Michael Liao
CSC 372

A2 Technical Report - IDA* Cube Solver in Python

Terms

- The "**cube**" refers to the 2x2x2 puzzle cube.
- A "**cubelet**," or, interchangeably a "**block**," will refer to one of the eight pieces of the cube. These can all be uniquely identified by the three colors on their faces.
- Cube "**depth**" or "**complexity**" refers to the number of turns it's been scrambled.

Framing

In this search problem, any action (a quarter-turn on the cube) corresponds to an arc cost of 1. Path cost, then, is simply the number of actions taken on the cube in order to get to any given state. The search algorithm's goal is to find a minimal path cost from the start state (a randomly scrambled cube) to a goal state (a solved cube).

Heuristic Function

The heuristic function I implemented was calculating the Manhattan distance of all cubelets by arbitrarily designating one cubelet as "in-place" and determining how many turns all others would require to be placed in their relative-correct positions.

Since the cube uses an XYZ coordinate system for each block, and each turn displaces a given block (±1) on one axis, we can find the absolute difference of the goal coordinates of the correct block and the current coordinates of the incorrect block to determine how many turns away it is from being in location (however, neighbor blocks must differ from the correct block on one coordinate, so we subtract 1 also from this value). We will perform this six times, designating a near corner and a far corner (whose coordinates are exactly opposite from the near corner's, and therefore can also be calculated using the absolute difference described above) and moving the neighbors to the same position. We then divide the total by 4 to account for the fact that a single turn moves four blocks at a time.

Admissible heuristic functions must always underestimate the actual path remaining to a goal state. This can be achieved by relaxing the actual constraints of the problem, which applies here when we stop worrying about the orientation of the blocks and about whether turning to move an incorrect block into place will affect other blocks' Manhattan distance.

Use

This solver is primarily for experimental use, and therefore when solver.py is run, it runs 10 trials each of 16 different cube complexities, starting from 0 (the trivial, solved cube) to 16. It stores all data collected (nodes expanded and time taken) during the trials into a file named *results.csv* for easier data processing in a separate application. A snippet of *results.csv* is on the right, truncated for length.

The solver uses IDA* (starting with a limit equivalent to the evaluation score of the root node), beginning its search tree with the root holding the initial state of the cube in the frontier, the unexplored set (a stack, simulating depth-first search).

results.csv

```
depth,nodes,time (ns)
0,0,64092
0,0,52673
...
0,0,39378
0,0,65279
,,
1,4,249591
1,6,283573
...
1,6,287112
1,4,280517
...
```

It then repeatedly pops the last node off the frontier until the frontier is empty, generating children of this freshly-popped node as long as its current evaluation score is less than or equal to the limit. If the current evaluation score is greater than the limit, it is compared against the current minimum evaluated score that is greater than the limit; a value which will be used as the new limit when the frontier empties.

Code Breakdown

This implementation relies on Python and the standard *time* and *math* packages (used for collecting experimental data and finding the minimum evaluation score of a node, respectively). It's a fairly standard implementation of IDA*, to my knowledge; any speed-ups performed occurred in implementing the cube's check function to account for symmetrical variations on a canonical solved state and by utilizing the symmetrical property of the cube to reduce our branching factor to 5 (each of the 12 moves mirrors another, and we can avoid one pair of those by avoiding U-turn moves).

It consists of a class Solver, which takes a scrambled cube and runs IDA* on it until a solution is found. Accompanying the Solver class is the bookkeeping class Node, which (upon instantiation) performs its designated action and evaluates the heuristic function for its state.

Results & Analysis

I also ran this program with no heuristic function (meaning the evaluation function was just the depth of the node). IDA* scales much better than IDS (iterative deepening search), but because of the specifics with how I implemented my code (Python, having to deep copy cube states each node, etc.), the amount of overhead made IDA* run comparatively the same at smaller complexities. (Graphs depicting the scalability are on the next page.)

Worth noting is that IDS, in this implementation, ran out of memory in the middle of its 8th complexity-15 trial. IDA* processed 8x less nodes on average than IDS at that complexity (11.1 million nodes vs. 93.2 million nodes), and took a little less than half than the time it took IDS to process those nodes at that complexity (32.6 minutes vs. 53.9 minutes).

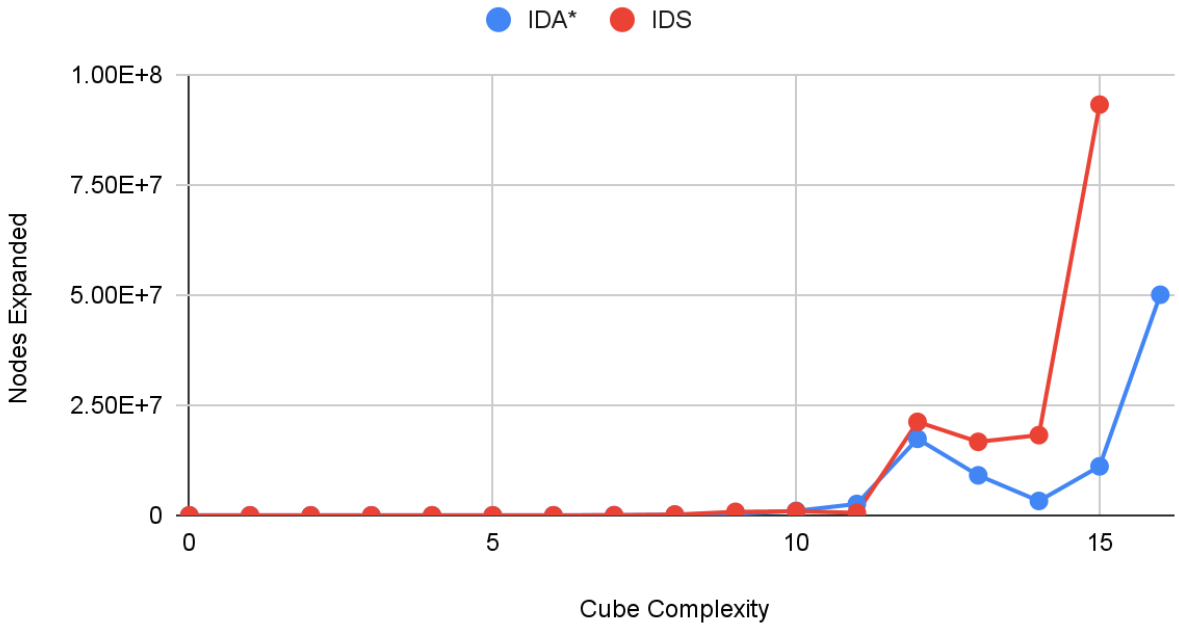
Although it is not easily visualized on the graphs, the table following it illustrates the comparative point where IDA* begins to outperform IDS (around complexity 3).

