Patrick Hawks
Ryan Larson
Rui Yang
Professor Cesare Tinelli
CS:442Ø
Course Project

Informed Search Algorithms and Heuristics

Introduction

The sliding block puzzle family provide a fertile area to develop heuristic functions for informed search strategies. This is by no means a new area in either puzzles or computer science, and puzzle problems contain a variety of strategies to use. This work takes a 10,000 foot perspective and evaluates several heuristics approaches using the traditional 8 and 15 puzzles. We are measuring the heuristic's performance on the number of nodes expanded, branching factor, time, and cost. Namely we characterize empty-tile-relaxed (Manhattan and Linear Conflict) methods, adjacency-relaxed (N-Max-Swap), and pattern databases (non-additive and disjoint pattern databases) while using a grid search of different algorithms to illustrate the trade-offs of each. We hypothesize that in order of increasing performance, the empty-tile-relaxed and adjacency-relaxed will be similar with Linear Conflict out edging N-Max-Swap, which will be both be dominated by the pattern databases.

Prior Work

Heuristics

As mentioned this is by no means a cutting edge field. The N-Puzzle provides a neatly packaged problem space, one whose complexity provides challenge for new search strategies as the number of tiles rapidly increases. While more elaborate heuristics have been developed towards this problem we will constrain our experiments to three broad heuristic families: linear, tile, and pattern based. We will briefly discuss the background and attributes of each family.

Empty-tile-relaxed: This aptly named family includes the Manhattan and Linear Conflict heuristic. The family is characterized as adhering to an adjacency constraint, meaning only tiles in adjacent rows or columns can be manipulated. The Manhattan heuristic regards each tile independently, taking the linear distance between two tiles at right angles. It is a relaxed solution to the N-puzzle, and is deterministic. In contrast the Linear Conflict calculates the Manhattan cost, and increments it's value by counting pairs of tiles that are within their correct row or column, but not their correct position. At any value N the Linear Conflict is expected to expand less nodes than Manhattan, at the cost of spending more time at each. In addition as N increases the Manhattan heuristic will suffer from a steep cost to reverse any errors.

Adjacency-relaxed: The adjacency-relaxed search strategies remove the adjacency constraint while enforcing empty tile involvement, resulting in a lower bound on the number of moves. The efficacy depends on the coding solution, randomly choosing a out-of-place tile yields better performance than an ordered solution. We are using the NMaxSwap heuristic, whose performance is similar to the empty-tile-relaxed heuristics. However the pattern databases will exceed NMaxSwap's performance.

For adjacency-relaxed heuristics, swapping the empty tile with other tiles until the goal state is achieved. The heuristics depends on the order of moves, and random moves make it be not admissible. Here we use NMaxSwap that guarantee the heuristics is admissible, we swap the empty tile with the tile whose position in goal state is the one empty tile currently is located at. Once the empty tile is moved to its goal position, we randomly pick up a tile that is not in its goal position and swap the empty tile with it. Then, moves are executed as the previous approach, until it reaches the goal state.

We can partition the tiles into different groups to prove its admissibility and why random picking up next tile to swap when the empty move to the goal position works.

3	1	2
4	6	5
7	E	8

Take a 3 by 3 puzzle for example: tiles occupied the positions of each other are considered as one group. Here 3, 1, 2 is a group, 6 and 5 is a group, E and 8 is a group. Swapping within a group i takes |group i|-1 moves, where |group i| is the cardinality of group i. Once we reordering the tiles in group that empty tile belongs to, the empty moves back its goal state. Now we need to join the empty tile into another group, which takes one move, and execute previous steps repeatedly until it reaches the goal state.

Thus, the minimum number of swaps for adjacency-relaxed heuristics is

$$\sum i=1K \ (|\text{group } i|-1) + k - 1$$

where k is the number of groups in initial state, that is NMaxSwap takes. Also, the next tile to pick up for the empty tile to swap with when one group is finished reordering doesn't matter, since it won't affect the total number of moves.

