Sliding Puzzles: A*

Eckel, TJHSST AI1, Fall 2023

Background & Explanation

So far, you've used BFS (and possibly DFS and BiBFS, if you did more advanced assignments) to solve sliding puzzles. Even if you did BiBFS, you saw that we haven't made it very far into the `4x4_puzzles.txt` file – there's a lot of room for improvement.

This assignment introduces a different way of thinking about search that will get us a massive improvement in results. But first we have some details to take care of.

A Brief Note About O(n^2!) (yes, that says "oh of en squared *factorial*"; **be afraid**)

On the last assignment, I asked you to find the hardest 8-puzzle, or 3x3 puzzle. This probably took a couple seconds. Your computer had to generate 181,440 states to find it, $\frac{9!}{2}$.

On this assignment, we'll be mostly focusing on 15-puzzles, or 4x4 puzzles. So, let's say I asked again: what's the hardest 15-puzzle? How many states would you need to try? Well, $\frac{16!}{2}$ is a liiiiiiittle bit larger – the number is 10,461,394,944,000 states.

Yeah.

When I say "a liiiiiiittle bit larger", I mean, like, "AAAAAAAAAAAHHHHHHHHHH!?!" That is *more than fifty seven **million** **times*** as many states. O(n^2!) is no joke!

Think about it this way – if generating 181,440 states took 2 seconds, generating 10,461,394,944,000 states would take 115,315,200 seconds. Or 1,921,920 minutes. Or 32,032 hours. Or 1,334 days. Or ***3.66 YEARS.*** That's years, *plural*.

So, um. Suffice it to say, I will **not** expect you to find the hardest 15-puzzle!

More seriously: this means that the 15-puzzle in general will be a *vastly* more challenging problem, straining our computer's time and memory resources in a way that the 8-puzzle absolutely did not. In fact, we won't "solve" the 15-puzzle completely in the end – we won't be able to find code that will resolve this shortest path question for any arbitrary board. This makes it a great problem to explore, though, because we can keep making progress to the limit of our ingenuity and knowledge! Any improvement in our code will be noticeable in our results.

Let's go exploring.

Parity & Impossible Puzzles

First and foremost, right now, if a puzzle is impossible, the only way our code has of discovering that is running an entire BFS to completion, failing to find the goal state, and retuning None. This is not going to work on the 15-puzzle!

Recall from our conversations in class that any odd size board is solvable if the number of out of order pairs of tiles (disregarding the blank) is even. *(If we haven't done that classwork activity yet, ask me for help!)* Consider these boards:

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

The first has 9 out of order pairs (14, 24, 34, 38, 56, 57, 58, 68, 78). Therefore, it is unsolvable! The second has 28 out of order pairs (all of them). Therefore, it *can* be solved. (If you need a refresher on how this works, ask for help!)

This same process applies to any board with an odd size, because moving a single tile affects its relationships with an *even number* of other tiles, resulting in an *even change* to the count of out of order pairs. Since that count needs to end up at zero when the board is solved, only boards with even out of order scores are solvable. Period.

On even size boards, 2x2 or 4x4, we must be more careful. On a 4x4 15-puzzle, for instance, swapping the blank horizontally continues to have no effect on the number of out of order pairs, but swapping the blank vertically can change the count by +3, +1, -1, or -3. See the following four boards in order as examples. For the purposes of later discussion, the four rows are numbered this time. In each board, consider swapping the blank with the tile above it in row 0.

```
Row 0:          A B C D          B A C D          C A B D          D A B C
Row 1:          . E F G          . E F G          . E F G          . E F G
Row 2:          H I J K          H I J K          H I J K          H I J K
Row 3:          L M N O          L M N O          L M N O          L M N O
             swap up for +3   swap up for +1   swap up for -1   swap up for -3
```

No longer can we simply rely on finding even parity. However, there is still a parity pattern to be found. When solved, the blank must end on row 3 with the puzzle having an out of order pairs score of zero. Working backwards, that means that when the blank was last on row 2, the parity of the board's total out of order score must have been **odd**. This is because the blank, when moving from row 2 to row 3, would have changed the board's parity, since adding or subtracting an odd number always changes a quantity's parity. So puzzles with **odd** parity look solvable when the blank is on row 2.

