inzva Algorithm Programme 2018-2019

Bundle 7

Graph-2

Editor
Uğur Uysal

Reviewer
M.Besher Massri

Contents

1 Introduction

In Graph-01 bundle, we were introduced to the basics of graph theory via a combination of definitions and concepts like DFS and BFS. We also learned about trees and tree-related data structures like heaps and BSTs. It is beneficial for the participants to recall the Graph-01 bundle before studying this bundle as some materials from the previous bundle are needed for this week's bundle.

In this bundle, we will cover more in depth topics about graph theory. First we will cover some new definitions in graph theory, then we will move on the following topics:

- Bipartite checking
- Topological sort
- Shortest path problem
- Minimum spanning trees

So let's move forward and discuss each topic in detail!

2 Definitions

2.1 Bipartite Graphs

A bipartite graph is a graph whose vertices can be divided into two disjoint and independent sets U and V such that every edge connects a vertex in U to one in V. Vertex sets U and V are usually called the parts of the graph. [1]. The figure is shown in below. It is similar to graph coloring with two colors. Coloring graph with two colors is that every vertex have a corresponding color, and for any edge, it's vertices should be different color. In other words, if we can color neighbours two different colors, we can say that graph is bipartite.

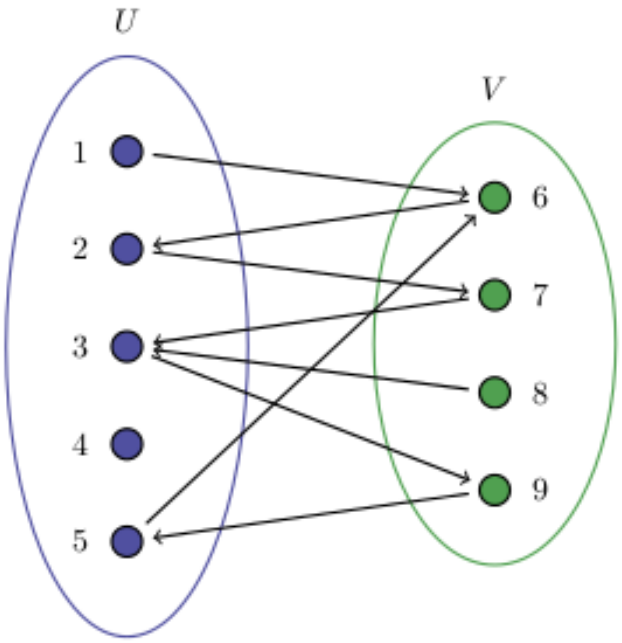

Figure 1: Example bipartite graph, all edges satisfy the coloring constraint

We have some observations here.

- A graph 2- colorable if and only if it is bipartite.
- A graph does not contain odd-length cycle if and only if it is bipartite.
- Every tree is a bipartite graph since trees do not contain any cycles.

2.2 Directed Acyclic Graphs

A directed acyclic graph(DAG) is a finite directed graph with no directed cycles. Equivalently, a DAG is a directed graph that has a topological ordering (we cover it in this bundle), a sequence of the vertices such that every edge is directed from earlier to later in the sequence [2]. DAGs can be used to encode precedence relations or dependencies in a natural way [4]. There are several applications using topological ordering directly such as finding critical path or automatic differentiation on computational graphs (this is extremely useful for deep learning frameworks [5]).

