Theory Assignment-4: ADA Winter-2024

Krishna Shukla (2022254) Kshitij (2022256)

References used:
https://www.geeksforgeeks.org/number-of-paths-from-source-to-destination-in-a-directed-acyclic-graph/

Took a little bit help of GPT for documentation

1 Preprocessing

n is the total number of vertices in the graph. Please change the value of n, source and destination and add or remove edge as per your requirement

2 Algorithm Description

1. Dynamic Programming (DP):

Role in the Algorithm:

- In the provided algorithm, dynamic programming is employed to optimize the process of finding all paths from the source to the destination in a directed acyclic graph (DAG).
- It addresses the issue of redundant calculations that may arise when traversing the graph to find all possible paths.

Memoization:

- It maintains a memoization table, often referred to as dp in the code, to store the results of intermediate calculations.
- Each entry in the dp table represents the number of paths from a particular vertex to the destination.
- By storing these results, the algorithm avoids recalculating the same paths multiple times, leading to significant efficiency gains.

Benefits:

- Dynamic programming optimizes the time complexity of the algorithm, especially when dealing with graphs with a large number of vertices and edges.
- It enables the algorithm to efficiently explore the graph and find all possible paths from the source to the destination without redundant computations.

2. Topological Sorting:

Significance in the Algorithm:

- The algorithm employs topological sorting to establish a valid order in which to traverse the vertices of the graph.
- It guarantees that dependencies or precedence relationships among vertices are respected during traversal.

Kahn's Algorithm:

- Kahn's algorithm, a popular method for topological sorting, is utilized in the algorithm implementation.
- It starts by identifying vertices with zero in-degree (i.e., no incoming edges) and adds them to a queue.
- It iteratively dequeues vertices, updates the in-degree of neighboring vertices, and adds vertices with zero in-degree to the queue.
- The process continues until all vertices are visited, resulting in a valid topological ordering.

Importance:

- Topological sorting ensures that the algorithm traverses the graph in a manner that adheres to the directional constraints imposed by the edges.
- It plays a crucial role in correctly ordering the vertices, especially when dealing with tasks or events that have dependencies or prerequisites.

Conclusion:

- Together, dynamic programming and topological sorting enable the algorithm to effectively find all paths and common vertices in a directed acyclic graph while maintaining optimal performance.

3. Main Steps:

- **Graph Representation**: The code represents the DAG using an adjacency list, where each node is associated with a list of its neighboring nodes. This representation allows efficient traversal of the graph.

- **Topological Sorting**: Before finding paths, the code performs a topological sort on the graph. Topological sorting ensures that nodes are visited in an order that respects the direction of edges in the graph. This step is crucial for correctly traversing the graph and finding paths.

- **Finding Paths**: After obtaining the topological order, the code recursively finds all possible paths from the source to the destination node. It maintains a list of visited nodes and the current path being explored. Once a path from the source to the destination is found, it is appended to the list of all paths.

- **Dynamic Programming (DP)**: To optimize the process of finding paths, the code employs dynamic programming. It utilizes memoization to store and reuse intermediate results, avoiding redundant calculations. This optimization significantly improves the efficiency of the algorithm, especially for large graphs.

- **Identifying Cut Vertices**: After finding all paths, the code identifies any common vertices (cut vertices) present in all paths between the source and destination. These cut vertices are nodes that must be traversed in all paths, indicating their significance in connecting the source and destination nodes.

- **Output**: Finally, the code prints the cut vertices found between the source and destination. If no cut vertices are present, it indicates that there are no common vertices required to traverse all paths.

Time and Space Complexity:

1. Topological Sorting:

- The time complexity of topological sorting using Kahn's algorithm isO(V+E), where V is the number of vertices and E is the number of edges in the graph.

2. Finding All Paths:

- However, the dynamic programming optimization implemented in the code helps avoid redundant computations, reducing the effective time complexity.

