Graph

April 5, 2004
Graph Terminology

- Graph: vertices + edges
- Induced subgraph of a subset of vertices
- Connected graph: a path between every pair
  - Maximal connected: there is no path from this subgraph to an outside vertex
Representation

- **Adjacency matrix**
  - V by v array
  - Adv./disadvantages

- **Adjacency list**
  - Linked list for each vertex
  - Adv./disadvantages

```c
typedef struct { int v; int w; } Edge;
Edge EDGE(int, int);

typedef struct graph *Graph;
Graph  GRAPHinit(int);
void   GRAPHinsertE(Graph, Edge);
void   GRAPHremoveE(Graph, Edge);
int    GRAPHedges(Edge [], Graph G);
Graph  GRAPHcopy(Graph);
void   GRAPHdestroy(Graph);

typedef struct node *link;
struct node { int v; link next; };
struct graph { int V; int E; link *adj; };
Graph  GRAPHinit(int V)
{ int v;
  Graph G = malloc(sizeof *G);
  G->V = V; G->E = 0;
  G->adj = malloc(V*sizeof(link));
  for (v = 0; v < V; v++) G->adj[v] = NULL;
  return G;
}
```
Hamilton Path

- Hamilton path:
  - Given two vertices, is there a simple path connecting them that visits every vertex in the graph exactly once?

- Worst case for finding Hamilton tour is exponential
  - Assume one vertex isolated; and all v-1 vertices are connected
  - (v-1)! Edges need to be checked
Euler Tour/Path

- Euler Path
  - Is there a path connecting two vertices that uses each edge in the graph exactly once?
  - Vertices may be visited multiple times
- Euler tour: Is there a cycle with each edges exactly once
  - Bridges of konigsberg
- Properties:
  - A graph has a Euler tour iff it is connected and all the vertices are of even degree
  - A graph has a Euler path iff it is connected and exactly two of its vertices are of odd degrees
- Complexity?
Graph Search

- Depth First Search

void dfsR(Graph G, Edge e)
{
    link t; int w = e.w;
    pre[w] = cnt++;
    for (t = G->adj[w]; t != NULL; t = t->next)
        if (pre[t->v] == -1)
            dfsR(G, EDGE(w, t->v));
}

#define dfsR search

void Graphs may not be connected

V^2 for adj matrix
V+E for adj. list

static int cnt, pre[maxV];

void GRAPHsearch(Graph G)
{
    int v;
    cnt = 0;
    for (v = 0; v < G->V; v++)
        if (pre[v] == -1)
            search(G, EDGE(v, v));
}
DFS for graph problems

- Cycle detection
  - Back edges
- Simple path
- Simple connectivity
  - The graph search function calls the recursive DFS function only once.
- Two way Euler tour
  - Each edge visited exactly twice
- Spanning tree
  - Given a connected graph with V vertices, find a set of V-1 edges that connects the vertices
    - Any DFS is a spanning tree
- Two coloring, bipartiteness check
Separability and Connectivity

- **Bridge**
  - An edge that, if removed, would separate a connected graph into two disjoint subgraphs.
  - Edge-connected graph – has no bridges
    - In a DFS tree, edge v-w is a bridge iff there are no back edges that connect a descendant of w to an ancestor of w
Separability and Connectivity

- Articulation point (separation/cut)
  - Removal results in at least two disjoint subgraphs
- $K$-connected - for each pair:
  - At least $k$ vertex disjoint paths
  - Indicates the number of vertices that need to be removed to disconnect a graph
  - Biconnected: $2$-connected
    - Removal of a vertex does not disconnect
- $K$-edge-connected - for each pair:
  - At least $k$ edge disjoint paths
  - Indicates the number of edges that need to be removed to disconnect a graph
BFS Search

- Instead of Stack
  - Use a Queue
- Can be used to solve
  - Connected components
  - Spanning tree
  - Shortest paths

```c
#define bfs_search
define bfs_search
void bfs(Graph G, Edge e)
{
    int v, w;
    QUEUEput(e);
    while (!QUEUEempty())
    {
        if (pre[(e = QUEUEget()).w] == -1)
        {
            pre[e.w] = cnt++; st[e.w] = e.v;
            for (v = 0; v < G->V; v++)
            {
                if (G->adj[e.w][v] == 1)
                {
                    if (pre[v] == -1)
                    {
                        QUEUEput(EDGE(e.w, v));
                    }
                }
            }
        }
    }
}
```
Directed Graph

- **Digraph: Vertices + directed edges**
  - In-degree: number of directed edge coming in
  - Out-degree: number of directed edge going out
  - DAG – no directed cycles
  - Strongly connected
    - Every vertex is reachable from every other
  - Not strongly connected: set of strong components

- **Kernel K(D) of digraph D**
  - One vertex of K(D) corresponds to each strong component of D
  - One edge in K(D) corresponds to each edge in D that connects vertices in different components
  - K(D) is a DAG
Reachability and Transitive closure

- **Transitive closure of a graph**
  - Same vertices + an edge from s to t in transitive closure if there is a directed path from s to t

