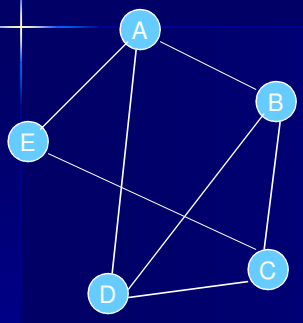
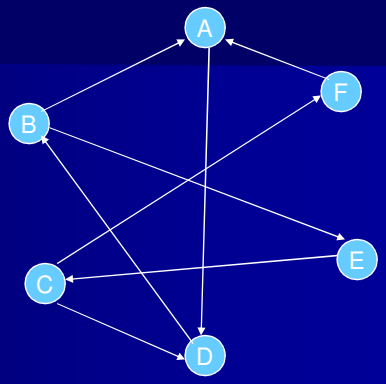
Finite Graphs

- **Def**. A finite graph G consists of a finite set of vertices, V, and a finite set E of pairs of vertices from V, called edges. Notation: G = {V, E}.
- We say that two vertices v_1 and v_2 are **adjacent** if there exists an edge connecting them.
- A graph is called **directed** (or a digraph) if the edges are ordered pairs. The graph is **undirected** if the edges are unordered pairs.
- A graph is called **weighted** if every edge has a real number associated with it, called weight or cost.

Examples



undirected graph
5 vertices
7 edges



directed graph
6 vertices
8 edges

Path

- If v_1 and v_2 are vertices from V, we denote the edge connecting them as (v_1, v_2) for a directed graph and $\{v_1, v_2\}$ for an undirected graph.
- A **path** in a graph is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that for every i between 1 and n-1, (v_i, v_{i+1}) or $\{v_i, v_{i+1}\}$ is an edge.
- A path is called a **cycle** if the first and last vertices are the same: $v_1 = v_n$.
- The length of a path is the number of edges in the path. In this notation, it's n-1.
- A path is called **simple** if no vertex is repeated in the path. A cycle is called simple if by removing the last vertex, it becomes a simple path.

Connectivity

- An undirected graph is called **connected** if for every pair of vertices x and y in the graph there is a path from x to y.
- Def. A graph H = {W, F} is a subgraph of a graph G = {V, E} if $W \subseteq V$ and $F \subseteq E$.
- **Def**. A **connected component** in a graph is the subgraph containing all the vertices for which there is a path to a particular vertex and all the edges that these vertices belong to.
- A graph is called **complete** if it contains all possible edges.

Graph Implementations

- There are two classical approaches: list of vertices or adjacency matrix.
- For a list of vertices, for every vertex we keep a linked list of the vertices adjacent to it.
- For an adjacency matrix, we keep the edges in a matrix with a nr of rows and columns equal to the number of vertices in the graph.
- We represent an edge (x, y) by assigning adj[x][y] = 1; All the other elements of the matrix are 0.
- The adjacency matrix is used when the number of edges is big (close to n2).

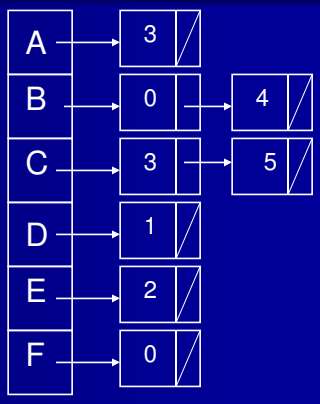
Adjacency Matrix

- Used when the graph is small or when the number of edges is high.
- Adjacency matrix for the first graph in the examples:
- For un undirected graph the matrix is symmetric.

	A	B	C	D	E
A	0	1	0	1	1
B	1	0	1	1	0
C	0	1	0	1	1
D	1	1	1	0	0
E	1	0	1	0	0

Adjacency List

- The most common implementation of graphs.
- Example for the second graph.
- If the vertices have non-trivial names, they can be stored in a hash table separately.
- If the graph is weighted, the weights are contained in the nodes of the adjacency lists.



