Search Algorithms on Sliding Puzzles, Part 2

Eckel, TJHSST AI1, Fall 2021

Background & Explanation

So far, you've used one simple search algorithm to solve the sliding puzzle - BFS. Now, we're going to dive much deeper into different search algorithms, exploring their strengths and weaknesses.

This assignment is pretty dense, so get ready for some deep thinking! If it helps, there's more reading in this assignment than in part 1, but there's probably a bit less coding?

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, "AAAAAAAAAAHHHHHHHHHH!?!" 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.

One of the files on the course website is `15_puzzles.txt`. To start off this assignment, read in this file. (You'll need to modify your input/output code; this file does not have a digit at the beginning of each line specifying the size. They're all 4x4.) Once you read in the puzzles, solve as many of them as you can with your BFS code, printing out the solution path length after each puzzle before your code starts the next one. You should see that the solution lengths increase in a regular pattern – each puzzle has a solution length exactly one greater than the previous puzzle.

What is the longest solution length that your BFS code is able to find in less than a minute? **Direct message me your answer to this on Mattermost, I'm curious.** (And, so that I know what you're talking about, don't just send me a number. Write me a sentence like "the longest 15-puzzle solution that my BFS can find in less than a minute is ____".)

If you did the BiBFS Outstanding Work, try that and **DM me that answer as well**.

…and then see how much farther the file of puzzles goes.

Yep, you got it. We're going to need better algorithms!

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.

Depth-First Search

Ok, impossible puzzles are handled. Next: let's try to find some better algorithms for the puzzles that *can* be solved.

The ridiculous enormity of 10,461,394,944,000 states poses two problems: a memory problem and a processing time problem. From an unlikely direction, we will first make progress on the memory problem.

Specifically, your next task is to try an experiment. Make one small change to the code you turned in for Sliding Puzzles 1. 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. 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?

So – what behavior is happening here? Why are your path lengths so much longer for the 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!* So, 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, as we just learned, in the 4x4 puzzle there are 10,461,394,944,000 states – so many that a random wandering search is likely to just get lost entirely!

To be clear: **depth-first search is a completely reasonable way to find a correct solution if you're searching a <u>tree</u>.** In a tree, there's only one path from any node to any other node; there are no loops in the graph. So DFS and BFS will both find the correct path. And which algorithm has a runtime advantage just depends on the structure of the tree, so in certain circumstances DFS will run faster than BFS and just as accurately. Which is good to know!

But in **this** case, not so much. **In a graph that isn't a tree, DFS is emphatically not guaranteed to find the shortest path**.

So: what could we do to get a version of DFS that would actually work? And is there any reason to do that, when BFS works just fine?

As it turns out, yes!

We can modify DFS to remove this random wandering problem, and then we can produce a version of this algorithm that is slower than BFS **but takes up much less memory!**

This is called Iterative Deepening Depth-First Search, or ID-DFS.

Iterative Deepening Depth-First Search

Once again, the goal for this section: we will make an algorithm that, compared to BFS, will take longer to run but use much less memory.

> **Important Note**
>
> I should be clear up front – modern computers have so much memory that this isn't actually going to enable us to solve any problems we couldn't solve with BFS already. To be specific, a recent change from 32-bit memory addressing to 64-bit in most programs (including python) means that a single application can claim/address a theoretically enormous amount of RAM, orders of magnitude more than the RAM you actually have on your computer. When I taught this class for the first time in 2018, though, this wasn't the case on everyone's computer; several students still used 32-bit versions of programs, and we were able to see instances in which BFS would max out the available addressable RAM for a single 32-bit process (about 2 GB) and crash. In those cases, waiting a very long time would enable ID-DFS to solve the puzzle when BFS would fail.
>
> Now, though, this is more a theoretical exercise than a practical solution to a the 15-puzzle problem. However, it remains an outstanding example of the way related algorithms can have very different strengths and weaknesses. This tradeoff of memory vs speed is one that emerges frequently in a deeper study of algorithms, so ID-DFS continues to be instructive and well worth the time to explore.

Ok. The first thing we need to do is solve this problem of DFS just wandering randomly until it finds the goal. We'll do this in a way that seems hilariously crude – we will simply limit the depth that DFS can search. Specifically, in the part of our algorithm where the children are generated, we won't generate any more children if a node is already at maximum depth. Simple as that. We'll run the whole search limited to a certain depth, and if the complete DFS search fails to find the goal state, we'll add one to the depth we allow, and *run the entire DFS search from scratch over again*. And if that fails, we'll once again increase the depth we allow by one, and do it over again from scratch once more. Etc, etc.

Ridiculous, right?

Well, actually, this isn't as bad as it sounds! Since most nodes in a sliding puzzles graph have more than two children, each layer of the graph is at least twice as big as the prior layer. This exponential growth means that if we are looking for a solution that turns out to have a minimum solution length of *k*, the maximum time that the algorithm might take to do the final *k*-limited DFS search is about the time that the algorithm takes to do *all of the previous ones from 1 to k-1 combined*. This plus some other inefficiencies mentioned later on means that ID-DFS might take anywhere from 2x to 5x as long as an equivalent BFS search, but in a problem where many quantities grow exponentially, this kind of inefficiency is not so bad if we get an additional benefit.

