Tutorial 0 - Graph Representation, Traversal and MST

COMP9418 – Advanced Topics in Statistical Machine Learning

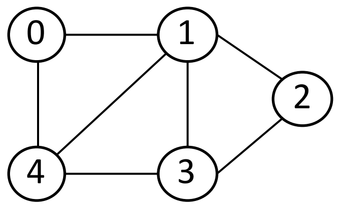
Lecturer: Gustavo Batista

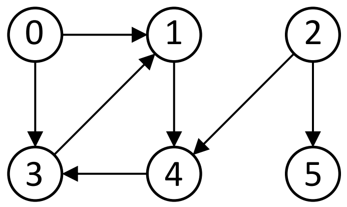
Last revision: Tuesday 27th May, 2025 at 12:49

This tutorial will review some basic concepts of graph representation, traversal and minimum spanning trees (MST). In particular, we will look at some relevant problems we can solve with graph traversal and efficient implementations of MST algorithms. We will not cover this material in the lectures, but it will be essential to understand and implement several algorithms of Probabilistic Graphical Models.

Question 1 - Graph representation

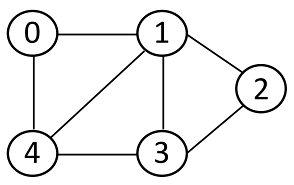
There are two frequently used representations for graphs: adjacency matrix and list. We can use both representations for directed and undirected graphs, like the ones in the following figure.



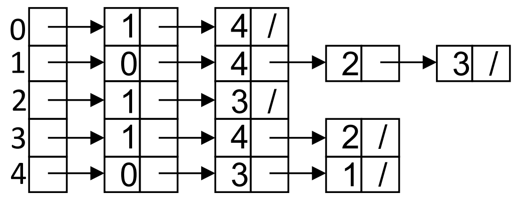


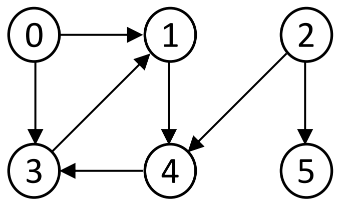
Represent both graphs using the matrix and list representations. You do not need to write any code here. Use a graphical representation such as boxes liked by arrows. Which of these two representations is typically preferred to implement operations over graphs? Why?

Answer

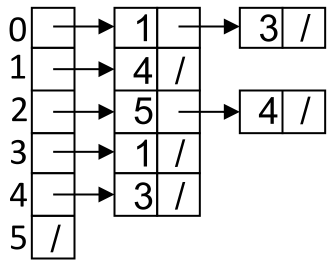


	0	1	2	3	4
0		1			1
1	1		1	1	1
2		1		1	
3		1	1		1
4	1	1		1	





$$\begin{array}{c|cccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & & 1 & & 1 & & \\
1 & & & & & 1 & \\
2 & & & & & 1 & 1 \\
3 & & 1 & & & & \\
4 & & & & 1 & & \\
5 & & & & & & \\
\end{array}$$



We prefer the adjacency list representation since it allows more efficient implementations of the basic operations on sparse graphs. For instance, for a graph with n vertices and m edges, graph traversal operations are implemented in $O(n^2)$ operations with the matrix representation and $O(n+m)$ operations with an adjacency list representation. For a sparse graph $m \ll n^2$.

Question 2 - Graph traversal

Graph traversal is the process of visiting each vertex in a graph. There are two main variations: depth-first (DFS) and breadth-first search (BFS). Write the algorithms for each of these operations. Use a simplifying notation to write a single algorithm for both graph representations. For instance, assume the existence of a function that gives each adjacent edge for a given vertex. For problems that both algorithms can solve, is there one search strategy (depth or breadth) that is preferred? Why?

