CSC384 Winter 2017

N-Puzzle

Akshay Hariharan, *harihar9*
Bipen Grewal, *grewalb3*
Mohammad Umar Farooq, *farooq72*

Type of Project: Search

Roles	
Akshay Hariharan	Creating Heuristics (major), creating tests (major), project motivation (minor)
Bipen Grewal	Creating Heuristics (major), testing heuristics (major), gathering results (minor)
Mohammad Umar Farooq	Creating StateSpace (major), implementing IDA* (major), writing report (major)

April 7, 2017

1 Project Motivation

The inspiration for this assignment comes from a video game mini-game puzzle. In Runescape, you occasionally encounter a minigame that has a 25-puzzle. Rather than use a numbering system to represent a goal state of order, this game uses a scrambled picture that must be put back together. In lecture we also discussed the 8-puzzle and 15-puzzle variations of this problem when going through uninformed and heuristic based search.

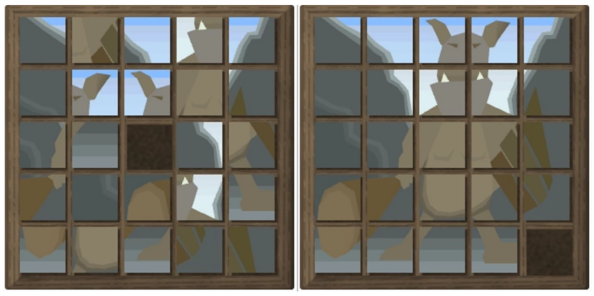


Figure 1: A minigame that involves a 25-puzzle variation

The N-Puzzle problem is a good platform to further explore search techniques. In this project, we will compare A* search and IDA* search in N-Puzzle. We also focused on creating a good heuristic to guide the search and explored what techniques bring a good heuristic with respect to run-time and states explored.

2 Methods

An N-Puzzle problem of size consists of $n!$ states, $n!/2$ of which are solvable [2]. Any order of legal moves, starting from a solved board, will yield a solvable state. It is clear that the number of states, with respect to n, grows as factorial function. N-Puzzle has also been proven to be NP-Complete to obtain optimal solutions (some N^3 algorithms exist for non-optimal). Because of this, we decided to use the A* and IDA* search techniques, both of which yield optimal solutions, to solve this problem.

2.1 N-Puzzle State

To formulate our problem, we created an N-Puzzle state space, inheriting from code supplied in A1. To improve efficiency, we implemented the 8-puzzle and 15-puzzle state spaces separately and then also an N-Puzzle which allows you to pick a custom size. The successor function returns up to 4 states, one for each action in up, down, left, right. If an action is invalid (move off the board) then there is no successor state for that action. N-Puzzles can have a variety of goal states but we decided to go with a goal of ordering tiles from 1 to N starting from top left with the blank tile in the bottom right position:

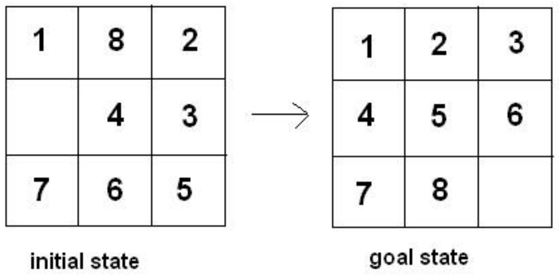


Figure 2: Initial state to goal state of the 8-puzzle variation

2.2 IDA*

The “God’s Number” is the maximum number of moves it takes to solve a puzzle. Small variations of the N-Puzzle such as 8-puzzle and 15-puzzle have known “God’s Number”. For example, in the 3×3 8-puzzle variation the God’s Number is 31 (only 2 such states, one of which we will solve in this project). For the 4×4 15-puzzle variation, Richard E. Korf showed that there are 17 different states that result in the God’s Number which is 80 moves[3]. For the 5×5 24-puzzle variation and beyond, there are no proven God’s Number’s yet but the current record is 208.

IDA* algorithm works by setting a cutoff value in the form of an f-value at each iteration in a DFS. At each iteration, the new cutoff value is the smallest f-value of any node that was pruned (exceeded the cutoff value) in the previous iteration. We hypothesize that IDA* will work well in N-Puzzle for two reasons:

1. N-Puzzle has a relatively small branching factor. At each state, you have a maximum of 4 possible moves. By comparison, Chess has an average branching of 35 and the game Go has a branching factor of 250. [4]

2. The optimal number of moves, upper-bounded by the “God’s Number” is relatively small. Also, the average number of moves is also relatively small. This means that, on average, there won’t be too many iterations needed.

This should mean that at each iteration we are getting a decent amount closer to the solution. Because we are not expanded a large number of states at each level (reason 1) and the f-value of the goal will be relatively close (reason 2), we hypothesize that IDA* will be able to solve more difficult problems with fewer states expanded.