This same kind of reasoning can be repeatedly applied. Any time the blank moves from one row to another, the parity changes. So even parity in row 3 implies odd parity in row 2, and that in turn implies even parity in row 1, and that in turn implies odd parity in row 0. So the rule for 4x4 puzzles (and, in fact, all even size puzzles) is this:

- If a puzzle currently has the blank in an even numbered row, then if the total number of out of order pairs is ODD, the puzzle is solvable.
- If a puzzle currently has the blank in an odd numbered row, then if the total number of out of order pairs is EVEN, the puzzle is solvable.
- Otherwise, the puzzle is not.

Programming Task: Implement a Parity Check

Your first task for this assignment: **implement a parity check function**. It must work on all sizes from 2x2 to 5x5. Test it on all the puzzles from Sliding Puzzles 1; it should say all of them are solvable. Then, make a new set of puzzles by going through each of the puzzles from that test file and performing exactly one swap of two tiles in each puzzle (eg, turning "AB" to "BA"). This alters the out of order count by 1, producing an impossible board. Run your parity check on all the new puzzles; it should say all of them are impossible. **Check in with me in person or DM me a message that says "I promise I tested my parity check function at all sizes and it works!"** I promise to believe you.

Informed Search & Taxicab Distance

Ok, impossible puzzles are handled. Now it's time for better algorithms for the puzzles that *can* be solved.

With BFS, we can only get up to puzzles of something like 20 to 23 moves when looking at a 15-puzzle. If you did the BiBFS extension, it does better – something around 40 moves, maybe – but this is still *clearly* insufficient!

It's time to level up. Let us leave **uninformed search** behind and move instead to **informed search** – a search that *chooses* which paths to attempt next. Instead of just exhaustively trying everything, we will prioritize boards that seem *closer to being solved*.

Congratulations, you're about to code your first Artificial Intelligence assignment that contains "Intelligence"!

The first thing to do here is to define what we mean by "closer to being solved". We want to be able to *estimate* how close a board might be to the solution state *without doing a search*. There are a lot of ways to do this, but probably the best one to use for now is **taxicab** or **Manhattan** distance. (There is more discussion on this in the RED assignment.)

The idea behind taxicab distance is to say that, *bare minimum*, each tile on the board needs to move directly back to where it belongs. If we're very lucky, the tiles won't get in each other's way, and the number of steps in the solution will be the sum of each tile's individual minimum journey. The actual number of necessary moves might be *more*, but it won't be *less*!

For example, consider the following board from before:

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

Let's consider each tile:

- Tiles 1, 3, and 7 are each in an opposite corner from where they belong. They each must move 4 spaces to return to their location in the goal state.
- The 2, 8, 4, and 6 tiles each need to move across the puzzle 2 spaces.
- The 5 tile is where it should be! 0 moves needed there.
- **DO. NOT. COUNT. THE. BLANK.** I'll explain why on the next page!

This makes this board have a total taxicab distance of 20. So, I know at least 20 moves are necessary to solve this puzzle!

Programming Task: Implement Taxicab Distance

WARNING: IT IS EXTREMELY COMMON TO SCREW THIS UP. I don't want to deprive you of your thinking, so I won't say why, but like 50% of students have a really strong instinct here that turns out to be close… but incorrect.

So write your function, then compare it **CAREFULLY** to the correct taxicab distances from `4x4_puzzles.txt`.

```
Line 0: 0     Line 1: 1     Line 2: 2     Line 3: 3     Line 4: 4     Line 5: 5     Line 6: 6
Line 7: 7     Line 8: 8     Line 9: 9     Line 10: 8    Line 11: 11   Line 12: 12   Line 13: 11
Line 14: 12   Line 15: 11   Line 16: 16   Line 17: 13   Line 18: 16   Line 19: 17   Line 20: 14
Line 21: 13   Line 22: 18   Line 23: 15   Line 24: 20   Line 25: 15   Line 26: 22   Line 27: 21
Line 28: 16   Line 29: 19   Line 30: 12   Line 31: 23   Line 32: 20   Line 33: 23   Line 34: 22
Line 35: 23   Line 36: 26   Line 37: 25   Line 38: 24   Line 39: 21   Line 40: 26   Line 41: 29
Line 42: 22   Line 43: 29   Line 44: 32   Line 45: 31   Line 46: 32   Line 47: 27   Line 48: 28
Line 49: 33   Line 50: 34   Line 51: 33   Line 52: 42   Line 53: 39   Line 54: 40   Line 55: 39
```

Do not move on until you've verified these numbers.

