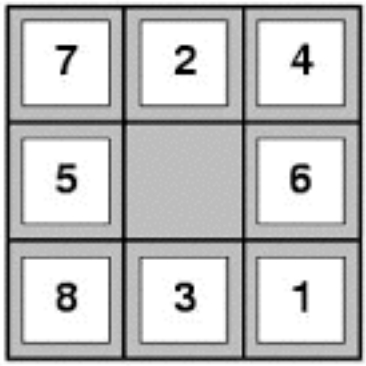
Kevin Peterson

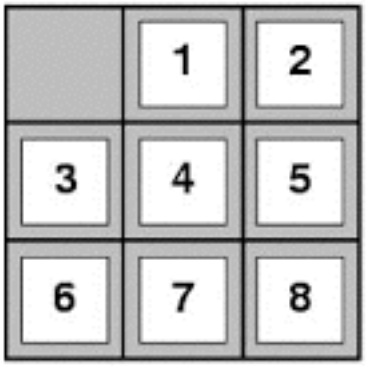
Solving the 8 Puzzle Problem with A*

The 8 Puzzle Problem

The 8 Puzzle Problem involves a board with nine squares. Eight of the squares hold a number and one of the squares is a free space. The goal is to end up with all the squares on the board in order starting at one and ending at eight, with the last square of the board being the free space. The challenge of the puzzle is that the player cannot "pickup" or swap two numbered squares. The player can only move blocks into the free space. The player must keep rearranging blocks into the free space until the board has successfully reached the goal state.



Start State



Goal State

In order to get a computer to solve the 8 Puzzle, this program implemented the A* search algorithm. A* is an algorithm that is commonly used for tree or graph traversal because it is complete and is an optimal algorithm if the heuristic function used is admissible.

Code

To implement the A* algorithm in code, data structures are required to represent nodes that A* has to search through. In my code, the nodes are simply structs that hold the board state and other information such as the f, g, h values, a unique node id, the id of that node's parent, and coordinates to track the free space of that specific node's board. I implemented my open and closed list using sets from the C++ standard template library. The open and closed lists were both represented as sets of nodes. In the code, I started with the original board, and created a node for it. I added this node to the open list. After this, the program followed a looping sequence that involved, taking the first item off of the open list, adding that item to the closed list, then expanding that node and adding it's children to the open list if an identical state wasn't already on the closed list(to prevent looping).

Sets were a great data structure to use for the lists because they are automatically sorted when elements are inserted or removed from the set. For the open list, the nodes were sorted by f values, and when two nodes had the same f values, id was used as a tiebreaker. The node with the newest (greatest) Id would be ordered first on the set. This means that the first value of the open list will always be the node with the lowest f value, which will result in the program finding the optimal path. The closed list was sorted by comparing the elements of the boards. When expanding nodes, the program has to check the closed list to be sure that state hasn't already been explored. The set find() function allows this search to have optimal speed, which wouldn't be possible with a normal linear search.

Heuristic

The program takes a command line argument that tells the program which heuristic to use. There were four different heuristics I implemented in this code. The first heuristic was simply h(x) = 0. This means that f(n) = g(n) and will result in a uninformed cost search. This heuristic will always find the optimal solution because it is admissible. The second heuristic was the number of tiles that were misplaced from their original location. We know that for the board to get to the goal state, we would at least have to move all the tiles that are misplaced one time, so this as well is an admissible heuristic and will never overestimate f(n). The third heuristic I used was the Manhattan distance. This is the sum of how far away all the nodes are from their goal positions. The Manhattan distance of a node could be represented by a line of code like this:
(abs(row - goalRow) + abs(column - goalColumn))

The fourth heuristic was a novel heuristic that I created. For this function I used a sort of modified Manhattan distance. For each tile that was misplaced, I calculator how far it would take to get the blank square to that position. Because the position would have to be open in order for you to move the right number into it. This ended up being less optimal than the Manhattan distance, but more efficient than h(n) = 0 and the number of tiles misplaced heuristics.

