









31 - Adjacency List

00:00

Suggested Problems

Status	Star	Problem 	Difficulty 	Solution
<input type="checkbox"/>		Clone Graph 	Medium	
<input type="checkbox"/>		Course Schedule 	Medium	

Adjacency List

An adjacency list is probably the easiest graph format of graphs to traverse. This is mainly because with an adjacency list, we can easily find all the neighbors of a vertex and we don't have to worry about going out of bounds.

Suppose we are given a list of directed edges and we have to build an adjacency list.

Adjacency List



The code below demonstrates how we can build an adjacency list. We can use a hashmap where the key is a vertex and it maps to a list or array of its neighbors, which are also vertices. A hash map works here because we are assuming that all of the values keys are unique.

Python**Java****C++****JavaScript****C#****Swift**

```
# GraphNode used for adjacency list
class GraphNode:
```

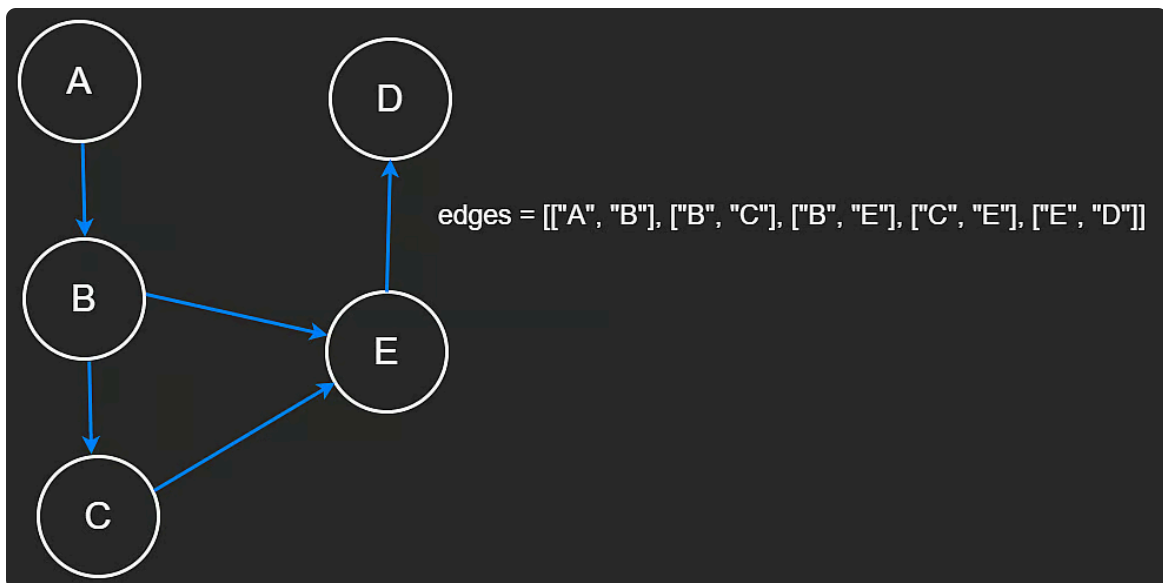
```
def __init__(self, val):
    self.val = val
    self.neighbors = []

# Or use a HashMap
adjList = { "A": [], "B": [] }

# Given directed edges, build an adjacency list
edges = [{"A", "B"}, {"B", "C"}, {"B", "E"}, {"C", "E"}, {"E",
"D"}]

adjList = {}

for src, dst in edges:
    if src not in adjList:
        adjList[src] = []
    if dst not in adjList:
        adjList[dst] = []
    adjList[src].append(dst)
```



DFS on an adjacency list

Adjacency List DFS



Let's say that we wanted to count the number of paths that lead from a source to destination.

In the code below, we have an adjacency list, a source, and a `target`. Similar to matrix traversal, we will make use of a hashset called `visit` to keep track of the vertices that we have already visited.

We will then recursively run DFS on our list until we reach the target node, after which we will return `1`. Once we have found a path, we will backtrack by removing nodes from our `list` and return the `count`.