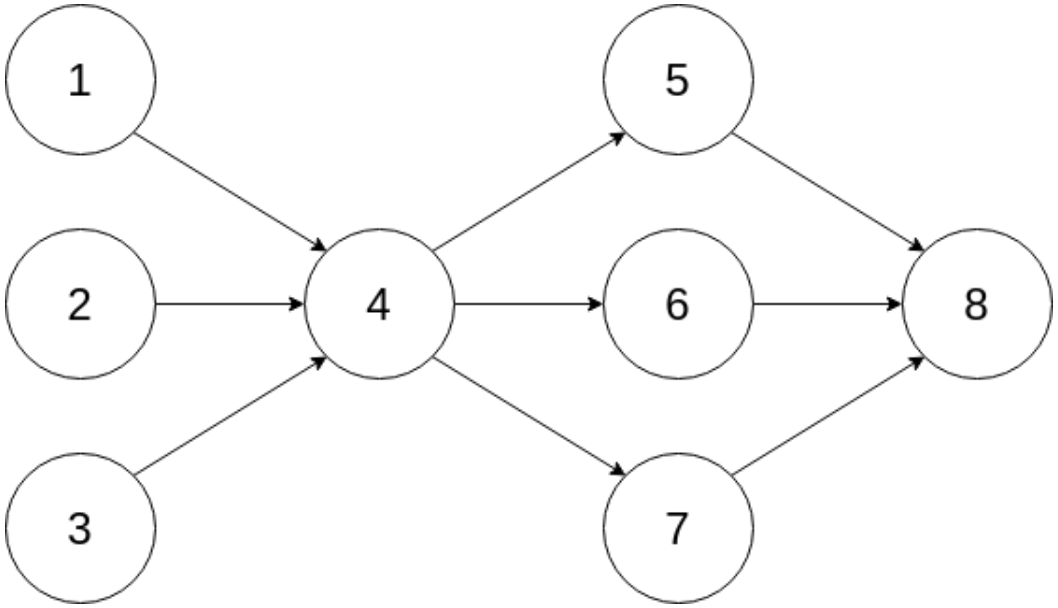

Figure 2: Example Directed Acyclic Graphs

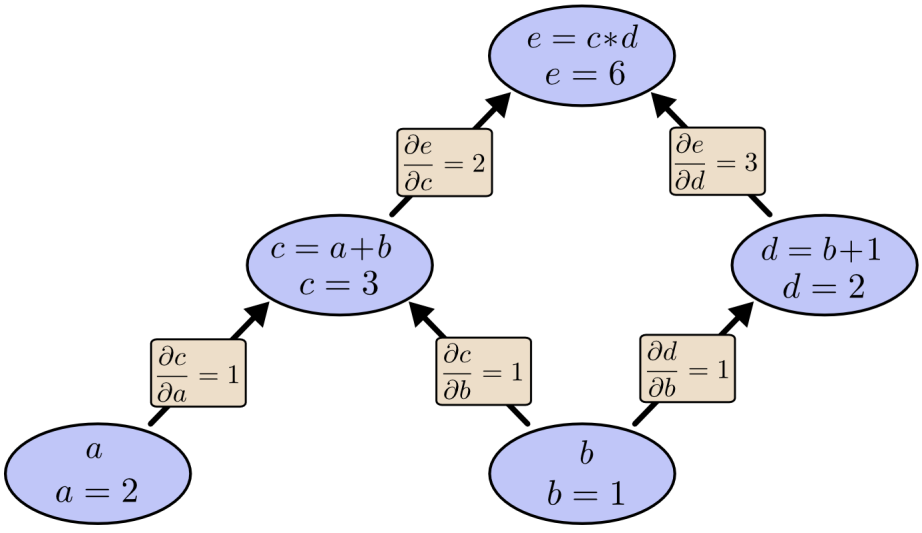

Figure 3: Example computational graph also a DAG, partial derivatives are written to edges respect to topological order

3 Bipartite Checking

The question is in the title. Is the given graph bipartite? We can use BFS or DFS on graph. Lets first focus on BFS related algorithm. This procedure is very similar to BFS, we have an extra color array and we assign a color to each vertex when we are traversing the graph. Algorithm proof depends on fact that BFS explores the graph level by level. If the graph contains an odd cycle it means that there must be a edge between two vertices that are in same depth (layer, proof can be found on [4]). Let's say the colors are red and black and we traverse the graph with BFS and assign red to odd layers and black to even layers. Then we check the edges to see if there exists an edge that its vertices are same color. If there is a such edge, the graph is not bipartite, else the graph is bipartite.

