**Part 01: introduction to graph.**

**Linear or sequential data structure:** Array,linked list,stack,queue.

**Non-linear data structure(hierarchical):** Tree,graph.

**Graph:**

* A graph is an ordered pair of a set v of vertices and a set of edges. G=(V,E).
* its just like a tree, is a collection of objects or entities tha we called nodes or vertices connected to each other through a set of edges.
* Edges can connect nodes in any possible way.
* No spesefic rules dectating the connection among the nodes.

**Tree:**

* connections are bound to be in a certain way. In a tree there are rule dectating the connection among the nodes.
* Tree with 1 nodes have exactly (n-1) edges.

One edge for each parent child relationship.

* Except root node every nodes have exactly one parent.
* Every nodes can be reachable from root a nd there axactly one possible way to the node from root.
* Tree is only a special kind of graph.

**Types of pair:**

* Ordered pair: (a,b)!=(b,a) if a!=b.
* Ordered pair: (a,b)==(b,a)

**Types of Edges:**

1. **directed.** Connection is one way. Edges are unidirectional. (u,v)→ first element is origin and second one is destination. So, (u,v)!=(v,u).
2. **Underected.** Connection is two way. Edges are bidirectional. (u,v)→ origin and destination are not fixed. So, (u,v)==(v,u).

***NOTE:*** *Typically a graph can be directed or undirected. Its possible thal a graph can have both of them.*

1. **Self-loop or self-edge.** Involves only one vertex. Both points of edge origin and destination is same. In can be both directed and undirecte. Ex: if a web page have link to itself or semiler scenarios.
2. **Multi-edge.** Involve two vertices. Can be both directed and undirected. Ex: consider about a flit connection among cities. where can be different flit to travel between two city. The flites can differ in cost, time etc. so their weights are not same . So, we need multiple edges between two cities to represent the scenario.

***NOTE:*** *if a graph have self loop or multi edge than we need to take extra care while solving problems. So, this type of edge complicates the graphs.*

**Simple Graph:** if a graph contains no self-loop and multi-edge than it is a simple graph.

**Degree of nodes:**

* In case of directed graphs, number of edges going into a node is known as **in degree** of the corresponding node and number of edges coming out of a node is known as **outdegree** of the corresponding node. For any graph sum of totalindegree should be equal to total outdegree.
* In undirected graph there is **no difference** between in degree and out degree of node. The **number of total adjacent nodes** is the degree of the node.

**Handshaking lemma:**

* Number of odd degree nodes in a graph is always even.
* The sum of all degree of nodes is double to the number of edges.
* The degree sum formula implies that every *r*-[regular graph](https://en.wikipedia.org/wiki/Regular_graph) with *n* vertices has *nr*/2 edges. In particular, if *r* is odd then the number of edges must be divisible by *r*.
* In [graph theory](https://en.wikipedia.org/wiki/Graph_theory), a regular graph is a [graph](https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)) where each vertex has the same number of neighbors; i.e. every vertex has the same [degree](https://en.wikipedia.org/wiki/Degree_(graph_theory)) or valency.

**Directed vs undirected graph:**

* a directed graph or digraph has directed or unidirectional edges with ordered pair of nodes.
* an undirected graph has undirected or bidirectional edges with unordered pair of nodes.
* We can draw an undirected graph as directed. Each undirected edges will have two directed edges.
* We cant draw a directed graph as undirected.

**Weighhted vs unweighted graph:**

* sometimes all connections can not be treated as equal. Some connection can be prfarable to others. Like inter city road network.
* In a wighted graph the edges are weighted as preferance.
* in unweighted graph edges are unweighted or an weighted graph where all edges have 1 unit of weight.

