Search Algorithms on Sliding Puzzles, Part 1

Eckel, TJHSST AI1, Fall 2020

Background & Explanation

Sliding puzzles are famous. (Google "Simon Tatham fifteen" for an example.) For this assignment, we will be working with square sliding puzzles of sizes 2x2, 3x3, 4x4, and 5x5. The only difference for us is that, to make representing the game state easier, we will use a single character to represent each tile instead of the two-digit numbers often used. Each character might be a digit from 1 to 9 or it might be a capital letter. We will use a period to represent the blank.

To solve this game, 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. For an explanation of how to craft this algorithm, see the video posted on the course website. **I strongly recommend a thorough understanding of the video 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()"!

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.

Modeling the Board

It might seem counterintuitive, but we are going to use a string to represent the gameboard, reading each row left to right and concatenating them all. So this board:

```
1 2 3
4 5 6
7 8 .
```

…would become "`12345678.`"

You may think something like an *n* x *n* array of characters is more intuitive, but strings are *immutable* in Python, meaning every time one is modified, a new string is created. This prevents you from having to copy manually or accidentally pointing two different variables at the same instance. Debugging string manipulation is therefore *much* easier. Use a string, trust me.

If you find it difficult to think in one-dimensional indices, I recommend functions that convert back and forth between a coordinate location (x, y) and a single index of that location in a state string. This is a good way to figure out how to do the conversion *generally* (not hard-coding a specific board size) and then not have to worry about it elsewhere!

Modeling Tasks

1) First, make sure you can read the input and store the puzzles properly.

   On the course website, you'll find a file called `slide_puzzle_tests.txt`. Note that each line contains two things:
   - A number, indicating the size of the puzzle ("2" means 2x2, "3" means 3x3, etc).
   - A sequence of characters containing a period representing the initial state of the puzzle. This is formatted exactly the way you should store your board state in your code, a single string of length size^2.

   All the puzzles in this text file are solvable.

   First, create a `print_puzzle` function that will take the same two pieces of information – the size of a board and its string representation – and print it nicely to the console as a grid. (I recommend putting an extra space in between each character horizontally; it looks nicer.)

   Then, create a `find_goal` function that will take a board and return the goal state for that board. The goal state is always the characters in ascending order with a period in the bottom right space. (For example, the first board in the file is "`A.CB`". The goal is the characters in ascending order – "ABC" – with a period on the end, so "`ABC.`" The built in Python `sorted` function might be of some use here.)

   Then, read in each of the entries in the txt file and, for each row, nicely print the puzzle and its goal state. Recommended syntax for opening files is this:

```
with open("slide_puzzle_tests.txt") as f:
    for line in f:
        # "line" is a string and you can manipulate it as such; add your code here
```

   One note of recommendation – the Python `split()` command splits on any whitespace by default, returning a list of substrings. This is much easier than alternatives involving finding indices and cutting piece by piece.

2) Before any searching is done, the game must first function. Design a set of functions that can manipulate the game board correctly.

   Remember that your code must work *generally*, with any size of board.

   You must have the following functions:
   - A `get_children` function that takes a state and returns a list of the boards one move away from that state. (Test with different size boards! Verify by hand! A small amount of annoying effort now will save a lot of pain later if this is wrong!)
   - A `goal_test` function that returns true if the board matches the goal.

3) Check that all of the above is working. Specifically, read in every line of `slide_puzzle_tests.txt`. Then, for each line, use the `print_puzzle` command from #1 to nicely output the start state, the goal state, and all of the children *of the start state*. Your output should be clearly labelled, like "`Line 0 start state:`", "`Line 0 goal state:`", "`Line 0 children:`", etc. Compare your output with the output of another student; make sure they're identical. (Sharing **code** is plagiarism; sharing **output** is just efficient double-checking!) Then, **send it to me in a direct message on Mattermost**. See the Mattermost formatting guide (just google "mattermost formatting") to see how to format your message as code so it looks like console output.

Implementing BFS Searching

4) Before we solve, first let's use BFS to just generate the entire set of reachable locations. This is a BFS with no goal state; it should keep going until no new children are generated. You won't need to check for this, because the queue will simply be empty at that point and the algorithm will end!

   Specifically, use a no-goal-check iterative BFS algorithm to begin with a goal state and find how many possible game states are winnable (in other words, how many game states can be reached from the goal state, including the goal state). Note that you will need to find some way of avoiding repeated states. Our machines are only fast enough to do this for 2x2 boards (ie, goal state "`ABC.`") and 3x3 boards (ie, goal state "`12345678.`") **Verify your answers with myself or a classmate**! Past experience has shown that this is a good moment to eliminate subtle code errors before continuing on.

5) Now, let's use the BFS to find a solution path. Specifically, make a function that uses iterative BFS to find the shortest path from any given state to the solution state. Print the length of the path. (The length of the path is the **number of moves**, not the number of states. If you try to "solve" a board that is already at the goal state, it should return a path length of 0.)

   Then, add a function that will print the entire sequence of actions and all the boards along the path from the start state to the goal state. Check your code's solution against a puzzle on Simon Tatham's page (ie, generate a random puzzle there, type it into your code, run your solve function, then follow the steps and see if your code is correct.)

   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 code from question 4 is not 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` so it keeps track of which direction was taken to get to each child.