Pattern-based: The Pattern-based heuristics take advantage of pattern repetition within the N-Puzzle. The heuristic precomputes a lookup table for later use in the search. This means it will preform well in the nodes expanded at the cost of increased run time. Pattern-based heuristics can be further broken down into two further categories; non-additive and disjoint. The non-additive heuristics corner, fringe, and max use the entire board as it's index. This is opposed to the disjointed strategies, which partitions and evaluates the board horizontally, vertically or a combination of the two. We note that the Manhattan heuristic is a special case of disjoint pattern database, calculating every tile grid into a group. In theory, the disjoint should preform better than it's sibling, with a caveat: only on even number boards when it is able to partition equally. Otherwise the non-additive strategies will have similar performance.

Search Algorithms

Our A* search algorithm is general purpose. It takes as arguments:

1. An initial node (Can be any type that can be compared for equality)
2. A goal node
3. A makeNodes function which generates children of a node, and also returns the move from the current node to each child.
4. A heuristic function, which takes a node and returns the estimated distance from that node to the goal as an int. This is the h(x) function.

5. A cost function, which takes a child node and a step and returns the cost to that child by that step. This is the g(x) function.

A* will add generated nodes to a priority queue sorted by minimum value of a score which is calculated by ∑g(x) + h(x). As nodes are dequeued, they are compared to the goal node and, if they are not the goal, their children are generated and added to the priority queue.

Iterative Deepening A* works similar to uninformed Iterative Deepening Depth-First search except that the depth limit is determined by the heuristic function. Because the heuristic function must underestimate the cost of reaching the goal, we are guaranteed that any solution we find is the optimal solution.

The advantage of IDA* over A* is that the priority queue is no longer necessary, thus the memory requirements are dramatically reduced. Branches that exceed the cost limit are never explored.

The disadvantage of IDA* is that the tree may be traversed multiple times if the cost was underestimated the first time. In practice however this is not an issue, and IDA* run time was usually lower than A*

Methods

At a high level our methodology follows: First we generate puzzle boards, then run multiple experiments over the same arbitrary subset of all possible puzzles using two algorithms (A* and IDA*) with each of our heuristics over two puzzle sizes (8 and 15). For each combination we record the board size, algorithm and heuristic used along with measuring the board cost (steps in optimal solution), time (milliseconds) elapsed, number of nodes expanded, and branching factor. Finally we aggregate our results, and conduct an multivariate analysis.

The 8 and 15 puzzles are small enough that we can store all the states which are solvable in 20 or fewer moves. Thus for replication and uniformity we first generate all permutations and store them to text files. We did this by using breadth-first search plus memoization, starting from the goal state at the root and branching down via legal moves until we have generated all boards that are 20 moves or fewer away from the goal state. Because we used a breadth-first search, we could be certain that any board we

saw was solvable in no fewer moves than the depth at which it was generated. After each state is generated it is saved to a text file, for which our application has a parser function. As previously noted each board is categorized into a directory hierarchy according to the board's cost, N-number moves.

All of our tests were run on the same machine for uniformity: A Mac Pro with a 2.8GHz Quad-Core Intel Xeon processor, 16 GB of RAM, and Scala version 2.12.2. To give our solver flexibility with memory, we configured the Java Virtual Machine to start with an initial heap size of 4 GB, and a max heap size of 10 GB. We only ran out of memory once, while running A* with NMaxSwap on a 15-puzzle board that was solvable in 20 moves. Due to time constraints we are only able to perform one set of experiments.

+The 3x3 board only has 181,400 possible states. Because of it's relatively small size our first 8 puzzle experiment used A*. We use the same set of boards for each of our experiments, varying only the algorithm and heuristic. After setting the algorithm and heuristic we feed boards to our program in increasing cost. At lower N-cost boards we are able to exhaust the low number of possible states, and it isn't until {!!!stat, look for number of records by cost, 3x3} that we pick a limited number of state subsets.