So… what about that additional benefit?

To get an algorithm that uses less memory, we're going to have to get rid of another key feature of BFS – the visited set. As mentioned on the first page, 181,440 nodes doesn't present us much of a challenge, either time-wise or memory-wise. 10,461,394,944,000 nodes is a bit more of a problem! Eventually, storing every node we visit will fill up the computer's available memory. But: we still don't want our algorithm to go searching in futile circles; we still need to prevent that somehow.

The solution is to make a change and store a node's ancestor information *locally* instead of *globally*. By this, I mean that when we put any state onto the fringe as our algorithm progresses, we will pair that state with an **ancestor set**, a set of all the states visited on the path from the start node to that state. Then, when we remove that state from the fringe and

generate its children, we will skip over any child that is in the ancestor set. In other words, the path from the start node to any given node will never return to a previously visited node along that path. This will result in no loops!

This **does** mean that sometimes a path will wander down to a level, generate a child not yet seen by that particular path that resides in a previous level, and start wandering back upwards again. But because of the depth-limiting, this isn't actually a problem: say that, when running a 20-limited DFS, a path of length 20 has wandered back up to level 12. **If the solution were on that level, the previously run 12-limited DFS would have found it already.** So, we're wasting some time (here's that extra inefficiency I promised earlier) but we aren't in danger of getting a wrong answer.

In other words, a *k*-limited DFS will search *every possible non-looping path of length k*, many of which *will not terminate on level k*, but that's not a problem because it *will* nonetheless find every node on level *k* (even though it also finds many others).

This is a good moment to pause and take stock. Read that last paragraph again. Any questions, please reach out to me!

Now: it might not make sense to you yet why this saves us memory; at first glance, keeping an ancestor set for every node may seem *more* inefficient. Let us be clear about the specific pseudocode, then we will return to this question in detail.

Pseudocode for *k*-limited DFS and ID-DFS:

```
function k-DFS(start-state, k):
  fringe = new Stack()
  start_node = new Node()
  start_node.state = start-state
  start_node.depth = 0
  start_node.ancestors = new Set()
  start_node.ancestors.add(start-state)
  fringe.add(start-node)
  while fringe is not empty do:
    v = fringe.pop()
    if GoalTest(v) then:
      return v
    if v.depth < k:
      for every child c of v do:
        if c not in v.ancestors then:
          temp = new Node()
          temp.state = c
          temp.depth = v.depth + 1
          temp.ancestors = v.ancestors.copy()
          temp.ancestors.add(c)
          fringe.add(temp)
  return None

function ID-DFS(start-state):
  max_depth = 0
  result = None
  while result is None:
    result = k-DFS(start-state, max_depth)
    max_depth = max_depth + 1
  return result
```

NOTE once again: this is still not Python code! For instance, a stack in Python is just a list manipulated using only `.append()` and `.pop()`. But it says Stack in the pseudocode because this algorithm could be implemented in any language, and so we want to be a specific about what's important as possible.

Also, this pseudocode is written as if there is a "Node" class, but objects in python are awkward and slow and I recommend avoiding them. My advice here is to use simple tuple packing and unpacking for a python implementation. **I strongly recommend that you make your fringe a stack of tuples** with each tuple having three elements – a state string, a depth, and an ancestor set of strings – in addition to any other information that may be necessary to solve a particular problem.

Walk your way through that pseudocode and make sure it makes sense.

Now: why is this more memory efficient?

The reason is that this fringe has *waaaaaaay* fewer nodes on it than a BFS fringe. Let's imagine our solution path length is 20. To find it, either BFS or ID-DFS will have to go down to a depth of 20. Let's imagine freeze-framing both searches as they pop a node off the fringe with a depth of 20 for the first time.

- BFS at a depth of 20: The visited set contains 19 entire levels of states. The fringe contains *the entire 20th level* of nodes. Remember the approximately exponential growth here! $2^{20} = 1048576$. So right now, the fringe contains approximately a million nodes!
- *k*-limited DFS at a depth of 20: The particular node we are examining has an ancestor set of 19 previous nodes, one at each level. The fringe contains *only the direct siblings of each of those nodes*. (This is because, when each node is popped off the stack, all of its children are added to the stack, then one of those children is popped off next, leaving the rest behind.) So, that's between 0 and 2 additional nodes per level, for a total of about 20. Aaaand… that's it! The fringe will always contain between 1 and 20 layers' worth of 0-2 neighbors, and it will slowly progress across the whole tree left to right, checking a set of neighbors on level 20 before retreating to the next option on level 19, then going through another group of level 20 options, then retreating back to level 18, etc.

