Basic Graph Theory Algorithm

Definitions and Notations:

A graph G = (V, E) consists of a set of vertices V, and a set of edges E. In an **undirected** graph, we represent an edge by a pair of vertices (u, v) indicating that vertex u and v are connected by an edge. In the case of a **directed** graph, we represent an edge by an **ordered** pair (u, v) indicating a directed edge from vertex u to v.

When analyzing the runtime of algorithm, we will use V to indicate the number of vertices in the graph and E to indicate the number of edges in the graph. In the pseudo-codes, we assume that the vertices are labeled from 0…V-1.

A graph may be **weighted** - each edge has a weight associated with it.

Graph Representation:

There are two ways to store the graph in your program:

1) Adjacency Matrix
   - Use a 2D array `bool graph[][]` to represent the graph.
   - `graph[u][v] = true` iff (u, v) is an edge in the graph
   - **Cons**: use lots of memory, take O(V) to check all edges leaving a vertex.
   - **Pros**: can check existence of edge in O(1) time.
2) Adjacency List
   - Use a `vector< vector<int> > graph` to represent the graph.
   - `graph[u]` is a vector storing all vertex v for which the edge (u, v).
   - **Cons:** slow to check on the existence of an edge in the graph
   - **Pros:** low memory usage, fast to check all edges leaving a vertex.

Graph Exploration:

The most basic thing we want to do with a graph is to explore all of its vertices from some **source** vertex s. This is useful for many purposes such as checking the connectivity of the graph. There are two basic algorithms:

1) Depth First Search
   - A recursive algorithm that explore each branch of a graph as much as possible.
   - At a vertex v, we explore all edges e = (v, u) that leaves v and recurse on the vertex u.
   - After all edges are explored, we backtrack to the vertex where we came from.
   - In the main function, we call `dfs(s)`.

```
bool seen[];
void dfs( int v ) {
     seen[v] = true;
     for ( each edge e = (v, u) leaving from v )
          if ( !seen[u]  ) dfs( u );
}
```

2) Breadth First Search
   - Explore the vertices of in the sequence of when we first "discover" it.
   - We maintain a queue that stores all nodes that we've discovered but not yet processed. At each stage, we process the node at the front of the queue.

- When we process node v, we add all newly discovered nodes reachable from v to the end of the queue we are maintaining.
- At the very beginning, only the source vertex is in the queue.

```
bool seen[];
void bfs( int s ) {
     seen[s] = true;
     queue<int> q;
     q.push(s);
     while ( !q.empty() ) {
          int v = q.front();
          q.pop();
          for ( each edge e = (v, u) leaving from v ) {
               if ( !seen[u] ) {
                    q.push(u);
                    seen[u] = true;
               }
          }
     }
}
```

Both BFS and DFS visit exactly each edge and vertex in the graph once. So the runtime of the algorithm is $O(V + E)$ if using an adjacency list or $O(V^2)$ if using an adjacency matrix. In terms of exploring the graph, both DFS and BFS are equally efficient. However, note that the memory usage of BFS depends on the **density** of the graph while the memory usage of DFS depends on the **depth** of the graph.

Beyond graph explorations, BFS and DFS have many other applications. One of the main applications for BFS is to find the shortest path from the source vertex in an **unweighted** graph. The algorithm makes use of the fact that at every stage, the vertex being process is actually the one closest to the source among all the un-processed vertices. Implementation as follows:

```
int dist[];
int parent[];
void bfs(int s) {
     for ( all vertex v ) dist[v] = infinity;
     for ( all vertex v ) parent[v] = -1;
     dist[s] = 0;
     queue<int> q;
     q.push(s);
     while ( !q.empty() ) {
          int v = q.front();
          q.pop();
          for ( each edge e = (v, u) leaving v ) {
               if ( dist[u] == inifinity ) {
                    dist[u] = dist[v] + 1;
                    parent[u] = v;
                    q.push(u);
               }
          }
     }
}
```

DFS also has many other applications such as **Bridge Detection**, **Articulation Vertex**, and **Strongly Connected Component**. We will not go into the details here.

Shortest Path:

We have already seen that the shortest path in an unweighted graph can be found by BFS. However, in the case of a weighted graph (where the shortest path is now defined as the path in which the sums of the weight of edges in the path is minimal), BFS will output an incorrect result. This is because the vertex in the front of the queue is guaranteed to be closest to the source vertex in terms of number of edges, not the weight of the path.

