BFS & DFS – Path Finding Algorithms

Raymond Kim
rkim2022@chadwickschool.org
Chadwick International School

1. Introduction

Path-finding refers to the concept of finding the shortest route between two distinct points. The concept has been long explored by mathematicians and computer scientists alike, so much so that it has evolved into an entirely separate field of research. This field is heavily based on Dijkstra's algorithm for pathfinding on weighted paths, where each path takes a certain amount of time, or weight, to traverse [1]. However, for the sake of simplicity, in this paper we shall deal only with unweighted paths, where we assume traversing each path requires the same amount of time. Here, though Dijkstra's algorithm would still apply, we will explore two new algorithms for path-finding on an unweighted path – DFS and BFS.

2. Background

a) Graphs and Dictionaries

When humans view a diagram such as a path, due to their brains having evolved for advanced pattern recognition, they can efficiently analyze the diagram and immediately discern possible routes from point A to point B. As such, diagrams are all humans require as input for pathfinding. However, algorithms are usually iteratively run on a computer which, without extensive coding, cannot analyze and recognize images. This poses a problem when analyzing paths. After all, in order for our algorithms to function, our code must first understand the given data. So the question arises: How do we input paths into computers?

To solve this problem, mathematicians incorporated a section of mathematics called "Graph Theory", which deals with mathematical structures called "graphs". As shown in Figure 1, graphs consist of nodes, depicted by circles, and edges, depicted by lines between the circles.

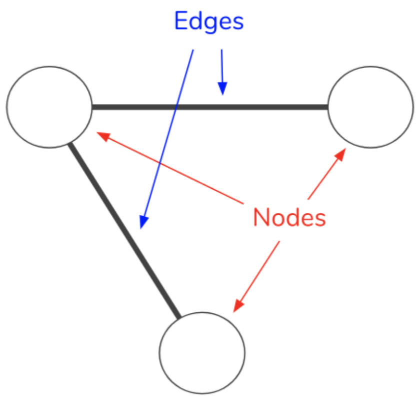


Figure 1: Graph with edges and nodes labeled and color coded

To illustrate exactly how graphs can be used to simulate paths, let us have an example path to work with. We will work with the path shown in Figure 2.

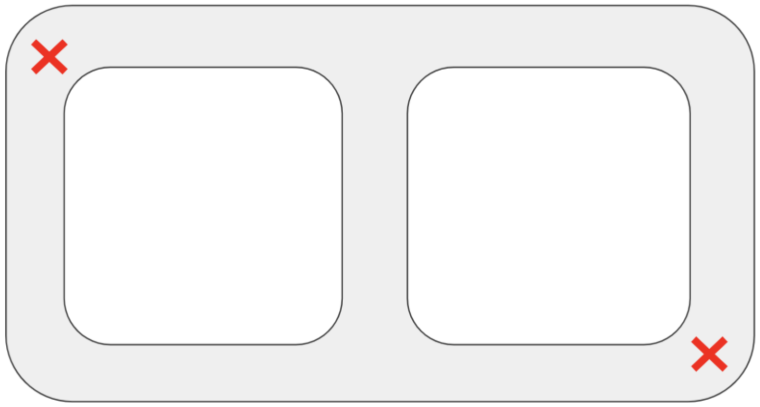


Figure 2: Example path in a 2 by 1 rectangular array formation

We turn the path into a graph by changing the individual paths into edges and intersections of two or more paths into nodes. We then label the nodes with any categorization, though letters are often used. The final graph representation of the path in Figure 2 is shown in Figure 3.1.

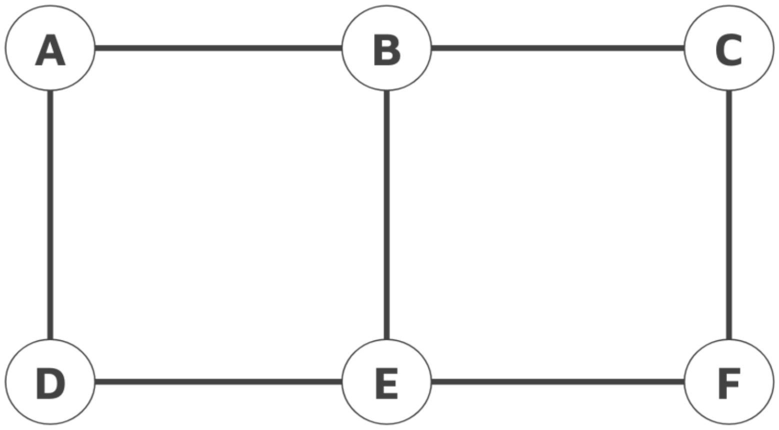


Figure 3.1: Graph representation of path shown in Figure 2

Here, it is extremely important to note that the graph we are using is an unweighted graph, meaning that each edge doesn't carry its own particular weight. As such, the actual lengths of each edge or path itself is meaningless; the graph shown in Figure 3.2, for example, is completely identical to the graph shown in Figure 3.1 as the relationships between each node do not change. The specification of the path in Figure 2 being in a '2 by 1 rectangular array formation' can therefore be treated as meaningless information.

