Graphs

Definitions

• A graph $G$ consists of a set $V$, whose members are called the vertices of $G$, together with a set $E$ of pairs of distinct vertices from $V$.
• The pairs in $E$ are called the edges of $G$.
• If the pairs are unordered, $G$ is called an undirected graph.
• If the pairs are ordered, $G$ is called a directed graph (or digraph).
• Two vertices in an undirected graph are called adjacent if there is an edge from the first to the second.
Definitions: Paths & Links

- A *path* is a sequence of distinct vertices, each adjacent to the next.
- A *cycle* is a path containing at least three vertices such that the last vertex on the path is adjacent to the first.
- A graph is called *connected* if there is a path from any vertex to any other vertex.
- A *free tree* is defined as a connected undirected graph with no cycles.
- In a directed graph a path or a cycle means always moving in the direction indicated by the arrows. Such a path (cycle) is called a *directed path (cycle).*

Definitions: Connected components

- A directed graph is called *strongly connected* if there is a directed path from any vertex to any other vertex.
- If we suppress the direction of the edges and the resulting undirected graph is connected, we call the directed graph *weakly connected.*
- The *valence (or degree)* of a vertex is the number of edges on which it lies, hence also the number of vertices adjacent to it.
Examples

1. Selected South Pacific air routes
2. Benzene molecule
3. Message transmission in a network

Examples

1. Connected (a)
2. Path (b)
3. Cycle (c)
4. Disconnected (d)
5. Tree (e)
Examples

(a) Directed cycle
(b) Strongly connected
(c) Weakly connected

Representation

Definition: A graph $G$ consists of a set $V$, called the vertices of $G$, and, for all $v \in V$, a subset $A_v$ of $V$, called the set of vertices adjacent to $v$.

Directed graph

Adjacency sets

<table>
<thead>
<tr>
<th>vertex</th>
<th>set</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(2, 3)</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>(3, 4)</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>(1, 2, 3)</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Adjacency table

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>
Linked List

Contiguous List
Graph Traversal
Graph Traversal: Depth First

Depth-first traversal

Depth First Search

/* DepthFirst: depth-first traversal of a graph.*/
Pre: The graph G has been created.
Post: The function Visit has been performed at each vertex of G in depth-first order.
Uses: Function Traverse produces the recursive depth-first order. */

void DepthFirst(Graph G, void (*Visit)(Vertex))
{
  Boolean visited[MAXVERTEX];
  Vertex v;
  for (all v in G)
    visited[v] = FALSE;
  for (all v in G)
    if (!visited[v])
      Traverse(v, Visit);
}

void Traverse(Vertex v, void (*Visit)(Vertex))
{
  Vertex w;
  visited[v] = TRUE;
  Visit(v);
  for (all w adjacent to v)
    if (!visited[w])
      Traverse(w, Visit);
}
Graph Traversal: Breadth First

Breadth-first traversal

Breadth First Search

```c
void BreadthFirst(Graph G, void (*Visit)(Vertex))
{
    Queue Q; /* QueueEntry defined to be Vertex. */
    Boolean visited[MAXVERTEX];
    Vertex v, w;

    for (all v in G)
        visited[v] = FALSE;
    CreateQueue(Q);

    for (all v in G)
        if (visited[v])
            do {
                Serve(v, Q);
                if (!visited[v]) {
                    visited[v] = TRUE;
                    Visit(v);
                }
                for (all w adjacent to v)
                    if (!visited[w])
                        Append(w, Q);
            } while (!QueueEmpty(Q));
}
```
Topological Sorting

Let $G$ be a directed graph with no cycles. A **topological order** for $G$ is a sequential listing of all the vertices in $G$ such that, for all vertices $v, w \in G$, if there is an edge from $v$ to $w$, then $v$ precedes $w$ in the sequential listing.

Directed graph with no directed cycles
Topological Sorting: Depth First

Idea: Depth-First

- By Depth-first traversal find the last node which has no successor.
- Place it in the last order.
- By recursion, when the routine returns, put its immediate successors into topological order.
- Use a variable ‘place’ to indicate the rank in the topological order.
void DepthTopSort(Graph *G, Toporder T) {
    Vertex v; /* next vertex whose successors are to be ordered*/
    int place /* next position in the topological order to be filled*/

    for (v = 0; v < G->n; v++)
        visited[v] = FALSE;
    place = G->n - 1;
    for (v = 0; v < G->n; v++)
        if (!visited[v])
            RecDepthSort(G, v, &place, T);
}

void RecDepthSort(Graph *G, int v, int *place, Toporder T) {
    Vertex curvertex; /* vertex adjacent to v */
    Edge *curedge; /* traverses list of vertices adjacent to v */

    visited[v] = TRUE;
    curedge = G->firstedge[v]; /* Find the first vertex succeeding v. */

    while (curedge) {
        curvertex = curedge->endpoint; /* curvertex is adjacent to v. */
        if (visited[curvertex])
            RecDepthSort(G, curvertex, place, T); /* Order the successors of curvertex. */
        curedge = curedge->nextedge; /* Go on to the next immediate successor of v. */
    }

    T[*place] = v; /* Put v itself into the topological order. */
    (*place)--;
}

Since each of the nodes and links are visited only once the complexity is O(n+e)
Topological Sort: Breadth-First

- Setup an array “predecessorcount[]” to keep a count of immediate predecessors to a node.
- The first vertices has zero count.
- Put these vertices with zero count into a queue.
- Visit each of them in the queue.
- When visit them
  - remove them from the queue,
  - assign the next place in the sorted list,
  - reduce the predecessor count of each of their successors by one.
  - If any of its successor’s count becomes zero, put it in the queue.

Since each of the nodes and links are visited only once the complexity is $O(n+e)$

Errata: link 9-3 does not exist.