Our quasi grid-search used several heuristics; Manhattan, linear conflict, non-additive, and disjoint pattern databases. The non-additive heuristic had three variations; horizontal, vertical, and maximum which uses the two previous variations. The disjoint pattern database also had fringe, corner and maximum variants. Each experiment is evaluated on four metrics, each is calculated by the formula in the table below:

Metric	Formula
Cost	N-number permutations away from goal state, ground truth from generation
Time	Elapsed time in milliseconds, taken from end time - start time
Nodes Expanded	A counter kept within the search function
Branching Factor	We use the textbook provided to calculate the E.B.F.

After completing our 8-puzzle experiments we turn towards the 15 puzzle. A*'s effectiveness is limited by it's large fringe, so we use an iterative deepening A* instead. Our second set of experiments involve a much more constrained subset of attributes, we run the algorithms on both size boards with only one heuristic. Due to it's performance in our 8-Puzzle experiments we choose the Linear Conflict as our control heuristic when comparing A* and IDA.

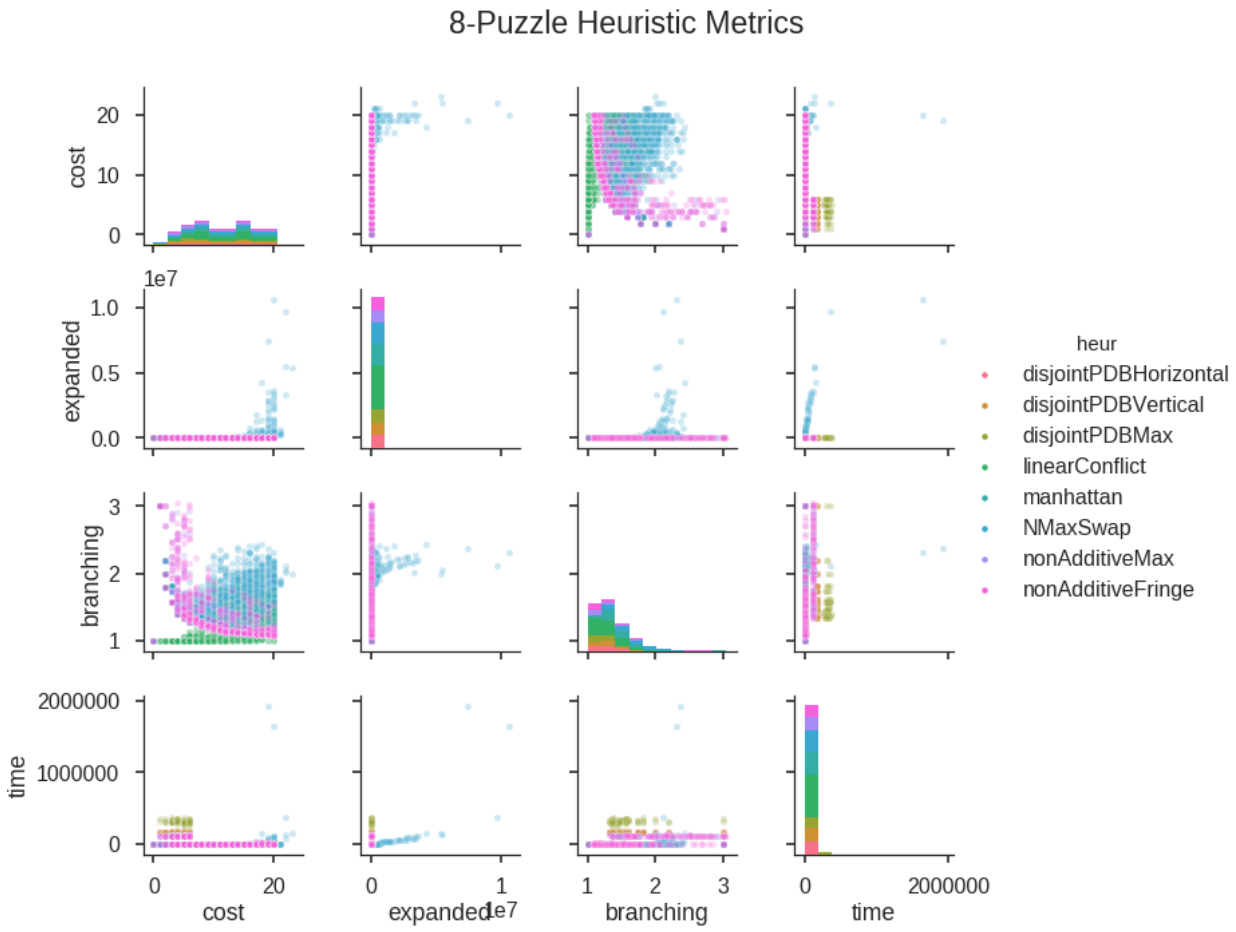
Finally, after running both sets of experiments we perform a multivariate analysis on the metrics results. This is done using the data analysis package Pandas 19.2 for Python, along with Seaborn 0.7.1 for statistical plots, and D3.js 3.0 to facilitate interaction.