3. Identifying Cut Vertices:

- Identifying cut vertices involves iterating through the list of all paths and finding common vertices.
- This process typically has a linear time complexity proportional to the number of paths and the average length of each path.

Overall Time Complexity:

- Considering the steps mentioned above, the overall time complexity of the algorithm can be approximated as
- O(V+E+P), where
- P represents the total number of paths explored.

Space Complexity:

1. Graph Representation:

- The space complexity for storing the graph using an adjacency list is O(V+E), where V is the number of vertices and E is the number of edges.

2. Topological Sorting:

- The space complexity for storing additional data structures used in topological sorting, such as the queue and the degree array, is O(V), where V is the number of vertices.

3. Dynamic Programming:

- The space complexity for dynamic programming involves storing the memoization table (dp array) to cache intermediate results.
- The size of the dp array is proportional to the number of vertices (V), resulting in a space complexity of O(V).

Overall Space Complexity:

- Considering the space requirements mentioned above, the overall space complexity of the algorithm is O(V+E).

3 Pseudocode

```
Function add_edge(a, b, adj_list, deg):
  Append b to the adjacency list of vertex a in the adjacency list adj_list
  Increment the in-degree of vertex b by 1 in the degree list deg

Function topological(num_vertices, adj_list, deg):
  Initialize an empty deque called q
  Initialize an empty list called lst to store the topologically sorted vertices

  For each vertex i in the range of 0 to num_vertices - 1:
    If the in-degree of vertex i is 0, append i to the deque q
```

```
    While q is not empty:
        Remove the front element u from q
        Append u to the list lst

        For each node in the adjacency list of vertex u:
            Decrement the in-degree of the node by 1 in the degree list deg
            If the in-degree of the node becomes 0, append it to q

    Return the list lst containing the topologically sorted vertices

Function compute_paths(source, destination, adj_list, paths, current_path, visited):
    Mark the source vertex as visited
    Append the source vertex to the current_path list

    If the source vertex is equal to the destination vertex:
        Append a copy of current_path to the paths list
    Else:
        For each child vertex in the adjacency list of the source vertex:
            If the child vertex has not been visited:
                Recursively call compute_paths with the child vertex as the new source also memoizing the solution
here

    Remove the last vertex from current_path
    Mark the source vertex as unvisited

Function find_common_vertex(paths):
    If the length of the paths list is 0:
        Print "NO CUT VERTEX" and return an empty list
    Initialize a set called common_vertex with the vertices from the first path

    For each subsequent path in paths:
        Update common_vertex by intersecting it with the current path

    Return common_vertex as a list

Function numberofPaths(source, destination, adj_list, dp):
    If the source vertex is equal to the destination vertex:
        Return 1
    If the value of dp[source] is not -1:
        Return dp[source]
    Initialize a variable c to 0

    For each neighboring vertex neigh of the source vertex in the adjacency list:
        Recursively call numberofPaths with neigh as the new source vertex
        If the result x is not -1:
            Update c by adding x modulo mod

    If c is equal to 0:
        Set dp[source] to -1
```

Else:
Set dp[source] to c

Return dp[source]

Set the number of vertices num_vertices and source and destination vertices
Initialize the graph adjacency list graph and the degree list deg
Populate the graph adjacency list and the degree list deg based on the given edges
Initialize the dynamic programming array dp with -1 for memoization

Print the number of paths from source to destination using the numberofPaths function
Perform topological sorting using the topological function and store the sorted vertices in s
Initialize an empty list called paths and variables current_path and visited
Find all paths from source to destination using the compute_paths function and store them in paths
Find the common vertex among all paths using the find_common_vertex function
Print the common vertex list after removing the source and destination vertices

4 Proof of Correctness:

Correctness of Topological Sorting:

- We need to show that the topological sorting process correctly orders the vertices of the graph such that all edges point in the direction of the sorting.
- Kahn's algorithm for topological sorting guarantees this property, ensuring that the algorithm traverses the graph in a valid order.

