BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

CS-F211: Data Structures and Algorithms

Lab 12: Graphs, Traversals and Applications

Introduction

Welcome to the 12th and final lab sheet of Data Structures and Algorithms!!! After going through different kinds of trees over the last few weeks, today we shall proceed to a generalisation of trees in the form of graphs. Mathematically, graphs are simply collections of nodes connected by edges. While trees had certain restrictions in the way nodes can be connected to each other, as we shall see, graphs are much freer. We shall see different kinds of graphs, how they can be represented digitally and the different ways of traversing along a graph. [1]

Topics to be covered in this labsheet

- Introduction to Graph Representations
    - Adjacency Matrix
    - Adjacency List
- Graph Traversals
    - Breadth-First Search
    - Depth-First Search
- Connectivity

[1] In this lab sheet, we have provided files input0.txt to input4.txt representing different graphs as described in Task 1. You can further make use of the tool - https://csacademy.com/app/graph_editor/ to visualise the graphs. (Just paste the edges from the files into the Graph Data field.)

Introduction to Graphs Representations

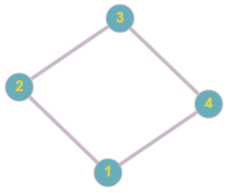
A graph is defined as a collection of vertices (V) and a set of edges (E) between these vertices, **ie G = (V, E)**. Based on whether these edges are unidirectional or bidirectional, we can have **undirected** graphs or **directed** graphs **(**or **digraphs)**. For a graph of small size, we can perhaps remember the nodes and the presence or absence of edges. Or similar to how we did for trees, create a structure for each node with pointers to all of its neighbours. However, while the trees we saw were rooted and it made sense to access the entire tree through a special node - the root, in a graph in general, there need not always be a source. So we may need to look up the edge between any two given nodes. We need a way to store this information in an organised and efficient manner, given which we are able to reconstruct the graph. That is, the representation must uniquely describe the graph for the given context. There are two major ways we use to store this information - Adjacency Matrix and Adjacency Lists

Adjacency Matrix

In an adjacency matrix representation of an unweighted graph G= (V, E), we assume some ordering (or numbering) of the vertices indexed from 0, 1, 2, …, |V| - 1. Then the adjacency matrix is a |V| x |V| matrix A = (a_{ij}) such that:

$$a_{ij} = 1 \; if \; (i, j) \in E$$

$$a_{ij} = 0 \; otherwise$$



0	1	0	1
1	0	1	0
0	1	0	1
1	0	1	0

Figure 1: Undirected graph and its adjacency matrix representation

In case of weighted graphs, this can easily be adapted where we store the edge weight in a_{ij} instead of simply 0 or 1. In this case, there might be ambiguity regarding whether 0 represents the absence of an edge or an edge with weight 0. Thus, it might make sense to store a special value denoting the absence of edges. However, in most applications, we can end up using 0 or INT_MAX based on the context. (Using INT_MAX when minimising costs and 0 when maximising with positive weights).

In the case of undirected graphs, we simply have the following property satisfied in the graph:

$$a_{ij} = a_{ji}$$

Thus, $A^T = A$ and the matrix is symmetric.

The advantage of an adjacency matrix is that because the edges are stored in a matrix, we can get any edge in `O(1)` time.

However, consider the following example:

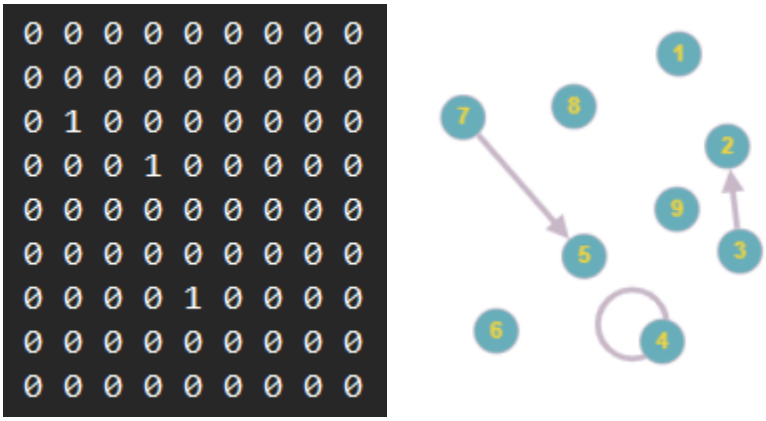


Figure 2: Adjacency Matrix and the corresponding graph for a sparse directed graph