algo	heur	size	boardID
ida	linearConflict	3x3	426
astar	nonAdditiveMax	3x3	426
astar	nonAdditiveFringe	3x3	426
astar	NMaxSwap	3x3	426
astar	manhattan	3x3	426
astar	linearConflict	3x3	426
astar	disjointPDBVertical	3x3	426
astar	disjointPDBMax	3x3	426
astar	disjointPDBHorizontal	3x3	426
ida	linearConflict	4x4	441
astar	nonAdditiveMax	4x4	84
astar	nonAdditiveFringe	4x4	84
astar	NMaxSwap	4x4	440
astar	manhattan	4x4	441
astar	linearConflict	4x4	441
astar	disjointPDBVertical	4x4	84
astar	disjointPDBMax	4x4	83
astar	disjointPDBHorizontal	4x4	84

After collecting our metrics; nodes expanded, branching factor, time elapsed, and cost we begin our analysis. To test our hypotheses, we calculate descriptive statistics (median, mean, standard deviation) for all costs for each heuristic. The results are informative but unsurprising; the non-additive Pattern-based heuristics had the lowest number of expanded nodes. As predicted the non-additive max performed the best with a median of 13, 17.2 mean nodes expanded. We confirm our hypothesis that the non-additive max would be the most performant with regards to nodes expanded on a small board.

	expanded			branching			time		
heur	median	mean	std	median	mean	std	median	mean	std
NMaxSwap	528	201534.68	889281.51	1.75	1.74	0.28	0.11	7.57	82.02
disjointPDBHorizontal	14	66.77	318.78	1.22	1.26	0.19	4.78	5.17	1.50
disjointPDBMax	13	20.49	31.08	1.19	1.23	0.18	6.46	6.94	1.98
disjointPDBVertical	14	90.40	326.54	1.22	1.27	0.19	4.77	5.18	1.64
linearConflict	29	1383.05	9021.14	1.29	1.30	0.22	0.05	0.08	0.15
manhattan	41.5	3806.74	19073.83	1.42	1.46	0.22	0.02	0.07	0.13
nonAdditiveFringe	15	94.64	409.75	1.20	1.33	0.32	5.11	5.49	1.61
nonAdditiveMax	13	17.15	16.90	1.18	1.24	0.20	5.11	5.50	1.66

This was in clear contrast to the Manhattan and Linear Conflict Heuristics whose mean and median expanded nodes were almost 20 and 30 nodes greater, respectively. This was not a total loss, as the two heuristics were much faster than their patterned counterparts. The median search times for Manhattan and Linear Conflict are 0.023 and 0.045 compared to the Pattern-based 5+ seconds. This is due to the pattern database generation, however the time vs. efficiency trade off could not be adequately investigated due to computational limits.



