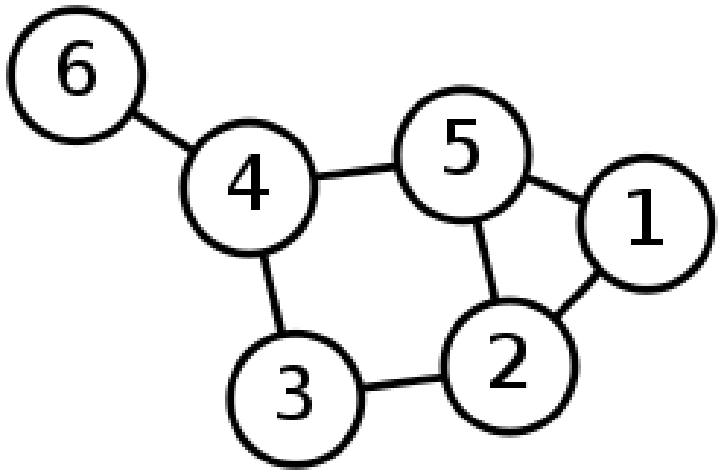
Graph Theory

IOI Training Camp 1 – 2017/2018

Bronson Rudner

What is a graph?

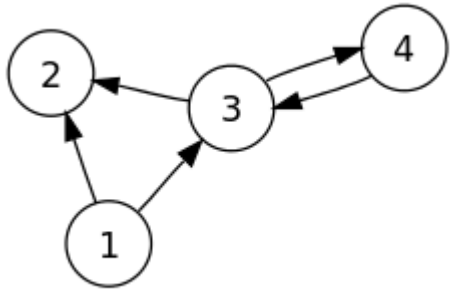
- Nodes
- Edges
- An edge joins exactly 2 nodes



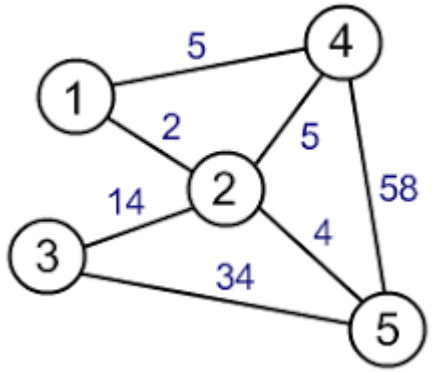
- E.g. Cities (Nodes) connected by roads (Edges).

Typical Variations

- Edges may be **directed**.



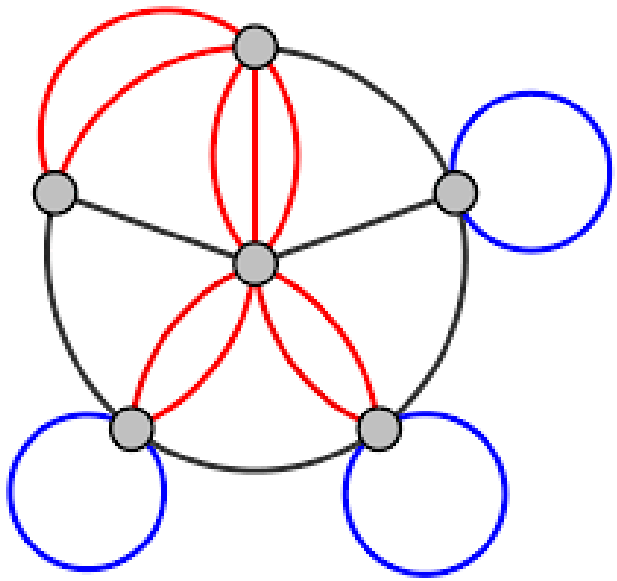
- Edges may be **weighted**.



More Variations

- A graph could have **loops**.

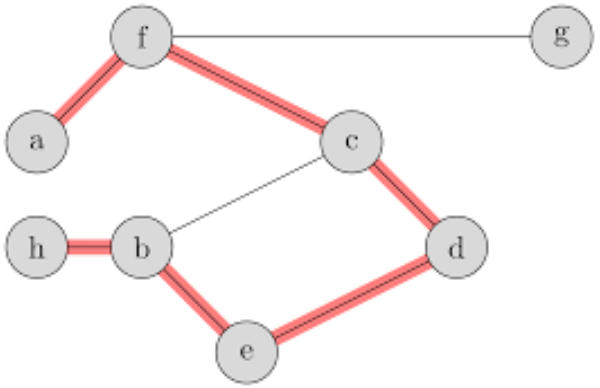
- Multiple edges between two nodes.



