Depth-First Search & Its Applications in Bioinformatics

Bolisetty Sujith
Department of Computer Science and Engineering
Amrita Vishwa Vidyapeetham
Amritapuri, Kerala, India
amenu4aie20018@am.students.amrita.edu

Guggilam Sai Prabhat
Department of Computer Science and Engineering
Amrita Vishwa Vidyapeetham
Amritapuri, Kerala, India
amenu4aie20032@am.students.amrita.edu

Raja Pavan Karthik
Department of Computer Science and Engineering
Amrita Vishwa Vidyapeetham
Amritapuri, Kerala, India
amenu4aie20060@am.students.amrita.edu

Suravarapu Ankith
Department of Computer Science and Engineering
Amrita Vishwa Vidyapeetham
Amritapuri, Kerala, India
amenu4aie20070@am.students.amrita.edu

ABSTRACT

Depth-First Search (DFS) is one of the searching algorithms using data structure Stack when it reaches a node or vertex which is connected in a graph. And the main aim of the DFS is to find each node. And worked on the applications of the DFS like Path Finding, Topological Sort, and connected components. Pathfinding is used to determine the shortest path between two places. Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering. A connected component or simply component of an undirected graph is a sub graph in which each pair of nodes is connected with each other via a path.

KEYWORDS

Depth-First Search (DFS), Path, Cycle, Node, Stack, Queue, Algorithm

INTRODUCTION

The algorithm for exploring a network or tree data structure is called depth-first search (DFS). The algorithm starts at the root (top) node of a tree and proceeds as far down a given branch (path) as it can, then backtracks until it discovers an unexplored path, which it then investigates. Backtracking is traveling backward on the same path to find nodes to traverse when you are moving forward and there are no more nodes along the current path. On the current path, all nodes will be visited until all of the unvisited nodes have been travelled, at which point the new path will be chosen. This is repeated by the algorithm until the entire graph has been studied. Graphs can be used to solve a lot of problems in computer science. Graph challenges include, for example, evaluating networks, mapping routes, scheduling, and identifying spanning trees.

Graph-search algorithms like depth-first search are beneficial for analyzing these difficulties.

Stacks can be used to implement DFS's recursive nature. The fundamental concept is as follows: Choose a starting node and stack all of the nodes that are near to it. To visit the next node, pop a node from the stack and push all of its nearby nodes into a stack. Carry on in this manner until the stack is depleted. Make sure, however, that the nodes you visit are highlighted. You will not be able to access the same node more than once as a result of this. You may find yourself in an infinite loop if you do not mark the nodes that [1] The paper discussed the implementation and analysis of the DFS algorithm for finding the longest path. And written by Sheila Eka Putr they discussed the longest path, Depth-First Search (DFS). It is one of the searching algorithms using data structure Stack when it reaches a node or vertex which is connected in a graph, Hamiltonian cycle. In that paper, they implemented a solution in determining the longest path in a graph based on the length of the graph. And they chose the one best algorithm from DFS and BFS from them. And the Longest path problem can be efficiently solved by the DFS algorithm only. And in that paper discussed the Method of Optimum Solution dynamic programming is also an important role in optimization problems. And the most important one, the Longest Path Problem, is developed in the paper, and in this problem finding the optimum solution, both maximization, and minimization. And did an experiment with the f programming language of Java and implemented a graph with an undirected weighted graph. And they used the DFS also to find the longest path. Their main aim is to show the DFS Algorithm to search the longest trajectory effectively and the DFS algorithm is an algorithm that is good enough for searching for an optimum solution of the longest path problem.

are visited and you visit the same node more than once. Depth-first searches are frequently incorporated into more complicated algorithms as subroutines. For example, the Hopcroft–Karp matching method incorporates a DFS into its algorithm to aid in the discovery of a match in a graph. The travelling-salesman issue and the Ford-Fulkerson method both use DFS in tree-traversal algorithms, also known as tree searches.

LITERATURE REVIEW

[2] In this paper, they discussed the Depth-first search, protein structure prediction, genetic algorithm, lattice model. And Solving the protein structure prediction problem using DFS. In this paper, they explained the HP Lattice Model, which is based on hydrophobicity where it divides the amino acids into two different beads – hydrophobic (H) and hydrophilic. And in the complexity of the lattice model in this model, it uses shorter sequences of the DNA. The Nondeterministic Conformational Search Algorithms are also explained for solving PSP using the lattice model. The concepts of GAs are also widely adapted within these DFS algorithms and it has more effective by combining it with DFS which can have a significant positive impact on solving the PSP problem. over and again bombing hybrid with a clogged however potential sub confirmation can be permitted a predetermined number of pathways for conceivable up-and-comer hybrid partners acquired by utilizing DFS assuming there exists somewhere around one way. And they did an experiment to empirically verify our hypothesis that combining DFS with crossover will be advantageous. They conclude with the depth-first search strategy at a low rate has been applied in combination with a powerful crossover operation and in their work will be

directed to exploring the biological significance and relevance of this novel approach.