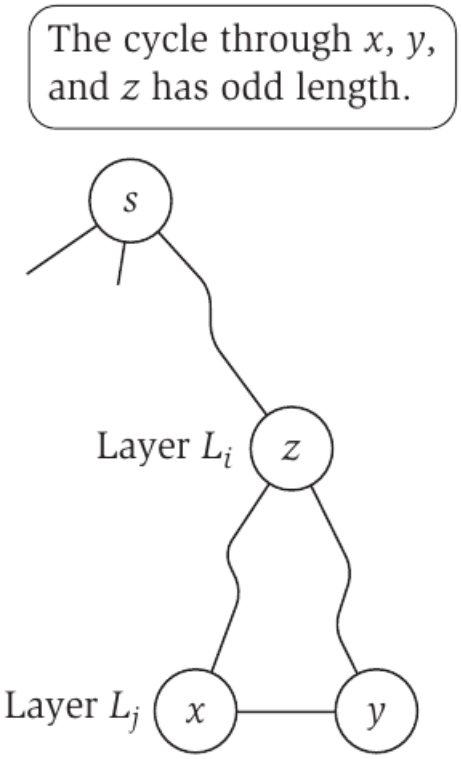

Figure 4: If two nodes x and y in the same layer are joined by an edge, then the cycle through x, y, and their lowest common ancestor z has odd length, demonstrating that the graph cannot be bipartite.

```cpp
typedef vector<int> adjList;
typedef vector<adjList> graph;
typedef pair<int,int> ii;
enum COLOR {RED, GREEN};
bool bipartite_check(graph &g){
    int root = 0; // Pick 0 indexed node as root.
    vector<bool> visited(g.size(),false);
    vector<int> Color(g.size(),0);
    queue<ii> Q( { {root,0}} ); // insert root to queue, it is  first layer_0
    visited[root] = true;
    Color[root] = RED;
    while ( !Q.empty() )
    {
        /*top.first is node, top.second its depth i.e layer */
        auto top = Q.front();
        Q.pop();
        for (int u : g[top.first]){
            if ( !visited[u] ){
                visited[u] = true;
                //Mark even layers to red, odd layers to green
                Color[u] = (top.second+1) % 2 == 0 ?  RED : GREEN;
                Q.push({u, top.second+1 });
            }
        }
    }
    for(int i=0; i < g.size(); ++i){
        for( auto v: g[i]){
            if ( Color[i] == Color[v] ) return false;
        }
    }
    return true;
}
int main(){
    graph g(3);
    g[0].push_back(1);
    g[1].push_back(2);
    g[2].push_back(3);
    cout << (bipartite_check(g) == true ? "YES" : "NO") << endl;
    return 0;
}
```

The complexity of algorithm is is $O(V+E)+O(E)$, BFS and loop over edges. But we can say it $O(V+E)$ since it is Big-O notation.

4 Topological Sort

4.1 Definition

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u->v, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG [6].

There are many important usages of topological sorting in computer science; applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers. It is also used to decide in which order to load tables with foreign keys in databases [3].

There are known algorithms (e.g Kahn's algorithm) to find topological order in linear time. Below, you can find one of the implementations:

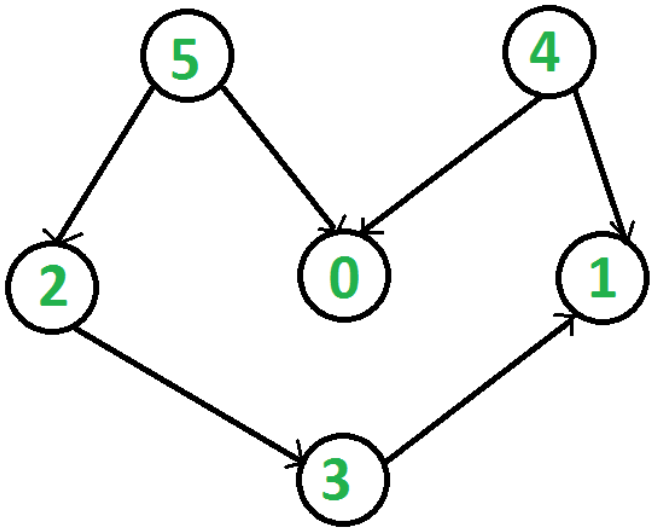

Figure 5: For example, a topological sorting of this graph is "5 4 2 3 1 0". There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is "4 5 2 3 1 0". The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges)[6].