2.3 Heuristics

We will also examine 5 different heuristics: Manhattan Distance, Linear Conflict, Misplaced Tiles, N-MaxSwap, and Tiles out of row and column [1].

One way to prove that a heuristic is admissible is to show that it solves a relaxed version of the problem. A relaxed problem is easier than the original so solving that will always underestimate the cost to solve the original problem.

In Misplaced Tile heuristic, we relax the problem so one move can put a tile in the correct position making it admissible. In Tiles out of row and column heuristic, one move for column and one move for row will fix a tiles position. This is also a relaxed version and so it is admissible. In Manhattan Distance heuristic, we ignore the fact that moving a tile can mess up other tiles so it is relaxed and thus admissible. In N-MaxSwap, we relax the problem to allow any tile to swap with the blank so it is also admissible. In Linear Conflict heuristic, when two tiles are in their goal row but reversed in position, we acknowledge the fact these two tiles will end up “conflicting” and so we account for this by adding vertical moves (to another row and back up) to Manhattan distance. This still does not account for tiles misplaced in the vertical moves so it is still relaxed and thus also admissible.

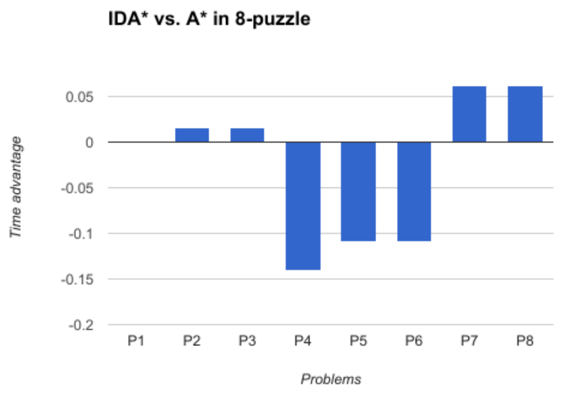
We hypothesize that the least relaxed problems will be the best performing. Linear Conflict will be the most accurate, followed by Manhattan distance. Misplaced tiles will be the least performing heuristic.

3 Evaluation and Results

Everything will be run on the same machine, for the same problems and, if required, using the same heuristic to ensure consistency. Sample problems were selected from https://www.cs.princeton.edu/courses/archive/spr10/cos226/checklist/8puzzle.html. We selected problems that were on the medium to difficult scale to magnify the effects each search technique and heuristic has.

3.1 IDA*

For the IDA* search, we first create a SearchEngine employing DFS with full cycle checking and then iteratively restart the search with a new costbound (until timeout or solution found/impossible). The costbound pruning works by checking if the (fval = gval + hval) of a successor node is greater than the costbound and if it is, it gets pruned (and thus not further explored). The initial costbound pruning is set to the fval of the initial state and after that, the costbound at the next iteration will be set to the smallest fval not explored in the current iteration. We needed to modify search.py to keep track of the smallest fval of nodes pruned.
We ran A* and IDA* searches on 8 problems for both 8-puzzle and 15-puzzle. The graphs show the time advantage that IDA* has over A*. If the bar is in the positive, it means IDA* took less time whereas if it's in the negative, IDA* took a longer time than A* for that problem.



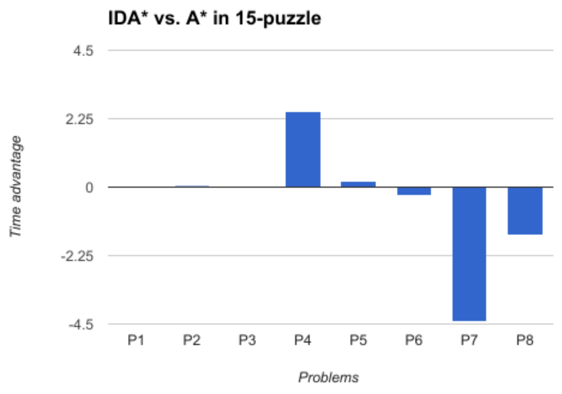
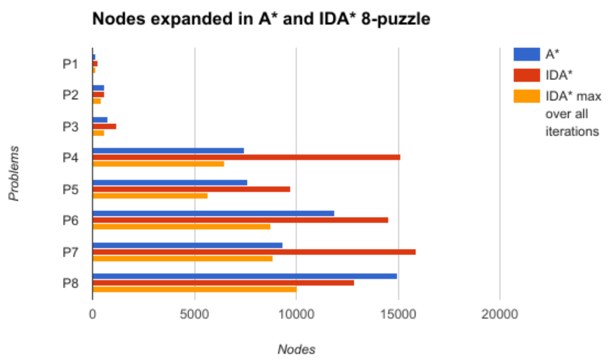


