Sliding Puzzles: Optimization & korf100

Eckel, TJHSST AI1, Fall 2022

Background & Explanation

It's time to focus in on optimization. In any real world scenario, this is likely to be just as important as the model and the algorithm – what can you do to get your code to run faster, or your algorithm to work more successfully?

We'll benchmark our success on this assignment in two ways. The first you already know – the `15_puzzles.txt` file. Our goal now is to solve all of those problems. The second way is a standard benchmark that's still used to report results of new search algorithms in research projects today – the korf100. It is a set of 100 reasonably difficult 4x4 sliding puzzles. It does not include any puzzles of the maximum solution length (that's beyond our abilities even after optimization), but provides a standard benchmark to evaluate the success of your algorithm. If you can solve the entire korf100 in less than an hour, that will be incredible. We're going to aim to solve the first half of the file in that time.

Start by reminding yourself how many puzzles in `15_puzzles.txt` you can solve before a single puzzle takes longer than a minute. Let's see how much better we can get!

Fairly Simple Optimization Strategies

For starters, you can make extraordinary gains just by making your current A* implementation more efficient. One student last year reduced their runtime by 94%! Implement these three improvements and see how fast your code gets:

1) Make your heuristic calculation incremental. For Taxicab, as an example, every time you move, the Taxicab heuristic estimate is changing by exactly 1 (plus or minus) because the single tile you slide is either getting closer to, or further from, its goal state location. Each time you generate a child, instead of calculating the entire Taxicab heuristic estimate from scratch, just determine whether it increases or decreases by 1. This will save a *ton* of time.
2) Adding on to part a, you can pre-store the desired location of each tile in the goal state in a dictionary (for instance, you might have "A": (0, 0)). Then, you can make the calculation in even easier. If the tile is moving horizontally, just check if it's getting closer to or further from the column number you've already stored. If the tile is moving vertically, just check if it's getting closer to or further from the row number you've already stored. No need to find the tile in the goal state!
3) Finally, it's good to have code that can store the path, but if you don't need to know the path (just the path length) then you can use the depth variable that you've got on your tuple already to report that. Remove the code that saves the path; just save the path length.

This is the low hanging fruit; after you do this, let me know what improvements you've made. How much further can you get in `15_puzzles.txt` now?

A Much Bigger Improvement: Giving A* a Better Heuristic

The more difficult way to improve your efficiency is to improve your heuristic. The idea is to find a heuristic that is *guaranteed to never overestimate* but that still gives better estimates (ie, higher estimates) than taxicab estimation in many cases. The way to do this is by adding *row and column conflicts* to your heuristic. **A video about this is available on the website**; it's really hard to explain in text.

Implementing this along with optimizations 1-3 in the section above will produce enormous gains. Step 1 will be harder to implement with this new heuristic – you'll need to find the most efficient way to figure out the change in the taxicab + conflict heuristic score every time you slide a tile. For just taxicab, this is easy – figure out if the tile is moving closer to or further from its goal – but here, it is harder. Every time you move a tile vertically, you have to figure out if you're removing a row conflict from the row you're leaving, and if you're adding a row conflict to the row you're moving to. You may wish to keep a separate taxicab count and row/column conflict count, updating the first incrementally and recalculating the second from scratch. It's up to you. (You might also need to try a few things, like me, before you get one you like.)

For what it's worth: by applying these techniques here, and then trying several ways to calculate the row/column conflicts quickly and going with the best one, I was able to get my solve time on the entire `15_puzzles.txt` file down to just over a minute (give or take some variability between computers). You don't have to do THAT well, but you should be able to solve the entire file in a reasonable amount of time. What is a reasonable amount of time? Send me a message on Mattermost with the time you can achieve, and I'll let you know.

In addition to the `15_puzzles.txt` file, as mentioned above, I want you to try running your code on the korf100. This has been provided to you in `korf100.txt`. These puzzles are not in any particular order, but they will certainly challenge any algorithm you write! It'll be a little awkward for us to use, since it defines the solution differently – with the empty space in the top left – so modify your goal state to `".ABCDEFGHIJKLMNO"` manually and then run them.

I'd like you to try your code on the korf100. I'm very curious about how well you do. Your solve time should be LONG – closer to 1 or 2 hours than 1 or 2 minutes – so you'll need to be patient! Have your code print the lines and solve times as you work through them; if 2 hours pass and you aren't done, you can report to me how many puzzles you solved. I've written code that can solve it in something like 40 minutes, it is possible, but I do not expect you to match that!

Once you have written your code, do these three things:

- **Check in with me in person or send me a DM on Mattermost** giving me some hard numbers about how much more efficient your code got (ie, how long it takes to do the `15_puzzles.txt` file).
- **Check in with me in person or send me a DM on Mattermost** with how well you were able to do on the korf100 in 2 hours or less.
- Set up your code to read the name of a file from the command line, and **expect that file to be like `15_puzzles.txt`** (ie, only 4x4 boards with no length or algorithm specification). Make sure you're using the old goal state, **not** the one in the korf100. Your code should run your new A* on each puzzle in turn, and report how long the puzzle takes to solve.

Specification

Submit a single Python script to the link on the course website.

This assignment is **complete** if:

- You follow the instructions on the submission form to format your submission properly.
- You told me your results on the two files and I OKed them.
- Your code does accepts a file name on the command line (see instructions above).
- Your submission runtime is in accordance with your reported results.