V Statistics (Total Number of Nodes Visited)

	V (Heuristic 0)	V (Number of Tiles Displaced)	V (Manhattan Distance)	V (Novel Heuristic)
Minimum	15	7	7	7
Median	28513	1983	372	1556
Mean	74627	6879	1120	4420
Maximum	400754	57580	6452	30990
Standard Deviation	98005	11004	1519	5937

N Statistics (Max Nodes Stored in Memory)

	N (Heuristic 0)	N (Number of Tiles Displaced)	N (Manhattan Distance)	N (Novel Heuristic)
Minimum	31	14	16	14
Median	38552	3177	620	2447
Mean	67463	10094	1815	6698
Maximum	221743	77277	10299	45083
Standard Deviation	71304	15343	2421	8808

d Statistics (Depth of Optimal Solution)

	d (Heuristic 0)	d (Number of Tiles Displaced)	d (Manhattan Distance)	d (Novel Heuristic)
Minimum	4	4	4	4
Median	18	18	18	18
Mean	17	17	18	19
Maximum	26	26	26	30
Standard Deviation	5	5	5	6

b Statistics (Effective Branching Factor)

	b (Heuristic 0)	b (Number of Tiles Displaced)	b (Manhattan Distance)	b (Novel Heuristic)
Minimum	1.6035	1.5098	1.3243	1.3976
Median	1.8024	1.5673	1.4476	1.5002
Mean	1.8320	1.5776	1.4642	1.5275
Maximum	2.3956	2.0000	2.0000	2.1147
Standard Deviation	0.1585	0.0625	0.0978	0.1106

Performance of Code With Different Heuristics

The statistics above were compiled by running the program 100 times with 100 different random boards for each heuristic. One very significant difference in the heuristics is the total number of nodes visited. With the heuristic of 0, which was the least efficient, an average of 74,627 nodes were expanded during a program run. This is very large compared to the Manhattan distance heuristic which only expanded an average of 1120 nodes per program run. The number of tiles displaced, Manhattan distance, and novel heuristic all had significantly lower means, medians, minimum, and maximums than the heuristic of zero. This provides a measurement of how valuable a good heuristic function is. The uninformed cost search visited up to 70 times more nodes than the A* search with a good heuristic function. Although V is measuring the total amount of nodes visited, it is a good indicator of time performance. We can conclude that more specific heuristics will improve time performance of A*.

The data for the N statistics are very similar to the V statistics. As the heuristics get increasingly specific, the N values go way down. This means that a specific heuristic that is still admissible will save space in addition to time performance.

The data for the d statistics were all virtually the same regardless of the heuristic. This is not surprising, considering that an optimal solution will be the same no matter how many nodes the program expands expand to find it. The means for all the heuristics were approximately 17-19.

The minimum, median, Maximum, and Standard Deviations were all approximately the same for all heuristics.
The b statistics showed that the Manhattan distance had the best performance. It had the lowest values on average for the branching factor with a mean of 1.4642. The Novel Heuristic and Number of Tiles Displaced had very similar mean values, and the heuristic of zero had the highest average branching factor.

Limitations and Improvements

The biggest limitation I originally had was searching through the closed list. I had originally searched through the closed list by using an iterator, starting at the beginning of the set, then going all the way through comparing every single board. This search method was far too slow and took up to 20 minutes for one run of the program with the zero heuristic. This limitation was improved drastically when I used the set find() method and overloaded the node < comparator to compare boards. This allowed the find() function to work optimally and sped up the closed list search. Without using a set or a similar data structure with an optimized find() function, the closed list search would take to long to ever practically use the program.

Sources

Pandare, Rohan. “Solving 8 Puzzle Using RBFS.” Google Sites, sites.google.com/site/rohanpandare/projects/solving-8-puzzle-using-rbfs.