CS51 Project Final Writeup
Solving Sliding Tile N-Puzzles With Genetic Algorithms and A*

Eric Chan ericchan@college.harvard.edu
Liam Mulshine lmulshine@college.harvard.edu
Luca Schroeder schroeder@college.harvard.edu
Ezra Zigmond ezigmond@college.harvard.edu

Video viewable at https://www.youtube.com/watch?v=5mZsh6kWxG4

1 May 2015

1 Overview

We successfully implemented our two core features: generating solvable N-puzzles and implementing a genetic approach to solving N-puzzles. Fortunately, we were able to also implement several neat extensions: using A* (best-first search) to solve N-Puzzles, implementing four different heuristic functions, extending our algorithms to work for m by n puzzles (non-square sliding tile puzzles), developing a basic command-line interface, and creating a basic dataset for comparing our two implementations (as well as the different heuristics).

We were also lucky to have been able to stick to our rigorous schedule: by the functionality checkpoint we had implemented puzzle.clj functionality, a nave A* implementation, and a preliminary genetic algorithm implementation with a 95% success rate on 2x2 puzzles. A few days later, we had dramatically improved our A* implementation and genetic approach and moved on to this write-up, our video, and some of our cool extensions (as well as of course cleaning up our code and documentation).

We decided to use Excel for graphing instead of Python since it was not particularly useful to the reader or to us to dynamically generate graphs based on output data and the interesting patterns were clearly visible in the static sets of data we created. Besides, the aim of the project was to learn something algorithmically challenging, not something visually interesting (which is also why we decided not to pursue our idea to build a GUI for displaying puzzle states and puzzle solutions, opting instead for a simple text-based command-line interface.

We were hoping to solve 4x4 sliding puzzles, or possibly even 5x5 puzzles, but we found the time and memory it took to solve these puzzles prohibitive and instead we have focused on 2x2, 2x3, 3x2 and 3x3 puzzles, which our algorithms can solve rather quickly. We collected some data on 3x4 puzzles using A*, but the size was prohibitive for the genetic algorithm, for reasons we will discuss more later.

2 Design and Implementation

The ideas of our initial spec, looking back, were a bit silly: implementing individual tiles as objects part of larger puzzle objects would have been incredibly inefficient and would have led to unintelligible code. Fortunately our final project specification was much more realistic, and indeed our signatures for the A* component are nearly identical to what we have turned in. The way we divided up the genetic algorithm into functions is quite similar to our spec, but the implementation details have changed significantly. (We think this is really cool: even though the internals of the algorithm changed, the basic outline is still similar.)

For the genetic algorithm, we found that our original idea of using lists of puzzles to represent the chromosomes was inefficient and had a poor success rate of solving. Performing mutation in the middle of a chromosome was nearly impossible with this implementation because every puzzle state following a particular puzzle was dependent on the list of moves leading up to that point, meaning that we could not mutate any one gene (puzzle state) without changing the whole chromosome. Our original approach to this was to start with very small chromosomes and have mutation grow out the solution by one random move rather than mutating in the middle of the chromosome. However, this implementation was slow and inefficient, and we realized that this sort of approach was more or less mimicked an A*-like approach in a stupid, random way, which defeated any possible advantage that the genetic algorithm might have. Instead, we went back to the original research paper we referenced about solving planning problems using genetic algorithms. The solution proposed in this paper was to use chromosomes consisting of floating point numbers between 0 and 1, in which each floating point number effectively represents a random move choice of all of the random valid moves at a state. What allows for easy evolution of the solution space is the fact that the translation of a floating point number into a move is dependent upon the state of the puzzle just before interpreting the move. We divide the max value of the floating point number, 1, by the number of possible moves in the current puzzle state. Then each move is assigned to a consecutive ranges of floating point numbers that add up to 1/(number of moves in state). For example if there are 3 possible moves in a current puzzle state then if the floating point number we are interpreting lies [0, 0.33) , we execute move 1. If it lies between [0.33, 0.67) then we execute move 2. and if it lies between [0.67, 1) then we execute move 3.