Python**Java****C++****JavaScript****C#****Swift**

```
# Count paths (backtracking)
def dfs(node, target, adjList, visit):
    if node in visit:
        return 0
    if node == target:
        return 1

    count = 0
    visit.add(node)
    for neighbor in adjList[node]:
```

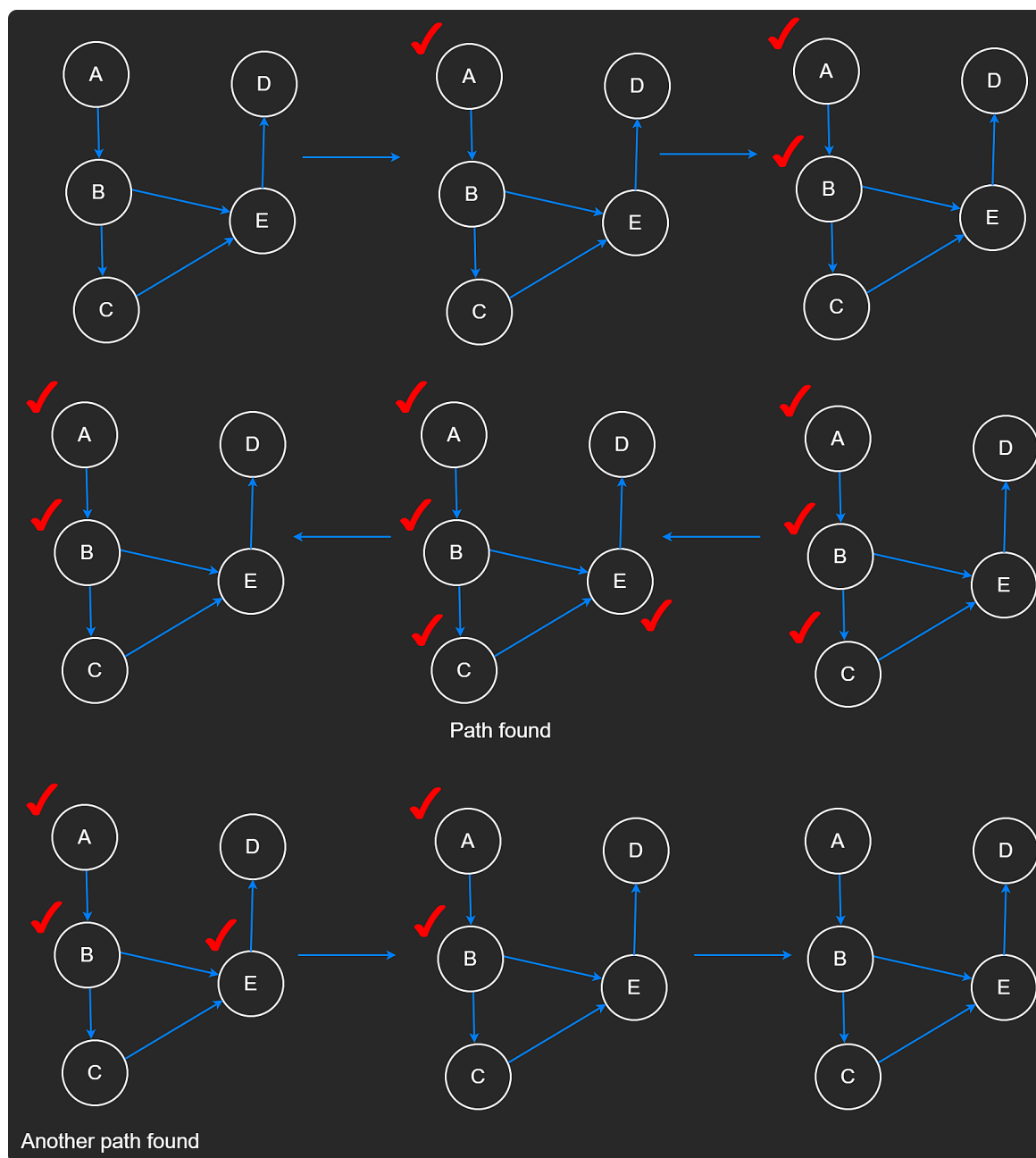
```

count += dfs(neighbor, target, adjList, visit)
visit.remove(node)

return count

```

In the image below, the above algorithm is demonstrated. The red check marks indicate that a node has been visited and is in the set.



Time Complexity

This backtracking is exponential. In the worst case, each node is connected to every other node in the graph. Recall the rule that $E \leq V^2$. So, let us say that each vertex has N edges. If we are to create a decision tree which determines how many vertices can be visited from each vertex, and the height of that tree is V , then in the worst case, we will have to do N^V work for reasons similar to what we discussed in the matrix chapter.

In the worst case scenario, N is equal to V , so the time complexity is $O(V^V)$.