Correctness of Path Finding:

- We must demonstrate that the algorithm accurately finds all possible paths from the source to the destination node in the given graph.
- This can be verified by examining the recursive path-finding function, ensuring that it explores all possible paths without missing any.
- The use of memoization with dynamic programming ensures that each path is computed only once, preventing redundant calculations.

Identification of Cut Vertices:

- We need to show that the algorithm correctly identifies any common vertices (cut vertices) present in all paths between the source and destination.
- This can be confirmed by checking that the algorithm correctly intersects the set of vertices in all paths to find the common ones.

Detailed Proof:

Topological Sorting Correctness:

- Kahn's algorithm guarantees that the topological sorting correctly orders the vertices of the DAG such that all edges point forward.
- The algorithm iteratively removes vertices with zero in-degree, ensuring that all predecessors of a vertex are visited before it is added to the sorted list.
- Therefore, the topological sorting process ensures that the traversal of the graph respects the direction of edges.

Path Finding Correctness:

- The path-finding function recursively explores all possible paths from the source to the

destination.

- At each step, the function explores neighboring vertices and continues the search until the destination is reached.
- Memoization ensures that previously explored paths are reused, preventing redundant computations.
- By exhaustively exploring all possible paths, the algorithm guarantees that no valid path is missed.

Identification of Cut Vertices:

- After finding all paths, the algorithm identifies common vertices (cut vertices) present in all paths between the source and destination.
- This is achieved by intersecting the sets of vertices in all paths, ensuring that only vertices traversed in every path are identified as cut vertices.
- Therefore, the algorithm correctly identifies the vertices necessary to connect the source and destination nodes in all paths.

Conclusion:

The provided algorithm demonstrates correctness by accurately finding all possible paths between a given source and destination node in a directed acyclic graph (DAG). Additionally, it correctly identifies any common vertices (cut vertices) present in all paths. The use of topological sorting and dynamic programming ensures the efficiency and accuracy of the algorithm, making it a reliable solution for the specified problem.

Naive Solution

A naive solution for this following which we thought earlier was to first check if there exist any path between s and t if there exists we remove each nodes from that path and check if there exist a path between s and t if we remove that node from the graph but in worst case it will take O(V[V+E]) ,space complexity for that(O(V+E)) checking DFS after removing each vertices whether it is possible to move after removing that edge or not in worst case we will remove V edges and for each v we will have DFS

code for that

```
from collections import defaultdict, deque

graph = {
   's': ['a', 'b'],
   'a': ['b'],
   'b': ['c','d','e','f'],
   'c': ['d','e','f'],
   'd':['f','e'],
   'e':['f'],
   'f':['h','g','i'],
```

```
    'g':['i'],
    'h':['i','g'],
    'i':['j','t'],
    'j':['t'],
    't':[]
}


s = 's'
t = 't'

V = graph.keys()
n = len(V)
flag = False


def dfs(node , rem , temp , visited):
    visited[node] = 1
    for v in graph[node]:
        if v != rem and visited[v] == -1:
            dfs(v,rem , temp , visited)

    temp.append(node)


def bfs(start,destination):
    queue = deque()

    visited = dict()
    for node in V:
        visited[node] = -1

    visited[start] = 1
    queue.append(start)

    while queue:

        node = queue.popleft()

        for v in graph[node]:
            if visited[v] == -1:
```

```
                visited[v] = 1
                queue.append(v)

        if node == destination:
            return True

    return False



cut_vertex = []


if __name__ =='__main__':

    path_exist = bfs(s , t)
    if not path_exist:
        print("NO CUT VERTEX")

    else:
        for vertex in V:
            if vertex not in ['s','t']:
                visited = dict()
                for v in V:
                    if v != vertex:
                        visited[v] = -1


                temp = []
                dfs('s' , vertex , temp , visited)
                if 't' not in temp:
                    cut_vertex.append(vertex)

        print(cut_vertex)
```