Graphs

1

DESIGN & ANALYSIS OF ALGORITHM

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.

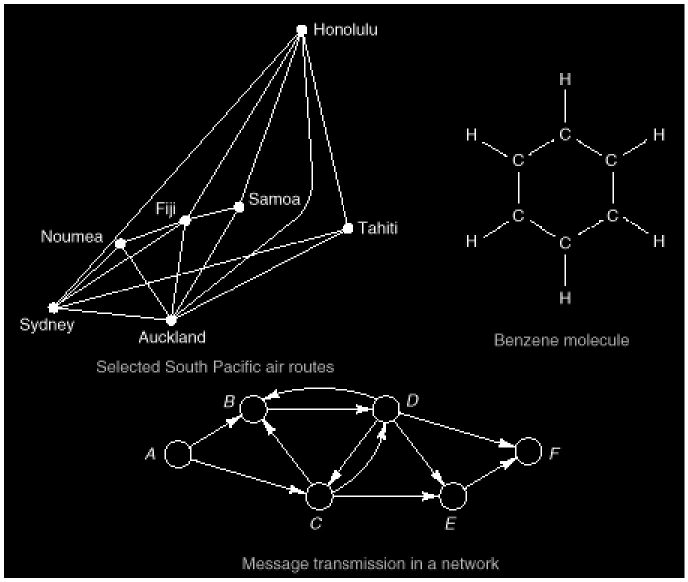
Definitions: Paths & Links

- 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

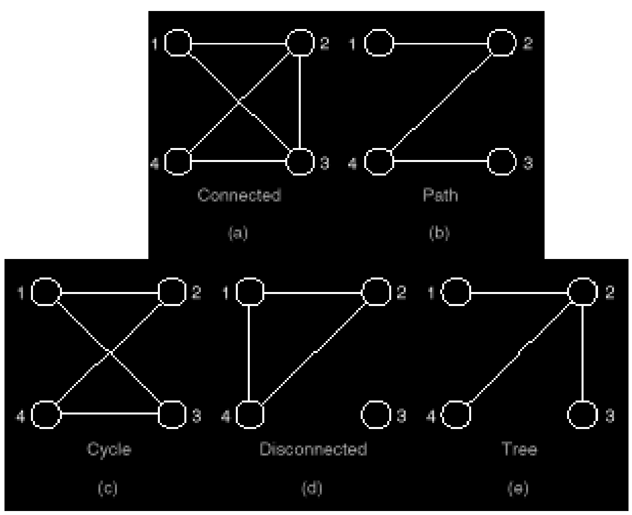
- A directed graph is called *strongly connected* if there is a directed path from any vertex to any other vertex.
- If we suppress the direction of the edges and the resulting undirected graph is connected, we call the directed graph *weakly connected*.
- The *valence (or degree)* of a vertex is the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

