible states of the environment:



A scrambled 3x3 puzzle



The goal configuration for a 3x3 puzzle

Actions

At any given state, 2-4 tiles can be moved. For example, in the scrambled 3x3 board above, tiles 7, 5, 3, or 8 can be moved.

In our implementation, actions are calculated for a particular state and are represented as 2D arrays of the form `[x, y]`, where `x` and `y` are the co-ordinates of the tile to be moved.

Rules

The rules for the puzzle are very simple. Tiles can only be moved in cardinal directions. That's actually it.

Goal States

Unlike path finding, the goal state for a particular board configuration always stays the same and only changes if the size of the board (`N`) changes. The goal is to order all of the tiles such that when the board is read in row-major order, all the numbers are in ascending order (excluding the "blank" tile). See the sample goal configuration above.

Inversions and Unsolvable Boards

There are certain random configurations of the puzzle board which are unsolvable using cardinal actions. For instance consider the following state:

An unsolvable state (inversions = 1)

If you write the tiles in row-major order (excluding the blank space): `1, 2, 3, 4, 5, 6, 8, 7`, you'll see that 8 > 7 but 8 appears before 7. This is called an inversion.

For an N*N board, if N is odd and the number of inversions is odd, then that board is unsolvable using cardinal/legal actions. Conversely, if the number of inversions is even, the board is solvable! This makes more sense when one notes that the number of inversions in the goal state is 0 and thus, also even!

For an N*N boards where N is even, the story is a bit more complicated. If N is even, the puzzle is solvable if one of the following conditions are met:

- The blank tile is on an even row counting from the bottom (second-last, fourth-last, etc.) and number of inversions is odd.
- The blank tile is on an odd row counting from the bottom (last, third-last, fifth-last, etc.) and number of inversions is even.

One consequence of unsolvable boards is that we couldn't just generate a random board, we had to take inversions into account.

In our implementation, instead of calculating inversions for every board, we just start off with the goal configuration (a solved board) and scramble it from there using random *legal* actions. So, it is somewhat like playing the game backwards and this ensures that every board scrambled in this manner is solvable! Also, this has the additional advantage of giving us a chance to experiment and see if it is easier to find a solution for a board that has been scrambled using 10 random moves versus a board that has been scrambled using 50 random moves, and so on. Our GUI allows the user to "shuffle" the goal configuration using a custom number of random moves and then experiment with the heuristics.

To learn more, see the `start()` and `shuffle()` functions in `GUI.js`.

Methodology

Overview and Rationale

We decided to use A* (using Graph-Search) for this project. We chose this algorithm because the model of the 8-puzzle environment seemed to fit the graph-based nature of the algorithm. We were also familiar with this algorithm from our previous coursework related to path finding.

One additional advantage of A* using graph search as opposed to A* using tree search is that in Tree-Search, a node may be visited multiple times, since there is no `closed` list and no attempt is made to keep track of nodes with a better `g(n)` value already in the `open` list. This can be quite expensive. Graph-Search fixes this by keeping track of already-visited nodes in a `closed` list and ***not*** adding nodes that are worse than what is already there in the `open` list. For N*N boards with a large N, we probably want to avoid visiting the same node multiple times, therefore, it makes sense to use Graph-Search here. This becomes increasingly important when one realizes that there are certain board configurations that are so far from the goal state that a solution is unfeasible using the heuristics we chose to implement.

Pseudocode

A* using Graph Search:

```
Function AStar(problem, h)
  closed = {}
  open = PriorityQueue(Node(problem.initial_state), f=g+h)
  while (true)
    if (open.empty) return fail
    node = pop_min_f(open)
    if (node.state is goal) return solution
    if (node.state in closed) continue
    closed.add(node.state)
    for c in Expand(node, problem)
      if (node n in open with c.state and n.g ≤ c.g) continue
      open.add(c)
```

A* using Graph Search selects nodes from the `open` list with the minimum value of `f(n)`. When adding a child node to the `open` list, Graph Search checks to see if there is already a node with the same state and a lesser `g(n)` value in the `open` list. If there isn't, it adds the child node to the `open` list, otherwise, it skips to the next search iteration. Graph Search also skips to the next search iteration if the current node is in the `closed` list.

Essentially, the last three sentences of the preceding paragraph are the only difference between A* using Graph Search and A* using Tree Search.

To see our implementation of the A* Graph Search algorithm, please see the `solve()`, `searchStart()`, and `searchIteration()` functions in the `AI.js` file.

Improvements and Optimisations

We only made one slight optimization, which helped make the process of checking if two states were equal a bit simpler, if not faster. In our implementation, states are represented using 2D arrays and every `Node` object has to store the entire 2D array. One consequence of this is that checking the `open` and `closed` list for nodes becomes extra painful because one has to iterate over each 2D array and make sure each and every element is equal to the corresponding element in the state being compared to.

We tried to make this a bit easier by calling `toString()` on a node's state. In JavaScript, for a 2D array such as `[[1, 2, 3], [4, 5, 6], [7, 8, 0]]`, calling `toString()` would return a deterministic string representation of the 2D array, such as `1,2,3,4,5,6,7,8,0`. We used these string representations in the `closed` list. Also, `self.isInOpen` in `AI.js` is an object where the key is the string representation of the state and the value is the number of times a node with the given state appears in the `open` list. In retrospect, this probably made a few look-ups faster, but not by much.

As for `toString()`, there must be some sort of iteration that goes on inside JavaScript to get the string representation of a nested array, so we are not entirely sure how much these "optimizations" helped speed and memory, but they definitely made the code easier and simpler to debug. Dealing with strings was easier, not to mention that using 2D arrays as keys for an object is tricky!

