Sliding Puzzles: Optimization & korf100

Eckel, TJHSST AI1, Fall 2023

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 `4x4_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.

Also, in case you're curious, there's a `5x5_puzzles.txt` file for you to play with.

Start by reminding yourself how many puzzles in `4x4_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 store the parent's taxicab estimate and then determine whether the child's estimate is more or less than that 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 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 `4x4_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.

This is a complicated idea to understand and an even more complicated idea to code, so deep breath.

Row/Column Conflicts

Let's say we're solving a 4x4 puzzle and the top row of the puzzle looks like this:

B A C D

Now, if we're looking at taxicab estimates for each tile, B and A would each have a taxicab estimate of 1. So, for these four tiles, the total estimate so far would be 2. But: use your brain for a moment. We can actually *guarantee* that it will take at least 4 moves to solve the four tiles on this row. Why? Well, look at the A and the B. They can't move through each other! So – one of them or the other will need to leave the row and then come back onto the row later, for an additional 2 steps. This is called a row conflict, and because we have 1 row conflict here, we can add an additional 2 steps to the heuristic estimate.

Since we still *guarantee* that these moves will be necessary, this is still a valid A* heuristic – we will never overestimate. We just have a better (ie, larger), but still valid, underestimate.

A row might have no conflicts or it may have several; in every case, the number of row conflicts on a given row is **the number of tiles that must at some point leave that row and come back for the row to become solved.** Note that this must, logically, only apply to tiles that are supposed to be on that row in the final solution, so we can ignore any tiles that belong on other rows.

Note also that the existence of a row conflict often does not imply which specific tile needs to leave the row and return. In the example above, moving either A or B down to the next row would resolve the conflict. Doing so would actually add 1 to that tile's taxicab estimate, but it would remove the 2 extra moves that were counted because of the conflict, for a total change of -1 to the estimate. Note that this still follows the other condition on the A* heuristic – a single move still cannot reduce the overall estimate by more than 1!

Here are examples of sequences of tiles you might get **in row 0**, along with the number of conflicts for that sequence. **Be sure this makes logical sense to you before moving on.**

B A C D	1 conflict, 2 additional moves necessary
C A B D	1 conflict, 2 additional moves necessary
C D A B	2 conflicts, 4 additional moves necessary (note either the A and B or C and D will need leave/return)
F A B C	0 conflicts, 0 additional moves necessary (ignore the F; the rest do not need to leave the row)
H G F E	0 conflicts, 0 additional moves necessary (ignore all 4 tiles – none of these belong on row 0)
D C B A	3 conflicts, 6 additional moves necessary
D C E A	2 conflicts, 4 additional moves necessary (ignore the E)
B D A C	2 conflicts, 4 additional moves necessary

If you aren't sure where these numbers are coming from, please stop here and ask me (or a classmate) about it.

Column conflicts work exactly the same way – look at the tiles that are on that column that will also be on that column when the puzzle is solved, and check for how many tiles would need to leave the column and return in order to solve it. Our new heuristic estimate, then, is total taxicab distance + 2 * total row conflicts + 2 * total column conflicts.

Implementing Row/Column Conflicts

So: how do we calculate how many conflicts there are on a given row or column? It turns out that this actually has a surprising connection to a problem from our Dynamic Programming interlude. This is the algorithm:

1. Consider a specific row/column.
2. Determine which tiles currently on that row/column will end up on that row/column in the solution. Consider only these tiles, preserving order.
3. Find the length of the **longest increasing subsequence** of tiles in this reduced, ordered list.
4. The number of conflicts is the number of tiles in the reduced list minus the length of the longest increasing subsequence.

Practice this on the examples given on the previous page, and convince yourself it's true. Logically, it does make sense – the longest increasing subsequence would be the number of tiles that could stay on the row/column and end up in the right place just by shifting back and forth. Any tiles not part of the longest increasing subsequence would need to leave and return.

So: now we can figure out the number of total row/column conflicts on a board. Just loop over each row and column and run the algorithm above.

But – this still faces an implementation problem! For every single board, we are going to do this complex calculation on every row and column? This is very slow, especially considering that from one board to its child, only at most 2 rows or 2 columns are changing, so there's a huge risk of repeated identical work.

After playing around with this myself in a number of ways, I found that the most efficient way to do this that was also relatively easy to code was to simply **pre-calculate the number of conflicts for every possible sequence in every possible row and column.** In other words:

1. Generate every possible permutation of tiles that could fit on one row/column. (eg, on a 4x4 board, this would be every possible 4-character permutation of "ABCDEFGHIJKLMNO.") Look up the `permutations` function in Python's built in `itertools` library; you don't need to code this yourself.
2. For each of those permutations, consider that permutation as if it were in row 0, then 1, then 2, etc. Then column 0, column 1, column 2, etc. For each placement, use the algorithm above to calculate the number of conflicts that permutation would have on that row/column. Store each result in a dictionary.
3. Later, to find the total number of row/column conflicts on each board, simply look at each row/column individually and look up the conflict number for each one in the dictionary you've made. Sum, multiply by 2, and add to the taxicab estimate you're keeping track of separately.