A* Search

So let us move on to a better algorithm at last! This is called A* Search (pronounced like "A-star") and it really isn't so different from BFS except for one crucial fact: the fringe is *sorted*. Or really not sorted, just *heaped*; we set it up so that we remove *the current most promising node* each time we remove from the fringe. We'll use a min-heap for this.

How do we decide the most promising node? For each node *x*:

- First, calculate $g(x) =$ path length so far. (In other words, current depth of the node.)
- Then, calculate $h(x) =$ estimated distance to the goal state. (For us: taxicab distance from this state to the goal.)
- Then, calculate $f(x) = g(x) + h(x)$. The node with the minimum value of $f(x)$ is the most promising.

A good intuition here is that, for any given node, we're trying to sort/heapify based on our *current best guess about the total path length, from beginning to end, through that node*. Which means how many moves we know that it already has been from the start state to here, plus how many moves we *guess* it will be from here to the end.

A Technical Aside

An optional note for those interested in the mathematically formal side of things.

To be technical, $h(x)$ has some more specific restrictions beyond simply being "estimated distance to the goal state". It's outside the scope of this course, but it can be proven that A* is guaranteed to find the minimal path while sorting on its $f(x)$ combined heuristic so long as the following conditions are true:

- $h(x)$ must be **consistent**. This means that the estimate obeys the triangle inequality, more or less; specifically, its estimate of distance to the goal must always be less than or equal to the sum of the actual distance to any neighbor plus its estimate of the distance from *that neighbor* to the goal. In other words, it **cannot** be true that a particular state has estimate 12, moves a distance of 1, and arrives at a new state with estimate 10.
- $h(x)$ must be **admissible**. This means that the estimate from any given node **cannot** be an overestimate. For instance, it must never be the case that a state is 10 moves away from the end but the estimate claims 12.

Taxicab distance meets both of these criteria, **but only if we do not count the blank**. Consider this board:

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

It is patently obvious that this board is precisely 1 move away from the end. However, if we count the taxicab distance for the blank as well, then our estimate would be two moves – one for the 8 to return to its location, and one for the blank. This is neither consistent nor admissible!

In any situation where an attempt is being made to find the shortest path through a graph, as long as a consistent and admissible estimating strategy can be found then A* search is proven to work.

Ok, let's explore the A* algorithm's pseudocode!

Pseudocode for A* search:

```
function a_star(start-state):
  closed = new Set()
  start_node = new Node()
  start_node.state = start-state
  start_node.depth = 0
  start_node.f = heuristic(start-state)
  fringe = new Heap()
  fringe.add(start_node)
  while fringe is not empty do:
    v = fringe.pop()
    if GoalTest(v):
      return(v)
    if v.state not in closed:
      closed.add(v.state)
      for each child c of v do:
        if c not in closed:
          temp = new Node()
          temp.state = c
          temp.depth = v.depth + 1
          temp.f = temp.depth + heuristic(c)
          fringe.add(temp)
  return None
```

Crucial note for Python implementation here – Python's built in heap commands automatically create a min-heap. As with last time, I strongly recommend a min-heap of tuples containing all relevant information, in this case the state, *f* value, and depth.

But note: when you create a heap of tuples, the heap is sorted on the *first tuple value*, and then if those are the same, the *second tuple value*, etc. As such, when we make this heap of tuples, we must have the *f* value that we're trying to minimize be the **first value in each tuple**. Otherwise, the heap will prioritize incorrectly!

You probably see that this has all the same components of our earlier search algorithms, but you might notice one pretty major change. Instead of a visited set or ancestor set we now have what's called a *closed* set, and we've changed where nodes get added to it. Previously, a node has been added to visited or to ancestors *when it was placed on the fringe*. This time, we add a state to the closed set *when we remove it from the fringe* instead.