Note: while testing your BFS, I recommend sticking with 3x3 and 2x2 puzzles for now. Your code will probably hang or crash on a 4x4 puzzle. Our algorithms aren't good enough yet to go that big. (More on this in class soon.) Once you've been able to solve a 3x3 puzzle from Simon, you've conclusively verified that your code works! This is another place you might spot an anomaly – a move that isn't actually legal, for instance – and track down a subtle bug in your code.

6) Try to find the 'hardest' 8-puzzle. That is, find a start state that requires the largest minimal path of possible moves to return to the start. How many puzzles are there with the longest possible solution length? Output the start state, solution, and solution length for all of them. **Send your results to me on Mattermost so I can verify them**. (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.)

Add Benchmarking

As we move forward, we'll want to compare the time it takes for various different search algorithms to work.

To benchmark your runtimes specifically, add `import time` to your import statements at the top of your file, and then use this code:

```
start = time.perf_counter()
# Whatever code you want to benchmark goes here
end = time.perf_counter()
print("Seconds to run: %s" % (end - start))
```

The code you are benchmarking **should not include any print statements**, since the time print statements take to execute varies according to outside factors, so any debugging console prints should be commented out when you benchmark. **Store** your results **inside** the timed section; **print** the results **after** marking the end time as above.

Get Your Code Ready to Turn In

I'd like your code to read in a file of sliding puzzle tests like the one given on the course website, run them all, and output certain information. The easiest way to explain this is by looking at an example run.

So, below you see a correct, complete output for the example file on the course website. Note that each puzzle outputs the **line number**, the **puzzle**, the **number of moves**, and the **time to solve**! For the record: my code is quite efficient; I would expect yours to be slower at this point! But: total run time must be less than two minutes.

```
>python 8_puzzle_efficient.py slide_puzzle_tests.txt
Line 0: A.CB, 1 moves found in 9.939999999999949e-05 seconds
Line 1: .132, 2 moves found in 4.809999999999537e-05 seconds
Line 2: 87436.152, 27 moves found in 0.7770692 seconds
Line 3: .25187643, 20 moves found in 0.21310989999999996 seconds
Line 4: 863.54217, 25 moves found in 0.6764337 seconds
Line 5: AB.CEFGDIJKHMNOL, 4 moves found in 0.00027870000000018713 seconds
Line 6: .BCDAEGHIFJLMNKO, 6 moves found in 0.0008493000000000528 seconds
Line 7: ABCDEF.HIJKGMNOPLRSTUQVWX, 6 moves found in 0.002289100000000044 seconds
Line 8: FABCE.HIDJKGMNOPLRSTUQVWX, 13 moves found in 0.5055432 seconds
```

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**.

Specification

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

This assignment is **complete** if:

- The "Name" field on the Dropbox submission form contains your **class period**, then your **last name**, then your **first name**, in that order.
- You sent me the **direct messages on Mattermost** in questions 3 and 6 that I asked for.
- Your code does all of the following:
    - Accept a single **command line argument** specifying a file name. (Do not hardcode the file name!!)
    - Read puzzles from that file just like the example file – a size and board on each line.
    - 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.)

For **resubmission**:

- Complete the specification correctly.

Specification for Outstanding Work: Bidirectional BFS

There is one way to extend this assignment and receive outstanding work credit: add another algorithm that applies BFS principles from both the source and the goal. As usual, the submission link is on the course website.

- Bidirectional BFS is not a terribly complex idea, in theory. Your goal is to write a new method that runs a BFS search from both sides simultaneously, creating a visited set and fringe from the source state and a different visited set and fringe from the goal state, alternating adding to each. As you add to each fringe, see if each state lies on the fringe from the other direction. Keep track of the path.
- Recreate question 5 above, but with both BFS and BiBFS. Show me that they both give the same path lengths and that BiBFS solves more quickly. Specifically, if you look at the sample output on the previous page, I'm looking for each puzzle to have two lines of output – one BFS, exactly as shown, and another BiBFS that gives the same information just beneath it. The BiBFS time should be shorter on any puzzle with more than a couple of moves!

This extension **receives outstanding work credit** if:

- Your code meets the complete original specification for this assignment but runs each puzzle *twice*, once with BFS and once with BiBFS. It should be clear that BiBFS is faster on the puzzles with sufficiently long path lengths.

Further Ideas to Ponder

- The obvious challenge here is simply efficiency. How close to my runtimes can you get? Can you get even faster? (Whether or not you implement BiBFS, you can always work on efficiency of your algorithm of choice.)

- There is another challenge to think about, though. So far, I have only given you solvable puzzles. What if you instead get an unsolvable puzzle? One way to determine that your puzzle is unsolvable is to run BFS on the puzzle until you've generated every single state that is reachable, and if you haven't seen the goal state, then the puzzle is impossible. Alternately, we can run BFS from the goal state once, and store all of the possible results (like we did in question 4). Then, we can simply check if our starting puzzle is in the set before we start solving. Unfortunately, both of these options require doing a complete BFS of all possibilities in one direction or the other. I promise: we do not, and will never, have enough memory to do this for a 4x4 puzzle or larger.

  So: can you think of a way to determine that a puzzle is impossible *without* storing a long list of states, or in fact doing a BFS at all? This is more of a math problem than a programming problem, but either way it's a good one!