Sliding Puzzles: BFS

Eckel, TJHSST AI1, Fall 2022

Background & Explanation

Now it's time to learn breadth first search!

To solve the sliding puzzle, we want to imagine all the possible game states connected to each other in a network, or **graph**. (This is a different sense of "graph" from algebra class; here it refers to any set of items & connections between them. Look up "Graph Theory" on Wikipedia for more context.) Any two game states are connected if a single slide *of a single tile* moves from one game state to the other. No sliding an entire row all at once!

The specific challenge is to search for a path to the **goal state** from any input **start state**. The path is a sequence of actions taken from {Up, Down, Left, Right}. **Each action describes the direction that the BLANK square moves**. (This is different from the way a human would probably think about the game, but it makes the problem easy to represent.)

If the graph of game states were a **tree** then there would be only one solution path, but this is not the case. It is possible to have two distinct sets of moves arrive at the same game state. Our graph has loops! This has implications for our search. We must keep track of all of the states we have previously visited and make sure we don't revisit them or else we may find ourselves going in circles. For a simple example of this, take a 2x2 square that contains the blank and rotate the other three cells around; eventually you'll return to where you started.

For part 1 of this assignment, we will solve sliding puzzles on a graph made up of possible states in the game with connections each time one state is one move of one tile away from another. Specifically, we will use **breadth first search (BFS)**, an example of an **uninformed search algorithm**, to find the shortest path. This is an algorithm that works by exhaustion – we try every state one move away from the starting state, then every state two moves away, then three, etc. If you're working ahead, or if you'd like a refresher on how to construct this algorithm, see the video posted on the course website. **I strongly recommend a thorough understanding of the algorithm before using the pseudocode below.** It is possible to implement this without understanding it, but then when I start asking trickier questions you'll discover that you are stumped! This is harder than APCS; take your time, make sure it really makes sense.

The pseudocode:

```
function BFS(start-node):
    fringe = new Queue()
    visited = new Set()
    fringe.add(start-node)
    visited.add(start-node)
    while fringe is not empty do:
        v = fringe.pop()
        if GoalTest(v) then:
            return v
        for every child c of v do:
            if c not in visited then:
                fringe.add(c)
                visited.add(c)
    return None
```

NOTE: this is not Python code! This is pseudocode, and you'll need to look up python's implementations of queues, sets, etc. In particular, you **don't** make a queue in python by writing "new Queue()" and you **don't** pop an item off a queue by using .pop() either!

NOTE ALSO: this pseudocode is *insufficient* to *fully* answer all of the questions on this assignment. You will need to augment this basic algorithm with other variables & code to answer later questions!

Later on we will build better algorithms – **informed** search – that enable us to solve harder puzzles! But for now: BFS.

NOTE: This assignment comes in two parts. **You probably want to read both parts before you start coding**; you may wish to work on both parts at the same time to save some work.

Part 1: Use BFS to Solve Sliding Puzzles

The first challenge in this assignment is to use the BFS to find the shortest solution path to a sliding puzzle. Specifically, we want a sequence of moves or board states that takes us, one move at a time, from the start state to the goal state and there should be no shorter possible sequence. Some puzzles do have more than one correct solution; in this case, you just need to find one of them.

Note that since you will either be keeping track of the path as you go or reconstructing it after the search is completed, the BFS pseudocode from the prior page is not, on its own, sufficient. You will need some way of tracking the additional path data as BFS runs. There are many ways to do this, some more efficient than others! You may also wish to modify `get_children` to send additional data; this is fine.

NOTE: while testing, don't randomly generate large boards (greater than 3x3) and use them to test; stick with the puzzle file. There are many 4x4 and 5x5 puzzles that you won't be able to come close to solving, and we'll deal with them in future assignments!

You'll also want to keep track of how long it takes your code to find the solution path for each puzzle.

Sample Run

This is what the output should look like for `sliding_puzzle_tests.txt`.