Figure 3: Advantage of IDA* over A* (positive means saved that much time)

We see that IDA* saves time in the two most difficult 8-puzzle problems but loses time in P4, P5 and P6. In 15-puzzle, IDA* actually loses time in the difficult problems but for the easier problems, IDA* out preforms A*. From this very limited data set, it seems that A* is overall faster than IDA* (average A* time is less than average IDA* time). This is contrary to what we hypothesized. To further explore this, let's see the nodes explored:



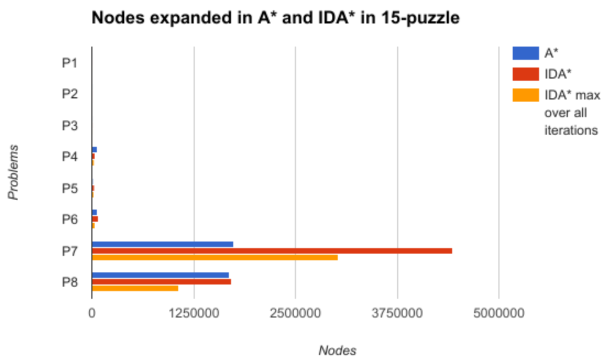


Figure 4: Nodes expanded in A* and IDA*

These graphs show the real strength of IDA*. The red is the total nodes expanded throughout the IDA* run whilst yellow shows the maximum nodes expanded of any iteration in IDA*. The total nodes expanded being more than A* explains why A* is overall faster. In lecture we discussed that a major flaw of IDA* is that it does not use something like dynamic programming so it does need to recompute many nodes and we see that indeed, this is the case. That being said, we see that for every problem except P7 in 15-puzzle, the maximum nodes expanded through all iterations is far less than in A*. This means that IDA* uses far less memory storage than A*. Although it may expand more nodes overall and thus potentially take more time, it does not need to keep storing as many nodes. This is in line with out hypothesis that IDA* will explore less nodes although there is a bit more nuance than that. IDA* expands more nodes in total but will keep less nodes in it's frontier. This also explains why our time hypothesis was wrong; IDA* actually expands more nodes overall and so it can potentially take more time. In 24-puzzle and N-puzzle variations larger, IDA* would almost become a necessity as memory becomes more an issue than time.

3.2 Testing Different Heuristics

To test which heuristics work and moreover, which work the best, we used the A* search. The time bound we set was 10 second. Once a search exceeded the time bound, further searches were not be run. Therefore how many problems and how fast for each problem, and the nodes expanded will be our metrics by which we will compare.

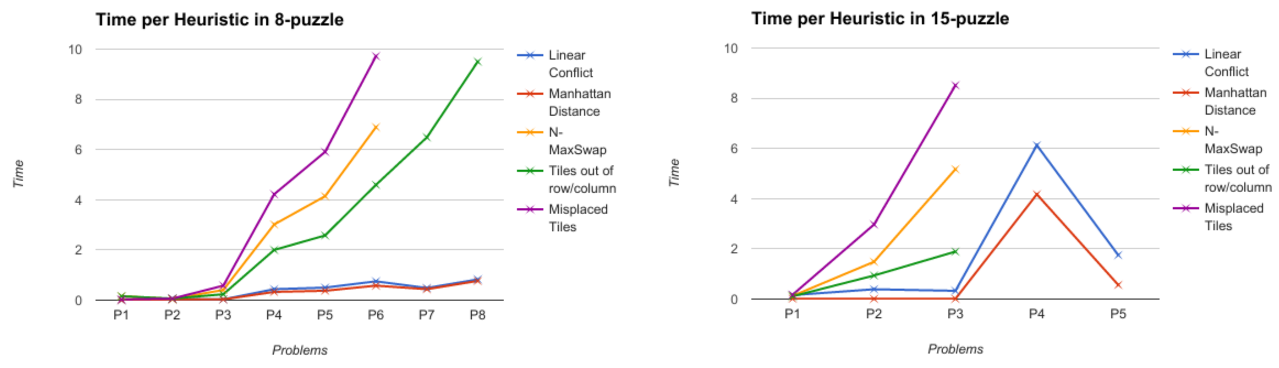


Figure 5: Time elapsed for each Heuristic per problem

From the time graph, we can see that Linear Conflict and Manhattan Distance are the two clear winners. Beyond that, misplaced tiles is the worst, N-MaxSwap and Tiles out of row/column follow respectively. If we consider by how much each heuristic relaxes the problem, this seems to agree with the performance. The more relaxed a problem becomes, the further from the original you are and the less your heuristic is helpful. That being said, Linear Conflict is Manhattan Distance plus some extra consideration for tiles that need to navigate around each other. In theory, Linear Conflict heuristically dominates Manhattan Distance as it builds upon Manhattan Distance. That is, for any state k, Heur_Lin_con$(k) \geq$ Heur_Manh_dist(k). So why does this not mean that Linear Conflict solves problems faster? If we see the nodes explored, this begins to make sense:
Linear Conflict is strictly better than Manhattan Distance; it expands less nodes. Both searches used A* so the only different variable is the h_val and indeed, Linear Conflict is better which lead to less nodes expanded. So why does it take longer if fewer nodes are expanded? This is where theory meets practicality. In theory, Linear Conflict is better but practically, a more complex heuristic, even if it is better, may not help in terms of time. Calculating a more accurate h_val takes more time and in this case, the trade-off was not worth it.