Graph Class

```
class Graph {
  struct vertex {
    list<int> neighbors;
    string name;
  };
  int nr_vertices, nr_edges;
  bool directed;
  vector<vertex> vertices;
};
```

Add a Vertex and an Edge

```
void Graph::add_vertex(const
 string the_name)
 {
   vertex new_vertex;
   new_vertex.name = the_name;
   vertices.push_back(new_vertex)
 ;
   nr_vertices++;
 }
```



```
void Graph::add_edge(const string name1,
const string name2)
{
  int index1 = index(name1);
  int index2 = index(name2);
  if (index1 >= nr_vertices)
    add_vertex(name1);
  if (index2 >= nr_vertices)
    add_vertex(name2);
  add_edge(index1, index2);
}
void Graph::add_edge(const int index1,
const int index2)
{
  vertices[index1].neighbors.push_back(inde
x2);
  if (!directed)
    vertices[index2].neighbors.push_back(in
dex1);
  nr_edges++;
}
```

Building a Graph "by Hand"

```
void make_graph1(Graph &gr)
{
  for (char n='A'; n <= 'E';
n++)
    add_char_vertex(gr, n);
  add_char_edge(gr, 'A', 'B');
  add_char_edge(gr, 'A', 'D');
  add_char_edge(gr, 'A', 'E');
  add_char_edge(gr, 'B', 'C');
  add_char_edge(gr, 'B', 'D');
  add_char_edge(gr, 'C', 'D');
  add_char_edge(gr, 'C', 'E');
}
```


Make Empty

```
void Graph::make_empty()
{
  if (nr_vertices) {
    for (int i=0;i<nr_vertices; i++)
      vertices[i].neighbors.clear();
    vertices.erase(vertices.begin(),
                   vertices.end());
    nr_vertices = 0;
    nr_edges = 0;
  }
  directed = false;
}
```

```
void Graph::print() {
  int i;
  cout <<"The graph contains " << nr_vertices
       <<" vertices and "<<nr_edges<<" edges"
       << endl << "The vertex names are: ";
  for (i=0; i<nr_vertices; i++)
    cout << vertices[i].name << ' ';
  cout << endl << "The adjacency list for each"
       " vertex is:" << endl;
  for (i=0; i<nr_vertices; i++) {
    cout << "vertex " << i << ": ";
    list<int>::iterator iter;
    iter = vertices[i].neighbors.begin();
    while (iter!=vertices[i].neighbors.end()){
      cout << *iter << ' ';
      iter++;
    }
    cout << endl;
  }
}
```


Storing the Graph in a File

- Starts with a character telling us if the graph is directed or not.
- The next line contains the number of vertices.
- The names of the vertices are listed afterwards on a line each or with spaces between.
- The edges are listed next, each edge on a line with a comma and a space (!) between the vertices. If the graph is weighted, the edge contains the weight too.

```
U
4121
Frankfurt
Tehran
Mexico City
Paris
...
South Bend, Paris, 4225
Nairobi, South Bend,
    7806
Boston, Calcutta, 13410
Sidney, Calcutta, 8007
```

```
void Graph::read(const char *filename)
{
  ifstream fin(filename);
  char buffer[20], buffer1[20];
  int nrv;
  fin >> buffer; // U or D
  if (buffer[0] == 'd' || buffer[0] == 'D')
    directed=true;
  fin >> nrv;
  for (int i=0; i<nrv; i++) {
    fin >> buffer; // vertex name
    add_vertex(string(buffer));
  }
  while (!fin.eof()) {
    fin >> buffer;
    buffer[strlen(buffer)-1]='\0';
    fin >> buffer1;
    if (strlen(buffer) && strlen(buffer1))
      add_edge(string(buffer),
               string(buffer1));
  }
  fin.close();
}
```


Graph Traversal

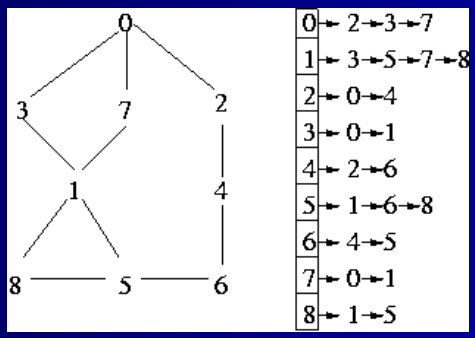
- There are two general graph traversal algorithms: breadth-first and depth-first.
- The breadth-first starts from a vertex, then explores all of its neighbors, the all of the neighbors of the neighbors, and so on.
- The depth first starts from a vertex and explores a path in the graph as far as it can go without repeating any vertex. When it can't go any further, it comes back and tries another path.

