20 Graphs and Topological Sort

Instructor : Ke Wei（柯韋）

A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP122/19/

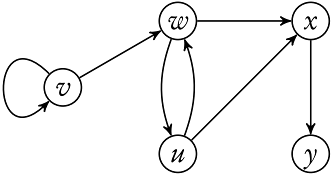
Bachelor of Science in Computing, School of Public Administration, Macao Polytechnic Institute

April 15, 2019

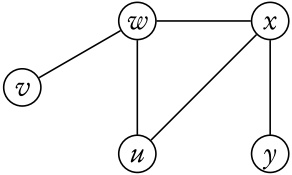
Outline

Graphs

- A graph $G = \langle V, E\rangle$ consists of a set of *vertices* (vertex): V, and a set of *edges*: E. Each edge is a pair (v, w), where $v, w \in V$.
- If the pairs are ordered, then the edges are *directed*, and the graph is called a directed graph. Otherwise the graph is *undirected*.
- A vertex w is *adjacent* to a vertex v if and only if $(v, w) \in E$. In an undirected graph containing edge (v, w), and hence (w, v), v is adjacent to w and w is adjacent to v.



A directed graph



An undirected graph

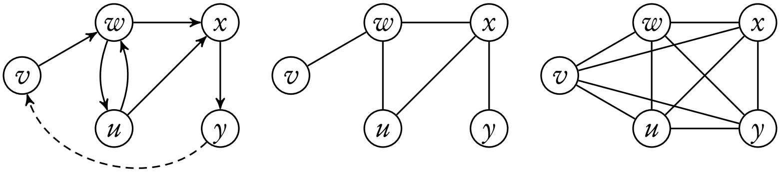
$V = \{u, v, w, x, y\}, E = \{(v, v), (v, w), (w, u), (u, w), (w, x), (u, x), (x, y)\}$ for the above directed graph.

Paths

- A path in a graph is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that $(v_i, v_{i+1}) \in E$, for $1 \leqslant i < n$.
- The *length* of a path is the number of edges on it, i.e. $n-1$. A path from a vertex to itself is allowed. If such a path contains no edge, then the length is 0.
- A *simple path* is a path such that all vertices are distinct, except that the first and last could be the same.
- A *cycle* in a directed graph is a path $v_1, v_2, \ldots, v_n$ of length at least 1 such that $v_1 = v_n$. A directed graph is *acyclic* if it has no cycles. A directed acyclic graph is abbreviated as *DAG*.

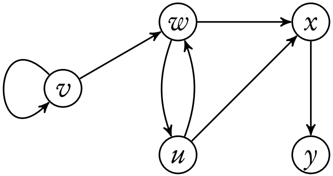
Connectivity

- An undirected graph is *connected* if there is a path from every vertex to every other vertex.
- A directed graph with such a property is called *strongly connected*.
- If a directed graph is not strongly connected, but it would be connected by ignoring the direction of edges, then it is called *weakly connected*.
- A *complete graph* is one where there is an edge between every pair of vertices.
- The *in-degree* of a vertex v is $|\{u \mid (u,v) \in E\}|$, the *out-degree* of v is $|\{u \mid (v,u) \in E\}|$.



Adjacency Matrix

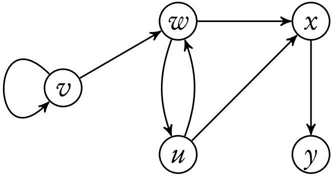
One simple way to represent a graph is to use a two-dimensional array a. It is known as an adjacency matrix representation. For each edge (u, v), we set $a[u][v] \leftarrow 1$; and all other entries in the array are set to 0.

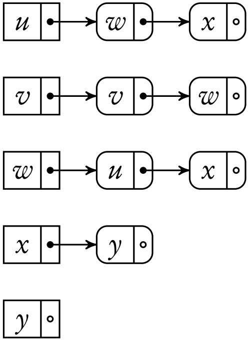


sources \ destinations	u	v	w	x	y
u	0	0	1	1	0
v	0	1	1	0	0
w	1	0	0	1	0
x	0	0	0	0	1
y	0	0	0	0	0

Adjacency List

- An improvement to the edge list is to take the common leading vertices out, and form multiple adjacency lists.
- An adjacency list starts from the leading vertex, followed by the vertices adjacent to the leading vertex.





Adjacency List — Vertices and Edges

- In practice, we use associative arrays instead of lists for more efficient insertion and deletion.
- We define classes for vertices and edges, and index them in an associative array by names.
- The classes also enable us to add more information to vertices and edges when needed.

```python
from avl import AVLAssocArray as AssocArray

class Vertex:
    def __init__(self, name):
        self.name = name
        self.adj_list = AssocArray()
        self.adj_list_values = self.adj_list.values()

class Edge:
    def __init__(self, src, dest):
        self.src, self.dest = src, dest
```

Adjacency List — Graphs

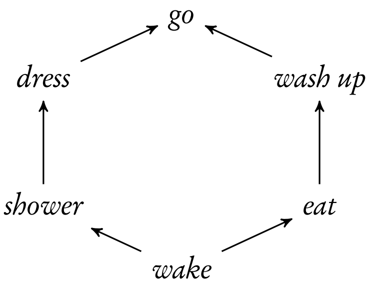
```
class Graph:
    def __init__(self):
        self.v_list = AssocArray()   # A graph is stored as a list of vertices.
        self.v_list_values = self.v_list.values()

    def add_vertex(self, name):
        if name not in self.v_list:
            self.v_list[name] = Vertex(name)
        return self.v_list[name]

    def add_edge(self, src_name, dest_name):
        src, dest = self.add_vertex(src_name), self.add_vertex(dest_name)
        if dest_name not in src.adj_list:   # Adjacent edges are indexed by dest names.
            src.adj_list[dest_name] = Edge(src, dest)
        return src.adj_list[dest_name]
```

Topological Ordering

- A topological sort is an ordering of vertices in a directed acyclic graph, such that if there is a path from v to w, then w appears after v in the ordering.
- It has an interpretation that the starting of w is dependent on the completion of v.
- There may be more than one topological orders for a given graph.



One of the topological orders: *wake*, *shower*, *dress*, *eat*, *wash up*, *go*

Topological Sort by Depth First Search

- We may start from any vertex, and recursively explore the reachable subgraph.
- Until we reach a vertex that has no outgoing edge or all vertices reachable from it have been visited.
- Then we output the vertex to a list (reversely) and mark it visited, there must not be any unvisited vertex that depends on it.
- We pick up any unvisited vertex and repeat the above procedure.
- If all the vertices are visited, the list should contain the vertices in the correct topological order.
- This traversal method is called depth-first-search (DFS).

Topological Sort

- The *explore* function recursively traverses the subgraphs adjacent to the starting vertex v, and then append v to list s, so that all the v's dependents are appended before v.
- The *topo_sort* function creates list s to receive the result of the exploration, and the set to mark the visited vertices, then explores the graph starting from every vertex, finally reverse the result to get the topological order.

```
def explore(v, visited, s):
    if v.name not in visited:
        visited.add(v.name)

        for e in v.adj_list_values:
            explore(e.dest, visited, s)

        s.append(v)
```

```
def topo_sort(g):
    s, visited = [], set()

    for v in g.v_list_values:
        explore(v, visited, s)

    s.reverse()
    return s
```