[3] They discussed Graph drawing algorithms are often used for visualizing relational information and used the Algorithm to find the distance and selective pivot nodes and to check the distance of the max distance and distance table and at last layouts. And they did a protein interaction with networks in that a large number of edges and nodes of their complexity of the protein in the interaction with the network to reduce the readability of the network due to cluttered edges and nodes. And they did a common problem with many force-directed layout algorithms in there but it is very slow and also a graphical representation of protein interactions has proven to be much easier to understand than a long list of interacting proteins and naive implementation of a graph drawing algorithm encounters real difficulties when drawing large-scale graphs such as protein interaction networks. At last, they concluded that in the graph drawing, protein interaction data is a major bioinformatics challenge, because it yields a large and complicated graph with an excessive number of edge crossings and it produces a disconnected graph with many connected components and the graph contains nodes of a wide range of degrees.

METHODS

Applications in Bioinformatics

Path Finding

Path-finding is the process of determining the shortest path between two places. The topic has long been studied by mathematicians and computer scientists alike, to the point where it has become its own field of study. Dijkstra's technique for pathfinding on weighted paths, where each path takes a set amount of time, or weight, to traverse, is frequently used in this subject. However, for the purpose of simplicity, we will only consider unweighted paths in this work, assuming that travelling each path takes the same amount of time. We will investigate novel methods for path-finding on an unweighted path, however, Dijkstra's technique will still apply.

DFS begins at node A and works its way up the stack. It verifies that node A is not node F, observes that nodes B and C are available, and chooses one of the two at random. Because actual randomness does not exist in computing, let us assume that the algorithm checks nodes on the upper side of the network first. (Because our graph is organized in a tree form, this will work in this case.) As a result, DFS would choose node B for evaluation. It then repeats this procedure until it reaches node H when it discovers that no node is accessible and returns to node D by removing H from the stack and making D the top. It starts with node I, the last available node from node D, and then moves backward until it reaches node B. It then travels via nodes E and J until it reaches node F, the desired ending node, where it comes to a halt. As a result, DFS would return the path by popping off the stack completely and reversing the order, yielding the path A→B→E→J→F.

Topological Sort

A directed graph with n vertices and m edges is supplied to you. You must number the vertices in such a way that every edge leads from the vertex with the lower number to the vertex with the higher number. To put it another way, you're looking for a permutation of the vertices (topological order) that corresponds to the order specified by all of the graph's edges. Non-unique topological order is possible (for example, if the graph is empty; or if there exist three vertices a, b, c for which there exist paths from a to b and from a to c but not paths from b to c or from c to

b).
If the graph comprises cycles, topological order may not exist at all (because there is a contradiction: there is a path from a to b and vice versa). The following is an example of a topological sorting problem. There are a total of n variables that have unknown values. We know that one variable is less than the other for some variables. Check whether these conditions are mutually exclusive and if they aren't, print the variables in ascending order (if several answers are possible, output any of them). It should be obvious that this is the same problem as determining the topological order of a graph with n vertices. We'll use a depth-first search to tackle this problem.

Assume the graph is acyclic, which means there is a solution. What is the purpose of a depth-first search? It tries to run along all edges outgoing from v when it starts from some vertex v. It skips the edges whose opposite ends have already been visited, and instead runs along the rest of the edges, beginning at their ends. As a result, by the time dfs(v) is finished, the search has already visited all vertices that are reachable from v either directly (through one edge) or indirectly. As a result, if we add vertex v to the beginning of a list at the time of exit from dfs(v), the list will eventually store a topological ordering of all vertices.

These reasons can also be expressed in terms of the time when the DFS function is exited. The moment a, when dfs(v) completed its task, is called the exit time for vertex v. (the times can be numbered from 1 to n). It's obvious that the departure time of any vertex v is always greater than the exit time of every vertex reachable from it (since they were visited either before or during the call dfs(v)). As a result, sorting vertices in descending order of their exit times achieves the necessary topological ordering.

Connected Components

A connected component of an undirected graph[4] is a subgraph in which each pair of nodes is connected with each other via a path[5]. A set of nodes forms a connected component in an undirected graph if any node from the set of nodes can reach any other node by traversing edges. The main point here is reachability. In connected components, all the nodes are always reachable from each other.