Examples



DESIGN & ANALYSIS OF ALGORITHM

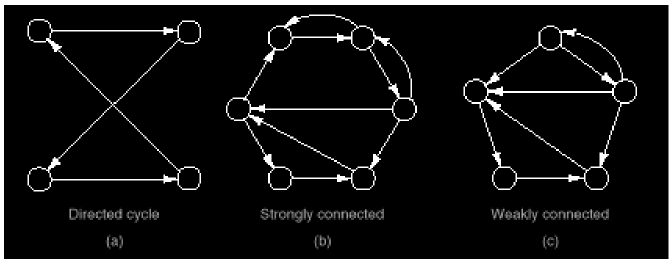
Examples



DESIGN & ANALYSIS OF ALGORITHM

Examples

DESIGN & ANALYSIS OF ALGORITHM

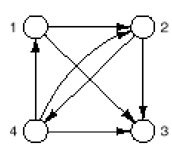


Representation

DESIGN & ANALYSIS OF ALGORITHM

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



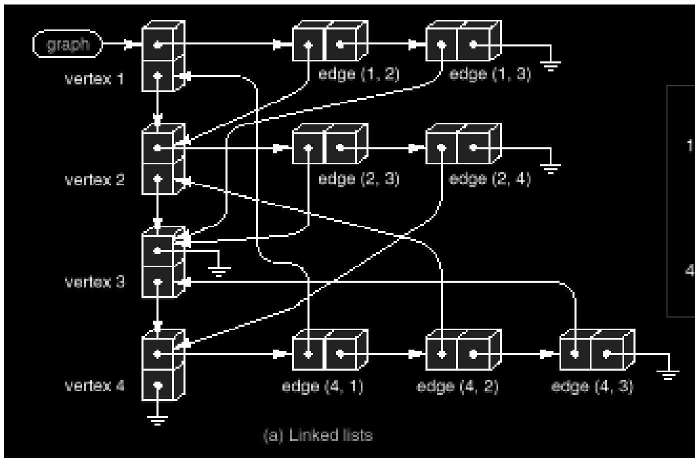
Adjacency sets

vertex	Set
1	{2, 3}
2	{3, 4}
3	ϕ
4	{1, 2, 3}

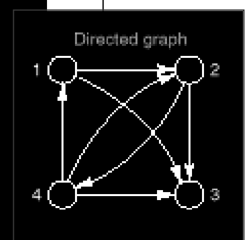
Adjacency table

	1	2	3	4
1	F	T	T	F
2	F	F	T	T
3	F	F	F	F
4	T	T	T	F

Linked List



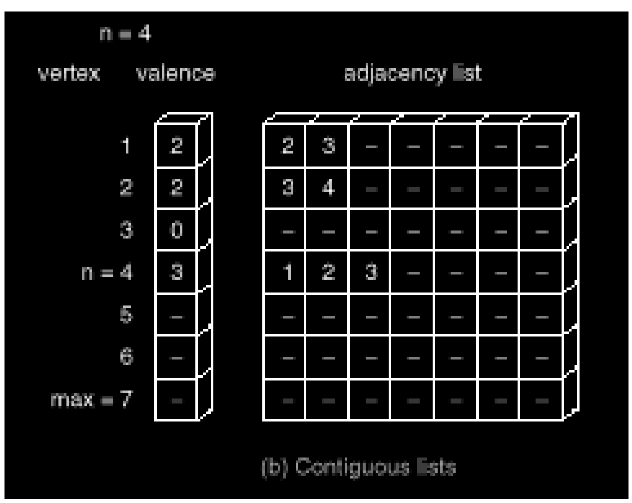
(a) Linked lists



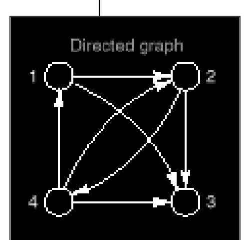
DESIGN &
ANALYSIS OF
ALGORITHM

ALG99F, javed@kent.edu
Javed I. Khan@1999

Contiguous List



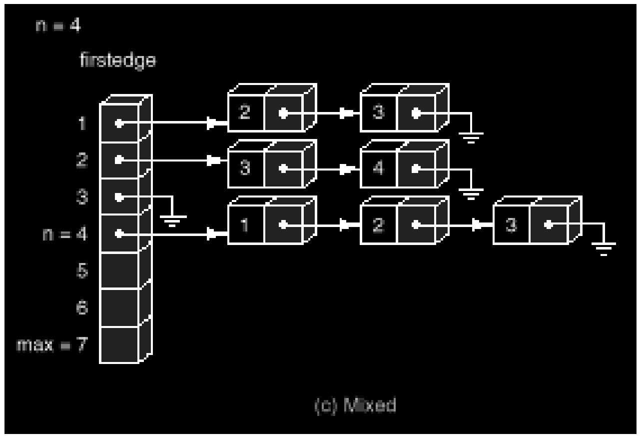
(b) Contiguous lists

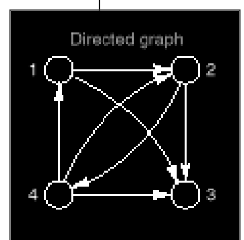


DESIGN &
ANALYSIS OF
ALGORITHM

ALG99F, javed@kent.edu
Javed I. Khan@1999

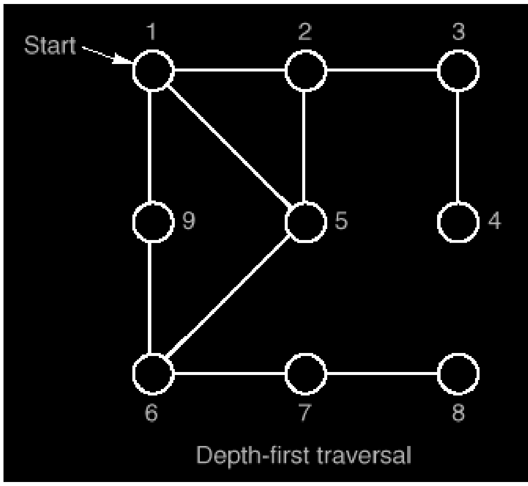
Mixed





Graph Traversal

Graph Traversal: Depth First



ertices w adjacent to v do
rse(w, Visit);