It turns out that A* may find several different paths to the goal state and add them to the fringe – we may have the goal state on the fringe multiple times! But, because of the heap's minimum invariant, we are guaranteed to *pop* the shortest path to the goal state off before any less efficient paths appear. So, we can't guarantee that the first time we *visit* the goal is the correct path, but we can guarantee that the first time we *pop* the goal off the heap *is* the right answer.

This applies to any other node as well. Any given node may be on the heap several times. As such, when we pop a node off, we need to check if that node's shortest path has already been found; thus, the extra line checking that `v.state` is not in `closed`.

This is another good moment to check yourself. Read those three paragraphs again and reach out if you have questions!

Programming Task: Implement A* with Taxicab Distance

Pretty straightforward – read the pseudocode, use your taxicab function for `heuristic()`, and make it happen!

When you run your A* search on the `4x4_puzzles.txt` file, you should be able to solve puzzles up to a path length of 40 in just a few seconds each and be 100% accurate. As you do a sample run, pay close attention to the solution lengths. Make sure they go up by exactly one for each consecutive puzzle!

Get Your Code Ready to Turn In

Look at `slide_puzzle_tests_2.txt` from the course website. You'll notice it looks a lot like slide_puzzle_tests from the prior assignment. There are two differences:

- Some puzzles are unsolvable
- Some puzzles are not solvable with BFS in a reasonable amount of time, so your A* will definitely need to work

Your job: take the file name on the command line, as usual. Read in each line. First, determine if a puzzle is impossible. If so, simply say that it has "no solution".

If a puzzle *is* solvable, run A* and **make sure you time how long it takes to solve each puzzle.**

Sample Run on slide_puzzle_tests_2.txt

Note the tiny numbers for discovering "no solution"; it may look at first glance like they represent some number of seconds, but they have powers of 10 at the end indicating they are much, much smaller. The time for finding "no solution" should be nearly instantaneous, even for a 5x5.

```
>python 4x4_puzzle_astar.py slide_puzzle_tests_2.txt
Line 0: A.CB, A* - 1 moves in 1.9199999999996997e-05 seconds

Line 1: .123, no solution determined in 3.4999999999896225e-06 seconds

Line 2: 87643.152, A* - 27 moves in 0.025455399999999999 seconds

Line 3: .25187643, A* - 20 moves in 0.0012221999999999789 seconds

Line 4: 836.54217, no solution determined in 6.999999999979245e-06 seconds

Line 5: BECDAFOGI.JHMNLK, A* - 15 moves in 0.0002495000000000136 seconds

Line 6: DCGJAE.BIMNHFKOL, A* - 39 moves in 1.5355892 seconds

Line 7: AFB.DEGCMOJKHILN, no solution determined in 1.870000000003813e-05 seconds

Line 8: ACGEJFBHD.LQMIWKUNRSPVXTO, A* - 37 moves in 0.12914789999999998 seconds

Line 9: FABCE.HIDJKGMNOPLRSTUQVXW, no solution determined in 3.16000000000205e-05 seconds
```

As usual, my code is much faster than I expect yours to be; your target here is a runtime of two minutes or less, total. (If you'd like to learn about how I made my code so fast, check out the RED assignment that follows from this one.)

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.
- 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.
  - Determine almost instantly if the puzzle is impossible. If so, output as such.
  - If possible, solve the puzzle to get the correct minimal path length using A*.
  - Output the line number, solution length, and time to solve each puzzle. (See sample output.)
- Total runtime is less than two minutes. (Runtime on my tests will be about the same as on the example file.)

Further Ideas to Ponder

We still haven't made it to the end of `4x4_puzzles.txt`! Can you continue to improve your code and get it to solve length 50+ puzzles in a reasonable amount of time?

Remember there are two different kinds of efficiency to speak of here. One is writing the algorithm you have more efficiently. Is there any redundant work? Are there any improper data structures? Excessively nested loops?

The other is *finding a better algorithm*. Can you improve A* or your Taxicab distance heuristic somehow?

You'll be able to explore these further in the Optimization lab for RED credit.