Breadth-First

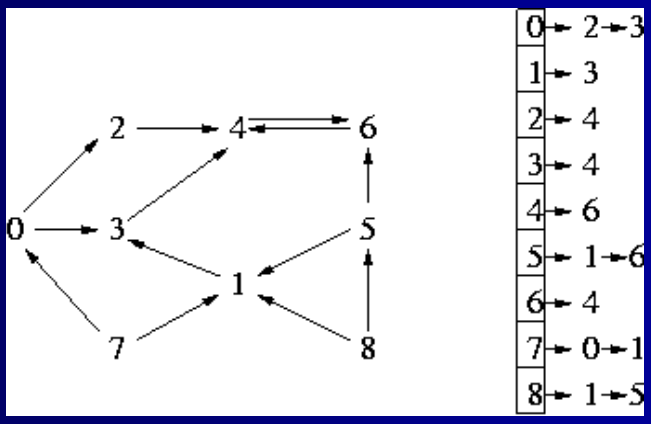
- In a breadth first search we start at some vertex v, process it, and mark it as having been "reached".
- Then we take each of its neighbors in turn, process the edges from v to those neighbors, process the neighbors themselves, and mark all the neighbors as having been reached.
- Then for each of those neighbors, taken one at a time, we process the edges out of them, and process and mark the neighbors at the ends of those edges.
- We continue in this way until all vertices that can be reached have been marked.


```
breadth_first_search (graph G, int v) {
   initialize all vertices as "unmarked";
   queue<int> Q;
   process and mark v;
   Q.enqueue(v);
   while (Q is not empty) {
      x = Q.dequeue();
      for each neighbor y of x in G {
        process the edge from x to y if
          necessary;
        if (y has not been marked) {
          process and mark y;
          enqueue (y, Q);
        }
      }
      perform final processing on vertex x;
   }
}
```

Graph Page GR-13



Graph Page GR-14



Depth-First

- We start at some vertex v , process it, and mark it as having been reached.
- Then we select one of its neighbors, call it w , process the edge from v to w , and mark w as having been reached.
- Then we select an unmarked neighbor of w , call it x , process the edge from w to x , and mark x as reached.
- We keep on going "deeper and deeper" into the graph until we cannot reach any more unmarked nodes.
- Then we backtrack until we get back to a node that has an unmarked neighbor, and we start down that path.
- We continue this process until we exhaust all possible paths from the original vertex v.


```
Graph::depth_first_search (int v) {
  bool mark[nr_vertices];
  for (int i=0; i< nr_vertices; i++)
    mark[i]=false;
  stack<(int, vertex*)> S;
  mark[v]=true; // perform preliminary processing on v;
  p = vertices[v].neighbors.begin();
  S.push(v,p);
  while (S is not empty) {
    (v,p)=S.pop();//perform intermediate processing on v;
    while (p != vertices[v].neighbors.end()) {
      w = *p; // process the edge from v to w;
      p++;
      if (!mark[w]) { // w has not been reached)
        S.push(v,p);
        v = w;
        mark[v] = true; //perform preliminary processing
  on v;
        p = vertices[v].neighbors.begin();
      }}
    perform final processing on vertex v;
}}
```

```
bool mark[nr_vertices];
Graph::depth_first_search(int v) {
  for (int i=0; i< nr_vertices; i++)
    mark[i]=false;
  recursive_depth_first_search(v);
}

Graph::recursive_depth_first_search (int v) {
   mark[v] = true;
   perform preliminary processing on v;
   p = vertices[v].neighbors.begin();
   while (p != vertices[v].neighbors.end()) {
      perform intermediate processing on v if necessary;
      w = *p;
      process the edge from v to w if necessary;
      p++;
      if (!mark[w])
         recursive_depth_first_search (w);
   }
   perform final processing of vertex v;
}
```

Shortest Path Problems

- How can we find a path in G from the origin to the destination with the fewest possible edges?
- It is possible to rewrite the breadth-first algorithm to find the shortest path to a vertex.
- For this, when we mark a new vertex, we can count the number of edges to that node.
- Every time we follow an edge, we increase the length of the path to that node by 1.
- The first time we encounter a node, it will be on the shortest path.

