SC1007 Data Structures and Algorithms **2023/24 Semester 2**

10: Graph Traversals

Nanyang Technological University

In this lecture, we will learn the most basic algorithms about graphs, for solving the problem of graph traversal. Two algorithms of graph traversal will be learned. One is Breadth First Search and the other is Depth First Search. Just like there are three orders of tree traversal, which are pre-order, in-order and post-order, the two algorithms discussed here explore a graph systematically in different orders.

10.1 Traversal of Graphs

Visiting every vertex of a graph in a systematic order is a typical and important algorithm for many graph problems. It can let us to do some processing at every vertex exactly once. Here we will introduce two traversal approaches, Breath First Searth (BFS) and Depth First Search (DFS).

10.2 Breath First Search (BFS)

BFS works similar to level-order traversal of a tree. Given a graph G = (V,E), and supposed we start the traversal from a source vertex x (in the example below, $x = 11$), the BFS algorithm systematically explores every unvisited vertex at the same distance/level from the source vertex and then moving on to visit vertices at the next level until all reachable vertices are visited. The BFS algorithm works on both directed and undirected graphs.

If a vertex has several unvisited neighbours, it would be equally correct to visit them in any order. Thus, the result may not be unique if we do not specify the visiting order.

If the **shortest path** from the source vertex to any vertex is defined as the path with the minimum number of edges, then BFS can be used to find the shortest path from the source to all vertices reachable from the source.

The tree built by BFS is called the **breadth first spanning tree** if the graph G is a connected graph. The tree is a set of $(|V| - 1)$ edges that connect all vertices of the graph.

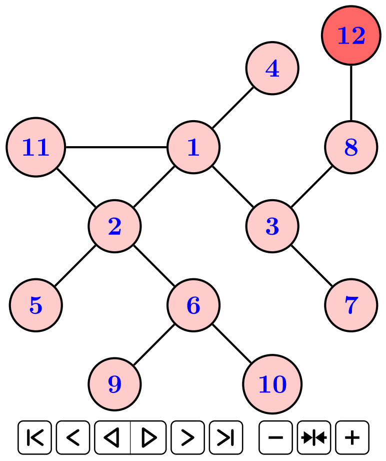


Figure 10.1: A BFS on a 12-node graph starting from node 11

To realise the BFS algorithm, we need a *queue* structure to store vertices to be visited next. Starting from the source vertex, we *enqueue* all unvisited neighbours of the visiting vertex into the *queue*. During visiting the vertex, we can take different action on it depending on the applications. Thus, BFS can resolve many real world problem like some shortest-path problems.

10.2.1 BFS Algorithm

Algorithm 1 Breadth First Search (BFS)

```
function BFS(Graph $G$, Vertex $v$)
    create a Queue, $Q$
    enqueue $v$ into $Q$
    mark $v$ as visited
    while $Q$ is not empty do
        dequeue a vertex denoted as $w$
        for each unvisited vertex $u$ adjacent to $w$ do
            mark $u$ as visited
            enqueue $u$ into $Q$
```

10.2.2 BFS on A Directed Graph

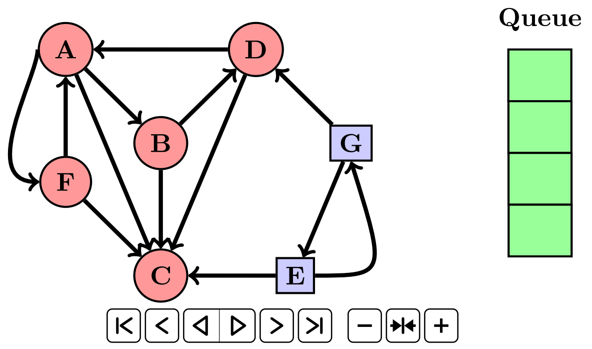


Figure 10.2: A BFS visiting starts from vertex A on a directed graph. Vertex visiting is in alphabetical order

In the example above, vertex G and vertex E are not reachable from vertex A. Thus BFS does not visit both of them in the end.