Note that we have not compared speed and memory between when `toString()` representations are used versus when one just iterates over all of the 2D arrays using a `for` loop, for instance. We simply chose to use the string representations (which worked perfectly, so the motivation to explore other options decreased) and didn't have the time to crunch the numbers, so we will refrain from making any substantive claims as to the effectiveness of said "improvements".

Heuristics

We used three heuristics to guide our A* search algorithm:

- **Manhattan distance -** For each tile, compute the Manhattan distance between that tile and it's goal position and add the distances all together (excluding the blank tile). We initially thought this could be an unadmissible heuristic, but these Stanford lecture slides and some experimentation proved otherwise.
- **Misplaced tiles/Hamming distance -** For a given state, count the number of tiles that aren't in the right position (excluding the blank tile).
- **Zero heuristic -** Only the `g(n)` value is used, `h(n)` is 0. This is essentially Uniform Cost Search. Also, each "move" (sliding one tile) has a cost of 1.

Further work

If we had more time, we would have worked further on the following:

- Combine the Manhattan distance with linear conflict between pairs of tiles and use that as a heuristic, which is apparently way faster than just standalone Manhattan.
- Construct pattern databases for encountered states and use those to speed up existing heuristics. This was quite ambitious and advanced and not possible in the time frame that we were given.
- Graph visualizations for various heuristics, the number of iterations, path length and compute time over thousands of random and solvable boards. We ran out of time but such visualizations would have been better for differentiating heuristics based on performance and optimality.

Results

Overview

Our experimentation procedure was simple. For each N*N configuration, run each heuristic with shuffle length 10, 30, 50 and 100 and record the number of search iterations, the path length, and the compute time. **Note that we scramble a board and solve the same board using different heuristics for apt comparison.** Shuffle length is the number of random moves to scramble the board. This option is a part of the GUI.

Machine Specs

The machine these experiments were run on has the following specifications:

- Intel Core i7-7500 @ 2.70 GHz 2.90 GHz
- 16 GB RAM
- NVIDIA GeForce 940M GTX

Experiment Results

M = Manhattan distance
H = Hamming distance/Misplaced tiles
Z = Zero heuristic
Iter. = Number of search iterations
Time = Compute time in milliseconds
TL = Takes too long to compute (script becomes unresponsive)

Shuffle Length (3x3)	Iter. (M)	Iter. (H)	Iter. (Z)	Path length (M)	Path length (H)	Path length (Z)	Time (ms) (M)	Time (ms) (H)	Time (ms) (Z)
10	5	5	243	4	4	4	3	7	44
30	23	41	478	10	10	10	11	18	59
50	57	164	108	12	12	12	11	30	34
100	322	1008	TL	16	16	TL	60	124	TL

Shuffle Length (4x4)	Iter. (M)	Iter. (H)	Iter. (Z)	Path length (M)	Path length (H)	Path length (Z)	Time (ms) (M)	Time (ms) (H)	Time (ms) (Z)
10	7	7	8397	6	6	6	9	8	9776
30	45	406	TL	14	14	TL	35	87	TL
50	4902	TL	TL	28	TL	TL	3337	TL	TL
100	3871	TL	TL	26	TL	TL	1713	TL	TL

Shuffle Length (5x5)	Iter. (M)	Iter. (H)	Iter. (Z)	Path length (M)	Path length (H)	Path length (Z)	Time (ms) (M)	Time (ms) (H)	Time (ms) (Z)
10	5	5	TL	4	4	TL	11	9	TL
30	17	32	TL	8	8	TL	29	29	TL
50	35	268	TL	20	20	TL	37	79	TL
100	271	TL	TL	30	TL	TL	86	TL	TL

Shuffle Length (6x6)	Iter. (M)	Iter. (H)	Iter. (Z)	Path length (M)	Path length (H)	Path length (Z)	Time (ms) (M)	Time (ms) (H)	Time (ms) (Z)
10	5	5	1891	4	4	4	12	8	994
30	11	11	TL	10	10	TL	21	13	TL
50	131	898	TL	20	20	TL	61	3541	TL
100	TL	TL	TL	TL	TL	TL	TL	TL	TL

The results of the experiment were very intuitive and predictable indeed. The Manhattan distance heuristic does overwhelmingly better than the other two in every case except the simplest and smallest of scrambles, where it will occasionally tie with the misplaced tiles heuristic. Moreover all the heuristics become unfeasible for larger boards, such as 6x6 with a 100 random moves/shuffles.

Conclusion and further research

To sum up, we were able to solve 3x3 puzzles. We were able to only partially solve 4x4 and 5x5 boards, there were some instances where 200 shuffles would lead to a board that took too long to solve with the heuristics we implemented. 6x6 is somewhat solvable, but only by the Manhattan heuristic for larger shuffles/scrambles and usually the misplaced tiles heuristic does very poorly when trying to solve it. The zero heuristic is unfeasible in most cases except for 3x3 puzzles.

We do think we were successful in achieving the goals we set out to achieve at the beginning of this project! For further research (if we had more time), we would focus on implementing more advanced heuristics that can solve bigger boards, including a heuristic that combines linear conflict between pairs of tiles and the Manhattan distance. We would then build a pattern database on top of the linear conflict + Manhattan distance heuristic that maps common patterns to their `h(n)` values for fast look-ups!

References

Here are the links and resources we consulted while working on this project:

- Implementing A-star(A*) to solve N-Puzzle
- Analysis and Implementation of Admissible Heuristics in 8 Puzzle Problem
- How to check if a 8-puzzle is solvable?
- How to check if an instance of 15 puzzle is solvable?

- Easy way to generate 8 piece sliding puzzle so that it is always solvable?
- Heuristic (Informed) Search - Stanford Lecture Slides
- Graph Search vs Tree Search