This is a 9x9 matrix with three 1s and 78 0s. While there are only three edges, we still end up having a space complexity of $O(V^2)$, independent of the number of edges in memory. While this is efficient for dense graphs where the number of edges itself is in the order of $O(V^2)$, it is not very efficient in sparse graphs.

Adjacency Lists

The adjacency-list representation of a graph G = (V, E) consists of an array of |V| lists, one for each vertex in V. For each vertex u, the list for u contains all the vertices v such that there is an edge (u, v) in E.

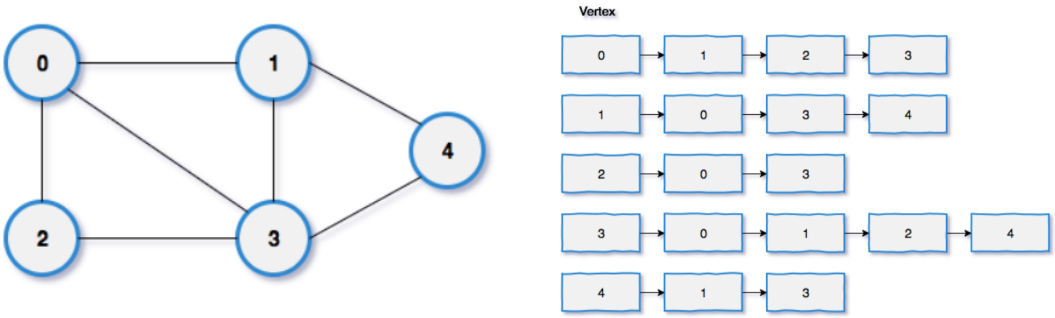


Figure 3: Adjacency List for an undirected graph

As seen in figure 3, it is convenient to store this as an array of linked lists. Each index of the array stores a linked list corresponding to each vertex. In order to check the presence of edges, one has to traverse the entire list of the node from which the node is incident. The advantage of an adjacency list is that the amount of memory required is $O(E)$ instead of the $O(V^2)$ required in the case of an adjacency matrix. However, the disadvantage is that the lookup for edges is now on the worst-case O(V) instead of O(1) as the adjacency list can have a length of up to |V|. We prefer adjacency lists when the graphs are sparse.

In an undirected graph, as each edge (u,v) would appear in the list for both u and v, the sum of lengths of all the lists is equal to 2|E|. In the case of a directed graph, it would appear in only one list and thus, the sum of the lengths of all the lists is |E|. In the linked list node, we can either store the index of the vertex or a pointer to it based on what attributes are present with the nodes. (Storing pointers is similar to our tree representation.)

In the linked list node representing an edge, we can also add a weight field that denotes the weight of the edge in a weighted graph. Here there is no ambiguity for the weight to be set for missing edges that we saw in adjacency matrices as those edges won't have any entry in the adjacency list.

Task 1: You are provided with graphs in text files *input0.txt*, *input1.txt* and *input2.txt*. The files have the description in the following format:
The first line contains the number of vertices |V|.
The second line contains the number of edges |E|.
The next |E| lines contain the edges in the form of "u v" where u and v are the indices representing the vertices (0-indexed).

The graph is directed and unweighted. Write a program that reads the graph from the file and prints the adjacency matrix and the adjacency list of the graph. Here we are defining a graph as a structure with the following fields:
`int V`: number of vertices
`int E`: number of edges
`int **adjacency_matrix`: a 2D array of size `V` x `V`
`int **adjacency_list`: an array of size `V`, each element of which is a linked list

You are given incomplete code in the file t1.c. Understand the code and complete it as per the instructions provided in the comments.