4.2 Algorithm

```
typedef vector<int> adjList;
typedef vector<adjList> graph;
typedef pair<int,int> ii;
void kahn(graph &g){
    vector<int> result;
    queue<int> q;//
    vector<int> degree(g.size(),0); // number of incoming egdes.
    for(auto &list: g){
        for(auto &node:list){
            degree[node]++;
        }
    }
    for(int i=0; i < g.size(); ++i){
        if (degree[i] == 0) q.push(i);
    }
    while( !q.empty()){
        int node = q.front();
        result.push_back(node);
        q.pop();
        for (auto &ng: g[node]){
            degree[ng]--;
            if ( degree[ng] == 0) q.push(ng);
        }
    }
    for(auto &i:result) cout << i << " ";
    cout << endl;
}
int main(){
    graph g(6);
    g[1].push_back(0);
    g[1].push_back(2);
    g[2].push_back(3);
    g[3].push_back(4);
    g[4].push_back(5);
    kahn(g);
    return 0;
}
```

As for time complexity: we traverse all edges in the beginning (calculating degrees) and in the while segment we remove edges (once for an edge) and traverse all nodes. Hence, the time complexity of this algorithm is $O(V+E)$. Note that this implementation assumes the graph is DAG. Try improving this code to support checking if the graph is DAG!

5 Shortest Path Problem

5.1 Definition

Let $G(V, E)$ be a graph, v_i and v_j be two nodes of G. We say a path between v_i and v_j is the shortest path if sum of the edge weights (cost) in the path is minimum. In other words, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized. [7]

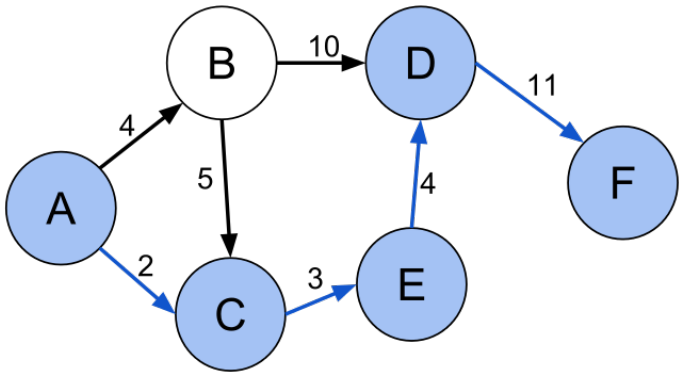

Figure 6: Example shortest path in graph. Source is A and target is F. Image taken from [7].

We will cover several shortest path algorithms in this bundle. One of them is Dijkstra's Shortest Path Algorithm but it has some drawbacks: Edge weights should be non-negative for the optimally of the algorithm. We will discover other algorithms in which these condition isn't necessary, like Floyd-Warshall and Bellman-Ford algorithms.

5.2 Dijkstra's Shortest Path Algorithm

Dijkstra's Shortest Path algorithm is straight forward. In brief we have a set S that contains explored nodes and d which contains the shortest path cost from source to another node. In other words, $d(u)$ represents the shortest path cost from source to node u. The procedure follows as that. First, add source node to set S which represents the explored nodes and assigns the minimum cost of the source to zero. Then each iteration we add node to S that has lowest cost ($d(u)$) from unexplored nodes. Let's say $S' = V - S$ which means unexplored nodes. For all nodes in S' we calculate $d(x)$ for each node x is S' then we pick minimum cost node and add it to S. So how we calculate $d(x)$?. For any x node from S', $d(x)$ calculated as that, let's say e cost of any edge from S to x then $d(x) = min(d(u) + e)$. It is a greedy algorithm.