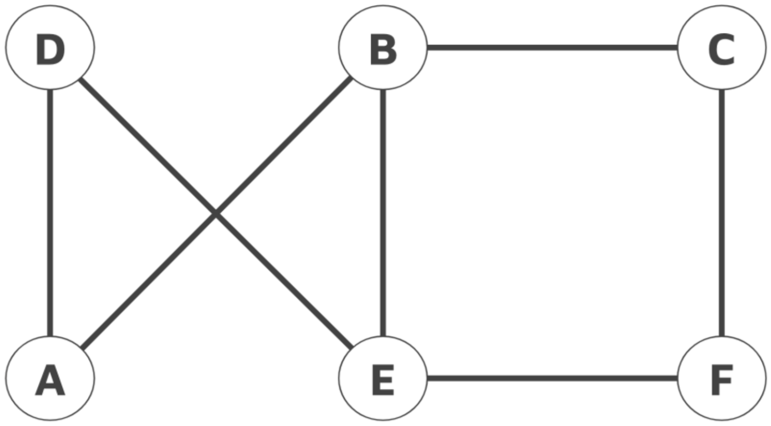
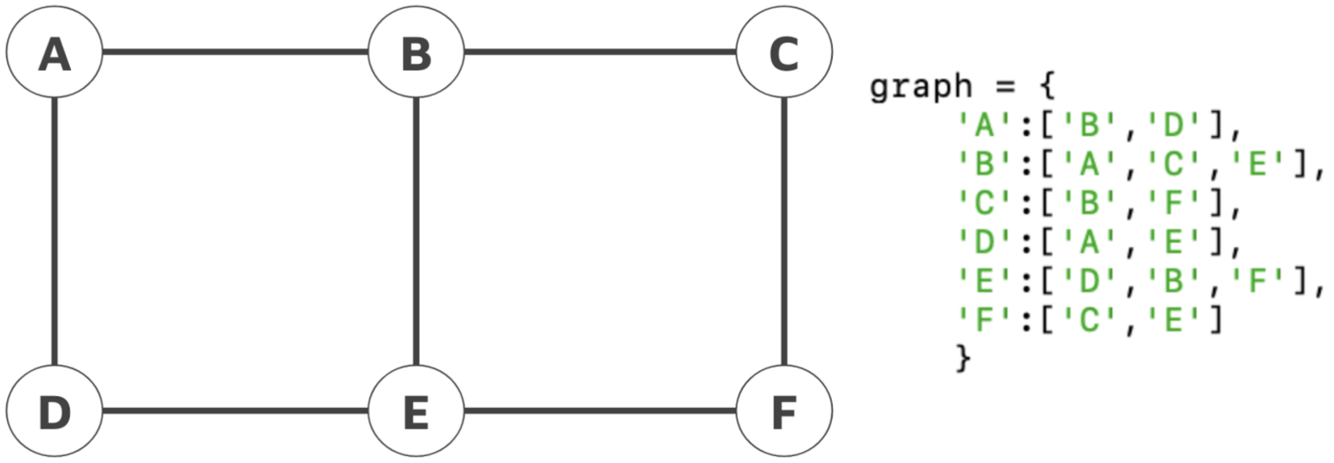


Figure 3.2: Different depiction of graph shown in Figure 3.1

We realize the only information necessary to be able to perfectly recreate the graph is the relationships between each node – in other words, the edges. For convenience, we will also store each individual node as well. These data will be stored in a data structure called a "dictionary".

Dictionaries are composed of "keys" and "values", where each key is associated with a singular value. Similar to how, in a normal dictionary, one would search up a certain word to retrieve the definition, one can look up a certain key to retrieve its value. This type of data structure is also referred to as hashes, hashtables, or hashmaps, although we will refer to them as dictionaries throughout this paper.

For our graph in Figure 3 and 4, we will create a dictionary and store each node as individual keys and a list of all possible nodes we can go to from the selected node as our associated values. An example dictionary written in Python is shown below in Figure 4.1.



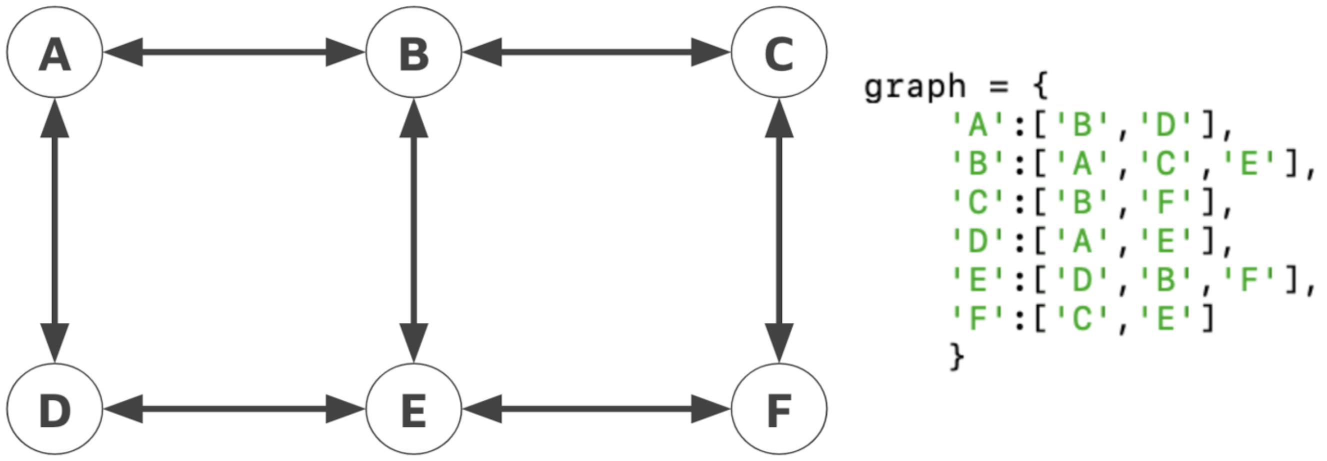
```
graph = {
    'A':['B','D'],
    'B':['A','C','E'],
    'C':['B','F'],
    'D':['A','E'],
    'E':['D','B','F'],
    'F':['C','E']
    }
```

Figure 4.1: Graph with dictionary implementation shown on the right

Let us observe Figure 4.1. For example, if we were looking at node A, by looking up the key 'A' in our dictionary, we retrieve all the possible nodes we can then go to: in this case, nodes B and D. This would then represent the two edges, AB and AD. Looking up node B would return us edges BA, BC, and BE.

Now, one might have noticed that our information seems repeated, as AB and BA seem like identical paths. However, this is not so. Paths can either be two-directional, meaning the path can be traversed both ways, or one-directional, where the path can only be traversed in a certain direction. This is also referred to as undirected and directed. AB and BA in the dictionary shows the two-directional nature of edge AB. The directions of each edge is usually represented by arrows. However, as our graph is an undirected graph, the usage of arrows becomes unnecessary.