10.2.3 Time Complexity of BFS

- Each edge is processed once in the while loop for a total cost of $\mathcal{O}(|E|)$.
- Each vertex is queued and dequeued once for a total cost of $\mathcal{O}(|V|)$.
- The worst-case time complexity for BFS is
  - $\Theta(|V| + |E|)$ if graph is represented by an *adjacency lists.*
  - $\Theta(|V|^2)$ if graph is represented by an *adjacency matrix.*

10.2.4 Applications of BFS

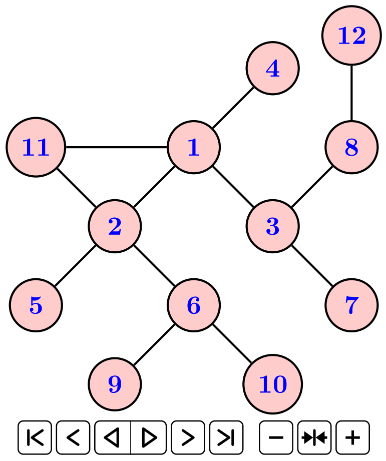
- Finding all connected components in a graph
- Finding all vertices within one connected component
- Finding the shortest path between two vertices

10.3 Depth First Search (DFS)

DFS works fimilar to preorder traversal of a tree. Given a graph G=(V,E) and supposed we start the traversal from a source vertex x(in the example below, x=11), the DFS algorithm systematically explores along a path from vertex x as deeply into the graph G as possible before backing up.

If a vertex has several unvisited neighbours, it would be equally correct to visit them in any order. It is the same as BFS on this issue. Therefore, the result may not be unique.

The tree T built by DFS is called the **depth-first search spanning tree** if the graph G is a connected graph. The tree is a set of $(|V| - 1)$ edges that connect all vertices of the graph. The same idea of the spanning tree can be generalised to directed graphs.



To realise the DFS algorithm, we need a *stack* structure to store visited vertices with unvisited neighbors. Starting from the source vertex, we visit it and push it into the stack. If the just visited vertex has any neighbor which has not been visited, DFS will visit the unvisited neighbor. DFS visits the vertices as far as it can reach. Every edge along the visiting path is traversed once in forward direction (exploring). Once DFS cannot find any new unvisited neighbor to move further, it will start to move backward (backtracking) to the predecessor vertex. Similarly, every edge is also traversed once in backward direction.

10.3.1 DFS Algorithm

Here we introduce two DFS algorithms, an iterative approach and a recursive approach.

Algorithm 2 Depth First Search (DFS)

```
function DFS(Graph $G$, Vertex $v$)
    create a Stack, $S$
    push $v$ into $S$
    mark $v$ as visited
    while $S$ is not empty do
        peek the stack and denote the vertex as $w$
        if no unvisited vertices are adjacent to $w$ then
            pop a vertex from $S$
        else
            push an unvisited vertex $u$ adjacent to $w$
            mark $u$ as visited
```

Algorithm 3 Recursive Depth First Search (DFS_R)

```
function DFS_R(Graph $G$, Vertex $v$)
    mark $v$ as visited
    for each neighbour $w$ of $v$ do
        if $w$ is unvisited then
            DFS_R($G$, $w$)
```

10.3.2 DFS on A Directed Graph

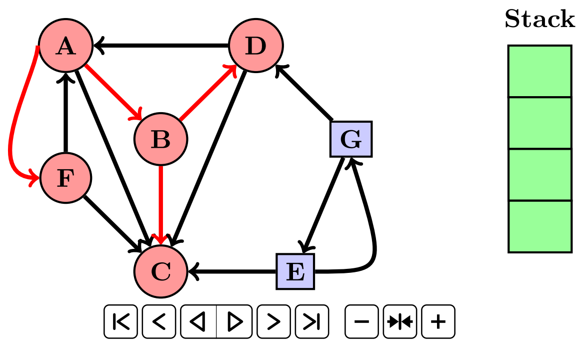


Figure 10.3: A DFS visiting starts from vertex A on a directed graph. Vertex visiting is in alphabetical order

In this example, the graph is not strongly connected. Thus, not all vertices are reachable from vertex A.

10.3.3 Time Complexity of DFS

The time complexity is the same as the BFS's:

- Each edge is processed once in the while loop for a total cost of $\mathcal{O}(|E|)$.
- Each vertex is pushed and popped once for a total cost of $\mathcal{O}(|V|)$.
- The worst-case time complexity for DFS is
  - $\Theta(|V| + |E|)$ if graph is represented by an *adjacency lists*.
  - $\Theta(|V|^2)$ if graph is represented by an *adjacency matrix*.