Least Costly Path

- For a weighted graph, we would like to find the path such that the sum of the weights of all the edges is minimal.
- The best known algorithm was invented in 1959 by E. Dijkstra.
- It's a modification of the BF algorithm using a priority queue instead of the simple queue.
- The queue will contain the least costly path to each vertex as known up to that point in the algorithm.

Dijkstra's Algorithm

- The algorithm starts with an origin and has to reach a destination.
- We will need to store for every vertex, the total cost of the path from the origin to it.
- We also need to know for every vertex, its predecessor in the path from the origin to it.
- We will classify the vertices in the graph as finalized or not. For the finalized vertices, the best path from the origin to it known so far is already the least costly path.
- The algorithm stops when the destination has been finalized.

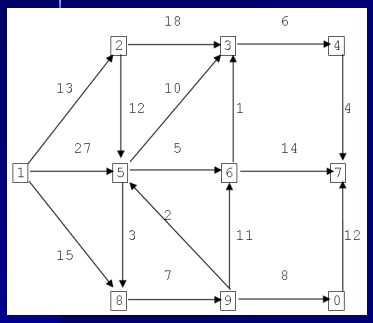
Dijkstra Initialization

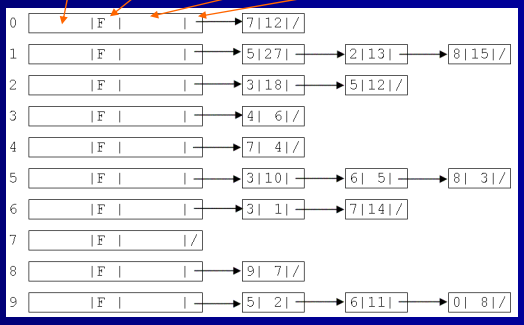
```
Initialization()
{
  for (int i=0; i<nr_vertices; i++) {
    vertices[i].total_cost = +infinity;
    vertices[i].finalized = false;
  }
  Priority_min_queue PQ;
  vertices[origin].total_cost = 0;
  vertices[origin].predecessor = -1;
  PQ.push(origin, 0);
}
```



```
dijkstra_algorithm (graph G, int origin, int
  destination) {
  Initialization(); // should return PQ somehow.
  while (PQ is not empty) {
    x = PQ.pop();
    if (vertices[x].finalized) continue;
    vertices[x].finalized = TRUE;
    if (x == destination)
      return;
    for each non-finalized neighbor y of x
      if (vertices[x].total_cost + cost(x,y) <
          vertices[y].total_cost) {
        vertices[y].total_cost=
           vertices[x].total_cost + cost(x,y);
        vertices[y].predecessor = x;
        PQ.push(y, vertices[y].total_cost);
      }
  }
}
```

Graph Example

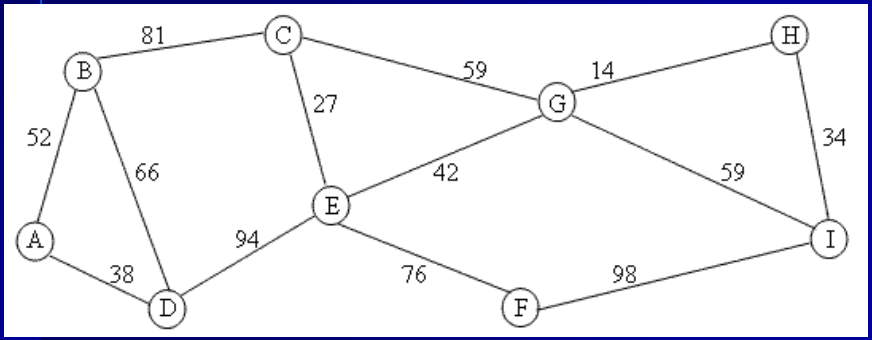




Minimum Spanning Tree

- **Def**. A free tree is an undirected, connected, acyclic graph.
- **Def**. A spanning tree for a given graph is a free tree that has the same vertex set as the graph.
- Given an undirected, weighted graph G=(V,E), find a spanning tree T such that the sum of the weights of the its edges is minimal.