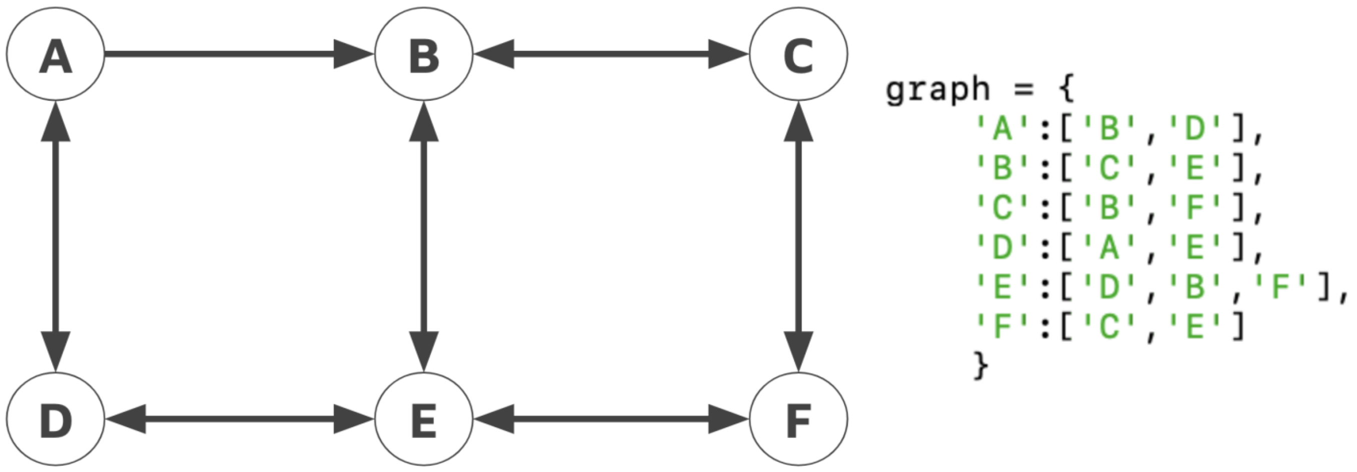
A graph with directional markings on the paths is shown in Figure 4.2.



```
graph = {
    'A':['B','D'],
    'B':['A','C','E'],
    'C':['B','F'],
    'D':['A','E'],
    'E':['D','B','F'],
    'F':['C','E']
    }
```

Figure 4.2: Extensive version of previous graph with arrows representing the possible direction(s) of each path

If we were to alter our graph a bit and change edge AB such that one could only traverse in the direction A to B, our graph and dictionary would alter accordingly, as shown in Figure 4.3.



```
graph = {
    'A':['B','D'],
    'B':['C','E'],
    'C':['B','F'],
    'D':['A','E'],
    'E':['D','B','F'],
    'F':['C','E']
    }
```

Figure 4.3: Alteration of graph and dictionary in Figure 6

Here, when drawing our graph, the usage of arrows for each path becomes necessary. Our dictionary is also altered; our key ‘A’ returns node B, but our key ‘B’ does not return node A, as we cannot reach node A from node B. We observe that our graph contains the minimal information necessary for a perfect recreation of our desired path. Any path can be turned into a graph and stored as a dictionary using the same process. As such, we can now successfully input any path into a computer for pathfinding.

b) Queues and Stacks

Many algorithms and programs utilize data structures that allow for the algorithm to run more efficiently. DFS and BFS use two of the most commonly used data structures – queues and stacks. Both are structured containers for objects and each have two operations available for the insertion and removal of objects of the data structure [2].

i) Queues

Queues function according to the first-in-first-out (FIFO) principle. To put it into perspective, one can imagine a queue in line for the opening of a new shopping mall. Whoever enters the queue first gets to enter the mall first – the line operates on a first-come-first-serve principle. Similarly, it is helpful to think of the queue as a narrow tunnel wide enough for only one object where the objects line up within, such as the diagram shown in Figure 5.

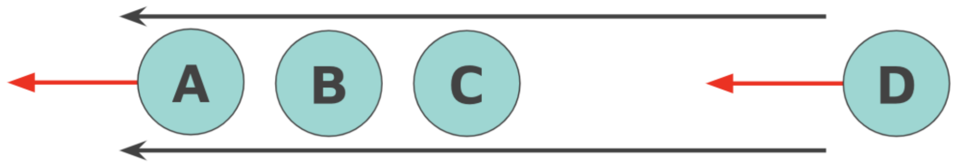


Figure 5: Queue data structure represented as a narrow, one-directional tunnel

Queues have two operations: enqueue and dequeue. Enqueuing inserts objects to the back of the queue, while dequeuing removes and returns the object at the front. Another operation commonly used is peeking, where the queue returns the object at the front without removing it from the queue.

ii) Stacks

Stacks function according to the last-in-first-out (LIFO) principle. A helpful analogy is to think of a stack of books; one can only add or remove books at the very top. Similarly, one can think of a stack as a large pit or tall container wide enough for only one object where the objects stack up within, such as the diagram shown in Figure 6. The last object to be pushed, or the object at the top, is commonly referred to as the "top", while the objects underneath are referred to as the "stack".

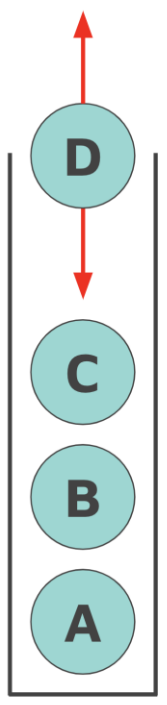


Figure 6: Stack data structure represented as a narrow pit or container

Stacks also have two operations: push and pop. Pushing inserts objects to the top of the stack to become the new top, while popping removes and returns the top while the next object becomes the new top. Stacks also have a peeking feature, where it returns the object at the top without removing it from the stack.