Example output for the file *input1.txt* is as follows:

```
V: 5, E: 10
Adjacency Matrix:
0 0 1 0 0
1 1 1 1 0
0 0 0 0 0
0 1 0 1 0
1 1 0 0 1
Adjacency List:
0: 2
1: 0 3 1 2
2:
3: 3 1
4: 4 0 1
```

File input1.txt is sparse while the file input2.txt is dense. You can appreciate the difference in the memory requirements of adjacency matrices and adjacency lists by observing the outputs of the three files.

Home Exercise 1: Using each of the two representations of the graph in *t1.c*, calculate the in-degree and out-degree of all the nodes and verify the following property holds:

$$\sum_{v \in V} Indegree(v) = \sum_{v \in V} Outdegree(v) = |E|$$

Searching

Given a graph, G = (V, E), we now would want to explore the nodes. Given the edges, we might want to study certain properties for the graph based on the configuration of the edges based on the configuration of the nodes and edges. For this, we start from a given node and try to explore multiple paths. The simplest problem is to find whether one can reach a certain node target node t starting from source node s. This is called **searching**. The way to **search** for a node is to **traverse** the entire set of nodes that are reachable from s until we reach the required destination t or we exhaust the set. Thus, we shall be using the terms searching and traversing interchangeably here as the meaning would be clear from the context.

The are different ways in which we might decide which edge to explore and discover new nodes of a graph. One can non-deterministically choose any edge, however, this may not be complete and you might end up going in an infinite loop. Thus we need to be able to systematically traverse the graph to search for a given node starting from a source u.

We shall study two of the simplest and most fundamental ways in which you can traverse a graph - Breadth-First Search and Depth-First Search.

Breadth-First Search (BFS)

Breadth-First Search is one of the simplest algorithms for searching a graph. It has a wide variety of applications and is a tool that would continuously be used while dealing with graphs or even some drastically diverse problems all of which can be modelled as some sort of search in a graph.

Given a graph G = (V, E) and a source vertex s, BFS computes the least number of edges required to reach every reachable vertex from s. It also produces a "breadth-first tree" rooted at s containing all the reachable vertices and the shortest (in the number of hops) paths to them. The algorithm works on both directed and undirected graphs and in the case of unweighted graphs, also yields the shortest path from s to t.

BFS is called so because we discover nodes in a monotonically non-decreasing order of their distances from the source. Another way of looking at it is we 'parallelly' start from every possible path from the source and at each iteration, take one step at every path.

While there are many ways of looking at this problem, we shall follow the technique described in the textbook *Introduction to Algorithms* of colouring the vertices white, grey or black. The basic principle is that all vertices are initially **undiscovered** and thus start out **white**. Each time a vertex is **discovered** we colour it **grey**. Here discovered means that the vertex is seen for the first time from any given path. And each time a vertex is **explored**, we colour it black. Explored means that every neighbour of a vertex has been discovered.

The major steps of the BFS traversal algorithm are as follows:

```
BFS(G, s):
   1. Start off with all nodes being white. Initialize an empty
      queue.
   2. Mark the source s as grey and enqueue it
   3. While the queue is not empty:
         a. Dequeue a node u
         b. For each white node v adjacent to u:
             i.   Mark v as grey
            ii.   Enqueue v
         c. Mark u as black
```

Let us now look at the instance of BFS on a graph:

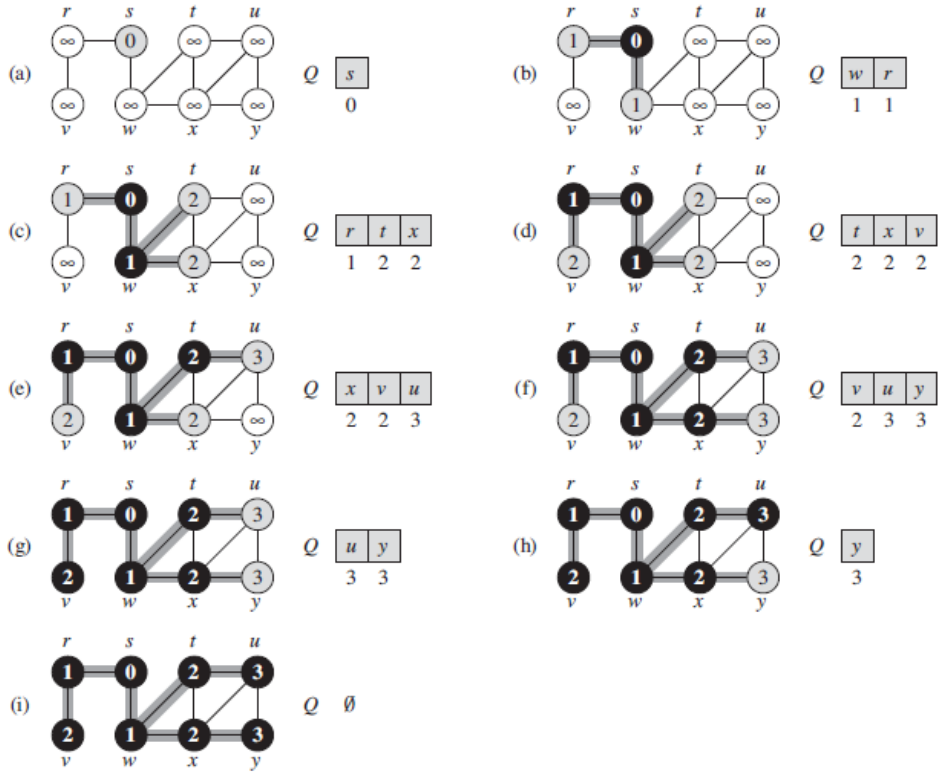


Figure 4: Illustration of BFS

Initially, in Figure 4.a, the root s is marked grey and added to the queue. Then it is explored (marked black) where its neighbours w and r are marked grey and added to the queue. Then we dequeue w in Figure 4.b, and add its undiscovered neighbours t and x to the queue. Note that since the neighbour s was already marked black, we did not add it to the queue again. We only add discover white nodes, paint them grey and add them to the queue. Similarly, we proceed as per the algorithm and traverse the entire graph.

Task 2: Complete the implementation of BFS traversal following the instructions given in the file *t2.c*. You can test it out on the same input files given in Task 1. Here we have modified the graph structure defined in task 1. It now only stores the adjacency matrix. We have also defined a structure graph_node that stores the attributes of each node. The graph structure now has an array of 'graph_node's called vertices as well. For the purpose of this example, we assume that the data of a node is the same as its index. However, it can be different and can hold a lot of attributes belonging to a node as required by different algorithms. Follow the instructions given in the file and use the structures defined to complete the implementation.

Home Exercise 2: Modify the implementation of BFS in Task 2 to also store the distance of each node from the source. Add the distance attribute to the graph_node struct and initialize it to 2 * |V|. Update it in the BFS algorithm such that it stores the correct distance from the source.

Home Exercise 3: The implementation of BFS that we have seen till now was the BFS traversal. We haven't actually searched in our implementation of Breadth-First-**Search** yet. Add a function to *t2.c* that takes a directed unweighted graph `G`, a source `s` and a target `t` and prints the path from the source to the target. [**Hint**: You might want to add a `predecessor` attribute in the graph node]

Depth-First Search (DFS)

Depth-First Search works on the principle of completely exploring one path to a leaf (node from which we cannot discover any new node) before backtracking and exploring the other paths. Suppose we are at node s with neighbours u and v. In the case of BFS, we would first discover unfollowed by v and then the neighbours of u, then the neighbours of v, … In contrast in DFS, we would first discover all the nodes along paths from s via u before discovering v and the nodes reachable via v. It is called **depth-first** because we proceed as **deep** as we can along any given path before backtracking and exploring other paths.

Interestingly, all we need to modify our BFS implementation to a DFS implementation is replacing the queue with a stack. This ensures that the most recently discovered node is expanded first resulting in a depth-first traversal of the graph.

