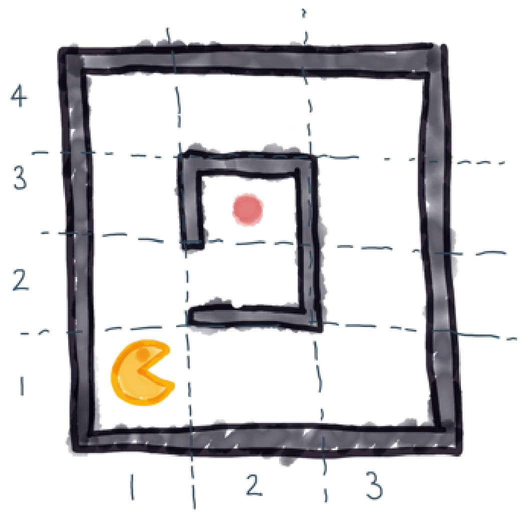
lennyboyatzis
Software Engineer
Aug 5, 2017 · 5 min read

AI—Teaching Pacman To Search With Depth First Search

The goal of this article is to explain ***Depth First Search (DFS)*** through looking at an example of how we use can use it to help Pacman navigate from a start state (1,1) to a goal state (2,3) as shown in the illustration below.



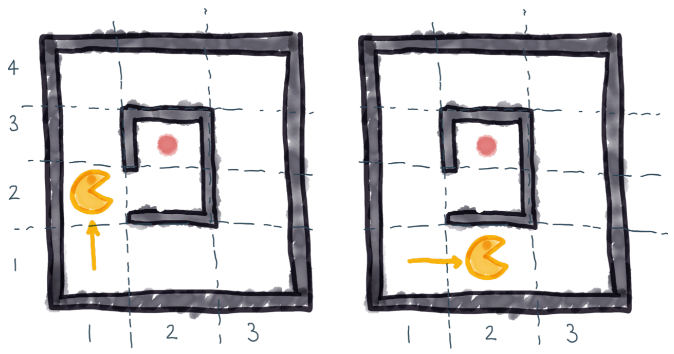
What is Depth First Search?

> *DFS is an algorithm used for performing an uninformed search through tree or graph data structures.*

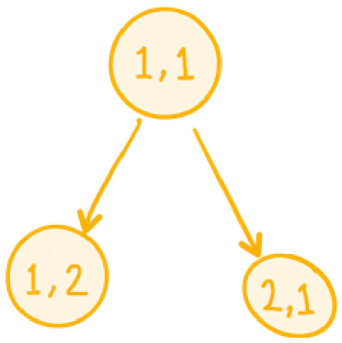
What do we mean by uninformed? DFS is not as clever as other tree/graph search algorithms as it does not take into account the cost of exploring a particular path or use a heuristic to guide exploration decisions. In saying that, understanding DFS will make it easier to understand informed search algorithms such as Uniform Cost Search (UCS) or A* search which we will be covering in subsequent posts.

What does all of this have to do with Pacman? As Pacman navigates around the maze it must decide where it should explore next in order to find the goal state. Lets visualise what this looks like for the first tier of nodes in our tree. Starting in position (1,1) and have two options;

- Move right to (1,2) or
- Move up to (2,1).

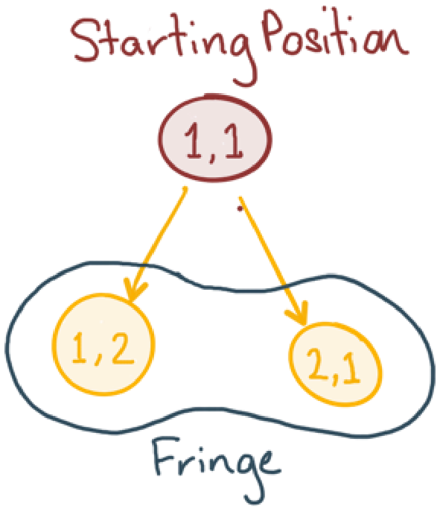


Representing this in a tree looks like this



Introducing the Fringe

The fringe can be thought of as a queue which contains the next available nodes in paths that have not yet been explored. Therefore, from the starting position the fringe is a queue containing the nodes (1,2) and (2,1).

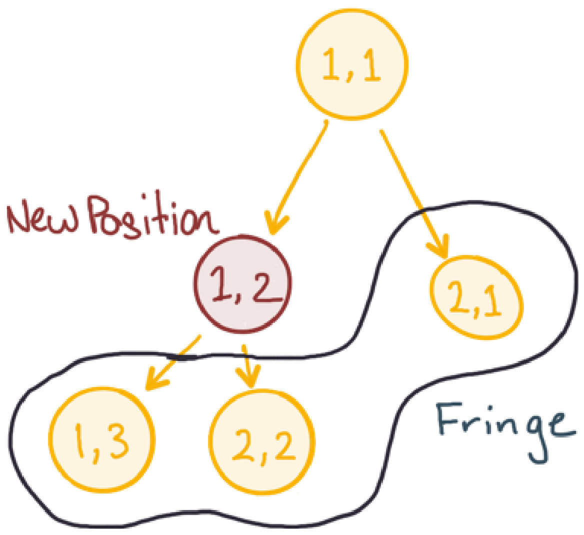


We have a fringe with nodes in it, time to explore!