Example

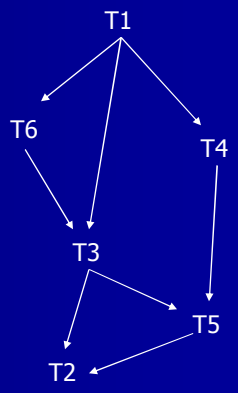


Kruskal's Algorithm (1956):

- Let G=(V, E).
- Initialize T = (V, ET) , ET = ∅ - totally disconnected.
- Let C = {connected components of T}.
- while (| ET | < n - 1 and E is not exhausted) {
  - Remove a lowest cost edge e ={a, b} from E;
  - If a and b belong to different connected components of T, then add e to ET and form the union of the two components containing a and b, creating a single component of C.

Topological Sorting

- Suppose we have to perform a number of tasks, some of which depend on others, and we can only do one at a time.
- We can organize the tasks in a dependency graph.
- We must find an ordering of the tasks respecting the dependencies.
- Example: T1, T6, T3, T4, T5, T2.



The Problem

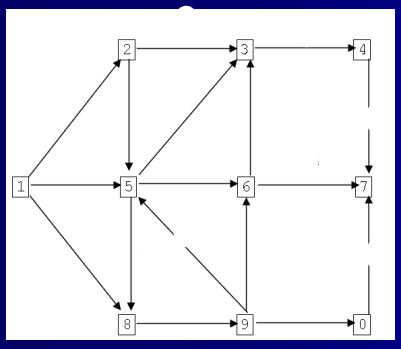
- **The Topological Sorting Problem**: Given a digraph G, find, if possible, a non-repetitive listing of all its vertices in such an order that for every pair of vertices x and y, if the edge (x, y) is in the graph G, then x precedes y in the list.
- Any listing of the vertices with these properties is called a **topological ordering** of the vertices, and finding such a listing is called **sorting the vertices into a topological order**.
- If we number the vertices in the order in which they appear in such a list, we say that we have a **topological numbering** of the vertices.
- **Note**: If the graph contains a cycle, then it cannot be topologically sorted.

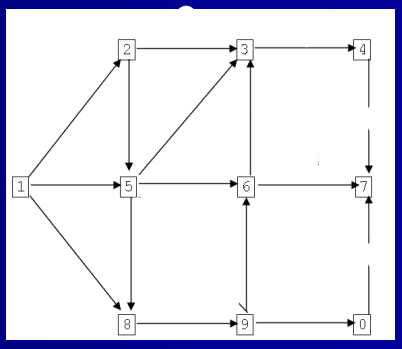
```
boolean topologically_number (digraph G)
{
  for (ever vertex v in G) {
    visited[v] = false;
    numbered[v] = false;
  }
  int topol_counter = number_of_vertices(G);
  for (each vertex v in G)
    if (!visited[v]) {
      visited[v] = true;
      if (!recursively_number(G, v,
                              topol_counter))
        return false;
    }
  return true;
}
```



```
boolean recursively_number (digraph G, int v,
  int & counter)
{
  for (each w that can be reached from v) {
    if (numbered[w]);
    else if (visited[w]) // but not numbered
      return false;  // a cycle has been found
    else {
      visited[w] = true;
      if (!recursively_number (G, w, counter))
        return false;
    }
  }
  vertex[v].topol_number = counter;
  -- counter;
  return true; // no cycles were detected
}
```

Example





Tours or Circuits

- **Hamiltonian** path or circuit. A path or a circuit in a graph that visits every vertex once and only once. NP-complete.
- **Eulerian** path or circuit. A path or a circuit that visits every edge once and only once.
- **Traveling Salesman problem**. In a totally connected, weighted graph, find the Hamiltonian circuit minimizing the total cost. NP-complete.

Eulerian Circuit

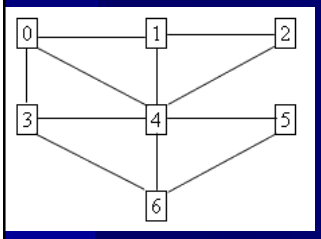
- Let G denote an undirected, connected graph.
- An **Eulerian path** in G is a path that includes every edge in G exactly once. An **Eulerian circuit** is an Eulerian path that ends at the same vertex at which it started.
- **Def.** The **degree** of a vertex v in an undirected graph is the number of edges that v belongs to.