DESIGN & ANALYSIS OF ALGORITHM

LECT-13, S-13
ALG99F, javed@kent.edu
Javed I. Khan@1999

Depth First Search

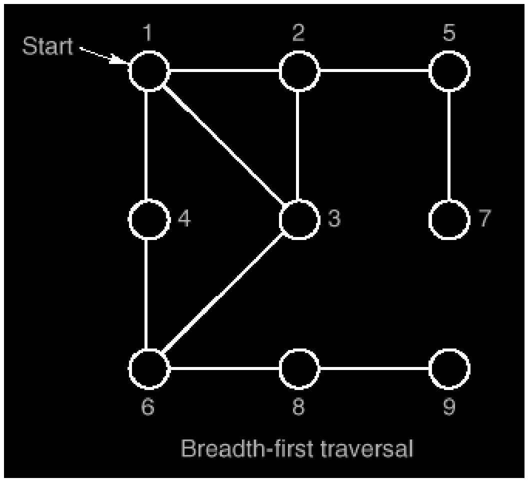
```
/* DepthFirst: depth-first traversal of a graph.
Pre:  The graph G has been created.
Post: The function Visit has been performed at each vertex of G in depth-first order.
Uses: Function Traverse produces the recursive depth-first order.  */
void DepthFirst(Graph G, void (*Visit)(Vertex))
{
   Boolean 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);
}
```

DESIGN & ANALYSIS OF ALGORITHM

LECT-13, S-14
ALG99F, javed@kent.edu
Javed I. Khan@1999

Graph Traversal: Breadth First

DESIGN &
ANALYSIS OF
ALGORITHM



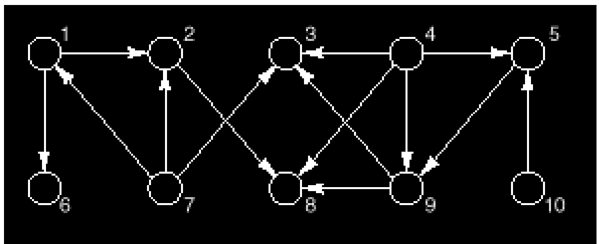
Breadth First Search

DESIGN &
ANALYSIS OF
ALGORITHM

```
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]) {
         Append(v, Q);

         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.

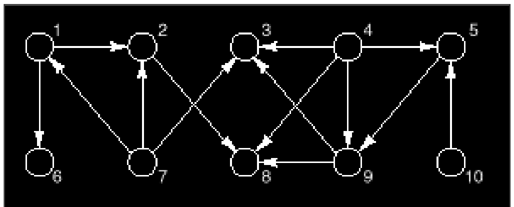


Directed graph with no directed cycles

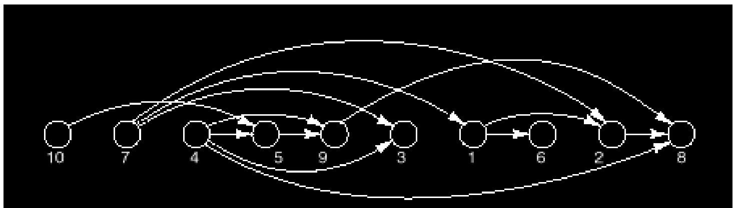
DESIGN & ANALYSIS OF ALGORITHM

Topological Sorting

Topological Sorting: Depth First



Directed graph with no directed cycles



Depth-first ordering

DESIGN & ANALYSIS OF ALGORITHM

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.

DESIGN & ANALYSIS OF ALGORITHM

Code

```
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);
}
```

DESIGN & ANALYSIS OF ALGORITHM


```
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)

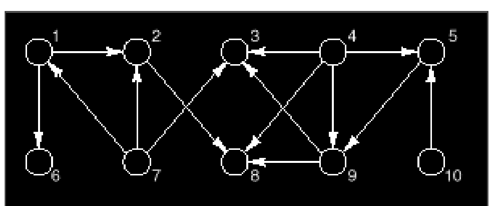
DESIGN & ANALYSIS OF ALGORITHM

Topological Sort: Breadth-First

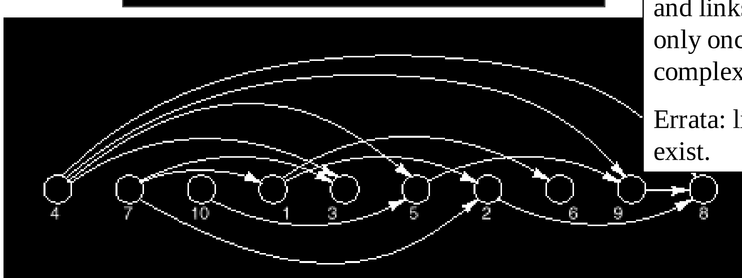
- Setup an array “predecessorcount[]’ to keep a count of immediate predecessors to a node.
- The first vertices has zero count.
- Put these vertices with zero count into a queue.
- Visit each of them in the queue.
- When visit them
  - remove them from the queue,
  - assign the next place in the sorted 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.

DESIGN & ANALYSIS OF ALGORITHM

Topological Sorting: Breadth First



Directed graph with no directed cycles



Breadth-first ordering

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

Errata: link 9-3 does not exist.

DESIGN & ANALYSIS OF ALGORITHM