DFS: Sliding Puzzles vs. Peg Solitaire

Eckel, TJHSST AI1, Fall 2023

Background & Explanation

You used BFS – breadth first search – to solve sliding puzzles. Now we're going to explore a slightly different algorithm with an intriguing set of strengths and weaknesses.

A Brief Experiment

Your first task is to try an experiment. Make one small change to the submission code you wrote for the BFS: Sliding Puzzles assignment. Instead of using `.popleft()` to remove a node from the deque, use `.pop()`. This will change the behavior of the deque object from a queue to a stack, becoming a last-on, first-off processing method. (Some of you even did this by mistake when you were writing the assignment in the first place!) Run the code again on the slide_puzzle_tests.txt file and compare with the BFS runs (or with the sample run given on that assignment). What happens?

Really, go do it – this will be much more fun if you have numbers to look at!

…ok, you really did it?

Wasn't *that* weird?

Depth First Search

So – what behavior is happening here? Why are your path lengths so much longer for some of the 2x2s and 3x3s? You probably even had the 4x4s and 5x5s not finish at all!

Well, what we have done with that small change is switch from a Breadth-First Search (one layer at a time) to a **Depth-First Search**, an algorithm that goes all the way down as far as it can go first before looking sideways. We pop a single node, put its children on the stack, then pop *one of those children*, then put *its* children on the stack, etc etc. So we try a single node in layer 1, then a single node in layer 2, then a single node in layer 3…

But: why does that result in such long path lengths? Remember – the graph for the sliding puzzle *is not a tree!* The paths rejoin and make loops, meaning the DFS doesn't just go down layer by layer, it actually sort of wanders around the entire interconnected web of the graph, finding a completely random path to the goal. Given that the graph of a 3x3 connects 181,440 states, this wandering might result in visiting a huge number of nodes before randomly stumbling upon the goal state! And in the 4x4 puzzle there are many, many more states than that (more on this in a future assignment) – so many that a random wandering search is likely to just get lost entirely!

So – this seems like DFS is maybe a terrible idea, then, if it doesn't find the shortest solution. But as with any tool, there are situations where that tool applies quite nicely. Let's explore one.

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.

To solve Peg Solitaire, we want to **precisely invert the start state**; that is, not only do we want to end with only a single peg, we want to end with that single peg **inside the only empty hole in the starting state**. The starting hole may be anywhere; when submitting your code later, it will be given as a command line argument.

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.

If you do this right, you will need to modify very little of your actual BFS / DFS code – you might even be able to use the exact same function! All you'll need are goal_test and get_children functions that are appropriate to peg solitaire.

Now, use both BFS and DFS to solve the puzzle. Your code needs to 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!)

You should see that BFS and DFS both solve the puzzle quite handily.

So: Why?

Send me a message on Mattermost where you answer these questions:

- Why did DFS work just as well as BFS on Peg Solitaire, but not on Sliding Puzzles?
- Could there be a situation where DFS might be preferable to BFS, not just equivalent? Why or why not?

Feel free to discuss with classmates or do some independent searching if you're not sure.

Get Your Code Ready to Turn In

You're going to submit a python file here that takes a single command line argument – an index representing the starting hole. **Don't forget to convert this to an int, as it will be read in as a string.** As you might expect, index 0 should be top center, then across each row in turn, ending with index 14 being the bottom right hole. Your code should then run a BFS, nicely print the results, and then runs a DFS, and nicely print those results. Clearly identify which search is which, and clearly write your code so that as I scan through I can easily find your BFS and DFS functions.

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 sent your answers to the "So: Why?" question to me on Mattermost, and I OKed them.
- Your submitted code runs a BFS and a DFS to solve Peg Solitaire when I run it starting with a hole in the given location on the command line. (Don't forget to convert the string command line arg to an int!)
- Total runtime is effectively instantaneous.