Typically problems involve **simple graphs**, which have no loops and at most one edge between two nodes.

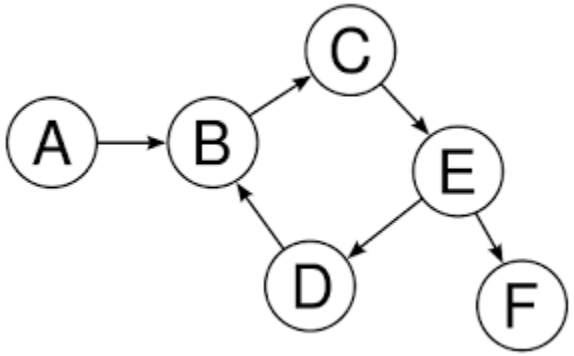
Definitions

- Path



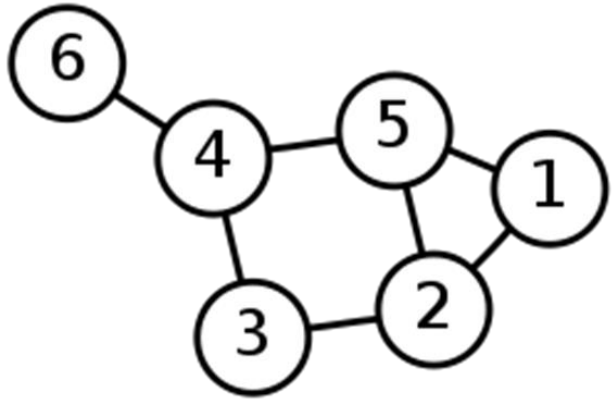
- Cycle

B-C-E-D



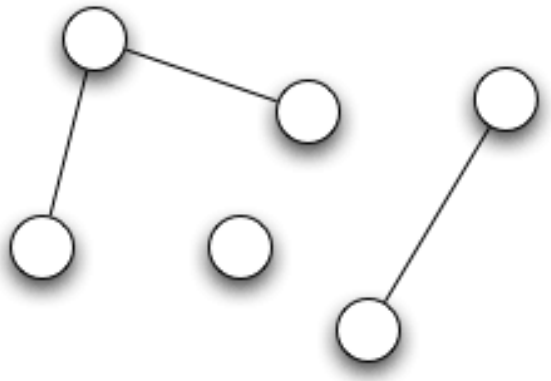
Definitions

- A **connected** graph



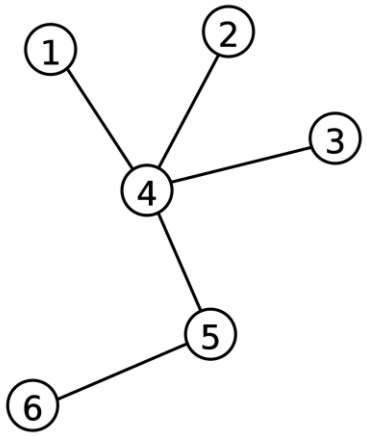
- A **disconnected** graph

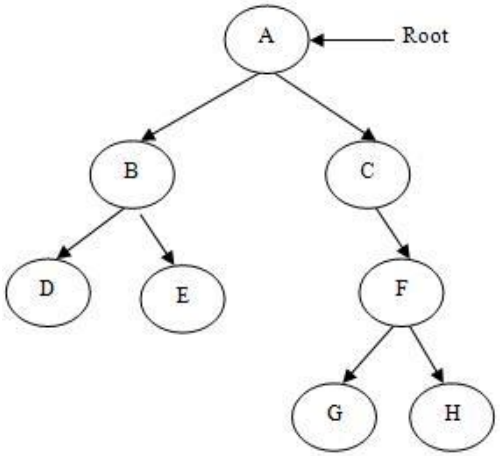
3 **Components**



Definitions

- Tree

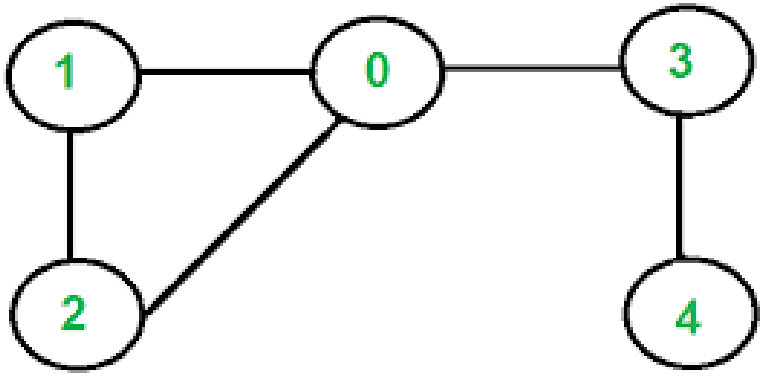




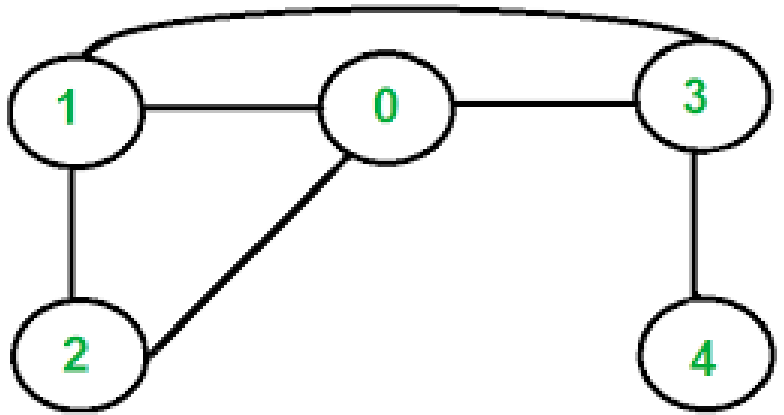
- Forest – a collection of trees

Definitions

- Eulerian path

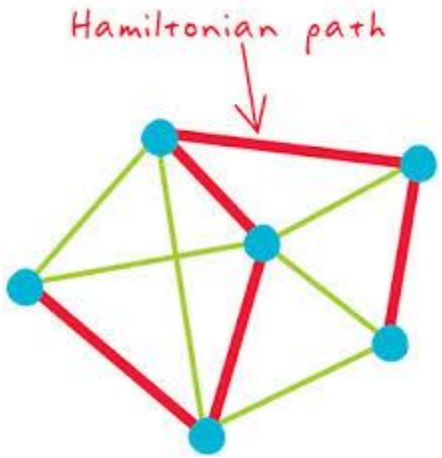


The graph has Eulerian Paths, for example "4 3 0 1 2 0", but no Eulerian Cycle. Note that there are two vertices with odd degree (4 and 0)



The graph is not Eulerian. Note that there are four vertices with odd degree (0, 1, 3 and 4)

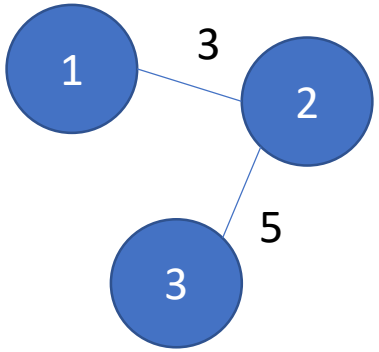
