Graphs

Chapter #9 in Weiss

Graphs

$G = (V, E)$

$V$ are the vertices; $E$ are the edges. Edges are of the form $(v, w)$, where $v, w \in V$.

- ordered pair: directed graph or digraph
- unordered pair: undirected graph

Terminology

- A weight or cost can be associated with each edge.
- $w$ is adjacent to $v$ iff $(v, w) \in E$.
- path: sequence of vertices $w_1, w_2, w_3, \ldots, w_N$ such that $(w_i, w_{i+1}) \in E$ for $1 \leq i < N$.
- length of a path: number of edges in the path.
- simple path: all vertices are distinct.

More terminology

cycle:
- directed graph: path of length $\geq 1$ such that $w_1 = w_N$.
- undirected graph: same, except all edges are distinct.
- connected: there is a path from every vertex to every other vertex.
- loop: $(v, v) \in E$.
- complete graph: there is an edge between every pair of vertices.

Digraph terminology

directed acyclic graph: no cycles. “DAG”
strongly connected: there is a path from every vertex to every other vertex.
weakly connected: the underlying undirected graph is connected.

Representation

- adjacency matrix:

$A[u][v] = \begin{cases} \text{weight}, & \text{if } (u, v) \in E \\ 0, & \text{if } (u, v) \notin E \end{cases}$
### Representation

- **adjacency list:**

```
  1 ----> 2 ----> 3 ----> 4
      |        |        |
      2        1        3
      |        |        |
      3        1        4
```

### Topological Sort

- Given a directed acyclic graph, construct an **ordering** of the vertices such that if there is a path from $v_i$ to $v_j$, then $v_j$ appears after $v_i$ in the ordering.
- **indegree** of $v$: # of edges $(u, v)$.

```cpp
void Graph::topsort()
{
    Vertex v, w;
    for (int counter = 0; counter < NUM_VERTICES; counter++)
    {
        v = findNewVertexOfDegreeZero();
        if (v == NOT_A_VERTEX)
            throw CycleFound();
        v.topologicalNum = counter;
        for each w adjacent to v
            w.indegree--;
    }
}
```

```cpp
void Graph::topsort()
{
    Queue q(NUM_VERTICES);
    int counter = 0;
    Vertex v, w;
    q.makeEmpty();
    for each vertex v
        if (v.indegree == 0)
            q.enqueue(v);
    while (!q.isEmpty())
    {
        v = q.dequeue();
        v.topologicalNum = ++counter;
        for each w adjacent to v
            if (--w.indegree == 0)
                q.enqueue(w);
    }
    if (counter != NUM_VERTICES)
        throw CycleFound();
}
```