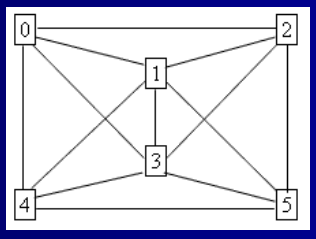
Theorems

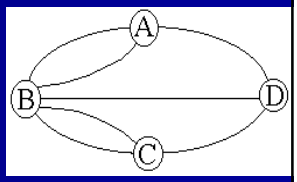
- **Theorem**: Let P denote a path in a undirected graph G, and let v denote a vertex on P. Suppose that P passes exactly once over every edge that touches v.
  a) If the degree of v is odd, then P must either start or stop at v, but cannot do both.
  b) If the degree of v is even, then P must both start and stop at v or else must neither start nor stop at v.
- **Corollary**:
- If a graph contains more than two vertices with odd degree, then it cannot contain an Eulerian path.
- If a graph contains exactly two vertices with odd degree, then any Eulerian path must start at one of these vertices and end at the other (so it cannot be an Eulerian circuit).
- If G contains no vertices with odd degree, then any Eulerian path in G must be an Eulerian circuit.

Euler's Algorithm

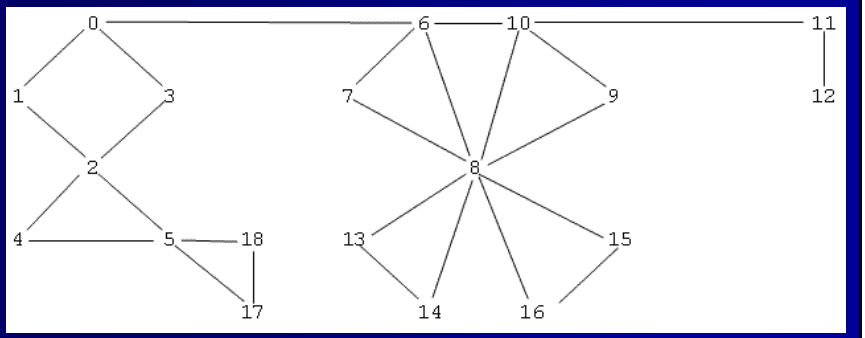
```
Print_Eulerian (graph G, int v)
{
  for each non-eliminated edge {v,w} {
      mark the edge {v,w} as eliminated;
      Print_Eulerian (G, w);
   }
   print v;
}
```







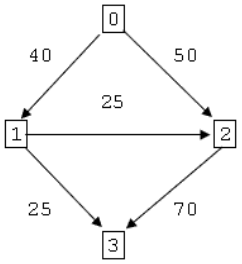
Example

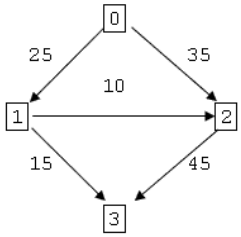


Maximum Flow

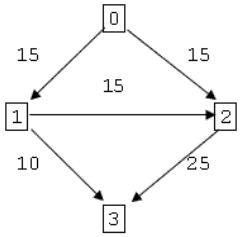
- Let G be a weighted digraph with one vertex, s (**source**), having no incoming edges and another vertex, t (**sink**), having no outgoing edges. The weights of the edges will be called **capacities**.
- Find an assignment of **flows** (non-negative numbers) to all the edges such that the following conditions are satisfied:
  1) each flow along an edge is less than or equal to the capacity of that edge;
  2) for each intermediate vertex v , the sum of the flows on edges *into* v is equal to the sum of the flows on edges *out of* v ("conservation of flow");
  3) the sum of the flows into t is as large as possible.

Example





Flow Graph



Residual Graph

Edmund-Karp Algorithm

- Start with a trivial flow graph (where all the flows are 0).
- Repeat the following steps while possible:
  - Find a path from the source to the sink in the residual graph such that none of the weights on that path is 0.
  - Add the minimal weight on that path to all the edges in the flow graph and remove it from the residual graph.

Example