**Exaple:**

1. social network is an **unweighted,undirected bidirectional graph.**
2. Interlinked web Web pages or WWW can be represented as **unweighted, directed,**

**unidirectional graph.**

1. Inter city road network can be represented as **weighted, undirected bidirectional graph.**
2. Inter city road network can be represented as **weighted, directed unidirectional graph.**

**Resource:**

1. <https://www.youtube.com/watch?v=gXgEDyodOJU&list=PL2_aWCzGMAwI3W_JlcBbtYTwiQSsOTa6P&index=38>
2. <http://www.shafaetsplanet.com/?p=143>

**Part 02: Properties of graph.**

**Number of edges:**

* **Mimimum nuber of edges** of graph with n vertices can be **zero(0)** for both **directed** and **undirected** graph. Because there can be no connection between the nodes. So, E={}.
* **Maximum number of edges** of a **simple directed** graph with n vertices would be n(n-1).

As, we can have maximum two directed edges between two nodes.

If, **|v|=n then, 0 <= |E| <= n(n-1).**

* **Maximum number of edges** of an **simple** **undirected** graph with n vertices would be

{n\*(n-1)}/2. As, we can have maximum one bidirected edges between two nodes.

If, **|v|=n then, 0 <= |E| <= n(n-1)/2.**

***NOTE:*** *This theories are true only when there is no self-loop or multi-edge in the graph.*

**Vertices vs edges:**

* from last portion we can conclude that number of edges would be close to square of vertices. So, |E| ∞ |V|^2.

EX: if, |V|=10, |E|<=90.

if, |V|=100, |E|<=9900

**Dense graph:**

* if the number of edges is close to maxmum possible number of edges. That is the number of edges of the order square of vertices.

**Parse graph:**

* if the number of edges are realy less. Tipically close to number of vertices, and not more than that.

***NOTE:*** *actually there is no boundary between dense and parese graph. It all depend on the context.But these are important to take important dicitions,* ***like we tipically store a dense graph in adjecency matrix and parse graph in adjacency lists****.*

**Walk/Path:**

* a sequence of vertices where agjacent pair is cinnected by edge. The vertices and edges can be repeated. Like: <A-B-F-H-E-B-A-D>. where node A and B, edge A-B are repeated. Edges are undirected.

**Simple path:**

* a walk in which no vertices (and thus no edges) are repeated. Like: <A-B-F-H>.

**Trail:**

* a walk in which no edges can be repeated and edges can be. Like: < A-B-F-H-D-A-C>.

***Note:*** *walk and path are often used as senonym. But most often when we say path we mean simple path. Between two vertices if there is an walk where vertices and edges are repeated there must also be a simple path.*

*Ex: walk < A-B-F-H-D-A-C > and simple path <A-C>.*

**Strongly Connected graph:**

* if there is a path from any vertex to any other vertex the graph is strongly connected.
* If its an **undirected** graph we simply call it **connected**.
* if its an **directed** graph we simply call it **Strongly connected**.
* If a **direected** graph is **not strongly connected** but can be turnd into **connected** by treating the **edges as undirected** than such a directed graph called **weackly connected**.

**Closed walk/Cycle:**

* starts and end at same vertex and the length > zero (must). Length = number of edges in the walk/path.

Like: <A-B-E-H-D-A>. length = 5.

**Simple Cycle:**

* No repeation other than start and end (thus edges cant be repeated).  **Like:** <A-B-E-H-D-A>.

***Note:*** *Some may call closed walk as cycle but generally* ***we use the term cycle for simle cycle.***

**Acyclic graph:**

* A graph with no cycle.
* An **undirected tree** can be treated as **undirected acyclic graph**. Because here we have closed walk but no simple cycle. Ex: if B is a left child of root A than we have a closed walk <A-B-A>. A repeated two times and edge A-B(undirected) repeated two times. So its not a simple cycle as well as cycle.
* We can also have **Directed Acyclic Graph(DAG)**.
* Cycles creates issues to find the sortest rout to ane vertex to another.