3. Depth First Search

DFS, or Depth First Search, is arguably the simpler of the two algorithms. Although both algorithms are driven by iterative processes, the rules determining each iteration is simpler for DFS. The fundamental mechanism for DFS, although perhaps not explicit, is often a stack that will push and/or pop nodes for each iteration. Any DFS program that doesn't explicitly have a stack in use will still usually run by a similar principle.

a) DFS Explanation

The basis of DFS is as such. For each node in consideration, the algorithm first pushes it into the stack. It then checks whether that node is our desired final node. If so, the algorithm returns the route taken by popping out all the objects in the stack in order and stops. If not, the algorithm checks all possible nodes available from its current node that have not yet been considered. It then chooses a random node out of the available list and repeats. If all possible nodes from the current node have been considered, the algorithm backtracks its path by popping the stack until it comes across a node that has available nodes. If the algorithm has traversed the entire graph without meeting the final node, the algorithm determines there is no route available from the starting node to the final node.

b) DFS Runthrough with Example

To demonstrate how DFS functions, let us start with an example path shown in Figure 7.

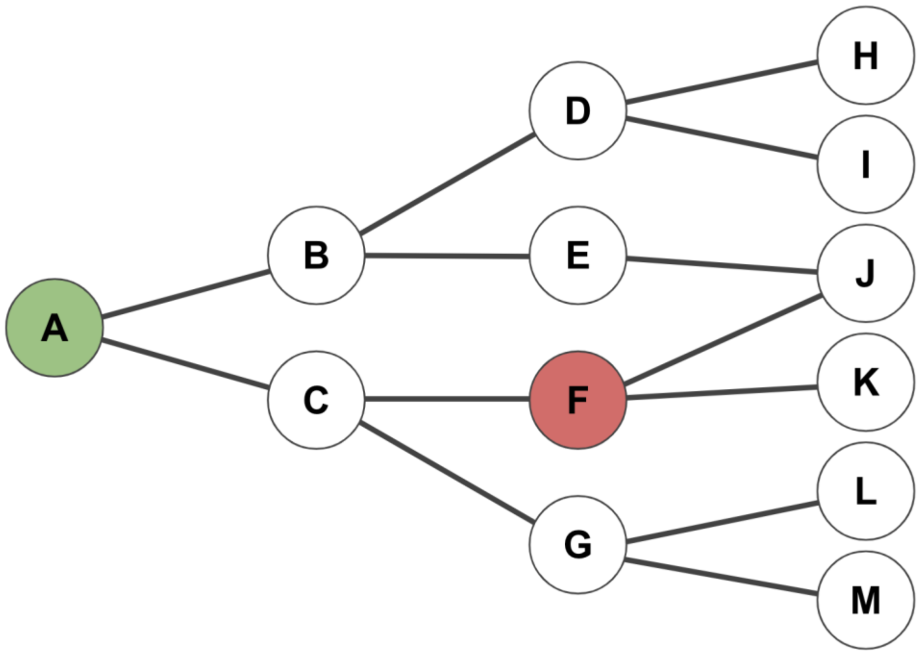


Figure 7: Example graph with starting node A colored green and finishing node F colored red

DFS first starts at node A, pushing it into the stack. It checks that node A is not node F, observes that nodes B and C is available and selects a random node out of the two. As true randomness does not exist in the computing world, for our intents and purposes let us assume that the algorithm has an upwards priority; that is, it checks nodes on the upper side of the graph first. (This would work in this instance as our graph is organized in a tree structure.) DFS would therefore select node B for consideration. It then repeats this process until it reaches node H, where it observes there is no node available and backtracks to node D by popping out H from the stack and having D be the top. It checks node I, the last available node from node D, then backtracks until node B. It then traverses node E and J until it finally reaches node F, the desired final node, and stops. DFS would therefore return the path by popping out the stack entirely and reversing the order, resulting in the path A→ B→ E→ J→ F.

From the example above, we realize why DFS is aptly named “depth”, as it goes as far, or deep, as possible in a certain direction until failing (or succeeding), after which it backtracks and repeats the process.

We also notice the major downside of DFS in that it doesn’t return the shortest route possible, instead only returning whether a path exists. Furthermore, the runtime of DFS depends entirely on the order in which DFS selects and considers the nodes. In the graph above, DFS could theoretically finish in two iterations, or it could traverse nearly the entire graph before succeeding. DFS is therefore quite unreliable, not only in terms of optimization but also in terms of time consistency.

4. Breadth First Search

BFS, or Breadth First Search, works quite differently from DFS. The fundamental mechanisms for BFS are a queue and a dictionary named “parents”. For the sake of this algorithm, we will define a parent-child relationship between two nodes to be the following:

If node A is selected and node B is enqueued into the queue via node A, then node A is the parent node and node B is the child node.

In “parents”, both the keys and values will be singular nodes. The keys will be child nodes, and the values will be the parent node for the corresponding child node. Hence, if one looks up a node in “parents”, they will receive the parent node.

a) BFS Explanation

The basis of BFS is as such. We first start by enqueuing the starting node into the queue and setting the starting node’s parent to be None in “parents”. We then run the following iterative process. We first dequeue a node from the queue, thereby ‘selecting’ it. We then enqueue all the nodes available from the selected node that we haven’t yet enqueued into the queue. At the same time, we update the “parents” dictionary accordingly, making sure not to alter the nodes that have already been enqueued once. We repeat this iterative process until the dequeued node is our desired final node. We then run another iterative process where we ‘backtrack’ our steps using the “parents” dictionary, then print our route. As long as there exists a route between the starting node and the final node, the queue should never be empty at any point in the iterative process. Therefore, if the queue is ever empty, the algorithm determines there is no route available from the starting node to the final node.

b) BFS Runthrough with Example

To demonstrate how BFS functions, let us start with an example path shown in Figure 8.1.