Yes, each node has a bigger footprint in *k*-DFS (since each node contains its whole ancestor set), but when you compare twenty or so nodes to a million or so nodes, the advantage to *k*-DFS becomes clear!

Video Demonstration: ID-DFS vs BFS

If you would like to actually watch this happening, **there is a video on the course website where I run BFS and ID-DFS on sequentially larger puzzles from the `15_puzzles.txt` file and watch the RAM usage of the Python process.** The difference is … stark. Watching this is optional, but super nifty.

Programming Task: Implement *k*-DFS and ID-DFS

Code the *k*-DFS and ID-DFS methods seen in pseudocode on the previous page.

Then, write code that will load the 15_puzzles.txt file and sequentially run BFS and then ID-DFS on each puzzle. You should see that the two find identical solution path lengths, and that ID-DFS takes a bit – but not too much – longer to execute than BFS does. A small snippet of your sample run might look something like this:

```
Line 15: ABCDFJGHENMK.IOL, BFS - 15 moves in 0.5867623000000002 seconds
Line 15: ABCDFJGHENMK.IOL, ID-DFS - 15 moves in 1.0629732 seconds

Line 16: .FBHAEDLIJCOMNGK, BFS - 16 moves in 1.4209836000000005 seconds
Line 16: .FBHAEDLIJCOMNGK, ID-DFS - 16 moves in 1.7156145999999994 seconds

Line 17: ABDJFGCHENK.IMOL, BFS - 17 moves in 3.6332322 seconds
Line 17: ABDJFGCHENK.IMOL, ID-DFS - 17 moves in 4.063648499999999 seconds

Line 18: AIBCFOGD.EKHMJNL, BFS - 18 moves in 5.390547 seconds
Line 18: AIBCFOGD.EKHMJNL, ID-DFS - 18 moves in 12.171945800000003 seconds
```

If you're on a PC where you have admin rights, you can bring up the task manager with CTRL + SHIFT + ESC and watch the RAM usage just like I did in my demonstration. That will *really* verify that you have this right!

Informed Search & Taxicab Distance

Ok but ID-DFS still isn't solving the problem! With either BFS or ID-DFS, 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 35 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 next 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 `15_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 `15_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!

Then take a moment to congratulate yourself. You have learned a *lot* of algorithms in one very big assignment – well done!

Get Your Code Ready to Turn In

Let's set this up so it's easy for me to grade. Look at `slide_puzzle_tests_2.txt` from the course website. You'll notice each line has three things on it now – a size, a puzzle, and a third argument that is a single character.

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, the third argument's single character should tell your code which algorithm(s) to run on that puzzle**.** Interpret them as follows:

- B means standard BFS.
- I means iterative deepening DFS.
- A means A*.
- ! means all of the above, ***IN THAT ORDER***, on the given puzzle.

Then, **make sure you time how long it takes to solve each puzzle.**

And finally, **set your code to read the puzzle file's filename from the command line.**

Note: it will not be possible for you to run all algorithms on all puzzles. I'll ask you to run A* on puzzles that are much too difficult for BFS or ID-DFS to solve. So be sure you read in the character and interpret it carefully; code that's running forever in Mr. Eckel's grading scripts makes him a sad teacher.

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 15_puzzle_astar.py slide_puzzle_tests_2.txt
Line 0: A.CB, BFS - 1 moves in 2.3500000000002685e-05 seconds
Line 0: A.CB, ID-DFS - 1 moves in 1.130000000000575e-05 seconds
Line 0: A.CB, A* - 1 moves in 1.2600000000001499e-05 seconds

Line 1: .123, no solution determined in 5.399999999995686e-06 seconds

Line 2: 87643.152, A* - 27 moves in 0.0226661 seconds

Line 3: .25187643, BFS - 20 moves in 0.269124 seconds
Line 3: .25187643, ID-DFS - 20 moves in 0.9732208999999999 seconds
Line 3: .25187643, A* - 20 moves in 0.0015969999999998485 seconds

Line 4: 836.54217, no solution determined in 1.3399999999830214e-05 seconds

Line 5: BECDAFOGI.JHMNLK, BFS - 15 moves in 0.6697618999999999 seconds
Line 5: BECDAFOGI.JHMNLK, ID-DFS - 15 moves in 0.9780612999999998 seconds
Line 5: BECDAFOGI.JHMNLK, A* - 15 moves in 0.00015509999999974156 seconds

Line 6: DCGJAE.BIMNHFKOL, A* - 39 moves in 0.8239703 seconds

Line 7: AFB.DEGCMOJKHILN, no solution determined in 2.9900000000360194e-05 seconds

Line 8: ACGEJFBHD.LQMIWKUNRSPVXTO, A* - 37 moves in 0.051114000000000104 seconds

Line 9: FABCE.HIDJKGMNOPLRSTUQVXW, no solution determined in 4.8599999999954235e-
05 seconds
```

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.
- 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, board, and algorithm choice 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 the algorithm or algorithms specified.
    - 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 `15_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 Sliding Puzzle Extensions for BLACK credit.