With this more intelligent floating point approach, mutation and crossover became extremely easy. To mutate a chromosome, we only have to pick a random index in the chromosome and replace the existing floating point number with a new, random floating point number. This is guaranteed to correspond to a valid move because each floating point number is interpreted in context. We accomplish cross over by randomly choosing two indices in two different chromosomes and concatenating the left section of the first chromosome with the right section of the second. Again, the chromosome is still guaranteed to correspond to legal moves. We now start the chromosomes at a much larger size and mutation keeps the length the same, however crossover can still change the length. Although this implementation was much faster, we were still getting slower performance to solve 3x3 puzzles than we wanted, which took a few minutes. We used timbre, a profiling library for clojure, to find that most of the slowdown in the algorithm came from sliding puzzles and checking valid directions of puzzles hundreds of thousands of times. To resolve this, we used Clojure's built in memoization feature which creates an associate map so that if a memoized function is called with the same arguments twice, clojure will look up the value in table rather than recomputing the function. This was great for speed, making the genetic algorithm nearly 10x faster, but ended up costing a lot of memory. This wasn't much of a problem for puzzles up to 3x3, but the genetic

algorithm could not solve a 4x4 puzzle without running out of allocated memory. We hoped to find a balance between speed and memory that would allow us to solve 4x4 puzzles, but we were ultimately unable.

For the A* team, the major problem was also efficiency. We were initially planning to implement our open set (nodes—i.e. puzzle states—that we plan on visiting) as a binary heap and the closed set using Clojure's set type (which allows constant or logarithmic lookup), but were frustrated by Clojure's immutable data-types (binary heaps only really work if you have mutable arrays, and we were not yet aware of Clojure's transients). We wanted to use the same data-type for both our open and closed sets to maintain a concise and consistent interface, however, and were thus forced to implement both our open and closed sets as vectors (arrays) in our first nave approach. After our first checkpoint, we looked again at Clojure's data structures documentation and decided that implementing both our open and closed sets as maps would be a much better approach. We ended up implementing the closed set as a hash-map, where the keys are a list of puzzle tiles (which are guaranteed unique in the closed set based on our implementation) and the values are the cost of the N-puzzle that was associated with that puzzle state (so we can judge if our path to the same puzzle state at a later point in our search is better than one that we had before). Our open set was implemented as a priority-map, with similar keys but with puzzle nodes as values (so-called TreePuzzles), where lowest cost items have the highest priority. These changes saved an enormous amount of memory (we had previously stored all the data in each traversed TreePuzzle in our closed list, which grew astronomically in size as time went on) and reduced lookup, removal, and insertion from linear time operations to constant time or logarithmic operations. In fact, our new implementation was so fast that previously diabolical puzzles that had taken close to ten minutes to solve took just five or six seconds!

For both the A* and genetic algorithm implementations, we were able to use different heuristic functions to calculate the closeness of puzzles to their respective solved states. In total, we implemented 4 heuristics: manhattan-distance, linear conflict, number of tiles out of row and column, and misplaced-tiles. Manhattan-distance is perhaps the most commonly used heuristic for determining the score of a given n-tile-puzzle. It sums the distance of each tile in a given puzzle from the tile's correct location in the puzzle's goal state along perpendicular axes (this is not euclidean distance because diagonal moves are not allowed). The heuristic function tiles-out-of calculates the number of tiles that are not in the corresponding row and column of the tiles in the goal state. Misplaced tiles' calculates the number of tiles in a given puzzle that are not in the same position as each corresponding tile in the puzzle's goal state. Linear conflict is a variant of the manhattan distance heuristic function which takes into account the fact that when there is a linear conflict in a row, the linear conflict heuristic will add 2 moves for each pair of tiles that are conflicting. If two tiles, t1 and t2 are in a row and the goal positions of both tiles are in the same row then a linear conflict is defined as having t2 to the left of t1 when the goal position of t2 is to the right of the goal position of t1.

