Sliding Puzzles: Iterative-Deepening DFS

Eckel, TJHSST AI1, Fall 2023

Background & Explanation

Ok, so there are some scenarios where DFS works fine, but Sliding Puzzles aren't one of them.

But: it turns out there *is* a *variant* on DFS that gives a surprising advantage when applied to sliding puzzles while still ensuring a shortest-path solution: Iterative-Deepening DFS. ID-DFS is slower than BFS **but takes up 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 any particular problem with sliding puzzles. 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.

Implementing ID-DFS

This algorithm can be quite challenging to wrap your head around; don't start coding until you're 100% positive you understand the idea.

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, we'll choose a maximum depth (which I'll call *k*). Then, in the part of our algorithm where the children are generated, we won't generate any more children if a node is already at depth *k*. 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 (at least for the first several layers). This exponential growth means that if we are looking for a solution that turns out to have a minimal 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? How does this save memory?

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. The visited set is huge – hundreds of thousands of nodes! If we don't want to store all that, it turns out we don't have to. 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). Here's what this looks like:

```
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.

This is a good moment to pause and take stock. 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. Why is this better?

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! $2^{20} = 1048576$. (A 3x3 puzzle has a smaller solution space, but on a 4x4 or larger this is a good estimate.) 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 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!

Get Your Code Ready to Turn In

To grade this, I just want to see what I showed in the snippet above. I'll provide a file of puzzles like `15_puzzles.txt` – they will all be size 4x4, and each line of the .txt file will only contain the puzzle, not a number at the beginning representing size. As usual, you'll get the name of this file on the command line. I want your code to run BFS and ID-DFS on each one and show the results like I did above. (I will verify that your code generates the path length in each search separately and that you aren't just printing the same number twice!)

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 `15_puzzles.txt` – nothing but a board on each line.
    - Output the line number, solution length, and time to solve each puzzle for both BFS and ID-DFS. (See sample output.)
- Your speed is no more than 5x what you see in on the previous page. (So, if you're taking 20 minutes to solve line 18 with ID-DFS, that's not good! If your speed is too slow, very carefully review the pseudocode.)