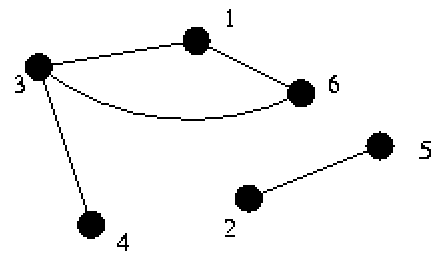
Programming in C/C++

Patrick Ho
peiqistar@gmail.com

(11/15/2014)

Graph

- Consists of a finite set of vertices or nodes (V) and a set of edges(E). G = (V , E).
- Each vertex could be any abstract object in real problem, like city, chess-board location, farm, etc.
- Each edge is a relation (adjacency) between two vertices. Edge could have weight value to associate with it (cost, distance length, etc).



In this example graph, V = {1, 2, 3, 4, 5, 6} and E = {(1,3), (1,6), (2,5), (3,4), (3,6)}.

Graph Terminology

- *self-loop*: edge from vertex is same as end vertex
- *simple graph*: no self-loop, no repeat edge
- edge (u,v) is *incident* to both vertex u and vertex v
- *degree* of a vertex is the number of edges which are incident to it
- vertex u is *adjacent* to vertex v if there's edge between them
- *sparse* graph if the total number of edges is small compared to the total number possible ((*N x (N-1))/2*)
- *directed graph*, in which case the edges have a direction, usually use add a flag in edge data structure
- *out-degree* of a vertex is the number of arcs which *begin* at that vertex; *in-degree* of a vertex is the number of arcs which *end* at that vertex

Graph Paths

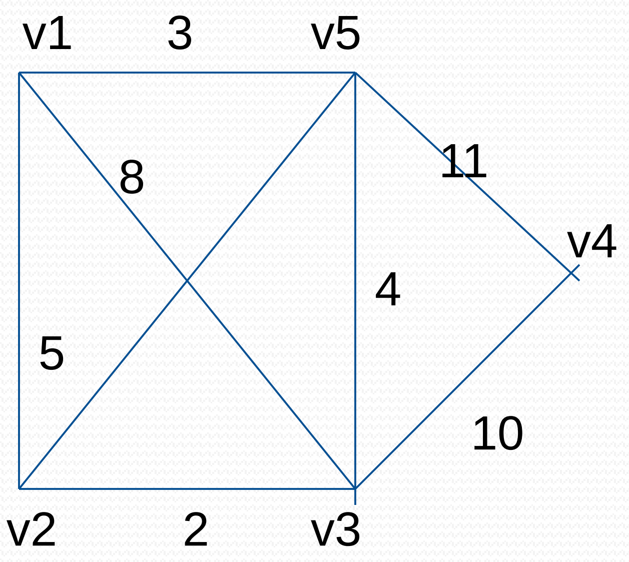
- A *path* from vertex *u* to vertex *x* is a sequence of vertices $(v_0, v_1, ..., v_k)$ such that $v_0 = u$ and $v_k = x$ and (v_0, v_1) is an edge in the graph, as is (v_1, v_2), (v_2, v_3), etc. The length of such a path is *k*.
- A path is *simple* if it contains no vertex more than once.
- A path is a *cycle* if it is a path from some vertex to that same vertex.

Graph Presentation (store data structure)

1. Edge List:
   keep a list of the pairs of vertices representing the edges in the graph.

e.g.

Edge#:	1	2	3	4	5	6	7	8
Vstart:	1	1	1	2	2	3	3	4
Vend:	2	3	5	3	5	4	5	5
weight:	5	8	3	2	6	10	4	11



Code using data structure + array/vector/list

Graph Presentation (store data structure)