A quick clarification about heuristic functions: heuristic function can be either admissible or non-admissible. The requirement of an admissible heuristic function is that it never returns a score that is greater than the actual number of moves left to solve the given puzzle. If an admissible heuristic is used in our A* implementation, by nature of the A* best-first-search algorithm, the solution to solving a puzzle will be optimal (the number of moves returned will be the fewest possible number of moves necessary to get from the starting puzzle to its goal state). However, although all admissible heuristics should return an optimal solution, some heuristics cause both

the A* and genetic algorithm implementations to be more time efficient. A heuristic function that returns a score that is closer to the actual number of moves remaining to solve a given puzzle optimally will decrease the amount of time spent solving a puzzle using both the A* algorithm and the genetic algorithm. For example, the heuristic function misplaced-tiles is not considered to be a strong heuristic function because it does not always give an accurate representation of the distance of a puzzle from its goal state. In addition to increasing the solve speed of a given puzzle, a better heuristic function will increase the probability that a given puzzle can be solved using the genetic algorithm.

3 Data and Analysis

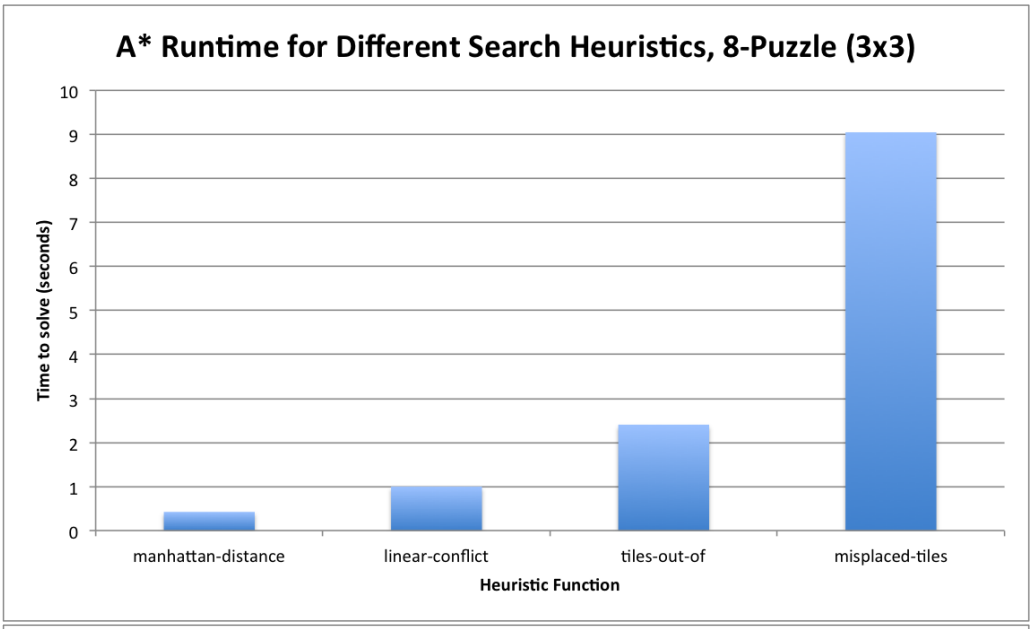
A* Runtime for Manhattan Distance Search Heuristic			
Puzzle Size	Number of Puzzles	Total Time (s)	Average Time (s)
2x2	1000	0.320930316	0.00032093
2x3	1000	8.2886176	0.008288618
3x2	1000	8.298161197	0.008298161
3x3	1000	369.8239737	0.369823974
3x4	10	200.7372551	20.07372551