There are also algorithms used to find Connected Components of a directed graph, some of them are Kosaraju's algorithm for strongly Connected Components[6], and Tarjan's Algorithm to find Strongly Connected Components[7], but as we are discussing DFS algorithms in this paper we will be discussing Connected Components of an undirected graph.

By using a DFS algorithm starting from every unvisited vertex, we get all strongly connected components.

Algorithm Connected Components:

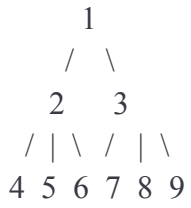
```
1) Initialize all vertices as not
visited.
2) Do the following for every vertex 'v'.
       (a) If 'v' is not visited before,
call DFSUtil(v)
       (b) Print newline character

DFSUtil(v)
1) Mark 'v' as visited.
2) Print 'v'
3) Do the following for every adjacent
'u' of 'v'.
     If 'u' is not visited, then
recursively call DFSUtil(u)
```

RESULTS AND DISCUSSION

Path Finding

Input: N = 10



Pair = {4, 8}

Output: 4→2→1→3→ 8

Given a directed graph with 's' as the source vertex and 'd' as the destination vertex, Take a look at the directed graph below. Let's say the s is 2 and the d is 3. There are three various courses to choose from, ranging from 2 to 3. All pathways from the letter 's' to the letter 'd' [16] have been printed using dfs.

Topological sorting

Input:

```
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 1 0 0
0 1 0 0 0 0
1 1 0 0 0 0
1 0 1 0 0 0
```

DFS has been modified to create a topological sort. Running DFS on a whole network and adding each node to the global ordering of nodes, but only after all of a node's children have been visited, is all it takes to perform topological sort. This ensures that parent nodes are ordered before their children, and that edges are ordered in the forward direction. From [17] we have obtained the topological sort using dfs algorithm.

Output:

```
Nodes after topological sorted
order: 5 4 2 3 1 0
```

Connected Components

Input: A simple graph with n ≤ 10^3 vertices in the edge list format.

```
12 13
1 2
1 5
5 9
5 10
9 10
3 4
3 7
3 8
4 8
7 11
8 11
11 12
8 12
```

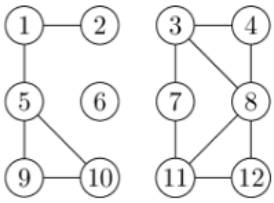


Fig1: The graph from the data set

Output: The number of connected components in the graph.

```
[[1, 2, 5, 9, 10], [3, 4, 8, 11,
7, 12], [6]]
results: 3
```

Given an undirected graph with vertices and edges as mentioned above, as we see from the above Fig1, there are three connected components. By running the code[18] in the

appendix, we got the accurate no of connected components.

CONCLUSION

Depth-first search is an interesting algorithm, and as you might suspect, it is used for transverse the graph or tree. In this paper, you can see the applications Topological Sort , Path finding , Connected components in Bioinformatics. Topological sort is used for partially defined orders and path finding is the technique of discovering the shortest path between two points and finding connected components of an undirected graph using an adjacency matrix which is more efficient than other algorithms that use the whole graph in their algorithms.

REFERENCES

[1] Implementation and Analysis of Depth-First Search (DFS) Algorithm for Finding The Longest Path

[2] DFS Based Partial Pathways in GA for Protein Structure Prediction

[3] Kyungsook Han, Byong-Hyon Ju, A fast layout algorithm for protein interaction networks, Bioinformatics, Volume 19, Issue 15, 12 October 2003, Pages 1882–1888,

[4] Undirected Graph CS241: Data Structures & Algorithms II. (n.d.). Retrieved January 26, 2022

[5] Wikipedia contributors. (2021, October 29). Path (graph theory). Wikipedia.

[6] Wikipedia contributors. (2021, August 12). Kosaraju's algorithm. Wikipedia.

[7] Wikipedia contributors. (2022, January 13). Tarjan's strongly connected components algorithm. Wikipedia.

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

[9] Adamchik, Victor S. "Stacks and Queues." Carnegie Mellon University, 2009.

[10] Topological sort algorithm for dag. Techie Delight. (2021, November 5).

[11] Kumar, A. (2020, December 29). Applications of Breadth First Search and Depth First Search. TutorialCup.

[12] Jena, S. (2019, April 18). Topological Sorting using Depth First Search (DFS). OpenGenus IQ: Computing Expertise & Legacy.

[13] Topological Sorting Using Depth First Search (DFS)." OpenGenus IQ: Computing Expertise & Legacy, 18 Apr. 2019

[14] Topological Sort | CodePath Android Cliffnotes. (n.d.). Code Path.

[15] Depth-First Search (DFS) | Brilliant Math & Science Wiki. (n.d.). Brilliant Math & Science Wiki.