- Hamiltonian path



Representing a graph

- A list of edges

```
1  struct edge {
2      int u, v, weight;
3
4      bool operator<(edge other) {
5          return weight < other.weight;
6      }
7  }
8
9  void foo(vector<edge> edges) {
10     sort(edges.begin(), edges.end());
11     for (edge e: edges) // do something
12 }
```

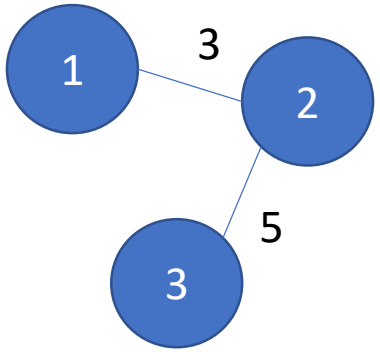


{ (1,2,3) , (2,3,5) }

Representing a graph

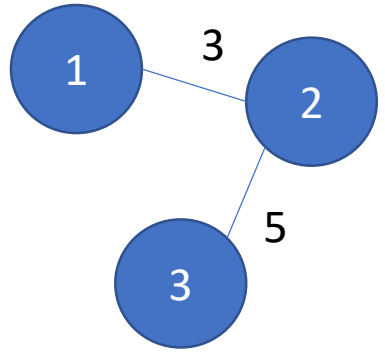
- An adjacency matrix

```
1   bool adj[N][N]; // adj[u][v] = true -> edge between u and v
2   int adj[N][N]; // adj[u][v] gives weight of edge between u and v
3   edge adj[N][N];
4
5   for (int i = 0; i < N; ++i) for (int j = 0; j < N; ++j) {
6       adj[i][j] = // some default value;
7   }
8
9   vector<vector<int>> adj(N, vector<int>(N, inf));
10
11  for (int i = 0; i < N; ++i) {
12      for (int j = 0; j < N; ++j) {
13          cin << adj[i][j];
14      }
15  }
```