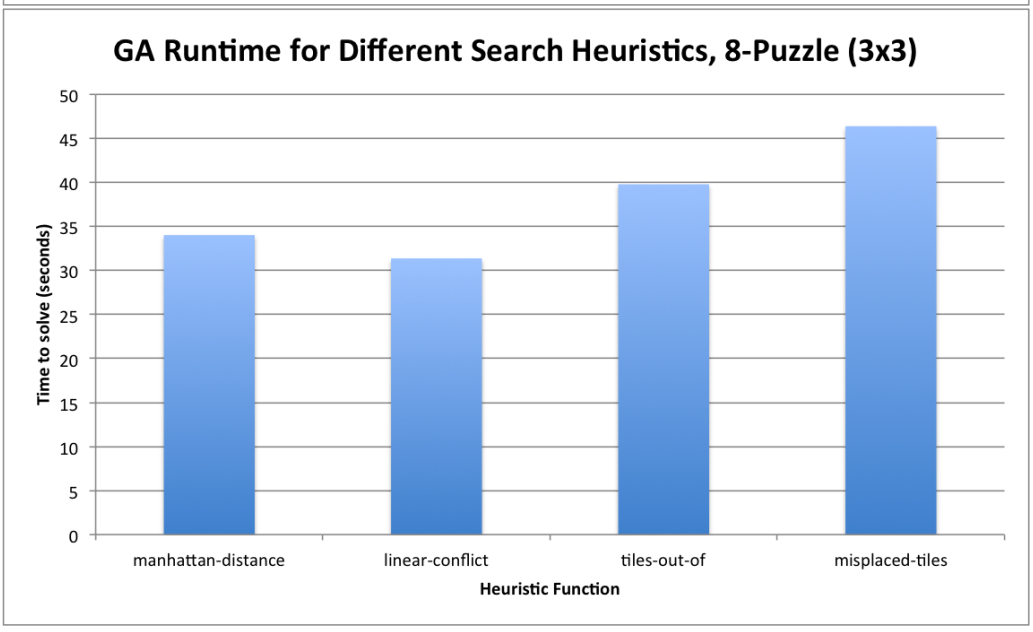
A* Runtime for Different Search Heuristics on 8-Puzzles (3x3)			
Heuristic Function	Number of Puzzles	Total Time (s)	Average Time (s)
manhattan-distance	100	42.46368304	0.4246368304
misplaced-tiles	100	904.829286404	9.048292864
tiles-out-of-place	100	240.629192361	2.406291924
linear-conflict	100	99.88026563	0.998802656

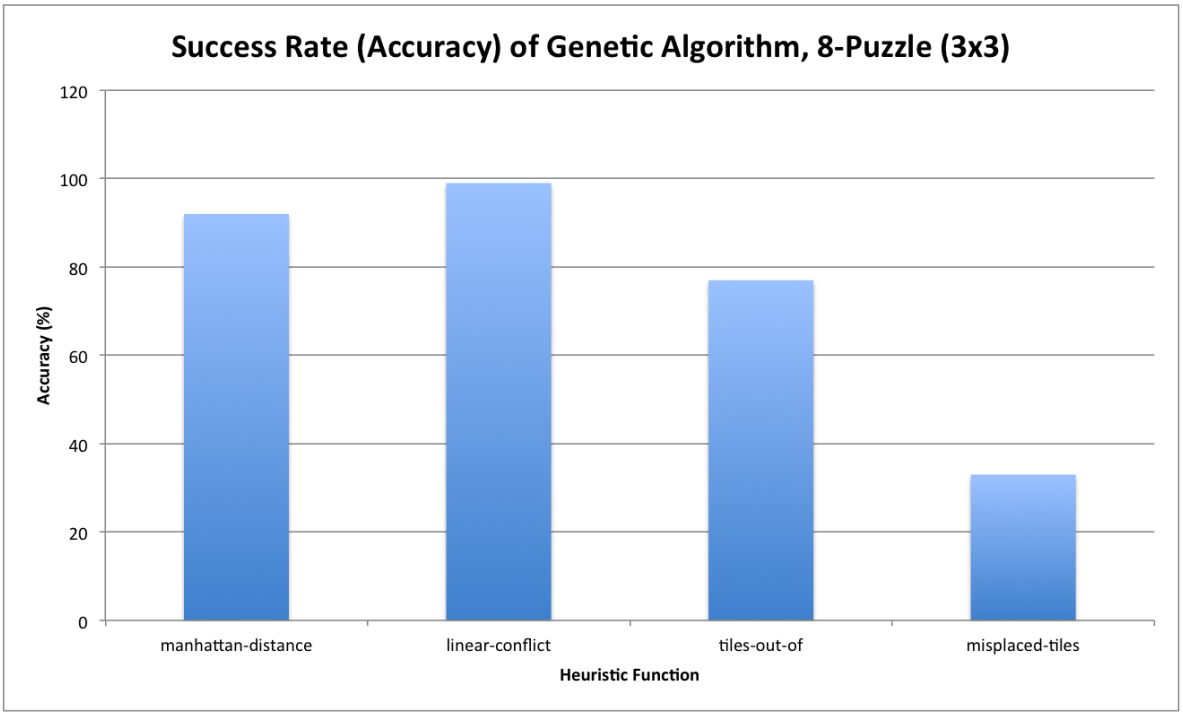
GA Runtime with Manhattan Distance Search Heuristic on 3-Puzzles (2x2)						
Trials	Pop. Size	Generations	Phases	Accuracy	Total Time (s)	Average Time (s)
1000	10	10	1	.898	6.78789569	.006787896
1000	20	10	1	.897	14.909315011	.014909315
1000	10	20	1	.897	10.978799156	.010978799
1000	10	10	2	.939	8.230210431	.00823021
1000	10	10	3	.952	8.110889346	.008110889
1000	10	10	4	.947	9.207165584	0.009207166
1000	10	10	5	.989	10.021810047	0.01002181