2. Adjacency Matrix

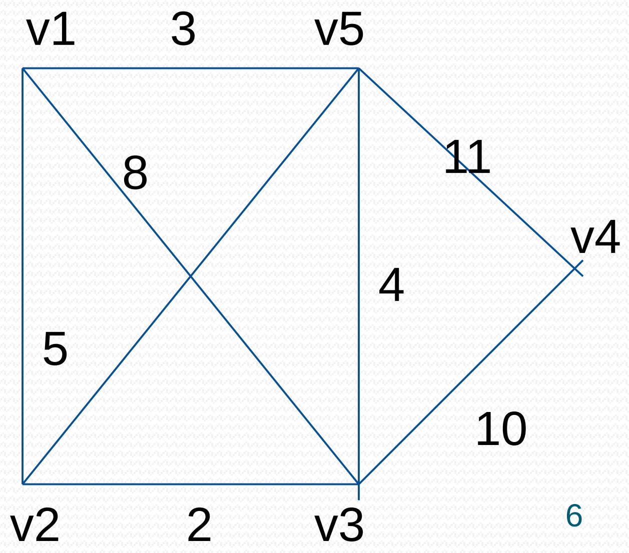
Matrix is NxN size (N is the number of vertex)

edge connection is the value of (i, j)

$$G[i,j] = \begin{cases} \text{1 or weight value, when i to j has edge} \\ \text{0 or -1, when i has no edge} \end{cases}$$

e.g.

$$G(V,E)= \begin{matrix} -1 & 5 & 8 & -1 & 3 \\ 5 & -1 & 2 & -1 & 6 \\ 8 & 2 & -1 & 10 & 4 \\ -1 & -1 & 10 & -1 & 11 \\ 3 & 6 & 4 & 11 & -1 \end{matrix}$$



Code using 2D array

Undirected graph is symmetric

Graph Presentation (store data structure)

3. Adjacency List

Given a index for each vertex; use a data structure and self pointer to link next object; one array for start point; on link list for each end point + weight

e.g.

$1 \rightarrow 2|5 \longrightarrow 3|8 \longrightarrow 5|3$

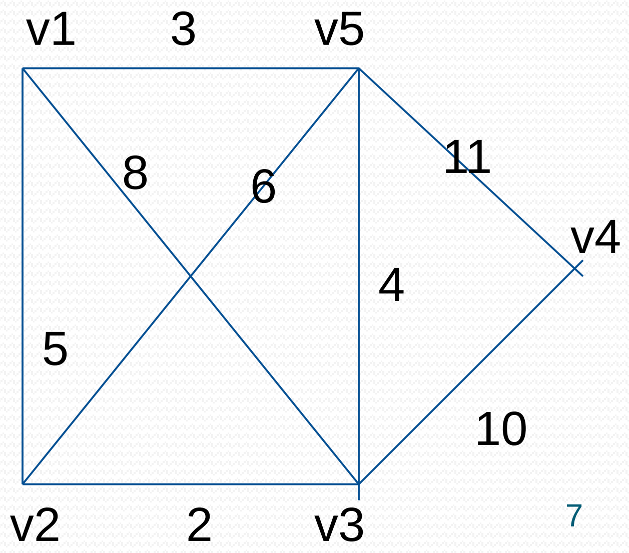
$2 \rightarrow 1|5 \longrightarrow 3|2 \longrightarrow 5|6$

$3 \rightarrow 1|8 \longrightarrow 2|2 \longrightarrow 4|10 \longrightarrow 5|4$

$4 \rightarrow 3|10 \longrightarrow 5|11$

$5 \rightarrow 1|3 \longrightarrow 2|6 \longrightarrow 3|4 \longrightarrow 5|11$

Code data struct + vector/list
Easy for editing graph



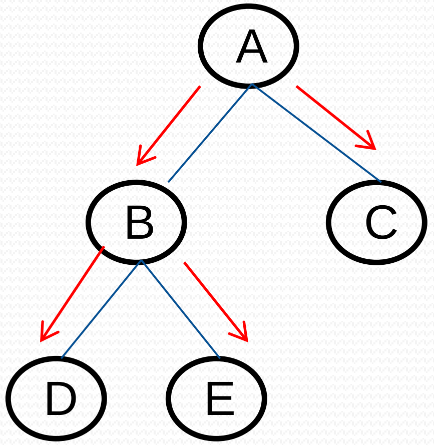
Graph Connectedness

- undirected graph is *connected* if there is a path from every vertex to every other vertex
- *component* is a maximal subset of the vertices such that every vertex is reachable from each other vertex in the component.
- directed graph is *strongly connected* if there is a path from every vertex to every other vertex.
- *strongly connected component* of a directed graph is a vertex u and the collection of all vertices v such that there is a path from u to v and a path from v to u.