Thus, the algorithm for DFS traversal is described as

```
DFS(G, s):
   1. Start off with all nodes being white. Initialise an empty
      stack.
   2. Mark the source 's' as grey and push it
   3. While the stack is not empty:
         a. Pop a node u
         b. For each white node v adjacent to u:
             i.   Mark v as grey
            ii.   Push v
         c. Mark u as black
```

Task 3: Implement DFS in the file *t2.c* in the same way you implemented BFS, replacing the queue and queue operations with a stack and the corresponding stack operations. Run both BFS and DFS on the input files. (In case you are running both of them in the same run, ensure that you reset the colours of all nodes to white before calling another traversal algorithm on the same graph instance). An example run on the file *input3.txt* is shown below.

The file *input2.txt* represents the following graph:

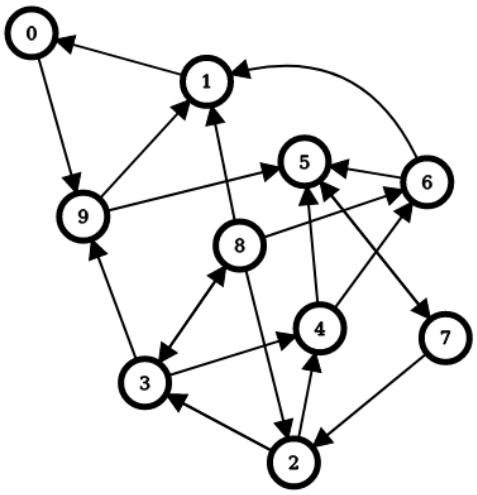


Figure 5: Graph represented by the file *input3.txt*

BFS: 0 9 1 5 7 2 3 4 8 6 **DFS: 0 9 5 7 2 3 8 6 4 1**

Task 4: Your reference book, *Introduction to Algorithms 3rd Edition by CLRS*, has a slightly different implementation of DFS. Look at the implementation on page 604 in section 22.3. Here no explicit stack is used. They have instead used recursion to accomplish DFS. Also, they don't have an explicit start node. They iteratively complete the traversal from undiscovered (white) nodes until all nodes are discovered. Further, they also maintained a global timer variable and updated two timestamp attributes for each vertex - discovery time (d) denoting when the node was discovered and finish time (f) denoting when the exploration of the node finished. While the discovery and finish times are not explicitly required, they are useful in many applications of DFS.

Implement the recursive version of DFS along with timestamps, predecessors and repeated starts as described in CLRS by implementing the following functions:

```
void recDFS(Graph *G);
void recDFSVisit(Graph *G, int u);
```

Is the traversal order the same as the older version of DFS? Why or why not?

Home Exercise 3: Now, let us consider an application of the above implementation. A directed acyclic graph (or DAG) is a graph with no directed cycles. A topological sort of a DAG is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG. Topological sorting is useful in scheduling jobs from the given dependencies among jobs. For example, in a computer science curriculum, there may be dependencies between courses, such as “Data Structures” course must be completed before the “Algorithms” course. In this case, a valid topological sorting of courses is “Data Structures” followed by “Algorithms”.

You are given a comma-separated file *CDCs.csv* that describes the prerequisites of CDCs of BE CS at BITS. The file has the following format:
The first line contains the number of CDCs (V)
The second line contains the number of prerequisites (E)
The next V lines contain the CDC code and CDC name
The next E lines contain prerequisite CDC code and CDC code

You have to first create vertices that store all the information about the courses and store it in such a way that you would be able to efficiently retrieve the data while printing the topological order. (Define a mapping between course code and index.) You have to print the topological sort of the CDCs. If there is no topological sort, print “No topological sort exists”.

Connectivity

Let us now consider the notion of connectivity. We shall be focussing on digraphs for now. In the case of undirected graphs as well, the same ideas work with some obvious relaxations. Formally, a strongly connected component of a directed graph is a maximal subgraph in which every pair of vertices is reachable from each other. A directed graph is said to be strongly connected if it has only one strongly connected component.