**Warshall’s algorithm**

- Complexity: $V^3$

```c
void GRAPHtc(Graph G)
{
    int i, s, t;
    G->tc = MATRIXint(G->V, G->V, 0);
    for (s = 0; s < G->V; s++)
        for (t = 0; t < G->V; t++)
            G->tc[s][t] = G->adj[s][t];
    for (s = 0; s < G->V; s++)
        G->tc[s][s] = 1;
    for (i = 0; i < G->V; i++)
        for (s = 0; s < G->V; s++)
            if (G->tc[s][i] == 1)
                for (t = 0; t < G->V; t++)
                    if (G->tc[i][t] == 1) G->tc[s][t] = 1;
}
```

```c
void GRAPHtc(Graph G)
{
    int i, s, t;
    G->tc = MATRIXint(G->V, G->V, 0);
    for (s = 0; s < G->V; s++)
        for (t = 0; t < G->V; t++)
            G->tc[s][t] = G->adj[s][t];
    for (s = 0; s < G->V; s++)
        G->tc[s][s] = 1;
    for (i = 0; i < G->V; i++)
        for (s = 0; s < G->V; s++)
            if (G->tc[s][i] == 1)
                for (t = 0; t < G->V; t++)
                    if (G->tc[i][t] == 1) G->tc[s][t] = 1;
}
```
Topological Sort

- Given a DAG
  - Renumber vertices such that every directed edge points from a lower-numbered vertex to a higher-numbered one

![Diagram of a directed acyclic graph (DAG) with vertices renumbered to demonstrate topological sort. The left diagram shows the original graph with vertices labeled 0 to 8, and the right diagram shows the renumbered graph with vertices labeled 0 to 6. The renumbering is indicated by the relabeling process.]
Topological Sort

- Process each vertex before processing the vertices it points

- Reverse topological sort
  - Scheduling applications
  - Postorder numbering in DFS yields a reverse topological sort
Topological Sort

// Reverse (adj list)

static int cnt0;
static int pre[maxV];
void DAGts(Dag D, int ts[])
{
  int v;
  cnt0 = 0;
  for (v = 0; v < D->V; v++)
    { ts[v] = -1; pre[v] = -1; }
  for (v = 0; v < D->V; v++)
    if (pre[v] == -1) TSdfsR(D, v, ts);
}

// Adj. matrix
void TSdfsR(Dag D, int v, int ts[])
{
  int w;
  pre[v] = 0;
  for (w = 0; w < D->V; w++)
    if (D->adj[w][v] != 0)
      if (pre[w] == -1) TSdfsR(D, w, ts);
  ts[cnt0++] = v;
}
Minimum Spanning Tree

- Weighted graph
  - To incorporate this information into the graph, a weight, usually a positive integer, is attached to each arc
  - capacity, length, traversal time, or traversal cost.
Minimum Spanning tree (MST)

- A spanning tree whose weight (the sum of the weights in its edges) is no larger than the weight of any other spanning tree

- Representation
  - weighted graph using an adjacency matrix is straightforward – use an integer matrix
  - In the adjacency list representation, the elements of the list now have two components, the node and the weight of the arc
A graph and its MST
A Cut
- A partition of the vertices into two disjoint sets
- Crossing edge is one that connects a vertex in one set with a vertex in the other

Cut Property
- Given some cut in a graph, every minimal crossing edge belongs to some MST of the graph, and every MST contains a minimal crossing edge


- **Proof**: Suppose that on the contrary, there is no minimum spanning tree that contains $X$. Take any minimum spanning tree and add the arc $X$ to it.

A cycle is formed after adding $X$. 

![Diagram of a cycle formed after adding arc X](image)
Cycle Property

- Cycle property
  - Given a graph G, consider the graph G’ defined by adding an edge e to G
  - Adding e to an MST of G and deleting a maximal edge on the resulting cycle gives an MST of G’
Prim’s algorithm

- **Prim’s algorithm**
  - Step 1: \( x \in V \), Let \( A = \{x\} \), \( B = V - \{x\} \)
  - Step 2: Select \((u, v) \in E, u \in A, v \in B\) such that \((u, v)\) has the smallest weight between A and B
  - Step 3: \((u, v)\) is in the tree. \( A = A \cup \{v\}, B = B - \{v\} \)
  - Step 4: If \( B = \emptyset \), stop; otherwise, go to Step 2.

- **time complexity:**
  - \( O(n^2), n = |V| \).
Kruskal’s algorithm

- **Step 1**: Sort all edges
- **Step 2**: Add the next smallest weight edge to the forest if it will not cause a cycle.
- **Step 3**: Stop if we have \( n-1 \) edges. Otherwise, go to Step 2.

<table>
<thead>
<tr>
<th>Edge</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2)</td>
<td>10</td>
</tr>
<tr>
<td>(3,6)</td>
<td>15</td>
</tr>
<tr>
<td>(4,6)</td>
<td>20</td>
</tr>
<tr>
<td>(2,6)</td>
<td>25</td>
</tr>
<tr>
<td>(1,4)</td>
<td>30</td>
</tr>
<tr>
<td>(3,5)</td>
<td>35</td>
</tr>
</tbody>
</table>
The shortest path problem has several different forms:

- Given two nodes A and B, find the shortest path in the weighted graph from A to B.
- Given a node A, find the shortest path from A to every other node in the graph. (single-source shortest path problem)
- Find the shortest path between every pair of nodes in the graph. (all-pair shortest path problem)
Shortest Path

- Visit the nodes in order of their closeness;
  - visit A first, then visit the closest node to A,
  - then the next closest node to A, and so on.
- Dijkstra’s algorithm

![Graph with labeled nodes and edges]

<table>
<thead>
<tr>
<th>Path</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_0v_2$</td>
<td>10</td>
</tr>
<tr>
<td>$v_0v_2v_3$</td>
<td>25</td>
</tr>
<tr>
<td>$v_0v_2v_3v_1$</td>
<td>45</td>
</tr>
<tr>
<td>$v_0v_4$</td>
<td>45</td>
</tr>
</tbody>
</table>
To select the next node to visit, we must choose the node in the fringe that has the shortest path to A. The shortest path from the next closest node must immediately go to a visited node.