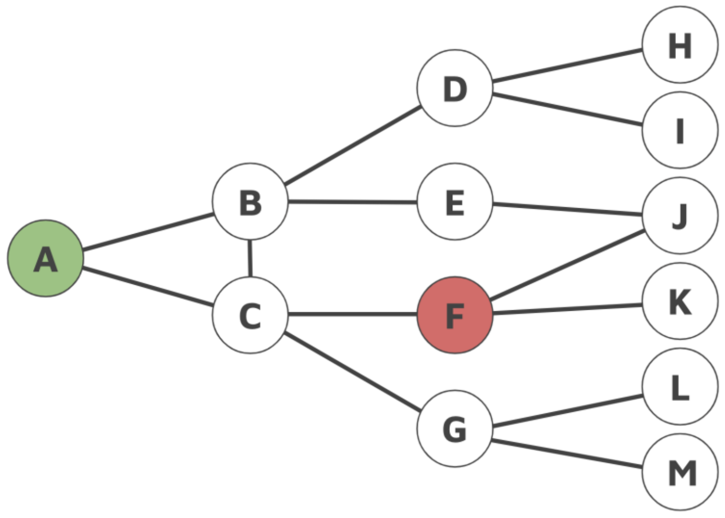


Figure 8.1: Example graph with starting node A colored green and finishing node F colored red

We first initialize the queue and “parents” with the starting node, as shown in Figure 8.2.

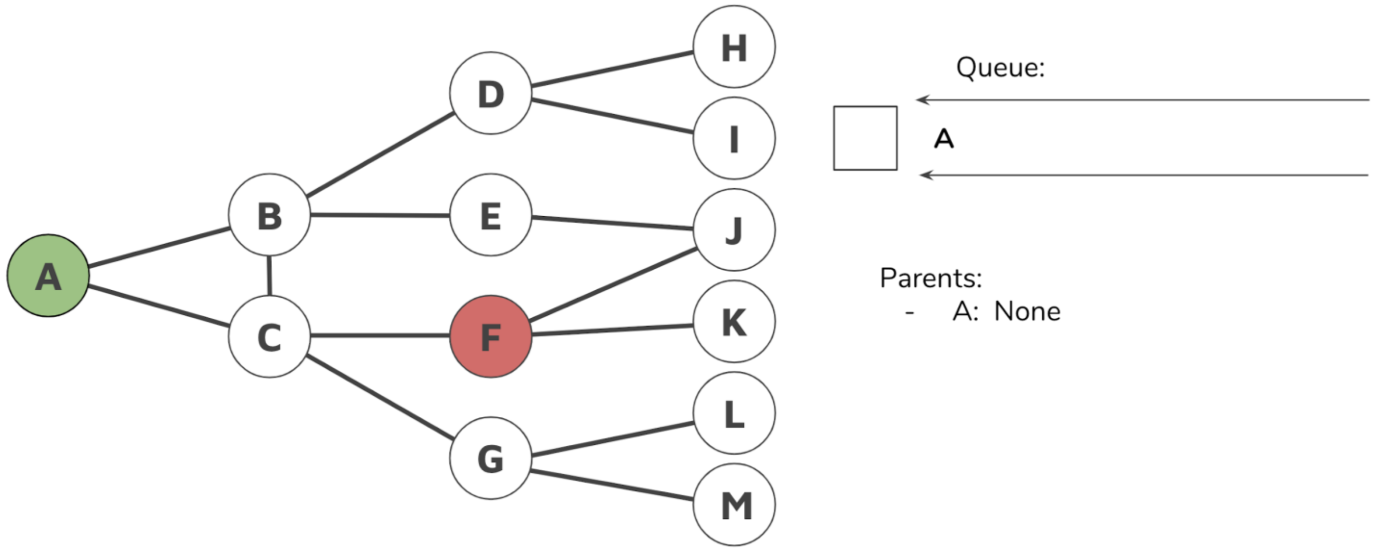


Figure 8.2: Initialization of queue and “parents” with starting node A

We then start our iterative process and dequeue the queue, selecting node A. As node B and C are available from node A and haven’t been enqueued before, they are both enqueued and updated in “parents”, shown in Figure 8.3.

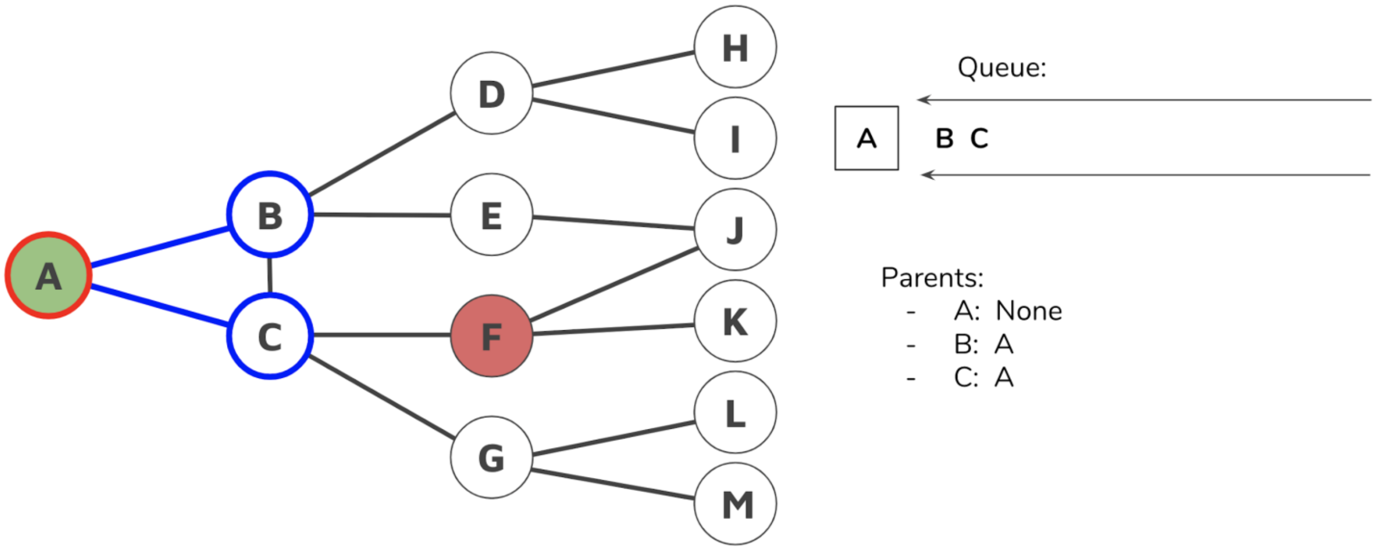


Figure 8.3: Nodes B and C are enqueued and updated in “parents”. Selected node is highlighted in red, and available nodes are highlighted in blue.