Appendix:

[16] Code - Path Finding

```
# Python3 implementation of the above approach
class PathFinding:

   """An utility function to add an edge in an
   undirected graph."""
   def addEdge(self,x, y):
       self.v[x].append(y)
       self.v[y].append(x)

   """A function to print the path between
   the given pair of nodes."""
   def printPath(self,stack):
       for i in range(len(stack) - 1):
           print(stack[i], end = " -> ")
       print(stack[-1])

   """An utility function to do
   DFS of graph recursively
   from a given vertex x."""
   def DFS(self,vis, x, y, stack):
       stack.append(x)
       if (x == y):

           """print the path and return on
           reaching the destination node"""
           self.printPath(stack)
           return
       vis[x] = True

       # if backtracking is taking place

       if (len(self.v[x]) > 0):
           for j in self.v[x]:
               # if the node is not visited
               if (vis[j] == False):
                   self.DFS(vis, j, y, stack)

       del stack[-1]

   """A utility function to initialise
   visited for the node and call
   DFS function for a given vertex x."""
   def DFSCall(self,x, y, n, stack):

       # visited array
```

```
        vis = [0 for i in range(n + 1)]

        #memset(vis, false, sizeof(vis))

        # DFS function call
        self.DFS(vis, x, y, stack)

    def __init__(self) -> None:
        self.main()

    def main(self):
        # Driver Code
        self.v = [[] for i in range(100)]
        n = 10
        stack = []

        # Vertex numbers should be from 1 to 9.
        self.addEdge(1, 2)
        self.addEdge(1, 3)
        self.addEdge(2, 4)
        self.addEdge(2, 5)
        self.addEdge(2, 6)
        self.addEdge(3, 7)
        self.addEdge(3, 8)
        self.addEdge(3, 9)

        # Function Call
        self.DFSCall(4, 8, n, stack)

PPF = PathFinding()
```

[17] Code - Topological sorting

```
# Python program to print topological sorting of a DAG
from collections import defaultdict

# Class to represent a graph
class Graph:
	def __init__(self, vertices):
		self.graph = defaultdict(list) # dictionary containing adjacency List
		self.V = vertices # No. of vertices

	# function to add an edge to graph
	def addEdge(self, u, v):
		self.graph[u].append(v)

	# A recursive function used by topologicalSort
	def topologicalSortUtil(self, v, visited, stack):

		# Mark the current node as visited.
		visited[v] = True
		# Recur for all the vertices adjacent to this vertex
		for i in self.graph[v]:
			if visited[i] == False:
				self.topologicalSortUtil(i, visited, stack)
		# Push current vertex to stack which stores result
		stack.append(v)

	# The function to do Topological Sort. It uses recursive
	# topologicalSortUtil()
	def topologicalSort(self):
		# Mark all the vertices as not visited
		visited = [False]*self.V
		stack = []

		# Call the recursive helper function to store Topological
		# Sort starting from all vertices one by one
		for i in range(self.V):
			if visited[i] == False:
				self.topologicalSortUtil(i, visited, stack)

		# Print contents of the stack
		print(stack[::-1]) # return list in reverse order


# Driver Code
g = Graph(6)
g.addEdge(5, 2)
g.addEdge(5, 0)
g.addEdge(4, 0)
g.addEdge(4, 1)
g.addEdge(2, 3)
g.addEdge(3, 1)
```

```
print ("Following is a Topological Sort of the given graph")

# Function Call
g.topologicalSort()
```

[18] Code - Connected Components

```
""" Python program to print connected
components in an undirected graph """

class Connected_components:

   def dfs(self,cc, v, visited, graph):
       visited[v] = True
       cc.append(v)
       for i in graph[v]:
           if visited[i] == False:
               cc = self.dfs(cc, i, visited,graph)
       return cc

   def connectedComp(self,vertice, graph):
       visited = [False for i in range(vertice+1)]
       connected_components = []
       for v in range(1, int(vertice)+1):
           if visited[v] == False:
               cc = []
               connected_components.append(self.dfs(cc, v, visited, graph))

       print(connected_components)
       print("results: {}".format(len(connected_components)))

   def main(self,f):

       vertice, edge = map(int, f.readline().strip().split(" "))
       graph = {i+1:[] for i in range(vertice)}

       for line in f:
           l = list(map(int, line.strip().split(" ")))
           graph[l[0]].append(l[1])
           graph[l[1]].append(l[0])
       f.close()

       self.connectedComp(vertice, graph)

   def __init__(self):
       data = "CC_data.txt"
       f = open(data, "r")
       self.main(f)

cc = Connected_components()
```