**Resources:**

1. <https://www.youtube.com/watch?v=AfYqN3fGapc&list=PL2_aWCzGMAwI3W_JlcBbtYTwiQSsOTa6P&index=39>

**Part 03: Graph Representatio\_01 (**Vertex list Edge List Representation)

**code:**

|  |  |
| --- | --- |
| Struct Edge  {  int startvertex;  int endvertex;  int weight;  };  string vertex\_list[MAX\_SIZE];  Edge edge\_list[MAX\_SIXE]; | Or,  Class Edge  {  int startvertex;  int endvertex;  int weight;  };  string vertex\_list[MAX\_SIZE];  Edge edge\_list[MAX\_SIXE]; |

**Complexity:**

* Space Complexity : **O ( |V| + |E| ).**
* Time Complexity: **O ( |E| ).**  ( some operations like taking all nodes adjacent to a given node or find if two given nodes are connected, we have to travers or linear search all elements in edge list.)
* But problem is maximum edges can be proportional to **|V|^2.**

**|v|=n then, 0 <= |E| <= n(n-1). →** for directed.

If, **|v|=n then, 0 <= |E| <= n(n-1)/2. →** for undirected.

* So, this representation is not good to choice as it is costly based on time complexity.

**Resources:**

1. <https://www.youtube.com/watch?v=ZdY1Fp9dKzs&list=PL2_aWCzGMAwI3W_JlcBbtYTwiQSsOTa6P&index=40>

**Part 04: Graph Representatio\_02** (Adjacency Matrix Representation)

**code:**

string vertex\_list [V]; → for vertices;

int adjacency\_matrix [V][V]; → As, maximum possible edges are proportional to V^2.

adjacency\_matrix[i][j]=1, if, there is an edge between I , j.

=0 , otherwise.

|  |  |
| --- | --- |
| **[Number of nodes are given.]**  int N;  int matrix[100][100]; // graph can have max 100 nodes.  scanf("%d",&N);  for(int i=1;i<=N;i++)  for(int j=1;j<=N;j++)  scanf("%d" ,&matrix[i][j]); | **[number of nodes and edges are given]**  int Node,Edge;  int matrix[100][100];  scanf("%d%d",&Node,&Edge);  for(i=0;i<Edge;i++)  {  int n1,n2,cost;  scanf("%d%d%d",&n1,&n2,&cost);  matrix[n1][n2]=cost;  matrix[n2][n1]=cost;  } |

**Complexity:**

* **Time Complexity:**

*finding adjacency node:* |v|+|v| or O(|v|). first scan vertex\_list to find index and than count 1’s in adjacency\_matrix eow corresponding to that index.

*Find if two nodes are connected:***O(1),** if indexes are given. Just check adjacency\_matrix[i][j].

**O(|v|)+O(1),** If vertex names are given. We have to scan the vertex\_list to identify the indexes.

* We are ok with time complexity.
* **Space Complexity:**

**O(v^2).** For adjacency matrix. This space is ok if the graph Is dense graph but waste of memory for parse graph. As most of the graph in real world like social network or WWW are parse graph. Because every user or web page are not linked to each other.

* Ex: assume there are |v|=10^9 user in a social network. And every user say connected with 1000 friends. As it is an undirected graph so,

|E|=v(v-1)/2 or 5\*10^9 Bytes or 0.5 TB memory. Which is ok.

But we estimeted |E|=V^2 or 10^9 bytes or 1000 PB memory. How funny.

* So, in terms of time we are ok with adjacency matrix representation but not with space.