For a 4x4 puzzle, this pre-calculation of every possible permutation should only take around a second or so, believe it or not! For a 5x5 puzzle, it may take 2-4 minutes. (It's still worth it.)

You'll want to add this code on top of your incremental calculation from the first part of this assignment. To be totally clear: our combined A* heuristic is now the total taxicab estimate + 2 * the number of row conflicts + 2 * the number of column conflicts. But we'll still want to keep track of the taxicab estimate separately in order to continue utilizing our incremental code from before. That means that when we're calculating depth + heuristic for each child, the calculation will be this:

parent_taxicab_total + incremental_taxi_change + parent_depth + 1 + 2 * total_board_conflicts(child)

If you pre-store all of the conflict totals, and use this calculation above incrementally, you should *vastly* out-perform your previous code on the `4x4_puzzles.txt` file. You're looking for a total time for the entire file of less than 5 minutes!

Optional Additional Optimizations

Feel free to skip this section and move to the next one. These are fully not required – if you hit the under 5 minutes time above, you're good, and if you *aren't* hitting that, you probably have something wrong with your conflict implementation. But once you hit less than 5 minutes, if you want to go even faster just for fun, here are some thoughts.

- Be very careful with how you generate children. To do a single swap, converting to a list of characters (or really length 1 strings), then doing a list swap, then joining is actually faster than assembling a string with multiple concatenations.
- Can you figure out how to calculate the conflicts incrementally as well? That is, knowing only the parent's combined taxicab + 2*conflicts score, can you figure out how that score should change for a given child? This could speed things up quite a lot – instead of doing 2*size different conflict lookups on each board, you will only need to do at most 2 total. Do you see why? (This might be quite hard to code; for me, it took almost as long to write this as it took to write the entire assignment up to this point, including the conflict calculations! It's quite finicky; I had many bugs.)
- Do you really need to generate every single child every time? Is there any situation where you could determine that a child wouldn't be useful anyway and shouldn't be generated at all? (If you've implemented the incremental conflict calculation, generating each child is relatively expensive now – not just the string operations, but also the incremental changes – so if we can skip those calculations, it will help.)
- **WARNING: COMMENT THIS PARTICULAR OPTIMIZATION VERY WELL AND MAKE SURE YOU TAKE IT OUT IF YOU USE A* FOR YOUR UNIT 4 CULMINATING PROJECT.** In general, A* is only guaranteed to work as shown on the A* assignment, where we goal test each node when it comes *off* the heap. However, in this particular problem, because of the fact that the graph isn't weighted (ie, each child is exactly its parent's depth + 1), we actually *can* goal test each node when it's generated. We will never have a situation where the first time a child is added to the fringe its depth is longer than the ideal path to that child. On a weighted graph this does not hold true! So **you can't keep this change** if you do the weighted graph assignment in unit 4! But in this case, you can, and that saves a great deal of runtime.
- Finally, as we learned in dynamic programming (and when pre-storing the conflict scores above), lookup tables can speed our code up tremendously. Start your code by doing a BFS from the goal state and storing the distance from the goal state to each node you can reach in, say, 30 seconds. Save them all in a lookup table. Then, every time any future A* search arrives on one of those nodes, you can calculate the total length from there easily. This is trickier to implement than you might think, and you also have to do a lot of puzzles for the time spent on this to pay off, but it's pretty helpful on the larger, harder problem sets (the 5x5 and the korf100).

My results, if you're curious – implementing the first 4 bullet points (everything but the lookup table), I get about a 50% reduction in runtime. That translates, on my computer, to the 4x4 file running in about a minute and the 5x5 file running in just under 10 minutes (including about a minute and a half pre-calculating all the 5-character permutation conflict scores).

Implementing the final bullet point is a little harder to measure (you sort of have to use a judgment call about how much time you want to spend searching before you start solving the puzzles), but with all of these optimizations combined I was able to gain a little bit more speed on the 4x4 and 5x5 files, and more significantly get my korf100 time reduced by almost two thirds, finishing in less than 20 minutes. (So, about 40 minutes on a school laptop.)

Of course, **if you have more ideas, feel free to try them!** You're not limited to what I think of.

Benchmarking

Run three benchmarks for me:

- How long does it take to do the `4x4_puzzles.txt` file? (You're aiming for less than 5 minutes)
- How long does it take to do the `5x5_puzzles.txt` file? (You're aiming for less than an hour)
- How far can you get on `korf100.txt` ** in two hours? (This set of problems has usually been used as the canonical benchmark for search algorithms.)

****Please note:** the korf100 file defines the solution differently – with the empty space in the top left – so modify your goal state to `".ABCDEFGHIJKLMNO"` manually and then run them. This will be a little awkward, but shouldn't be too bad.

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 the results of the three benchmarks above. Also: let me know if you tried any of the other optional optimizations!
- **Get your code ready to turn in.** Specifically, set up your code to read the name of a file from the command line, and **expect that file to be like `4x4_puzzles.txt`** (ie, only 4x4 boards with no length or algorithm specification). Make sure you're using the **old goal state**, with the period in the bottom right, **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.