Graph Search/Traverse/Visit

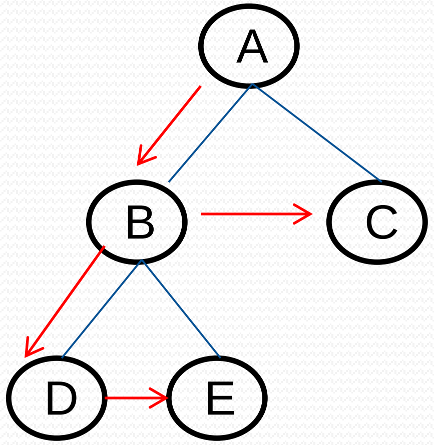
- Depth-First Search (DFS)

  starts at the root and explores as far as possible along each branch before backtracking.



- Breadth-First Search (BFS)

  begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on.

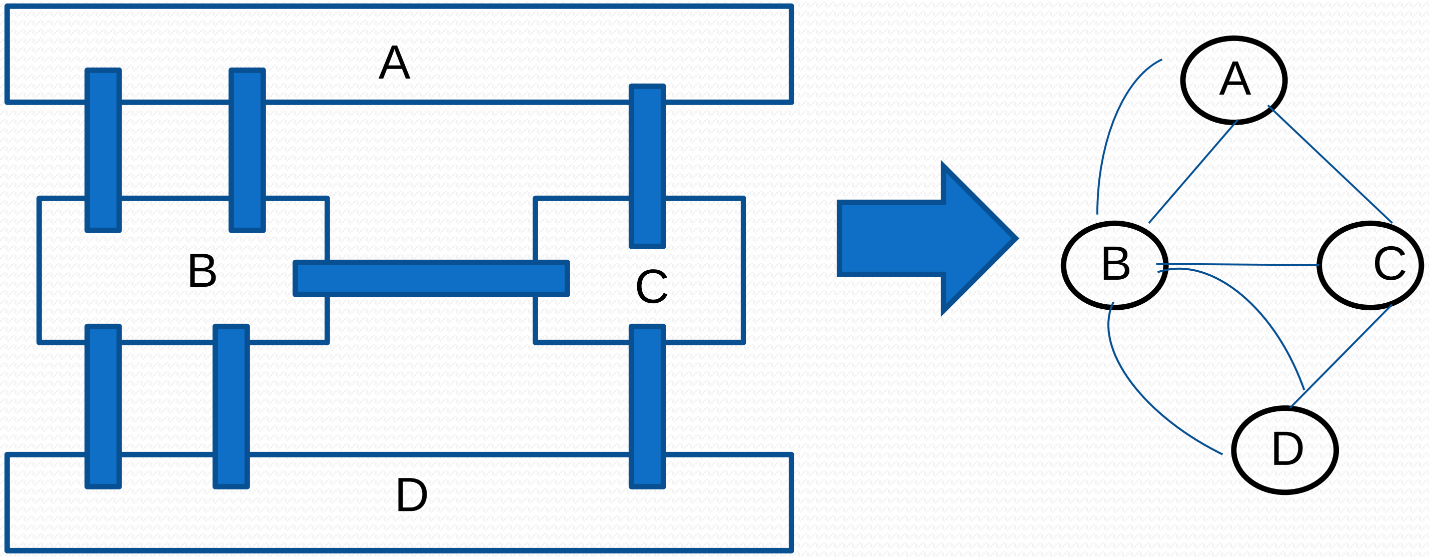


- Implement/Coding use Recursion

More Graph Problems (1)

- **Problem:**

There are 4 big cities(A, B, C, D) and 7 bridges are built in following map.
Is it possible that a man walk through each city, go through each bridge once and come back to his start city?



Eulerian Graph Problem

- **Solution:**
  - Form a Graph
    - Vertices (Nodes): city (4)
    - Edges: bridge (7)
  - Graph Theory (Typical Eulerian Graph problem)
    - Every vertex has an even degree
  - Solution:
    - Traverse each vertex
    - Count its connected edges (Degree)=> Odd
    - Answer: NO

      Euler Tours algorithm

More Graph Problems (2)

- **Problem:**
  - determines the area connected to a given node in a multi-dimensional array
  - check how many connected sub-graph/component
  - check whether moving A to B is posiible