DFS employs a strategy of exploring the deepest nodes on the fringe first. What does depth refer to?

> *Depth refers to the number of steps it takes to get from the start node to the node in question.*

If the nodes are all equally deep, like in the situation shown above, then we will select one randomly (to keep things simple). Node (1,2) seems like a decent candidate. Lets go there.



So to break down what has happened by moving to the new position (1,2)

1. We removed (1,2) from the fringe,

2. We checked to see whether (1,2) is the goal position (which it isn't)

3. We add (1,2) to our list of explored nodes

4. Then add (1,2)'s child nodes, (1,3) and (2,2), to the fringe

Our fringe is now made up of 3 nodes of varying depths in the tree. Using the explanation of depth from above;

- Nodes (1,3) and (2,2) will have a depth of 2, as it requires two steps to get to them from the start
- Node (2,1) has a depth of 1, as it only takes one step to get to it from the start

So we have 3 nodes, which one do we chose to explore next? The deepest ones of course! Remembering that if we have equally deep nodes on the fringe that we can just choose one randomly (or the next node in the queue). This brings us to the signature part of the DFS algorithm—**prioritising depth over breadth**. To do this we are going to use the help of a handy data structure known as a priority queue.

What is a Priority Queue?

> *A priority queue is an extension of a queue where every item in the queue has a priority associated with it (in this case the number of steps required to visit the node). Items are dequeued based on priority and if two elements have the same priority, they are served according to their order in the queue.*

Earlier on I mentioned that the fringe can be thought of as a Queue. This is true but to allow us to factor in depth we will use a priority queue. Every time we add a node to the fringe, we will use the nodes depth in the tree as it's priority level. So that way when deciding which nodes on the fringe to explore next we simply get the next node form the priority queue.

An Implementation of DFS in Python

Following the process mentioned above we can go ahead and implement DFS. Before we do that, we need a way to represent this graph/tree in code. There are a variety of ways to achieve this but we will use an adjacency list. The adjacency list stores;

- Keys: Representing each of the nodes in the graph
- Values: Set of adjacent nodes to the key node.

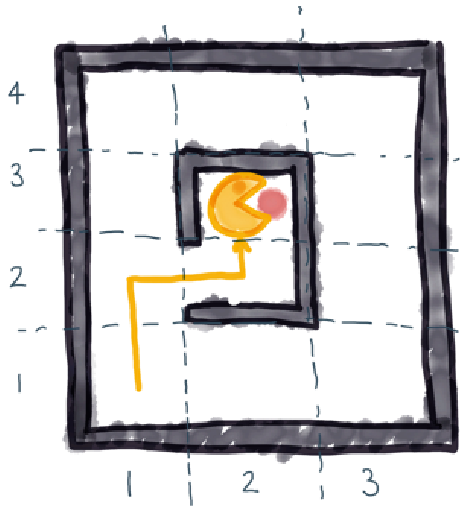
```
graph = {
    (1,1): set([(1,2), (2,1)]),
    (1,2): set([(2,2), (1,3)]),
    (1,3): set([(1,4), (1,2)]),
    (1,4): set([(1,3), (2,4)]),
    (2,1): set([(1,1), (1,3)]),
    (2,2): set([(2,3), (1,2)]),
    (2,3): set([(2,2)]),
    (2,4): set([(1,4), (3,4)]),
    (3,1): set([(2,1), (3,2)]),
```

Following the 4 key steps that we went through above we can implement an iterative DFS.

```python
from queue import PriorityQueue

def dfs(graph, start, goal):
    # initialise visited list, path list & fringe
    # Add starting node to the fringe
    visited = []
    path = []
    fringe = PriorityQueue()
    fringe.put((0, start, path, visited))

    # While there are still nodes in the fringe, keep expl
    while not fringe.empty():
        # 1. Remove the next most prioritised node from th
        depth, current_node, path, visited = fringe.get()

        # 2. Check to see if it is the goal node
        if current_node == goal:
            return path + [current_node]

        # 3. Add to our list of explored nodes
        visited = visited + [current_node]

        # If not goal, get its child nodes
        child_nodes = graph[current_node]
```

DFS for the win

If we run the code above passing in the graph from above, a starting state of (1,1) and a goal state of (2,3) our DFS algorithm returns the path [(1,2), (2,2), (2,3)] and with that, our little Pacman is a happy camper!



Combining the code snippets

Note: This particular example is using Python 3.6

```
from queue import PriorityQueue

def dfs(graph, start, goal):
    visited = []
    path = []
    fringe = PriorityQueue()
    fringe.put((0, start, path, visited))

    while not fringe.empty():
        depth, current_node, path, visited = fringe.get()

        if current_node == goal:
            return path + [current_node]

        visited = visited + [current_node]

        child_nodes = graph[current_node]
        for node in child_nodes:
            if node not in visited:
                if node == goal:
                    return path + [node]
                depth_of_node = len(path)
                fringe.put((-depth_of_node, node, path + [

    return path

if __name__ == "__main__":
    graph = {
```

I hope this post was helpful. DFS is just a starting point, stay tuned for more posts on how we can start instructing Pacman to search more intelligently!