GA Runtime with Manhattan Distance Search Heuristic on 8-Puzzles (3x3)						
Trials	Pop. Size	Generations	Phases	Accuracy	Total Time (s)	Average Time (s)
100	100	100	5	.64	485.281	4.85281
100	200	100	5	.87	973.048	9.73048
100	100	200	5	.63	659.714771774	6.597147718
100	200	500	5	.92	3402.231	34.02231

GA Runtime with Alternate Search Heuristics on 8-Puzzles (3x3), 100 Trials						
Heuristic	Pop. Size	Generations	Phases	Accuracy	Total Time (s)	Average Time (s)
misplaced-tiles	200	500	5	.33	4638.848504199	46.388485042
linear-conflict	200	500	5	.99	3137.028180018	31.3702818
tiles-out-of	200	500	5	.77	3979.574526834	39.795745268







As the tables make clear, the genetic algorithm is much slower, anywhere from 10 to 100 times slower for 2x2 and 3x3 puzzles, depending on the settings of the genetic algorithm. It is also clear that increasing the number of generations, phases, and initial population size will generally increase the level of accuracy but also cause the genetic algorithm to take longer. Even with a large number of generations and phases, the genetic algorithm never reaches the 100% accuracy of A*. The fact that we can tune the genetic algorithm makes it a good algorithm for other applications where there are a large number of correct solutions or a concept that "closeness" to an ideal is good. Unfortunately, for sliding tile puzzles, only solutions that really solve the puzzles are valuable.

In terms of comparing the heuristic function, as is clear from the graphs above, manhattan-distance and linear conflict provide the best speed for both A* and the genetic algorithm, suggesting that they provide the best indication of how close a puzzle is to being solved, minimizing the amount of time we waste searching incorrect solutions. This is further confirmed by the fact that manhattan-distance and linear conflict provide the greatest accuracy (success rate of finding a solution within a limited number of phases and generations)–92% and 99% respectively, compared to much lower accuracies for the other heuristics. It makes sense that linear conflict is slightly more accurate and slightly slower than manhattan-distance because, as you can see in the implementation, linear-conflict computes manhattan distance and then more additional information about which tiles are blocking other tiles.

4 Lessons Learned

Perhaps the most important lesson our group learned across the board was "don't think you're hot stuff; you're a hot mess." Often we thought ourselves more clever than the documentation of the language or the peer-reviewed research we based our work on, and we certainly paid the price,

implementing things that were not entirely wrong but certainly not entirely correct—probably the most difficult scenario to debug and get yourself out of.

One instance of this happened when Liam and Luca reimplemented the A* search algorithm with hash-maps and priority-maps. Priority-maps sort by value, and the documentation for proper priority-map construction warned us that if we were to sort our map using a custom comparator, we had to be sure there was no chance of a tie (all comparators must be total orders, meaning that you can't have a tie' unless the objects you are comparing are in fact equal.) We didn't quite understand why that was a problem, did a few test cases using a comparator that wasn't total ordered (which all worked), and then implemented it that way. The end result was an implementation that somehow worked for all 2x2 puzzles but for 3x3 puzzles had the strange property of corrupting keys of the priority-map very rarely into nonsensical values. Our mistake? Turns out you really do need to respect the documentation, and our few test cases, though convincing to us, did not manage to capture potential problems.