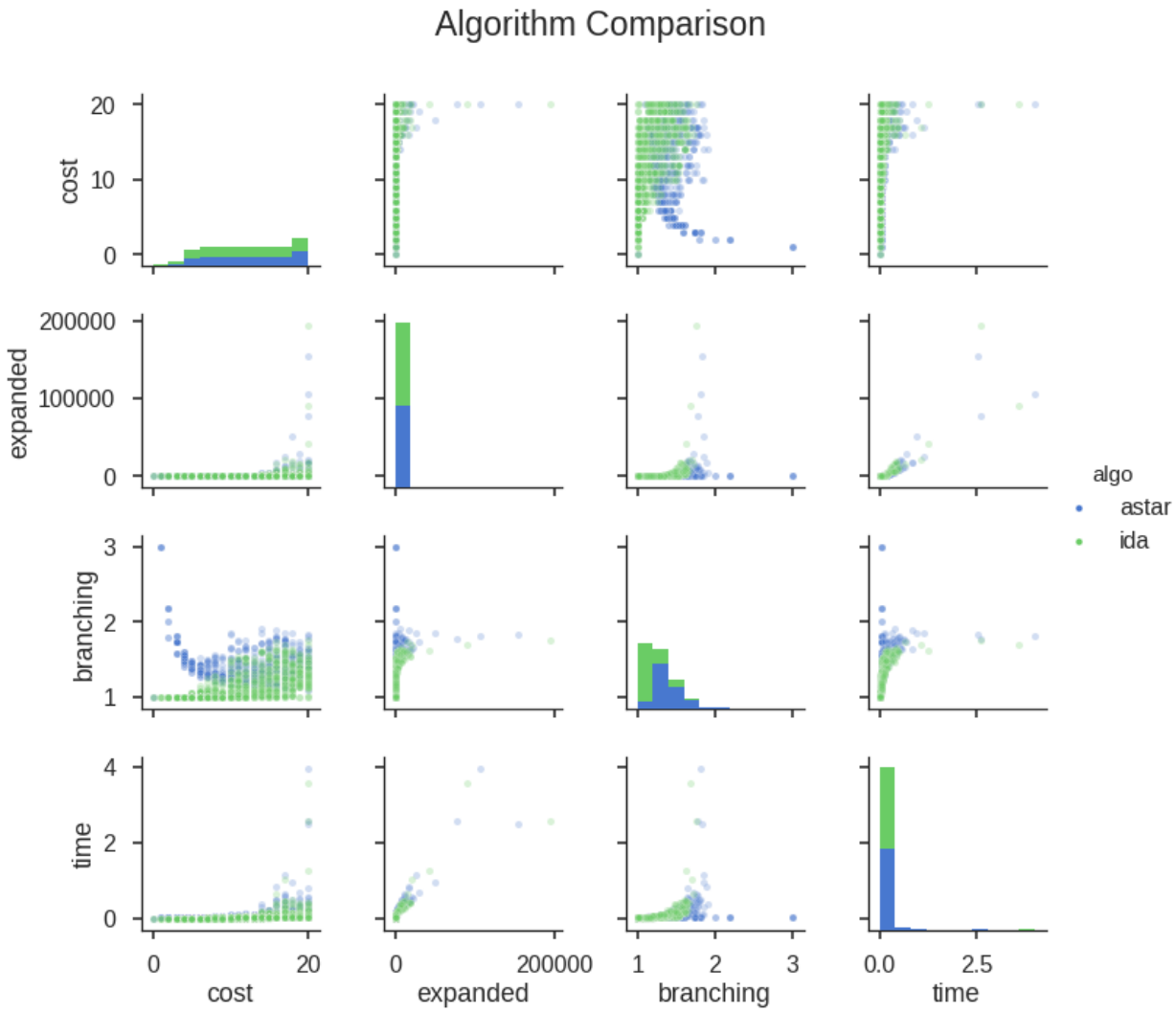
Our second set of experiments sought to evaluate A* and Iterative Deepening A*. In this section we discuss the 8 puzzle results followed by 15-Puzzle results. To this end we run the same experiments again with the algorithm used as our only experimental variable. The boards are fixed to the same subset used in the previous experiment and the only heuristic used is Linear Conflict. Across the board IDA performed slightly better than A*; mean expanded: 1278 vs. 1487, mean branching factor: 1.4 vs. 1.18, mean time: 0.1s vs 0.06s.

