Sliding Puzzle Extensions

Eckel, TJHSST AI1, Fall 2021

Background & Explanation

The sliding puzzles assignment so far barely scratches the surface of what can be learned about math, problem solving, and search algorithms from this task. Solving sliding puzzles is a famous AI task that continues to be researched today. Below is a large list of possible directions you can extend your learning.

You have four options below for BLACK credit. There are two BLACK credits available on this assignment. Any choice gets you the first; any remaining choice gets you the second. Three of the assignments have submission links on the course website, the other you'll need to show me in person.

For 1 BLACK Credit, You Can… Optimize: Make Your Code Faster & Better!

See how fast you can go!

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.

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.

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, another file of difficult 15 puzzles has been provided for this part of the assignment on the website. As I mentioned, the 15-puzzle is still studied in new AI papers even today; in the literature, many papers demonstrate the success of their algorithms using a standard set of 15-puzzles called the Korf 100; 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 – but just 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 never actually assigned the korf100 before, so I'm not sure what the results will be, but I'm quite intrigued. 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 two 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). Your code should run your new A* on each puzzle in turn, and report how long the puzzle takes to solve.

Then submit to the link on the course website, following the usual formatting rules, Period Last First.

For 1 BLACK Credit, You Can… Explore: Solve a Bunch of Nifty Brainteasers!

In the mood for some mathematical explorations? Choose any two of the explorations below and complete them for full credit.

Exploration #1: Path Lengths

Write code to get answers to these questions:

1) How many distinct 8-puzzles are there with each different solution length? (You know that there are exactly 2 puzzles with length 31 solution, for example.) Write a function that will figure this out and store it in a list, to be outputted later.

2) How many 8-puzzles are there that have *multiple correct solution paths* (that is, multiple *different* sequences of moves that are each the ideal number of moves)? This can be done using a BFS, but you'll have to modify it in a couple ways. As a hint, if you're starting from the goal state, then if there are multiple paths to a node, there must then be logically multiple paths to its children as well, so this can be treated like a property that gets automatically passed to children.

   When you find your answer, separate this out by length as well – ie, how may puzzles are there with multiple correct length 1 solutions, length 2, length 3, etc. Write a function that will figure this out and store it in a list, to be outputted later.

3) Conversely, how many 8-puzzles are there that only have one unique solution? Separate these out by length as well and store them in a list. (The sum of the numbers in b and c should equal a; every solvable state either has a unique solution or doesn't.)

Output all of your info for this exploration in a readable form to the console. You'll need four tab-separated columns:

- minimum solution length
- total # of states with that solution length
- # of those states with **unique** shortest solutions
- # of those states with **multiple** shortest solutions.

Again – a good sanity check is to make sure the last two columns add up to the second one in every row!

Exploration #2: Path Restrictions

1) What if you require that the solution to a 3x3 puzzle *must* contain at least one state that has the blank in the center of the board? Are there still 181,440 possible solvable states? How many distinct 8-puzzles are there with each different solution path length? Output these answers to the console in the form of a tab-separated pair of columns, pairing each solution path length with the number of states that have that length of minimal solution path.
2) What if you require that the solution *never* contain a state that has the blank in the center of the board? Are there still 181,440 possible solvable states? Print these to the screen like in the previous question, where each line of output gives a solution length and the number of boards with that minimal solution length.

So, combining these two questions makes another set of tab-separated columns:

- minimum solution length
- total # of puzzles with that minimum solution length after requiring one state have a blank in the center
- total # of puzzles with that minimum solution length after requiring NO states have a blank in the center

…and underneath that:

- how many total solvable states exist under condition 1
- how many total solvable states exist under condition 2

Exploration #3: Symmetry

1) Finally, given (almost) any board from size 2x2 to size 5x5 as a command line input, produce a distinct board that has the same minimal solution path length *without using any search algorithms*. For example, you know that there are two 31-length 8-puzzles, 8672543.1 and 64785.321. If one of them is given on the command line, the other should be produced. This should be almost instantaneous, requiring no finding of paths at all. In fact, it should even work on 4x4 and 5x5 puzzles with paths too long to feasibly solve. Code this and make it work with a command line input as suggested. The command line input will be a single string in `sys.argv[1]` representing a puzzle of any of those sizes. You can figure out the size yourself from the string, right?
2) Note that the question said *almost* any board. Find – with code or by hand – every board that this algorithm will not work on. Find out how many there are for each size, and print that tally.