BFS on an adjacency list

Adjacency List BFS



Running BFS is similar to what we have seen before. Let us say that our goal is to find the **shortest path** from node to target. By shortest path, we mean reaching the destination by visiting fewest vertices possible.

Our code looks very similar to when we did Matrix BFS, except in this case, we don't have to worry about edge cases. We will keep increasing the length at each level, until we find the target vertex.

Python

Java

C++

JavaScript

C#

Swift

```
# Shortest path from node to target
def bfs(node, target, adjList):
    length = 0
    visit = set()
    visit.add(node)
    queue = deque()
    queue.append(node)

    while queue:
        for i in range(len(queue)):
            curr = queue.popleft()
            if curr == target:
                return length

            for neighbor in adjList[curr]:
                if neighbor not in visit:
                    visit.add(neighbor)
                    queue.append(neighbor)

        length += 1
    return length
```

Looking at the image below, the above piece of code will return 2.

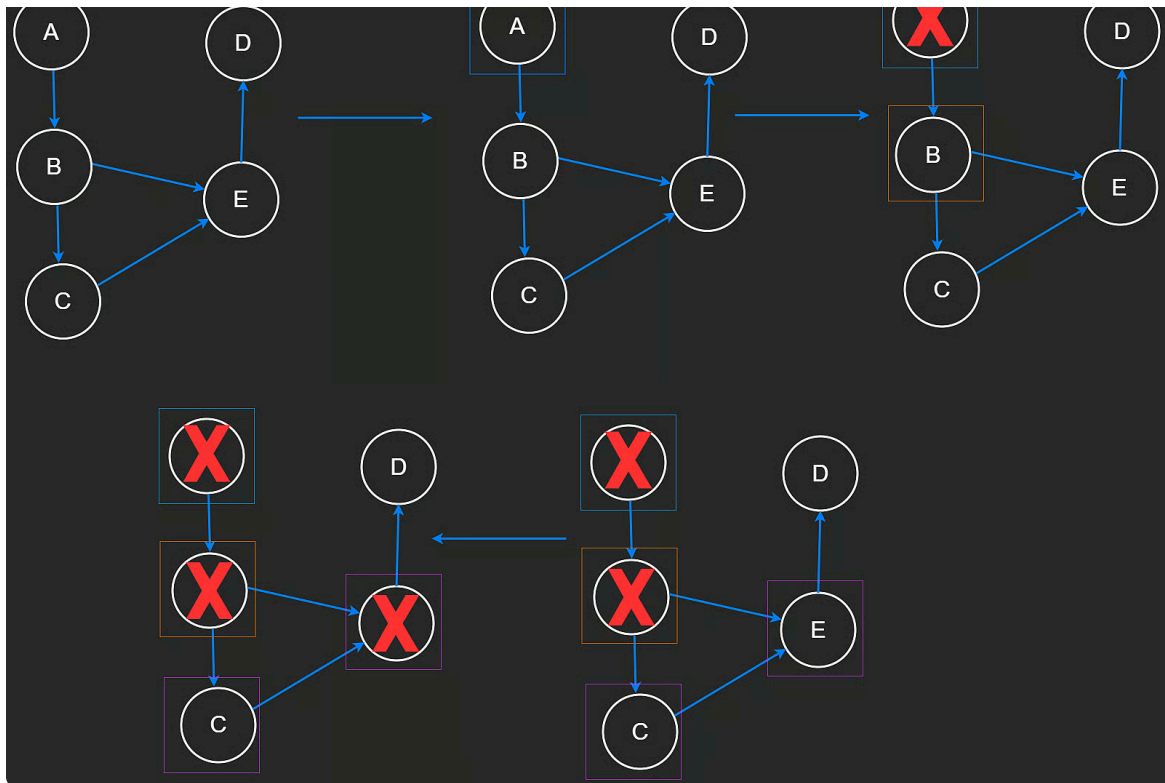


Mark Lesson Complete

Code Yourself

View Code





Time Complexity

We learned before that the number of edges in a graph is upper bounded by V^2 . However, we know that in this case, we don't have self loops and we don't have the maximal number of edges. Therefore, we can say that the time complexity is $O(V + E)$, where V is the number of vertices and E is the number of edges. This is because in the worst case, our BFS will visit every vertex and traverse every edge.

Closing Notes

That is all we are going to cover on graphs. You might not believe it but this is actually just scratching the surface and covering the basics. After all, there is a whole field of study called Graph Theory. Graphs can get even more complicated and there are a lot more algorithms that are

specific to graphs. You can find these in the **Advanced Algorithms** course.