In the genetic algorithm, Ezra and Eric didn't see why the paper we were reading had to use lists of floating point numbers to represent chromosomes, which felt indirect instead of just using lists of puzzle states. We originally implemented the algorithm using lists of puzzles as chromosomes, but it was inefficient and inaccurate. In the end, we rewrote the genetic algorithm according to the results we read in the original research paper we were referencing and found a significant increase in speed and efficiency.

The second lesson learned was really the value in getting familiar with the language. It's great if it allows you to code faster and use some syntactic tricks, but it's phenomenal if it allows you to be much smarter with the data structures you use. Moving from a nave implementation of A* search using vectors (arrays) to a more efficient one that made good use of Clojure's hash-maps and the data library's priority-map, as mentioned above, resulted in a 50-60x performance improvement and a significant positive trend in code readability, turning previously linear time lookup, remove, and insert operations into constant or logarithmic time operations. Puzzles that took five to ten minutes to solve now took five to six seconds.

But this is not to say that our nave implementation was useless: indeed, it was only because we had implemented it sloppily that we were able to understand where we needed to improve. And that was perhaps another lesson learned: writing code is really like writing prose. You write a draft, evaluate your thesis (in this case in terms of performance and readability), make macroscopic and microscopic changes to your implementation, then rinse and repeat. In fact, what was beautiful about our efficient A* implementation, for example, was that in entirely replacing the details our previous implementation but following its signatures and general structure we wrote code that basically completely worked on the first round through (excepting the priority-map comparator hiccup). That certainly could not have been possible if we had not been building up our code one layer at a time, or if we had not clearly modularized our project and written compelling signatures.

A final insight (not really a lesson though) we gained was a real appreciation for OCaml and a static type system—going back to a dynamically-typed language after a semester was disorienting to say the least (NullPointerExceptions thrown around like candy, for one). Additionally, Clojure's error messages are absolutely useless in terms of feedback and understanding what went wrong, and while OCaml isn't particularly good either in this respect, it certainly was better.

5 Division of Labor

Liam and Luca worked together on the A* implementation, with Liam later focusing on factoring out code and enforcing the modularity of our design (and implementing alternate heuristic functions) and Luca focusing on updating documentation and this writeup.

Ezra and Eric worked together on the genetic implementation, with Eric focusing a lot on the design of algorithm and understanding the papers we were reading and Ezra focusing on some of the grittier implementation details. Towards the end, Eric shifted over to working on the command line interface while Ezra finalized the genetic algorithm.

6 Thinking Differently

Clojure was a natural choice for us because we wanted to learn a new language, but a functional one that was similar to OCaml in the sense that we really wanted to be able to make use of first-class functions and immutable data types. One thing that we wish could have made better use of was Clojure's rich support for concurrency, but that was something we were only made aware of very late in coding process and after Prof. Morrisett's lecture on parallelism. If we were to do this again, however, it would have added a challenging level of complexity to implement A* with iterative deepening depth-based search that makes use of concurrency (have different threads searching simultaneously for the goal state on different branches of the tree of puzzle states).

If the A* team had more time, we would have also liked to implement the pattern database and recursive subgoals heuristics—these are part of an incredibly neat idea whereby you store a database full of subproblems that will culminate in a solution to the sliding tile puzzle. If you had a 4x4 puzzle, for instance, your first goal could be solving the fourth row and fourth column of the sliding tile puzzle, reducing the problem to an 8-puzzle; your second goal could be solving the third row and third column of the remaining 8-puzzle, reducing the problem to a 3-puzzle, and so forth. If this is all precomputed and stored the performance improvements are dramatic—but the complexity of such an endeavour would have been equal to another whole final project, so we obviously wouldn't have been able to realistically get to it.