We then dequeue again to select node B. As nodes A and C have already been enqueued, only nodes D and E are available. As such, they are enqueued and updated in “parents”, shown in Figure 8.4.

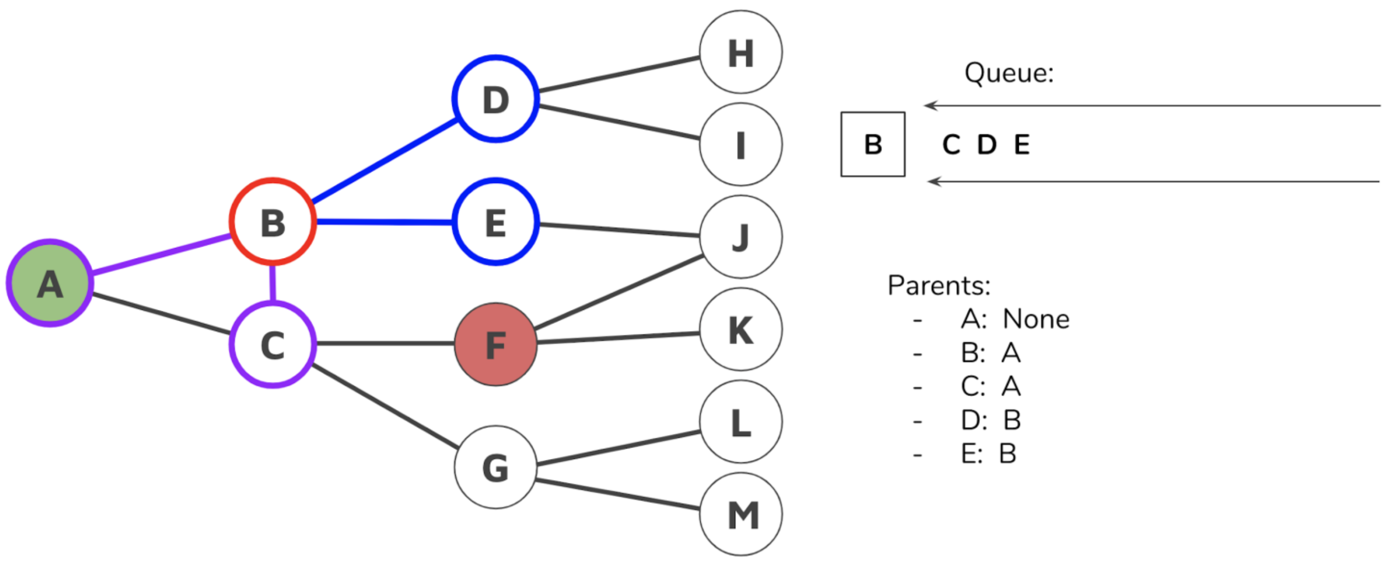


Figure 8.4: Nodes D and E are enqueued and updated in "parents". Selected node is highlighted in red, available nodes are highlighted in blue, and disregarded nodes are highlighted in purple.

Repeating this process, we dequeue node C to enqueue nodes F and G, dequeue node D to enqueue nodes H and I, and dequeue node E to enqueue node J. We then dequeue node F, and the iterative process stops. The queue and respective parents of each node is shown in Figure 8.5.

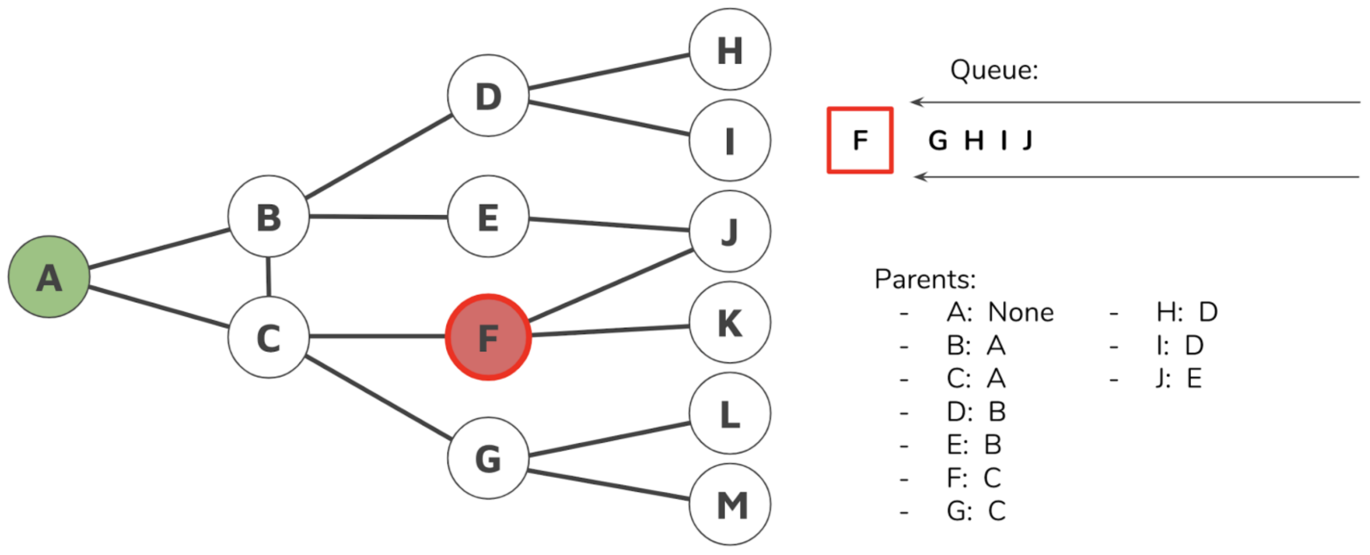


Figure 8.5: Completed iterative process. Parents of each considered node are shown.

Finally, to return the traversed route, we backtrack using "parents" using another iterative process. Node F's parent node is node C, and node C's parent node is node A. We know node A is the starting node as it has no parent node. For each iteration, each node is pushed into a stack in order of F, C, and A. The stack is then popped entirely to result in the path A → C → F. The final result of the BFS algorithm is shown in Figure 8.6.