There are some oddities in the data that I cannot explain save for random chance. Both IDS and IDA* dip down in time/space used for their ten trials after complexity 12. And at trial 11, IDA* actually performs marginally worse than IDS.

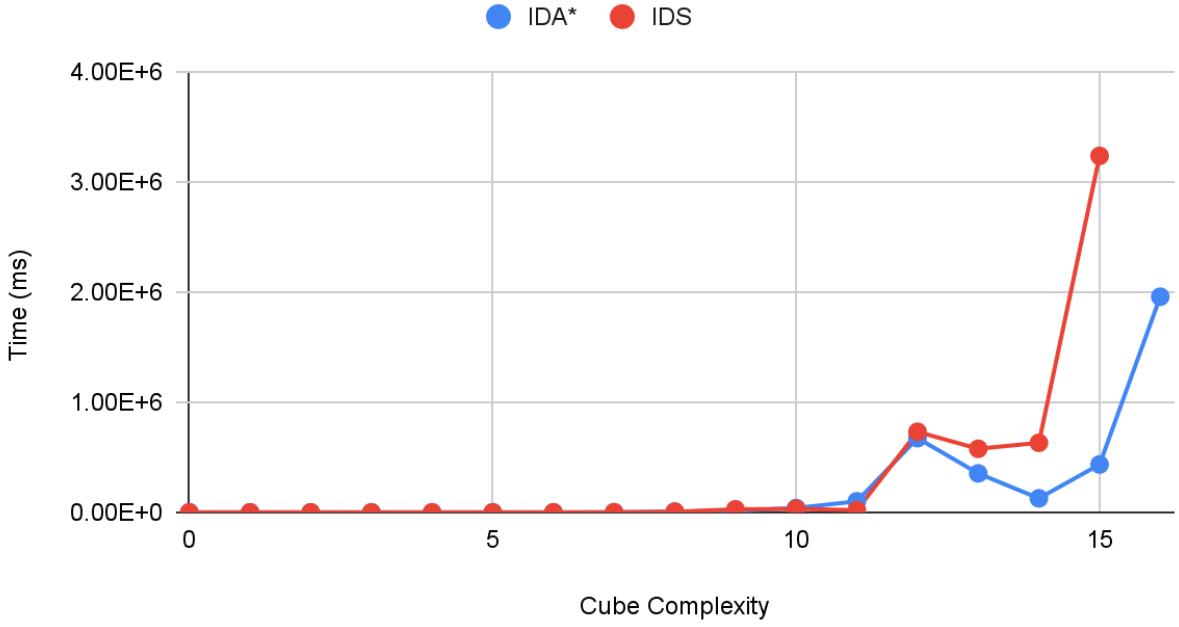
Acknowledgements & References

- My mother Haiyan helped me debug my deepCopy and check functions (mostly by grilling me about them until I figured out what I did wrong).
- Lam pointed out a potential slowdown to me in my limit-determining code within my implementation of IDA* and suggested a numbering system for my actions to make preventing U-turns easier to map.
- I implemented IDA* (specifically, the notes on how to increment the threshold for the repeated depth-limited search) from Dr. Richard E. Korf's paper, "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search." An online copy can be found here: https://www.cse.sc.edu/~mgv/csce580f09/gradPres/korf_IDAStar_1985.pdf

Cube Complexity vs. Nodes Expanded



Cube Complexity vs. Time



Cube Complexity	Nodes Expanded (IDA*)	Time (ms) (IDA*)	Nodes Expanded (IDS)	Time (ms) (IDS)
0	0	0.0509517	0	0.0432926
1	4.9	0.2754402	3.2	0.2470688
2	9	0.5598723	22	0.9424318
3	78	3.1079795	95.1	3.3865698
4	280.1	10.6671093	367.3	12.4306417
5	785	29.9645888	1906	64.0563405
6	7582.9	289.7398044	10505.2	356.6185791
7	35286	1352.464735	26134.3	892.9891691
8	177260.3	6825.451713	175612.4	6012.863286
9	349494	13465.02883	808784.3	27761.6883
10	1022196.3	39502.31644	930660.5	31970.60811
11	2589764.2	100312.8763	585348	20156.4194
12	17365117.5	673629.8444	21212409	731075.8947
13	9082393.9	353687.1532	16677768.3	577114.2485
14	3270552.4	127466.1865	18202377.4	631256.7531
15	11120158.9	434301.2911	93157829.43	3234172.761
16	50038076.7	1956334.049		