In some cases, the requirement of strongly connected components is too strict. For such cases, we can define a notion of weakly connected components as well. A weakly connected component of a directed graph is a maximal subgraph in which every pair of vertices is reachable from each other in the underlying undirected graph. In other words, it is a maximal 'connected' subgraph of the underlying undirected graph.

Both kinds of connectivity checks can be modelled by running modified versions of depth-first search.

For computing strongly connected components of a graph, we can just run the version of DFS described in Task 4 on both a graph and its transpose (the transpose of a graph is defined as the graph obtained by reversing all edges). We first run recursive DFS with timestamps and restarts on the graph. Then we sort the vertices in descending order of the finish times of this DFS traversal. Then we obtain the transpose graph and run the outer loop of recursive DFS with time stamps on the reversed graph. The nodes discovered in each run of DFS here correspond to a strongly connected component.

The implementation for the same is given below:

```
int strongly_connected_components(Graph *G)
{
    time = 0;
    printf("Computing strongly connected components\n");
    printf("Running DFS on the original graph\n");
    recDFS(G);
    printf("\n");
    graph_node descending_f_indices[G->V];
    for (int i = 0; i < G->V; i++)
    {
        descending_f_indices[i].data = i;
        descending_f_indices[i].f = G->vertices[i].f;
```

```
    }
    // Sort the vertices in descending order of f
    for (int i = 0; i < G->V; i++)
    {
        for (int j = i + 1; j < G->V; j++)
        {
            if (descending_f_indices[i].f < descending_f_indices[j].f)
            {
                graph_node temp = descending_f_indices[i];
                descending_f_indices[i] = descending_f_indices[j];
                descending_f_indices[j] = temp;
            }
        }
    }

    Graph *Gt = get_transpose(G);
    for (int i = 0; i < G->V; i++)
    {
        Gt->vertices[i].c = WHITE;
    }

    int num_scc = 0;
    time = 0;
    printf("\n");
    for (int i = 0; i < G->V; i++)
    {
        if (Gt->vertices[descending_f_indices[i].data].c == WHITE)
        {
            num_scc++;
            printf("SCC %d: ", num_scc);
            recDFSVisit(Gt, descending_f_indices[i].data);
            printf("\n");
        }
    }
    return num_scc;
}
```

Certain simple subroutines have been used here that can be easily implemented.

Let us try to build an intuition on why this works. Consider the extreme case of a Directed Acyclic Graph. Here, since there are no cycles, the number of strongly connected components

would be equal to the number of vertices |V| as no two vertices would be reachable from each other. In this case, when we run DFS on the reversed graph in the topologically sorted order (descending finish times), each run would have a single vertex. This would yield the number of strongly connected components as |V|. Now, for a general graph, if any run has multiple vertices, can we say that any pair of them can be reachable from each other? The answer surprisingly is yes. You are encouraged to go through the formal proof for the same in section 22.5 of *Introduction to Algorithms, 3rd Edition, by CLRS*.

Task 5: Understand the above implementation of strongly connected components and complete the implementations for the missing subroutines in the file *t5.c*. Now test out the above program on the file *input0.txt* and convince yourself that it works for a generic graph. The visual representation of the graph *input0.txt* is given below.

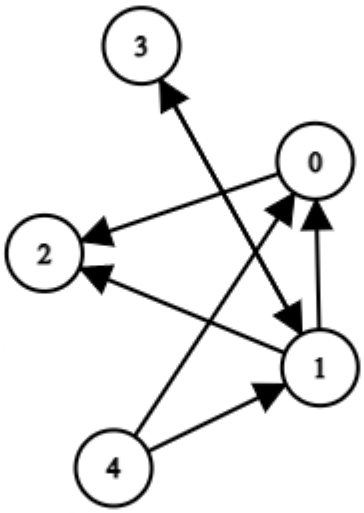


Figure 6: Graph represented by the file *input0.txt* (without the loops)

Now, write a similar simpler function for finding the number of weakly connected components. [Hint: You would need to run the traversal algorithm only once]. Find the number of strongly and weakly connected components in *input4.txt*.