**Array compression:**

* if the node value is very high like 2^30 than we can declare that sized array. Where array compression comes handy.
* <http://www.shafaetsplanet.com/planetcoding/?p=1388>

**Resources:**

1. <https://www.youtube.com/watch?v=9C2cpQZVRBA&list=PL2_aWCzGMAwI3W_JlcBbtYTwiQSsOTa6P&index=41>

**Part 05:** Graph Representatio\_03 (Adjacency List Representation).

**Code:**

|  |  |
| --- | --- |
| **///Taking input using linked list**  struct node  {  int data,weight;  struct node\* next.  };  string vertex\_list[MAX\_SIZE];  node edge\_list[v]; | **///taking grpah input using adjacency list tecknique by vector.**  #include <cstdio>  #include <vector>  using namespace std;  #define MAX 100000 //maximum node  vector <int> edges[MAX];  vector <int> cost[MAX]; //parallel vector to store costs;  int main() {  int numNodes, numEdges;  scanf("%d%d", &numNodes, &numEdges);  for (int i = 1; i <= numEdges; i++) {  int x, y,w;  scanf("%d%d%d", & x, & y,&w);  edges[x].push\_back(y);  edges[y].push\_back(x);  cost[x].push\_back(w);  cost[y].push\_back(w);  }  return 0;  } |

**stl resource:**

* <https://sites.google.com/site/smilitude/stl>
* <http://www.cplusplus.com/reference/stl/>

**Description:**

* We will alocate memory dinamically as it demands. We can use stl vector here.
* in this adjacency list of edges we just store the information about all the nodes those are connected to a specefic nodes.
* As most of the real world graphs are spares so, a specefic vertex may be connected to few other vertices. We dont need to store tha information to which vertices, our current vertex is not connected. Beacause this is redundent information, as we already know the vertices those are connected to current vertex.

**Copmlexity:**

* space complexity: **O(|E|) .** for a sperse graph its not a big deal. So, we are good with scape.
* Time complexity:

*finding all adjacency nodes* **O(V).** as we need to travers the whole row belonging to the given vertex.

*If two nodes are connected* **O(n)**. as we need to travers the whole row belonging to the given vertex. **O(logn),** if we maintain our list sorted than perform binary search.

* To do so we could use binary tree data structure by maintaining it balanced for list. We can also use linked list instead of array’s because dinamically array updating is costly than linked list.
* But if see the time cost to add an element to out list which is better linked list or binary tree ?
* Linked list will cost **O(v)** but binay tree will **O(logn)**. So using balanced binary tree is better.

**Resources:**

1. <https://www.youtube.com/watch?v=k1wraWzqtvQ&list=PL2_aWCzGMAwI3W_JlcBbtYTwiQSsOTa6P&index=42>

**part 06: Bredth first search.**

* The use of bfs is to determine the shortest path betweem two nodes. But it works only for unweighted graph or weight of every edge is 1.
* the level of a node indicates the shortest path from source to the node.

**Code:**

* If **linked list** for graph input we can check out this code:

<https://github.com/atique7465/Data_Structure_in_detail/blob/master/tree/binary_search_tree.cpp>

* If **Stl** used to take input graph than,

**Implementation:**

**BFS\_or\_level\_order\_traverse.01**

* nodes and edges are string. worst one. next implementation is better.
* The time complexity of BFS is **O(V + E)**, where V is the number of nodes and E is the number of edges.

**Code:** BFS\_or\_level\_order\_traverse\_01.cpp

input:

5

a b c d e

7

a b

a e

a c

b c

b d

d c

e c

output:

a b e c d

**BFS\_or\_level\_order\_traverse.02**

* nodes and edges are string. better implementation than first one.
* The time complexity of BFS is **O(V + E)**, where V is the number of nodes and E is the number of edges.

**Code:** BFS\_or\_level\_order\_traverse\_02.cpp

input:

5 7

a b c d e

a b

a e

a c

b c

b d

d c

e c

output:

a b e c d

**BFS\_or\_level\_order\_traverse.03**

* nodes and edges are integer.
* The time complexity of BFS is **O(V + E),** where V is the number of nodes and E is the number of edges.

**Code:** BFS\_or\_level\_order\_traverse\_03.cpp

input:

8 10

0 1

0 2

0 3

0 4

1 5

2 5

3 6

4 6

5 7

6 7

output:

0 1 2 3 4 5 6 7

**Applications:**

* 1. How to determine the level of each node in the given tree?
* the level of a node indicates the shortest path from source to the node.
* nodes and edges are integer.
* The time complexity of BFS is O(V + E), where V is the number of nodes and E is the number of edges.