10.3.4 Applications of DFS

- Topological Sorting
- Finding connected components
- Finding articulation points (cut vertices) of the graph
- Finding strongly connected components
- Solving puzzles

10.3.5 Topological Sorting

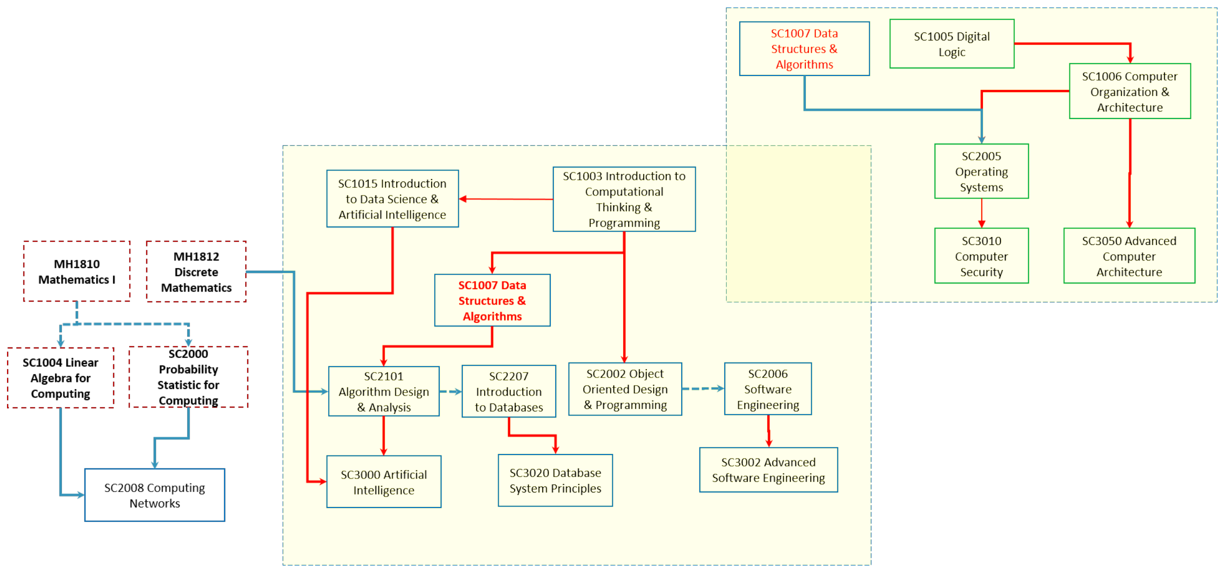

SC1007 Data Structures & Algorithms
SC1005 Digital Logic
SC1006 Computer Organization & Architecture
SC2005 Operating Systems
SC3010 Computer Security
SC3050 Advanced Computer Architecture
SC1015 Introduction to Data Science & Artificial Intelligence
SC1003 Introduction to Computational Thinking & Programming
MH1810 Mathematics I
MH1812 Discrete Mathematics
SC1007 Data Structures & Algorithms
SC1004 Linear Algebra for Computing
SC2000 Probability Statistic for Computing
SC2101 Algorithm Design & Analysis
SC2207 Introduction to Databases
SC2002 Object Oriented Design & Programming
SC2006 Software Engineering
SC2008 Computing Networks
SC3000 Artificial Intelligence
SC3020 Database System Principles
SC3002 Advanced Software Engineering

Figure 10.4: The relationship between SC1007 and other courses

Topological sorting is an algorithm used to sort a directed acyclic graph (DAG) such that for every directed edge from vertex u to vertex v, u comes before v in the ordering. For example in Figure 10.4, our course prerequisite structure is a typical topological sorted graph. In computer science, applications of it arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers.

Topological sorting must apply on a DAG which has no directed cycles. If the graph has a cycle, we cannot determine the order of vertices in the cycles.

The algorithm for topological sorting typically involves traversing the graph using depth-first search, keeping track of nodes that have been visited, and adding nodes to a stack in the order they are visited.

After the traversal is complete, the nodes in the stack are popped off in reverse order to produce the topologically sorted order of the nodes. If the graph has a cycle, it cannot be topologically sorted since there is no way to order the nodes to satisfy the ordering constraint.