If you do Exploration 3, your code should take a single command line argument – a puzzle for Exploration #3 to process. Otherwise, it should just output. I should see **two** of these things:

- The tab-separated table that answers Exploration #1
- The tab-separated table that answers Exploration #2, followed by two additional answers
- The distinct puzzle produced by your algorithm in Exploration #3 part 1, followed by the number of puzzles of each size that your algorithm won't work for that you found in #3 part b (which can be hardcoded)

Then submit to the link on the course website, following the usual formatting rules, Period Last First.

For 1 BLACK Credit You Can… Extend: Try Some Different Puzzles Entirely!

Tired of sliding puzzles? Do something else with your search algorithms! Note: the choice of which algorithm to use is not specified in these problems; do not assume you *must* use a certain choice. Everything we've learned in this unit is available.

I give you two puzzles to solve using these techniques. You must either solve BOTH or replace one with a proposal of your own in order to receive BLACK credit. You'll submit each one separately, but **please submit both at the same time** so I can grade both and give credit all at once.

Peg Solitaire:

Go here - https://www.pegsolitaire.org/ - and play the **Triangular5(15 holes)** variant. That variant *specifically*. Many of the others are too easy to model or, on the other end, too computationally complex; this one hits the sweet spot to make an interesting but eminently solvable problem.

Model the game in whatever way makes sense to you. You **don't** have to model a game of any size; hard code the size and shape to match the one on the site exactly if that makes your model easier. You can also hardcode the list of available moves; that doesn't have to be generated mathematically.

Then, use a search algorithm to find a path from the start state to the end. You can find the actual solution to this on the web in many places, but that's not the point – the point is for *your code* to figure it out!

Once you have this working:

- Your code should **not** take a command-line argument, it should just NICELY display every board state along the way from the start to the goal. (Don't just print strings on one line, make it look like a triangular game board!)
- Please comment where in your code you generate this solution with a comment that contains the text "EXTEND LEVEL 1" so I can find it in your code easily!

Then submit to the link on the course website, following the usual formatting rules, Period Last First.

Flood:

Go here - https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/flood.html - and play a few rounds.

1) Figure out how to model this in your code. Your code should be able to accept a board state of any size. Play a couple of games and make sure your model behaves correctly.
2) Run some kind of search to find the ideal solution to the game board. Start small and build up. See how big you can go.
3) At a few different sizes, generate a puzzle on the website, copy/input that puzzle into your code, run your code, follow its instructions, and see if you get the minimal path length that the website claims.

Once you have this working, **send me a DM on Mattermost** that contains the following information:

- Instructions for how I can create a board state and give it to your code. Are you using an input statement? Reading from a file? Whatever it is, it cannot require me to open your script and edit it; I have to be able to give a state myself as input.
- How large a board your code can generate a solution for in less than a minute.

Then submit to the link on the course website, following the usual formatting rules, Period Last First. I'll follow your instructions to generate a test case of my own and run it, comparing the solution length to Simon Tatham's page.

Or, replace one of the above options…

… by maybe suggesting a different puzzle you'd like to solve? Let me know and I'm happy to tell you if I feel like it's a good challenge or not!

For 1 BLACK Credit You Can… Visualize: Work to Show How These Algorithms Work

This one is genuinely kind of a reach, but every year I have a couple students take it on and do something cool.

Find an interesting way to **visualize** your searching algorithms, particularly A*. Find a Python graphics library you like, install it, and learn how to use it well enough to accomplish this task. Can you display the nodes in visited, fringe, ancestors, etc in some kind of visually appealing way? Experiment, find something that makes you say "COOL!", and explain how it works. You might have to share your screen to show it to me; I might not have the libraries you import!

You submit this by telling me "Mr. Eckel I'd like to submit Explain Level 2" and working with me to figure out the best way to do that. This is a bad choice to drop on me 12 minutes before the due date.