	1	2	3
1	-	3	inf
2	3	-	5
3	inf	5	-

Representing a graph



1: (2, 3)
2: (1, 3) , (3, 5)
3: (2, 5)

- An adjacency list

```
1   vector<vector<edge>> adj; // adj[u][i] is some edge from node u
2
3   vector<vector<pair<int,int>>> adj;
4   // adj[u][i].first is some neighbour v of node u
5   // adj[u][i].second is the weight of the edge from u to v
6
7   vector<vector<int>> adj; // adj[u][i] gives some neighbour v of node u
8   for (int u = 0; u < N; ++u) { // loops through each node u
9       for (int v: adj[u]) {// loops through each neighbour of node u
10          // do something
11      }
12  }
```

Depth first search

- Start from some node.
- Visit all nodes in a connected component.
- Walk as far as possible along a path in the graph before backtracking.

```
1  void dfs(int u) {
2      visited[u] = true;
3      for (int v: adj[u]) {
4          if (!visited[v])
5              dfs(v);
6      }
7      visited[u] = false;
8  }
```

Breadth first search

- Start from some node.
- Visit all nodes in a component.
- Visit nodes closest to the starting node first.

```cpp
void bfs(int start, vector<vector<int>> adj) {
    vector<int> visited(adj.size(), false);
    visited[start] = true;

    queue<int> q;
    q.push(start);

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int v: adj[u]) {
            if (!visited[v]) {
                visited[v] = true;
                q.push(v);
            }
        }
    }
}
```

Dijkstra's Algorithm

- Used in a weighted graph.
- Determine shortest distance from one node to each of the others.
- Breadth first search using a **priority queue**.

Dijkstra's Algorithm

```cpp
void djikstra(int start, vector<vector<pair<int,int>>> adj) {
    vector<int> distance(adj.size(), 1000000000);
    distance[start] = 0;

    priority_queue<pair<int,int>> q;
    q.push({0, start});

    while (!q.empty()) {
        int u = q.top().second;
        q.pop();

        for (int i = 0; i < adj[u].size(); ++i) {
            int v = adj[u][i].first, weight = adj[u][i].second;

            if (distance[u] + weight < distance[v]) {
                distance[v] = distance[u] + weight;
                q.push({-distance[v], v}); // ***
            }
        }
    }
}
```