Flood Fill Algorithm

- **Solution:**
  - start node, target color, replacement color

  Look for all nodes in array which are connected to the start node by a path of the target color, and changes them to the replacement color
  - many ways of implementation
  - DFS /BFS

Graph DFS (same in BackTracking)

- **DFS pseudocode:**
  - Recursion

```
dfs (node)
    for each child of node
        dfs (node)
```

Limitation: some problem will have stack overflow if recursion level is too deep

  - Iterating using STL stack

```
stack.push(root)
while stack not empty
    node = stack.top();
    stack.pop();
    for each child of node
        stack.push(child);
```

Graph (BFS)

- **BFS pseudocode:**
  - Using STL queue

```
queue.push(root)
while queue not empty
    node = queue.front();
    queue.pop();
    for each child of node
        queue.push(child);
```

Flood Fill (DFS)

- **Recursion code:**

```
// x, y is the coordinate of G[x][y]
// t_color is init color
// r_color is set color

Void FloodFill(int x, int y, int t_color, int r_color) {
    if (x < 0 || x>= size_x) return;
    if (y < 0 || y >= size_y) return;
    if (G[x][y] != t_color) return; // already filled
     G[x][y] = r_color;
    // for each direction (4 or 8) call FloodFill again
    FloodFill(x+1, y, t_color, r_color);
    FloodFill(x-1, y, t_color, r_color);
    FloodFill(x, y+1, t_color, r_color);
    FloodFill(x, y-1, t_color, r_color);
}
```

Flood Fill (DFS)

- **Iterating using STL stack :**

```
// x, y is the coordinate of G[x][y]
// t_color is init color
// r_color is set color
class node { public: int x; int y;}

Void FloodFill(int x, int y, int t_color, int r_color) {
    stack<node> s;
    node n1; n1.x = x; n1.y=y;
    s.push(n1);
    while(!s.empty()) {
        node n = s.top(); s.pop();
        if (n.x<0 || n.x>= size_x) continue;
        if (n.y<0 || n.y>= size_y) continue;
        if (G[n.x][n.y] != t_color) continue;
        G[n.x][n.y] = r_color;
        // for each direction (4 now)
        n1.x = n.x+1; n1.y=n.y; s.push(n1);
        n1.x = n.x-1; n1.y=n.y; s.push(n1);
        n1.x = n.x; n1.y=n.y+1; s.push(n1);
        n1.x = n.x; n1.y=n.y-1; s.push(n1);
    }
}
```

Flood Fill (BFS)

- **Iterating using STL queue:**

```
// x, y is the coordinate of G[x][y]
// t_color is init color
// r_color is set color
class node { public: int x; int y;}

Void FloodFill(int x, int y, int t_color, int r_color) {
    queue<node> q;
    node n1; n1.x = x; n1.y=y;
    q.push(n1);
    while(!q.empty()) {
        node n = q.front(); s.pop();
        if (n.x<0 || n.x>= size_x) continue;
        if (n.y<0 || n.y>= size_y) continue;
        if (G[n.x][n.y] != t_color) continue;
        G[n.x][n.y] = r_color;
        // for each direction (4 now)
        n1.x = n.x+1; n1.y=n.y; q.push(n1);
        n1.x = n.x-1; n1.y=n.y; q.push(n1);
        n1.x = n.x; n1.y=n.y+1; q.push(n1);
        n1.x = n.x; n1.y=n.y-1; q.push(n1);
    }
}
```

More Graph Problems (3)

- **Problem:**

There are n computers. Each wire connects 2 computers. Each wire has a cost. Make a network with lowest cost to connect each computer

Input: 1st is a integer of n. Rest of line is a graph adjacency matrix
e.g.

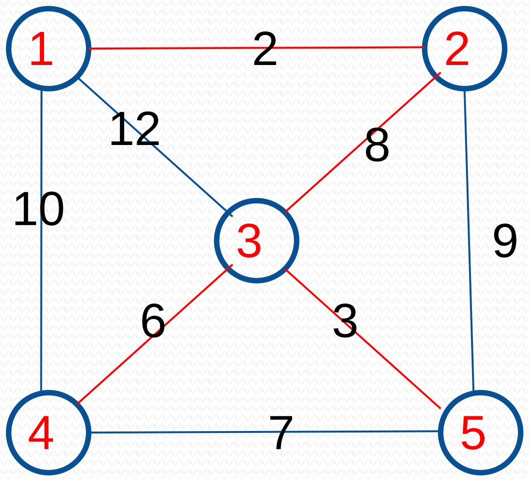
```
3
0  1  2
1  0  1
2  1  0
```

Output: minimum cost needed
e.g.

```
2
```

Minimum Spanning Tree (MST)