Answer

DFS algorithm.

```
procedure DFS(G, v):
    label v as grey
    for all edges e in G.adjacentEdges(v) do
        w := G.adjacentVertex(v, e)
        if vertex w is white then
            DFS(G, w)
    label v as black
```

BFS algorithm.

```
procedure BFS(G, v):
    Q := queue.new()              # Q is an empty queue
    Q := insert(v)
    mark v
    while not Q.empty()
        w := Q.dequeue()
        for all edges e in G.adjacentEdges(w) do
            x := G.adjacentVertex(w, e)
            if x is not marked then
                mark x
                Q.insert(x)
```

Both algorithms need a data structure to keep track of the search. DFS uses a stack, and BFS uses a queue. Most implementations of DFS are recursive, so the algorithm avails of the program stack memory, making this data structure implicit.

For problems that both algorithms can solve, we prefer DFS over BFS. The reason is the amount of memory necessary to run these search procedures, i.e., the maximum number of elements we can store in the stack or queue during the execution. For DFS, the memory complexity of $O(h)$ where h is the maximum depth of the search tree (maximum depth of recursive calls). BFS has a memory complexity of $O(n)$. For many practical problems, $h \ll n$.

Question 3 - Cycle detection in directed graphs

In the first part of this course, we will use Direct Acyclic Graphs (DAGs) to represent statistical independencies between random variables. Therefore, developing an algorithm to detect cycles may be helpful to verify if a directed graph is a DAG. The DFS algorithm can be adapted to detect the existence of cycles in a directed graph. A cycle is a non-empty path that links a vertex to itself. A well-known trick is to colour the vertices as we visit them. Vertices are all initially white and become grey as we first visit them. A grey vertex becomes black when we have visited all descendent nodes of this grey vertex and are ready to backtrack the recursion. Adapt your DFS algorithm to detect cycles in directed graphs using this colouring scheme. Is a cycle detected when we reach a white, grey or black node?

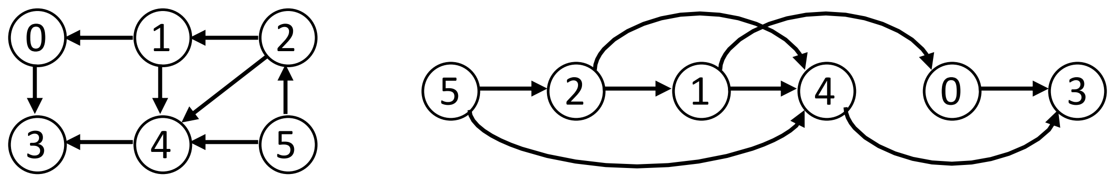
Answer

We will use the colouring vertex scheme of the DFS implementation to detect cycles. In this case, we stop when we identify the first cycle as an edge that links to a grey node. Such an arc is called a “back” edge in directed graphs. Notice that edges connecting to black nodes are “cross” edges and do not indicate cycles.

```
procedure findCycle(G, v):
    label v as grey
    for all edges e in G.adjacentEdges(v) do
        w := G.adjacentVertex(v, e)
        if vertex w is white then
            if findCycle(G, w) then
                return true
         else if vertex w is grey then
            return true
    label v as black
    return false
```

Question 4 - Topological sorting on DAGs

Many algorithms in this course require iterating through nodes, where we must process all parents before their children. For these algorithms, we use an ordering known as *topological ordering*. A DAG’s topological sort or topological ordering is a linear ordering of the vertices. For every directed edge from vertex u to vertex v, u comes before v in the order. The following figure shows a topological ordering for a DAG.



Adapt the DFS algorithm to produce a topological order of a DAG. Do you need any additional data structure to report the results in proper order?