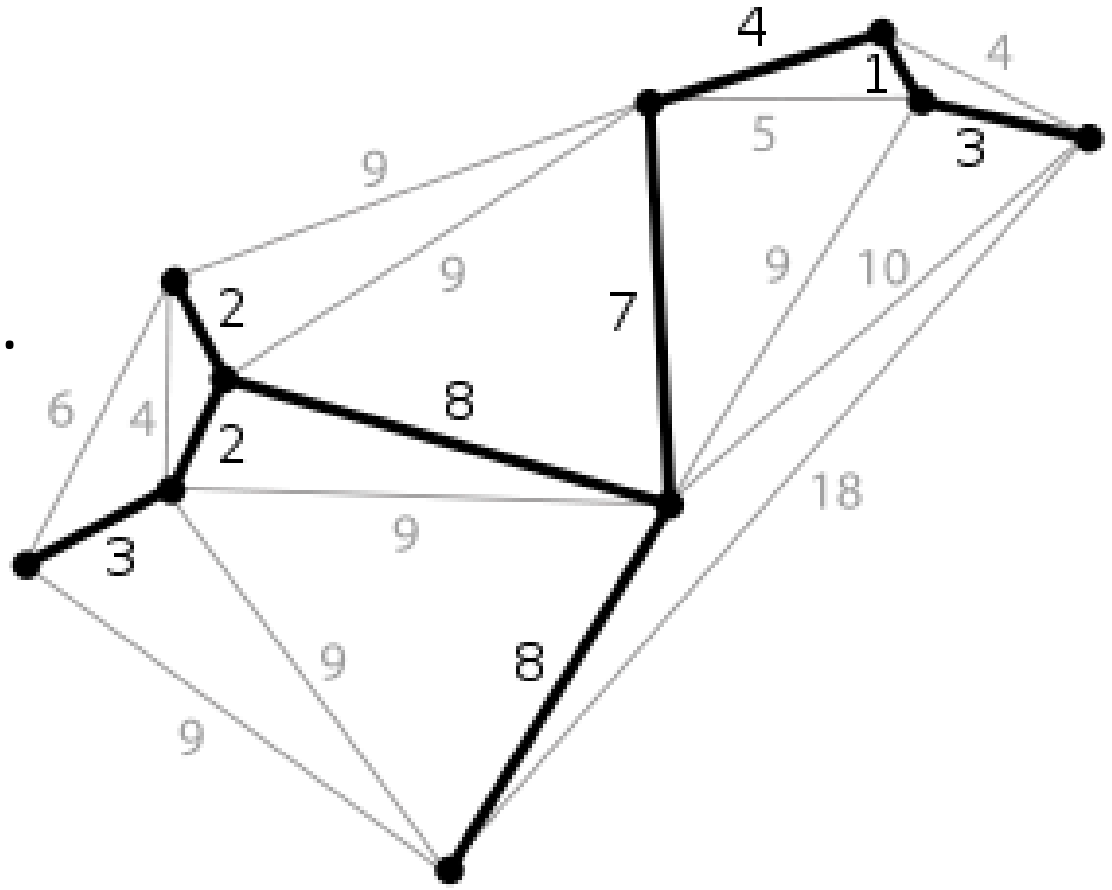
Floyd's Algorithm

- Used in a weighted graph.
- Determine shortest distance from one node to each of the others.
- Makes use of an adjacency matrix.

```
1  void floyd(vector<vector<int>> d) {
2      int N = d.size();
3      for (int k = 0; k < N; ++k)
4          for (int i = 0; i < N; ++i)
5              for (int j = 0; j < N; ++j)
6                  d[i][j] = min(d[i][j], d[i][k] + d[k][j])
7  }
```

Minimum Spanning Tree

- Subset of edges of a weighted, undirected graph.
- Tree, containing all nodes, with smallest sum of weights of edges.



- Can use Prim's or Kruskal's Algorithm to determine.

Prim's Algorithm

- Start with a set of 1 node.
- Keep adding the node with smallest distance to some node currently in your set.

```
int prim(vector<vector<pair<int,int>>> adj, int N) {
    vector<int> done(N, false);
    priority_queue<pair<int,int>> q;

    q.push({0,0}); // { distance, node }

    int cost = 0;
    while (!q.empty()) {
        cost += q.top().first;
        int u = q.top().second;
        q.pop();
        if (done[u]) continue;
        done[u] = true;

        for (int i = 0; i < adj[u].size(); ++i) {
            int v = adj[u][i].first, weight = adj[u][i].second;
            q.push({-weight, v});
        }
    }
    return cost;
}
```

Kruskal's Algorithm

- Start with each node its own component (so no edges).
- Loop through edges, from smallest to largest weight.
- If an edge joins two different components, add it to the graph.
- Makes use of the union-find data structure.

Kruskal's Algorithm

```
1  void kruskal(vector<edge> edges) {
2      int cost = 0;
3      sort(edges.begin(), edges.end());
4      for (edge e: edges) {
5          if (union_find(e.u) != union_find(e.v)) {
6              union_join(e.u, e.v);
7              cost += e.weight;
8          }
9      }
10     return cost;
11 }
```

Union-find Data Structure

```
1   vector<int> p, depth;
2
3   void kruskal_union_init(int N) {
4       p.resize(N), depth.assign(N, 0);
5       for (int u = 0; u < N; ++u) p[u] = u;
6   }
7
8   int union_find(int u) {
9       if (p[u] == u) return u;
10      p[u] = union_find(p[u]);
11      return p[u];
12  }
```

```
14  void union_join(int u, int v) {
15      int a = union_find(u), b = union_find(v);
16      if (a == b) return;
17      if (depth[a] < depth[b]) {
18          p[a] = b;
19      } else if (depth[a] > depth[b]) {
20          p[b] = a;
21      } else {
22          p[a] = b;
23          ++depth[b];
24      }
25  }
```