Note that each puzzle outputs the **line number**, the **puzzle**, the **number of moves**, and the **time to solve**.

```
>python.exe 8_puzzle_efficient.py slide_puzzle_tests.txt
Line 0: A.CB, 1 moves found in 2.3599999999998622e-05 seconds
Line 1: .132, 2 moves found in 1.959999999999809e-05 seconds
Line 2: ABCDEFG.H, 1 moves found in 1.8299999999998873e-05 seconds
Line 3: 87436.152, 27 moves found in 0.4085162 seconds
Line 4: .25187643, 20 moves found in 0.12215100000000001 seconds
Line 5: 863.54217, 25 moves found in 0.36367819999999995 seconds
Line 6: AB.CEFGDIJKHMNOL, 4 moves found in 0.00012599999999995948 seconds
Line 7: .BCDAEGHIFJLMNKO, 6 moves found in 0.0004654000000000602 seconds
Line 8: ABCDEF.HIJKGMNOPLRSTUQVWX, 6 moves found in 0.0013896000000001019 seconds
Line 9: FABCE.HIDJKGMNOPLRSTUQVWX, 13 moves found in 0.28023750000000003 seconds
```

For the record: my code is quite efficient and running on a very fast computer; I would expect yours to be slower at this point! But: total run time must be less than two minutes.

Note: a **very common mistake** on this assignment is to print **too many things**. I don't want **any more than this**! If you print out every game board along the solution path, for instance, that will be too much output and I won't be able to easily grade several students' submissions in a row. That will result in a request for resubmission; I won't grade code that doesn't follow the specification! So **be careful** to output **these things** and **nothing else**.

If you get the above output in two minutes or less, you are finished with Part 1.

Part 2: BFS Brainteasers

In addition to the primary task above, there are a couple of brainteasers I'd like you to do. For these, you must **send me your answers on Mattermost** when you find them. You **must** do this to get credit for the assignment. (We'll do this a lot this year; many assignments have a few fun facts to dig out in addition to the primary task.)

1) Use BFS to find out how many puzzles of size 2x2 and 3x3 are solvable. Keep in mind – if a path exists from a puzzle to the solution, then a path exists from the solution to that puzzle as well. So this can be done with a single BFS search starting from the goal state, if you modify your code right!

   Our computers aren't fast enough to do this for 4x4 and 5x5 puzzles, but find the answer for 2x2 and 3x3 and send me a message on Mattermost.

2) Find at least one 3x3 puzzle that isn't solvable. Send me the puzzle as well as how you know it's not solvable.

3) Find how many 3x3 puzzles there are that have a minimal path of 10 moves. (In other words, when you find the most efficient way to solve that particular puzzle, it takes exactly 10 moves – no more, no less.) Just send the number; you don't need to list them all.

4) Find the 'hardest' 3x3 puzzle(s). That is, find a start state that requires the largest minimal path of possible moves to reach the solution state. Then, find out if there are any more start states with the same path length. How many? What are they?

   (Hint: there is an obvious way to find this that will take days to run, and a clever way to find this that will take seconds to run. I'd aim for the clever way.)

   For each puzzle you've found, output the start state, **complete solution path**, and solution length for all of them. Send me all this info on Mattermost using backtick formatting.

5) Finally, this isn't really a brainteaser, just something I'm curious about. On the course website, you'll find another file – `15_puzzles.txt`. ("15 puzzle" is another name for the 4x4 sliding puzzle, since there are fifteen tiles to place.) This .txt file has a sequence of different puzzles where **each puzzle takes exactly one more move to solve than the previous puzzle**. This makes for an excellent benchmarking tool! How far can your code get before it bogs down? You'll have to modify your code so it's not looking for the size; all of these puzzles are the same size, so that information isn't provided.

   Let's define "bogs down" as "takes longer than a minute to solve a single puzzle". Send me a message on Mattermost with how far your code gets before it bogs down – what's the longest path it can find on a 4x4 puzzle in less than a minute?

Get Your Code Ready to Turn In

Just a reminder: if you've modified your code in order to solve the previous brainteasers, go back to the sample run on the previous page and make sure that is what the output looks like for the code you're submitting. In particular:

- Make sure you're accepting the filename as a **command line argument**.
- Make sure you're NOT outputting the whole path for each puzzle, just the length as shown in the sample run.

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 all the brainteasers on Mattermost, and I OKed them.
- Your code does all of the following:
    - Accept a single **command line argument** specifying a file name. (Do not hardcode the file name!!)
    - Read in and solve each puzzle using **BFS** to correctly get the **minimal** path length.
    - Output the line number, solution length, and time to solve each puzzle, as shown on the sample output on the previous page. Do **not** output the moves or any board states.
- Total runtime is less than two minutes. (Runtime on my tests will be about the same as on the example file.)