Here is the explanation of the algorithm step by step.

1. Initialize an empty set, distance array, insert source to set.
2. Initialize a min-heap, put source to heap with key is zero.
3. While heap is not empty, take the top element from heap and add its neighbours to min-heap.
4. Once we pick an element from the heap, it is guaranteed that the same node will never be added to heap with lower key value.

In implementatio,n we can use priority queue data structure in order to increase efficiency. If we put unexplored nodes to min - priority queue where the distance is key, we can take the lowest cost unexplored node in $O(log(n))$ time which is efficient.

```
typedef pair<int,int> edge;
typedef vector<edge> adjList;
typedef vector<adjList> graph;

void dijkstra(graph &g, int s){
    vector<int> dist(g.size(),INT_MAX/2);
    vector<bool> visited(g.size(),false);
    dist[s] = 0;
    priority_queue<edge, vector<edge>, greater<edge>> q;
    q.push({0, s});
    while(!q.empty())
    {
        int v = q.top().second;
        int d = q.top().first;
        q.pop();
        if(visited[v]) continue;
        visited[v] = true;
        for(auto it: g[v])
        {
            int u = it.first;
            int w = it.second;
            if(dist[v] + w < dist[u])
            {
                dist[u] = dist[v] + w;
                q.push({dist[u], u});
            }
        }
    }
}
```

5.3 Floyd-Warshall Algorithm

The Floyd Warshall Algorithm is used for solving the all pairs shortest path problem. The problem is to find the shortest shortest distances between every pair of vertices in a given weighted directed graph [8]. Instead of running Dijkstra's algorithm for every node as a source, Floyd-Warshall algorithm provides a simpler solution that uses the power of dynamic programming to achieve this task.

Lets state that in this algorithm, we have adjacency matrix representation of the graph. This algorithm works optimal even if there are negative edges but not negative cycles unlike the Dijkstra's shortest path. The algorithm looks for if any node can be an intermediate node for a path that decrease the cost. If the new cost is smaller, algorithm updates the cost in the adjacency matrix. For every k (as an intermediate node) in graph, we check all i,j pairs and we calculate $cost(i, k)+cost(k, j)$. Then we update $cost(i, j)$ with the new value if it is smaller than the current value.

```
void floydWarshall (int graph[][V])
{
    /* dist[][] will be the output matrix that will finally have the shortest
      distances between every pair of vertices */
    int dist[V][V], i, j, k;
    /* Initialize the solution matrix same as input graph matrix. Or
       we can say the initial values of shortest distances are based
       on shortest paths considering no intermediate vertex. */
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];
    for (k = 0; k < V; k++)
    {
        for (i = 0; i < V; i++)
        {
            for (j = 0; j < V; j++)
            {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
}
```

Time complexity of the algorithm can be seen here, there are three nested for loops over all nodes hence it is $O(V^3)$ and space complexity is $O(V^2)$ because we keep adjacency matrix in memory(after altering it with new values, it becomes a memoization table).

