CS420 Project 1 Report

Michael Vanderloo

Software Organization

The software is organized into 4 .go files.

- tile-puzzle.go contains the implementation of the tile puzzle
- agent.go contains the implementation of A*, A* nodes, and helper functions
- io.go handles both config and logging for the program
- util.go contains a few general functions that crowded tile-puzzle.go and agent.go, such as euclidean_dist with the lookup table

User Manual

All the details of compiling, execution, and IO are included in README.md under appropriate headers.

Problem Formulation

This program uses the A* algorithm to find optimal solutions to the sliding tile puzzle. The initial puzzle is a square 2d grid of tiles. This grid has a size denoting the length of 1 of the sides of the grid. The grid contains numbers 0 to n^2-1, with 0 being interpreted as an empty square that others can slide into. A 3 puzzle will contain numbers 0-8, and a 4 puzzle 0-15. The A* algorithm requires a few functions; checking the equality of states, checking if a state is final, generating successor states, getting the next state given a move, and estimating the distance from a current state to the solution.

The equality of states compares the position of all tiles. The solved state of a puzzle is having all the numbers in order from top to bottom, left to right. This puts 0 in the top left and the biggest number in the bottom right. In generating successors, there are 4 moves that can be made; up, down, left, and right. Moving up will swap the tile below empty (0) with the empty position, and so on. Moving up when empty is on the bottom is redundant and so on, because there is no change in state, thus the agent doesn't consider these moves. Getting the next state is done by applying the specified swap to the puzzle. Estimating the distance to the solution is handled by the heuristics below.

Heuristics

- H1 simply checks for each tile being in place
- H2 will calculate the Manhattan distance for each tile to its final location
- H3 converts the puzzle into a 1d array, and then counts the swaps done in the provided max sort algorithm. This algorithm works without modification because 0 is interpreted as the empty space, and thus the sorted array is the solved state.

- H4 calculated the euclidian distance of each tile to its final state and is implemented using a lookup table. This is effective because there are many repeated inputs that are expensive to calculate, but not a large number of unique values.

The A* algorithm has two lists, the open and closed list. The open list denotes a node that has been reached, but not yet evaluated. The closed list is nodes that have been evaluated, with all successors added to the open list. It starts with the open list containing the initial state, and the closed list is empty. It chooses the node from the open list with the minimum estimated cost and adds its successors to the open list if they have not yet been seen. If it has been seen but through a longer path, the node's path length and prev node are updated. The estimated cost is done by adding the cost thus far to the value of the heuristic. Thus choosing the correct path is based on how well the heuristic estimates the true path length. If the chosen node is in the solved state, the algorithm is finished. It can construct the path because each node keeps track of its previous node.

Program Performance

All puzzle generation for these configurations was done using a number of swaps that are randomly taken from an initial solved state. There is an option to specify a certain number of tiles out of place, but this seems to lead to less complex solutions as many of the tiles are 1 away from their original spot, as the generator is less likely to keep moving them because they will count towards the number of misplaced. This is still an option in the config to specify this. Each time a random puzzle is generated, the pseudo-random generator is re-seeded so two puzzles of the same size and swaps/misplaced numbers will be generated the same. These puzzle generation methods ensure that every possible input puzzle has a solution.

Random Seed: 1664776568247
Size: 3
Swaps: 100

Solution Path Length: 24

Heuristic	Execution Time (s)	Nodes Explored

h1	22.091	27296
h2	0.563	4417
h3	62.903	45251
h4	0.879	5424

It seems that the h1 and h3 are very poor heuristics based on the first input config. It is bad at identifying promising states so the number of nodes evaluated is extremely large. Though it was generated through 100 random swaps, the solution length is 20 so there was a shorter path identified than what was used while generating.

Random Seed: 1664776365346
Size: 4
Swaps: 20

Path Length: 20

Heuristic	Execution Time (s)	Nodes Explored
h1	0.178	1848
h2	0.010	446
h3	2.629	8110
h4	0.028	745

For this config, the agent seems to have used the same path as was used in generating, as the solution length is the same as the number of swaps. It is a good comparison of heuristics, showing the ordering of speed that seems to hold for every puzzle I've seen: h2, h4, h1, h3.

Random Seed: 1664776365346
Size: 4
Swaps: 21

Path Length: 21

Heuristic	Execution Time (s)	Nodes Explored
h1	0.843	4393
h2	0.038	945
h3	10.518	16423
h4	0.098	1515

This configuration is the same as the previous including random seed, except it is 1 swap further from solved than the last. This change multiplied the nodes explored somewhere from 2-2.3, and multiplied execution time by 3-6. This leads me to believe the complexity of the solution is somewhat exponential with respect to the length of the optimal solution.

Random Seed: 1664777480308
Size: 4
Swaps: 25
Heuristic: 2

Path Length: 25

Use prev node	Execution Time (s)	Nodes Explored
true	3.552	7363
false	2.477	7363

This configuration uses H2 but includes the previous node in the successor generation. Because there is no difference between making a move and undoing it, generating the previous node with the successor adds an extra linear search of the closed and open list to verify, though we already know it exists with a shorter path. This is proven by the fact that there is the same number of nodes evaluated, but an execution time almost 1.5 times in length. For all other configurations, this optimization is active.