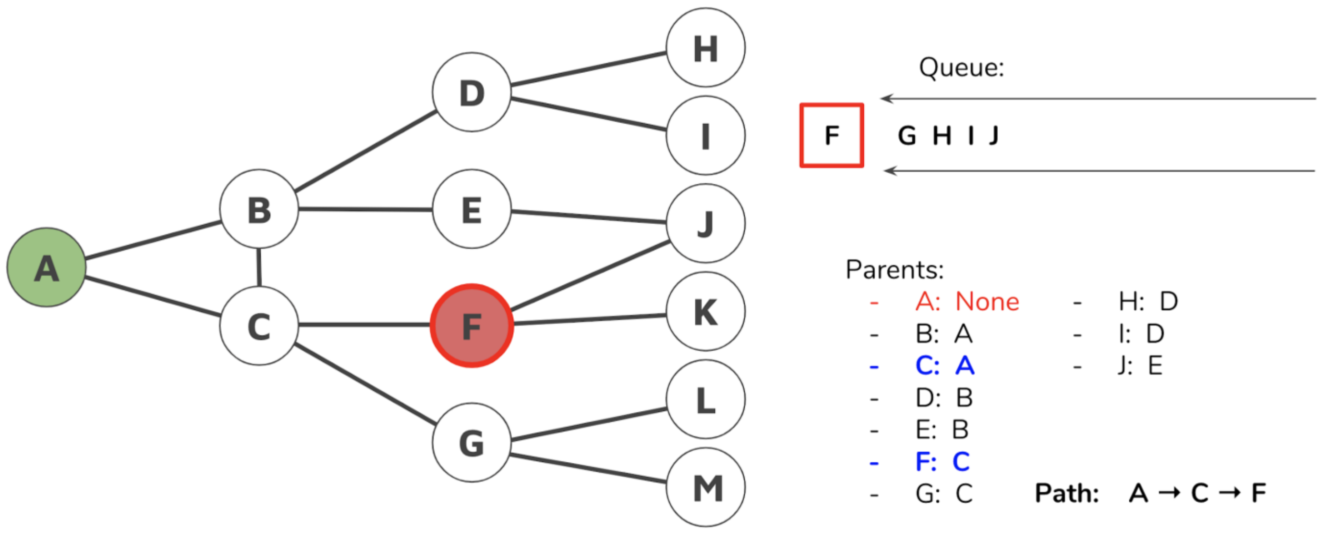


Figure 8.6: Final result of BFS algorithm with finalized and returned path

From the runthrough above, we observe why BFS is aptly named "breadth", as it traverses through the nodes layer by layer, with each layer being defined by the minimum number of steps it takes to reach a node from the starting node. For example, in the graph above, node B is considered part of Layer 1, as it takes 1 step from node A. Similarly, node C would be part of Layer 1 and nodes D, E, F, G would be Layer 2. Indeed, if we see the progression of the queue, all of Layer 1 is dequeued first before any node in Layer 2 gets dequeued. Hence, BFS always returns the shortest path from the starting node to the finishing node.

Furthermore, BFS is a lot more time consistent; although the order of enqueuing of available nodes is random, it has no significant effect on the code runtime as it only affects individual layers. Compared to DFS, where the iterations could span between one or two to potentially the entire graph, BFS is significantly more consistent and reliable.

5. Code Runtime Efficiency

When assessing the efficiency and effectiveness of a program, the term "code runtime" is often used. Code runtime refers to the maximum number of iterations a program has to complete, or the maximum execution time, relative to the size of the input, and is denoted as $O(\ )$. $O(1)$, for instance, would denote a linear runtime, meaning that it would take the same amount of time regardless of the size of the input. $O(n)$ would mean a linear proportionality between the input size and execution time.

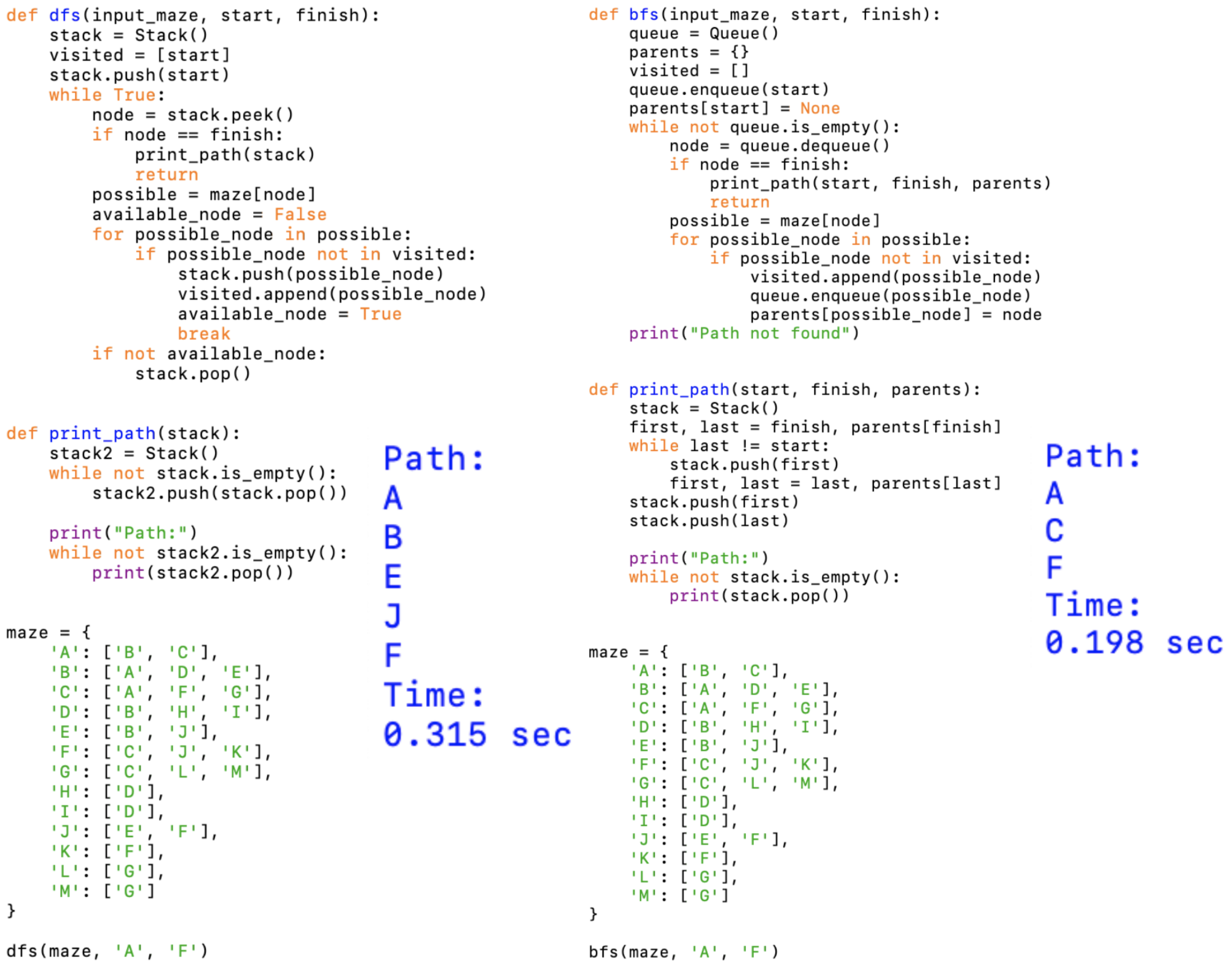
For both BFS and DFS, the code runtime efficiency is $O(|V| + |E|)$, where $|V|$and $|E|$denotes the number of vertices and edges respectively. This is because for both algorithms, the only operations executed (excluding executions of which their sole purpose was to return the answer in a recognizable format) are pushing / enqueuing nodes into stacks / queues and scanning for adjacent, available nodes. The maximum number of pushes / enqueues is the number of vertices, hence $O(|V|)$. Scanning for adjacent nodes take roughly the same amount of time, hence $O(1)$, and scanning only takes place when there is an edge, multiplying by the number of edges $|E|$ to become $O(|E|)$. Therefore, the final runtime for both BFS and DFS is as shown in Equation 1 below.