- spanning tree of a graph is a sub-graph that is a tree and connects all the vertices together
- a connected undirected graph with weighted edge
- a minimal spanning tree is a spanning tree which has minimal `cost' (where cost is the sum of the weights of the edges in the tree)



MST (Prim Algorithm)

The algorithm continuously increases the size of a tree, one edge at a time, starting with a tree consisting of a single vertex, until it spans all vertices.

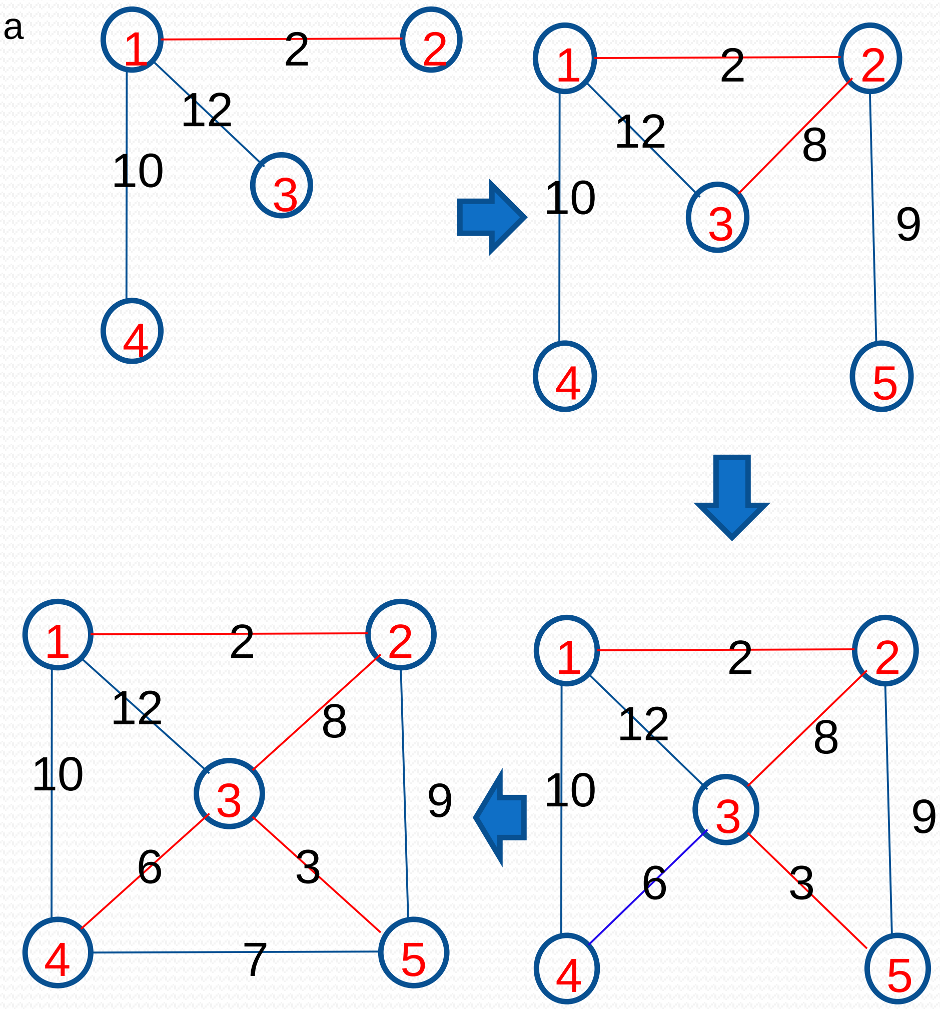
Input: A non-empty connected weighted graph with vertices *V* and edges *E* (the weights can be negative).