	expanded			branching			time		
heur	median	mean	std	median	mean	std	median	mean	std
NMaxSwap	528.00	201534.68	889281.51	1.75	1.74	0.28	0.11	7.57	82.02
disjointPDBHorizontal	14.00	66.77	318.78	1.22	1.26	0.19	4.78	5.17	1.50
disjointPDBMax	13.00	20.49	31.08	1.19	1.23	0.18	6.46	6.94	1.98
disjointPDBVertical	14.00	90.40	326.54	1.22	1.27	0.19	4.77	5.18	1.64
linearConflict	29.00	1383.05	9021.14	1.29	1.30	0.22	0.05	0.08	0.15
manhattan	41.50	3806.74	19073.83	1.42	1.46	0.22	0.02	0.07	0.13
nonAdditiveFringe	15.00	94.64	409.75	1.20	1.33	0.32	5.11	5.49	1.61
nonAdditiveMax	13.00	17.15	16.90	1.18	1.24	0.20	5.11	5.50	1.66

IDA's lead during the 15-Puzzle grew on all metrics as well; mean expanded: 684 vs. 919, mean branching factor: 1.37 vs. 1.1, and a mean time of 0.1s vs. 0.05s. Another interesting results were the growth characteristics of each algorithm. Our scatter grid charts show how IDA's branching factor grows in a distinct way from A*.

Results

Our first series of experiments were all conducted using A* on 8 and 15-Puzzle permutations. We completed 426 trials using each heuristic using the 8-Puzzle, due to time constraints we were not able to run all of the heuristics to completion on the 15-Puzzle. The Empty-Tile-Relaxed and Adjacency-relaxed heuristics were able to finish 440 trials, while the Pattern-based only ran for 84 trials. The exact figures are detailed in the table below.



Discussion

Heuristic Patterns

While Linear Conflict was a good heuristic for puzzles that can be solved in 20 or fewer moves, it seems that it would not be good enough for more complicated puzzles. Already in our test data, we noticed that the number of nodes expanded started growing rapidly around 16 step solutions. We did try to solve a couple puzzles with an 80 step optimal solution (the maximum for 15-Puzzles (Korf)), and neither A* nor IDA* was able to find a solution after 10 hours.

Algorithm Discussion

We only had one test case that ran our of memory, so IDA* did not solve any puzzles that A* was unable to solve. However, IDA* was a bit faster than A* in our tests. This was a surprise. Our intuition suggested that IDA* would be no faster, and possibly slower than A*, on account of possibly restarting the search multiple times. In practice, this was not an issue, and seems to be more than offset by not having to keep sorted a priority queue of nodes.

Since memory did not seem to be a limiting factor for us, we would have liked to explore a Bidirectional A* where one tree starts from the initial state and branches toward the goal, while another tree starts from the goal and branches.

Critique

If we had more time, we would have liked to calculate the amortized time of the Pattern Database heuristics over a large number of test cases with out regenerating the database each time. Once the database was generated, these heuristics resulted in a lower branching factor and fewer nodes expanded than the linear heuristics. It seems that if we could store this database instead of regenerating it each time, it would have eventually outperformed the others. In our tests, the time it took to generate this database was far greater than the time it took other heuristics to find a solution that the overhead was never worth it.

If we had more time, we would have liked to test every puzzle solvable in 20 or fewer moves, rather than just a subset of puzzles. It is conceivable (though not particularly likely) that we just happened to find a subset that one heuristic was especially well suited for, and testing more puzzles would have eliminated this possibility.