$$O(|V|) + O(|E|) = O(|V| + |E|) \tag{1}$$

The runtime of both BFS and DFS are linearly proportional to the number of vertices and edges of the path input. However, $O(|V| + |E|)$ refers to the expected average maximum code runtime of BFS and DFS. It does not represent the time consistency or reliability of the algorithms, of which BFS is superior over DFS.

6. Python Code Comparison

The following code shown in Figure 9 is a recreation of the DFS and BFS algorithms on Python. We observe a stark difference in the results and the execution time between the two at very similar code lengths.



```python
def dfs(input_maze, start, finish):
    stack = Stack()
    visited = [start]
    stack.push(start)
    while True:
        node = stack.peek()
        if node == finish:
            print_path(stack)
            return
        possible = maze[node]
        available_node = False
        for possible_node in possible:
            if possible_node not in visited:
                stack.push(possible_node)
                visited.append(possible_node)
                available_node = True
                break
        if not available_node:
            stack.pop()


def print_path(stack):
    stack2 = Stack()
    while not stack.is_empty():
        stack2.push(stack.pop())

    print("Path:")
    while not stack2.is_empty():
        print(stack2.pop())


maze = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F', 'G'],
    'D': ['B', 'H', 'I'],
    'E': ['B', 'J'],
    'F': ['C', 'J', 'K'],
    'G': ['C', 'L', 'M'],
    'H': ['D'],
    'I': ['D'],
    'J': ['E', 'F'],
    'K': ['F'],
    'L': ['G'],
    'M': ['G']
}

dfs(maze, 'A', 'F')
```

```
Path:
A
B
E
J
F
Time:
0.315 sec
```

```python
def bfs(input_maze, start, finish):
    queue = Queue()
    parents = {}
    visited = []
    queue.enqueue(start)
    parents[start] = None
    while not queue.is_empty():
        node = queue.dequeue()
        if node == finish:
            print_path(start, finish, parents)
            return
        possible = maze[node]
        for possible_node in possible:
            if possible_node not in visited:
                visited.append(possible_node)
                queue.enqueue(possible_node)
                parents[possible_node] = node
    print("Path not found")


def print_path(start, finish, parents):
    stack = Stack()
    first, last = finish, parents[finish]
    while last != start:
        stack.push(first)
        first, last = last, parents[last]
    stack.push(first)
    stack.push(last)

    print("Path:")
    while not stack.is_empty():
        print(stack.pop())


maze = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F', 'G'],
    'D': ['B', 'H', 'I'],
    'E': ['B', 'J'],
    'F': ['C', 'J', 'K'],
    'G': ['C', 'L', 'M'],
    'H': ['D'],
    'I': ['D'],
    'J': ['E', 'F'],
    'K': ['F'],
    'L': ['G'],
    'M': ['G']
}

bfs(maze, 'A', 'F')
```

```
Path:
A
C
F
Time:
0.198 sec
```

Figure 9: DFS and BFS Python Code Comparison. Results of each algorithm are shown on the right side of their respective code. Execution times of the algorithms are written beneath the results.

7. Conclusion

Through this investigation, we were able to learn about two different path-finding algorithms for unweighted graphs. We analyzed the fundamental mechanics of each algorithm, explored some of the mathematical field of Graph Theory, and learned multiple types of data structures along the way. The algorithms explored in this investigation, along with other concepts pertaining to computer science, serves as the foundation for numerous other path-finding algorithms, data searching and data management methods used in many servers and databases worldwide today.

8. References

[1] Schrijver, Alexander. Mathematics Subject Classification, 2010, pp. 155–156, *On the History of the Shortest Path Problem*.

[2] Adamchik, Victor S. “Stacks and Queues.” *Carnegie Mellon University*, 2009, www.cs.cmu.edu/~adamchik/15-121/lectures/Stacks%20and%20Queues/Stacks%20and%20Queues.html.