Initialize: $V_{new} = \{x\}$, where *x* is an arbitrary node (starting point) from *V*, $E_{new} = \{\}$

Repeat until $V_{new} = V$:

    Choose an edge (*u*, *v*) with minimal weight such that *u* is in V_{new} and *v* is not (if there are multiple edges with the same weight, any of them may be picked)
    Add *v* to V_{new}, and (*u*, *v*) to E_{new}

Output: V_{new} and E_{new} describe a minimal spanning tree



MST (Kruskal Algorithm)

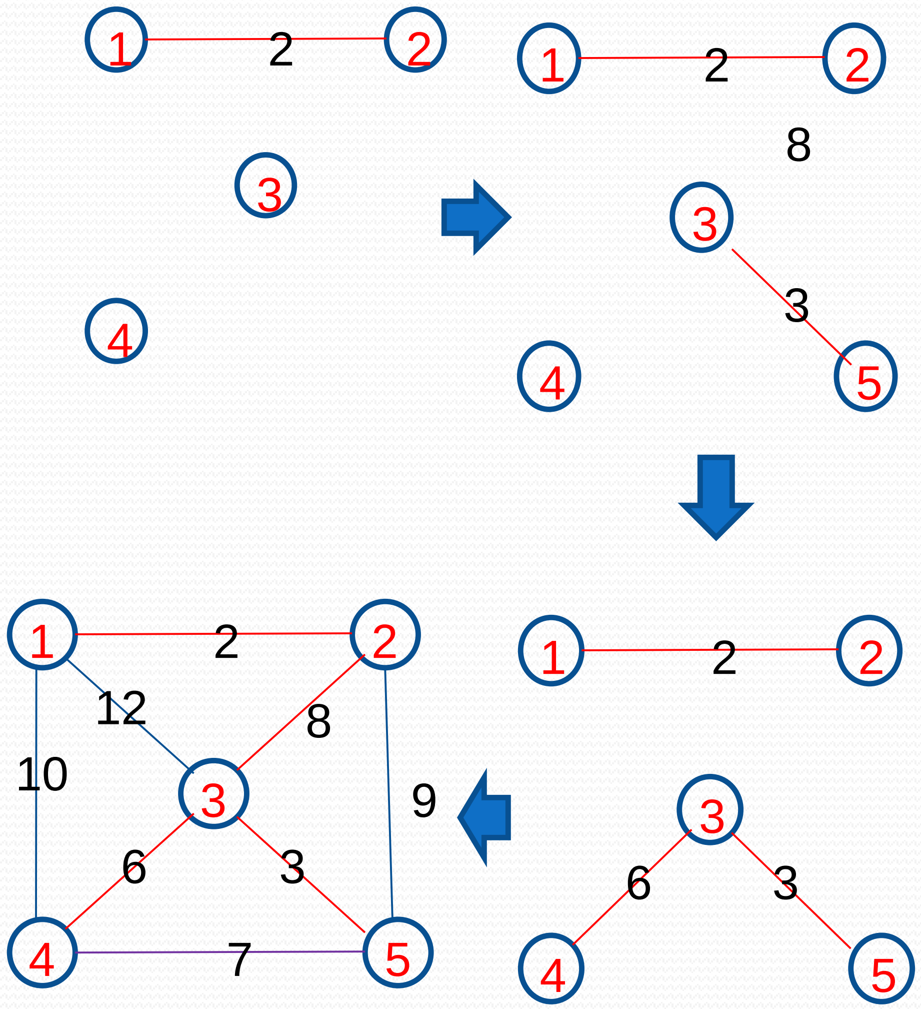
```
create a forest F (a set of trees), where
each vertex in the graph is a separate
tree

create a set S containing all the edges
in the graph

while S is not empty and F is not yet
spanning
    remove an edge with minimum
    weight from S

    if that edge connects two different
    trees, then add it to the forest,
    combining two trees into a single
    tree

    otherwise discard that edge.
```



Find Shortest Path

Dijkstra Algorithm

1. Assign to every node a distance value: set it to zero for initial node and to infinity(999, or INT_MAX, etc) for all other nodes. Mark all nodes as unvisited.
   2. Set initial source as current.
   3. Consider all its unvisited neighbors and calculate their distance. For example, if current node (A) has distance of 1, and an edge connecting it with another node (B) is 2, the distance to B through A will be 1+2=3
   4. When we are done considering all neighbors of the current node, mark it as visited. A visited node will not be checked ever again; its distance recorded now is final and minimal
   5. The next *current node* will be the node with the lowest distance in the *unvisited set* that is the topmost element of the priority queue