**Code:** BFS\_or\_level\_order\_traverse\_04.cpp

input:

8 10

0 1

0 2

0 3

0 4

1 5

2 5

3 6

4 6

5 7

6 7

output:

Node 0 at level 0

Node 1 at level 1

Node 2 at level 1

Node 3 at level 1

Node 4 at level 1

Node 5 at level 2

Node 6 at level 2

Node 7 at level 3

**Resources:**

* <http://www.shafaetsplanet.com/?p=604>
* <https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/>
* <https://www.geeksforgeeks.org/bfs-using-stl-competitive-coding/>

**Part 07: Minimum Spanning Tree(Prims algorithm).**

* MST\_prim's using priority queue.
* Time Complexity:

The time complexity of the Prim’s Algorithm is O((V+E)logV), because each vertex is inserted

in the priority queue only once and insertion in priority queue take logarithmic time.

Code: MST\_prim\_priority\_queue.cpp

input:

8 11

1 5 2

1 2 4

1 4 10

5 4 5

2 4 8

2 3 18

4 3 11

4 7 11

4 8 9

7 6 1

8 6 2

output:

34

Resources:

* http://www.shafaetsplanet.com/?p=692
* https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/tutorial/

**Part 08: Minimum Spanning Tree(Kruskal’s algorithm).**

* time complexity O(ElogE).
* most time consuming operation is sorting than we loop over edges.

Code: MST\_kruskals\_algo.cpp

input:

7 11

1 3 5

1 2 7

2 3 9

2 5 7

2 6 8

3 5 15

3 4 6

4 5 8

4 7 11

5 7 9

5 6 5

output:

Minimun cost is: 39

**Resources:**

* http://www.shafaetsplanet.com/?p=825
* https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/tutorial/

**part 08: Topological sort.**

* http://www.shafaetsplanet.com/?p=848

**part 09: Depth First Search(DFS)**

* Time complexity O(V+E) when implemented using the adjacency list.

**Applications**

***How to find connected components using DFS?***

A graph is said to be disconnected if it is not connected, i.e. if two nodes exist in the graph such that there is no edge in between those nodes. In an undirected graph, a connected component is a set of vertices in a graph that are linked to each other by paths.

Consider the example given in the diagram. Graph G is a disconnected graph and has the following 3 connected components.

* First connected component is 1 -> 2 -> 3 as they are linked to each other
* Second connected component 4 -> 5
* Third connected component is vertex 6

**Code:** DFS\_using\_recursion.cpp

input 01:

5 5

1 2

1 5

2 4

2 3

5 4

output:

5 4 3 2 1

Number of connected components: 1

input 02:

6

4

1 2

2 3

1 3

4 5

output:

3 2 1

5 4

6

Number of connected components: 3

**Resources:**

* <https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/>
* http://www.shafaetsplanet.com/?p=973

**part 10:** Dijkstrs algorithm shortest path.

* The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum.
* This problem could be solved easily using **(BFS)** if all edge weights were (1), but here weights can take any value. Three different algorithms are discussed below depending on the use-case.
* Time Complexity of Dijkstra's Algorithm is O(v^2)
* but with min-priority queue it drops down to O(V+E log V).

**Code\_01:** djkstra.cpp

**Code\_02:** dijkstrs\_using\_operator overloading.cpp

input:

8 11

1 5 2

1 2 4

1 4 10

5 4 5

2 4 8

2 3 18

4 3 11

4 7 11

4 8 9

7 6 1

8 6 2

Output:

Dijkstrs found min cost for nodes:

node 1 : 0

node 2 : 4

node 3 : 18

node 4 : 7

node 5 : 2

node 6 : 18

node 7 : 18

node 8 : 16

Summation of minimum costs 83

**References:**

* <http://www.shafaetsplanet.com/?p=1500>
* <https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/>