Answer

The idea is that a black node has no further outgoing edges to be explored. Consequently, all nodes that "depend" on this black vertex are already in the output stack. We use a stack to revert the order of the nodes.

```
procedure topologicalSort(G, v):
    label v as grey
    for all edges e in G.adjacentEdges(v) do
        w := G.adjacentVertex(v, e)
        if vertex w is white then
            topologicalSort(G, w)
    label v as black
    push v in s              # s is a stack with topological order
```

Question 5 - Strongly connected directed graphs

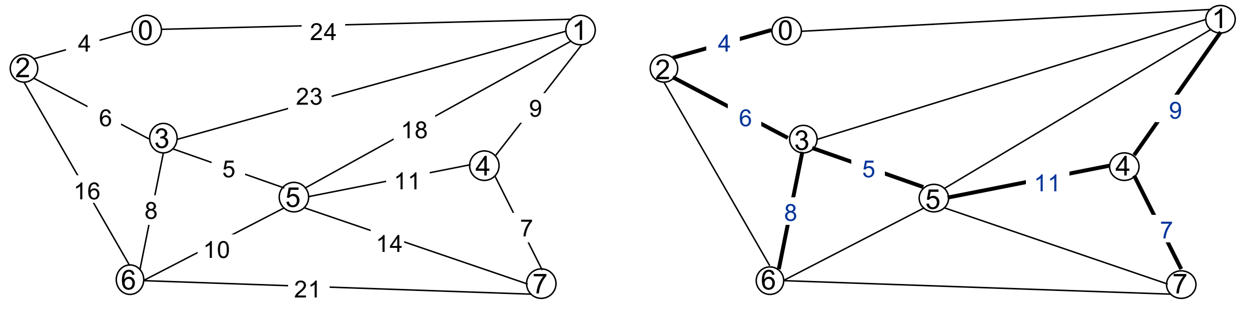
A directed graph is strongly connected if there is a path in each direction between each pair of vertices. We can easily (but inefficiently) test if a directed graph is strongly connected by running n DFS searches from each vertex and confirming that we can reach every node in each search. However, according to Kosaraju's algorithm, two searches suffice, the first one in the graph G and the second one in its transpose G^T. Explain the intuition behind this algorithm.

Answer

First, we obtain G^T by inverting the direction of each edge of G. We can get G^T for the matrix representation using the matrix transpose operation. A DFS search from a node v that reaches all vertices in G confirms the existence of a directed path between v and every node of G. A DFS search from the same node v that reaches all vertices in G^T indicates a directed path between all these nodes and v. Therefore, G is strongly connected since every node can reach every other node with a directed path that passes through v.

Question 6 - Minimal spanning trees

A minimum spanning tree (MST) is an operation over undirected graphs. Given an undirected graph G, MST finds a subset of the edges that forms a tree and includes every vertex of G. The sum of the edges' weights in the tree should be minimal. The following figure shows an example of a graph (left) and its minimum spanning tree (right).



Notice that for a graph with n vertices, the spanning tree must have exactly $n-1$ edges. Adding one more edge would create a cycle, and removing one edge would disconnect the tree, creating a forest.

There are several algorithms to compute an MST of a graph, including the well-known Prim and Kruskal algorithms. We can adapt algorithms to calculate the *maximum* spanning tree by selecting the edges with the highest cost instead of the smallest cost.

Write the pseudo-code of an MST algorithm such as PRIM. What is the time complexity of your algorithm? Which data structures are necessary to achieve such complexity?

Answer

The PRIM pseudo-code is the following.

```
procedure Prim(G, s):
    S := {s}
    Q := priority_queue.new()                          # Q is an empty priority queue
    for all edges e in G.adjacentEdges(s) do
        Q.insert([e.cost, e.from, e.to])               # e.from = s

    while (not Q.empty())
        [cost, v, u] := Q.remove()                     # edge with smallest cost in Q
        if u not in S
            S := S U {u}
            for all edges e in G.adjacentEdges(u) do
                if e.to not in S
                    Q.insert([e.cost, e.from, e.to])   #e.from = u
```

This algorithm has a $O(m \log m)$ time complexity, where m is the number of edges. We have to use a priority queue data structure that allows element insertion and removal with $O(\log n)$ cost to achieve such time complexity.