Assuming the graph has NO **negative weight**, the shortest path can be found by **Dijkstra's Algorithm**. The idea of the algorithm is very similar to BFS in the sense that we maintain a list of vertices that we have discovered and at every stage, we process the vertex that is closest to the source in terms of path weight. So for each vertex in the list, we keep track of the minimum path weight among all the paths from the source we've found so far. When we process a vertex v, we explore every edge e = (v, u) that leaves v and considers the new path from the source to u using (v, u) as the last edge. Based on the weight of the new path, we update the minimum path weight of vertex u. This can be implemented using by defining a comparator and using a set (which is sorted in C++):

```
int dist[];
int parent[];
struct cmp {
     bool operator()(const int& u, const int& v) {
          if ( dist[u] == dist[v] ) return u < v;
          else return dist[u] < dist[v];
     }
}

void dijkstra( int s ) {
     for ( all vertex v ) dist[v] = infinity;
     for ( all vertex v ) parent[v] = -1;
     dist[s] = 0;

     set<int, cmp> q;
     q.insert(s);
     while ( !q.empty() ) {
          int v = *q.begin();
          q.erase(q.begin());

          for ( each edge e = (v, u) leaving v ) {
               int d = dist[v] + weight(v, u);
               if ( d < dist[u] ) {
                    q.erase(u);
                    dist[u] = d;
                    q.insert(u);
                    parent[u] = v;
               }
          }
     }
}
```

This implementation has run time O((V + E) log V) if using an adjacency list. If the implementation uses an adjacency matrix, then the run time is $O(V^2 + (V + E) \log V)$. This is usually the preferred implementation for sparse graph. However, when the graph is dense (ie. E is approximately equals to V^2), then the run time is about $O(V^2 \log V)$. In this case, we can use an alternative approach to the implementation:

```
int dist[];
int parent[];
bool finished[];

void dijkstra(int s) {
     for ( all vertex v ) {
          dist[v] = infinity;
          parent[v] = -1;
          finished[v] = false;
     }

     dist[s] = 0;
     int numFinished = 0;
     while ( numFinished < V ) {
          int v = -1;
          int best = infinity;
          for ( all vertex u ) {
               if ( !finished[u] && dist[u] < best ) {
                    v = u;
                    best = dist[u];
               }
          }

          for ( each edge e = (v, u) leaving v ) {
               if ( dist[u] > dist[v] + weight(v, u) ) {
                    dist[u] = dist[v] + weight(v, u);
                    parent[u] = v;
               }
          }

          finished[v] = true;
          ++numFinished;
     }
}
```

In this implementation, instead of using a set to keep track of which vertex we should process next, we simply do a linear search through the array of vertices. The runtime of this algorithm is $O(V^2)$, which is faster than $O(V^2 \log V)$ and so is better than the first implementation for dense graphs.

Dijkstra's algorithm will only work on edges with no negative weight. If the graph has negative weight, then the algorithm will output an incorrect result. In such a case, we employ the **Bellman-Ford** algorithm, which will not be discussed here.

Minimum Spanning Tree (MST):

A tree is an undirected, connected, acyclic graph. Given an undirected graph G = (V, E), a spanning tree of G is a tree G' = (V, E'), where E' is a subset of E. The weight of a spanning tree is defined to be the sum of all the weight of the edges in the tree. The MST of G is the spanning tree with the minimum weight. There are two algorithms to find the MST: **Prim's Algorithm** or **Kruskal's Algorithm**. We will only discuss Prim's algorithm here.

Prim's algorithm is actually very similar to Dijkstra's algorithm: we start with a source vertex (which is arbitrary in this case), maintain a list of vertices we've discovered, and process the vertex with the smallest "distance" next. The main difference is how we maintain the dist array in the algorithm. In Dijkstra's algorithm, dist[u] is the minimal path weight among all paths we've found so far; in Prim's algorithm, dist[u] is the minimum weight of all edges we have found so far that has the vertex u as one of the endpoints.

```
int dist[];
struct cmp {
     bool operator()(const int& u, const int& v) {
          if ( dist[u] == dist[v] ) return u < v;
          else return dist[u] < dist[v];
     }
}

void dijkstra( int s ) {
     for ( all vertex v ) dist[v] = infinity;
     dist[s] = 0;

     set<int, cmp> q;
     q.insert(s);
     while ( !q.empty() ) {
          int v = *q.begin();
          q.erase(q.begin());

          for ( each edge e = (v, u) leaving v ) {
               if ( weight(v, u) < dist[u] ) {
                    q.erase(u);
                    dist[u] = weight(v, u);
                    q.insert(u);
               }
          }
     }
}
```

Similar to the Dikstra's algorithm, this implementation has run time O((V + E) log V). There is also an analogous $O(V^2)$ implementation which can be useful for dense graph.