2. For positive weight edge problem

Find Shortest Path:

Dijkstra Algorithm

Input:

1st line is total node number, n(n is from 1)

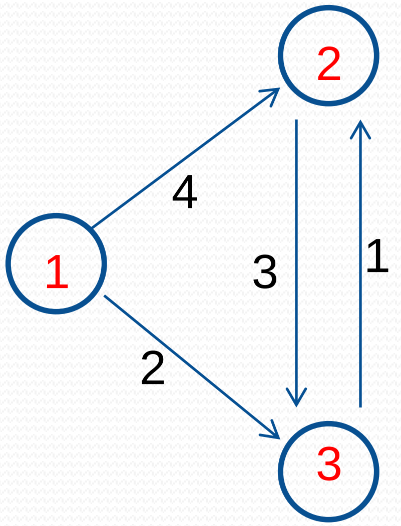
2nd line is total edge number, e

3 to e+2 line is edge start node, end node and weight(positive)

e+3 line is the start node

e.g. A

```
3
4
1 2 4
1 3 2
2 3 3
3 2 1
1
```



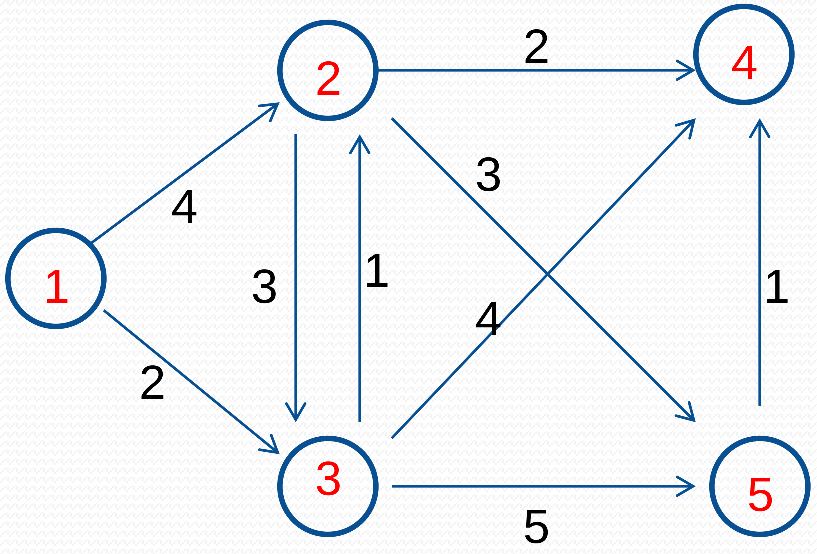
Find Shortest Path:

Dijkstra Algorithm

Input:
1st line is total node number, n(n is from 1)
2nd line is total edge number, e
3 to e+2 line is edge start node, end node and weight(positive)
e+3 line is the start node

e.g.

```
5
9
1 2 4
1 3 2
2 3 3
3 2 1
2 4 2
3 4 4
5 4 1
2 5 3
3 5 5
1
```



Find Shortest Path:

Dijkstra Algorithm

Output:
each line vertex number and its min weight from start node

Node 1, min weight = 0
Node 2, min weight = 3
Node 3, min weight = 2
Node 4, min weight = 5
Node 5, min weight = 6

Coding:
Using priority_queue

Find Shortest Path

Floyd-Warshall Algorithm

1. It compares all possible paths through the graph between each pair of vertices. It requires an adjacency matrix containing edge weights, the algorithm constructs optimal paths by piecing together optimal subpaths.

2. It belongs Dynamic Programming Algorithm.

3. See reference in, http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

4. It is very fast to get all pairs shortest path, if memory is not issue to store graph in adjacent matrix way.

5. The problem graph can be positive and negative weight, but no negative weight cycles.

Find Shortest Path

Floyd-Warshall Algorithm

Sample code:

```
int dist[N][N]; // For some N, use adjacent matrix
// Input data into dist, where dist[i][j] is the distance
   from i to j.
int i, j, k;
 for ( k = 0; k < N; k++ )
    for ( i = 0; i < N; i++ )
        for ( j = 0; j < N; j++ )
              dist[i][j] = min( dist[i][j], dist[i][k] + dist[k][j] );
```