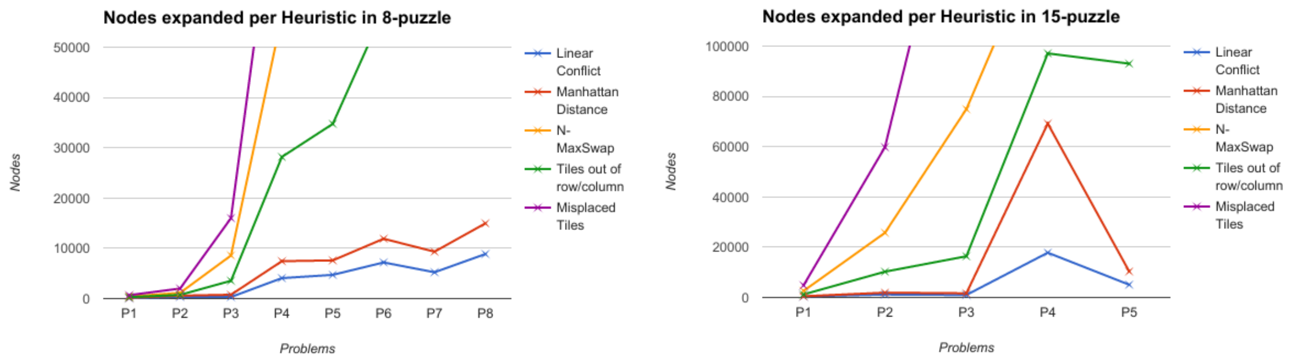


Figure 6: Nodes expanded for each Heuristic per problem

4 Limitations and Obstacles

4.1 IDA*

In section 3.1 we made the assertion that in 24-puzzle and N-puzzle variations larger, IDA* would almost become a necessity as memory becomes more an issue than time. Running these variations took far too long, especially if we have to run them multiple times during testing phase. In an ideal world, we'd have a super computer and have set up a pattern database (loading a corner and fringe pattern database for 24-puzzle took way too long) to aid in heuristics so we could compute 24-puzzles in a feasible amount of time. Because of this limitation, we can only extrapolate what happens to 24-puzzle from what we see in 15-puzzle and 8-puzzle results.

4.2 Heuristics

One of the main hurdles in this project was figuring out the heuristics. First, we came to the realisation that a pattern database would take too much time to set up and even then, it may not be worth it. Some papers we researched had pattern database where each entry was only 1 byte, the look-up time was incredibly fast and they had set it up days in advance. It is very likely that our implementation would not be at this level of efficiency. Another limitation was the runtime complexity of our heuristics. It may be the case that our implementations of these heuristics are not coded optimally. It may be that the data structures and their operations we used in some heuristics (like the lists in MaxSwap) are not efficient in Python or we simply did not program them efficiently. And finally, limiting ourselves to simpler problems and implementing a time-bound was needed. With only 5 heuristics and 13 problems we end up having 65 searches to run each time and thus it is impractical to do 24-puzzles or 81 move 15-puzzles that take hundreds of seconds instead of less than 10 seconds.

5 Conclusion

From our results we can conclude that our hypothesis regarding the time of IDA* was incorrect but was correct regarding memory usage. We can also confirm that Linear Conflict followed by Manhattan Distance are indeed the best performing heuristics but have realised that practically, a more accurate heuristic does not necessarily translate into less time.

If we were to try again in the future, time permitting, we may try to find an environment in which we can test IDA* against A* in 24-puzzle to confirm that memory problems do, in fact, bottleneck A*.

References

[1] N - Puzzle, Heuristics for the N-Puzzle. https://heuristicswiki.wikispaces.com/N+-+Puzzle. [Online; accessed 4-April-2017].

[2] William E. Johnson, Wm. Woolsey; Story. Notes on the 15 puzzle. *American Journal of Mathematics*, 2(4):397–404, 1879.

[3] Richard E. Korf. Linear-time disk-based implicit graph search. *Journal of the ACM*, 55(6):29–30, 2008.

[4] Alan Levinovitz. The Mystery of Go, the Ancient Game That Computers Still Can't Win. https://www.wired.com/2014/05/the-world-of-computer-go, 2014. [Online; accessed 4-April-2017].