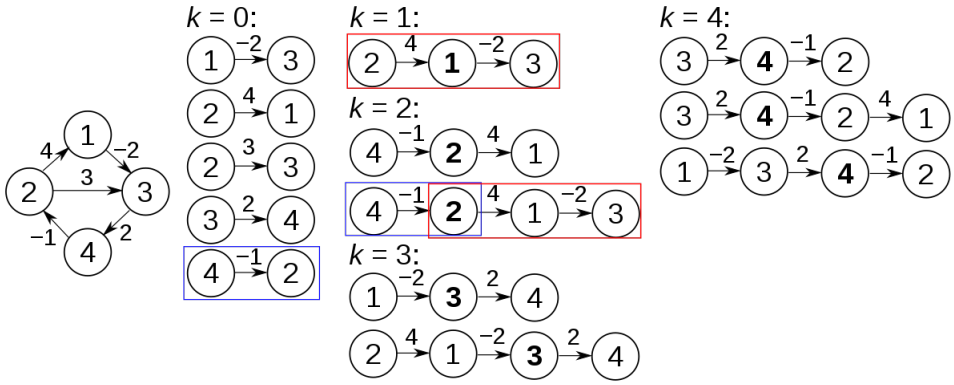

Figure 7: Example of Floyd-Warshall

5.4 Bellman Ford Algorithm

The Bellman–Ford algorithm is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph. It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers. Bellman-Ford algorithm also detects negative weighted cycles in the graph [9].

In this algorithm, we maintain distance and the previous arrays to save costs and path. This algorithm is for single source shortest path problem and we initialize distance[source] to zero and all others to infinity. Then we check for all vertices in graph with all edges if using this edge makes the distance smaller than the current distance to this node. If it is smaller, we update distance and previous arrays. After constructing the previous and distance arrays, we loop over the edges and if we find a change on distance array, it means that graph contains a negative cycle.

```
typedef pair<int,pair<int,int>> edge;
typedef vector<edge> weigthed_graph;
void BellmanFord(weigthed_graph g, int V, int src)
{
    int E = g.size();
    int dist[V];
    //this implemantation is just for negative cycle check
    // init distance array
    for (int i = 0; i < V; i++)
        dist[i]   = INT_MAX;

    dist[src] = 0;
    for (int i = 1; i <= V-1; i++)
    {
        for (int j = 0; j < E; j++)
        {
```

```
            int u = g[j].second.first, v = g[j].second.second;
            int weight = g[j].first;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }
    for (int i = 0; i < E; i++)
    {
        int u = g[i].second.first, v = g[i].second.second;
        int weight = g[i].first;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            printf("Graph contains negative weight cycle");
    }
}
```

There is a nested two for loops. One for vertices and one for edges so the time complexity of this algorithm is $O(EV)$.

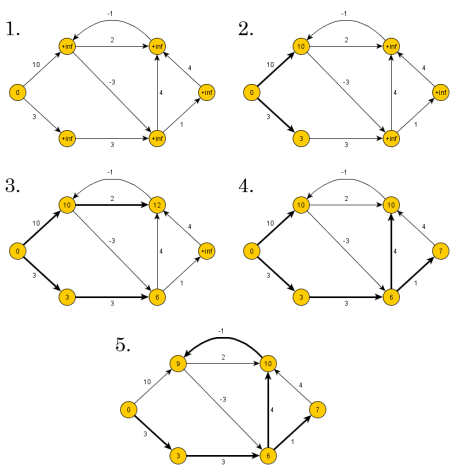

Figure 8: Example of Bellman-Ford Algorithm

Algorithm	Time Complexity	Space Complexity	Notes
Dijkstra Shortest Path A.	$O(ElogV)$	$O(V^2)$	Single Source Fails on negative edges
Floyd-Warshall A.	$O(V^3)$	$O(V^2)$	All Pairs Fails on negative cycle
Bellman-Ford A.	$O(VE)$	$O(V+E)$	Single Source Can detect negative cycle

As a result, we inspected three shortest path algorithm. Here is a brief conclusion as a table.

6 Minimum Spanning Tree

6.1 Definition

Given an undirected weighted connected graph $G = (V, E)$ Spanning tree of G is a connected acyclic sub graph that covers all nodes and some edges. In a disconnected graph -where there is more than one connected component- the spanning tree of that graph is defined as the forest of the spanning trees of each connected component of the graph.

Minimum spanning tree (MST) is a spanning tree in which the sum of edge weights is minimum. The MST of a graph is not unique in general, there might be more than one spanning tree with the same minimum cost. For example, take a graph where all edges have the same weight, then any spanning tree would be a minimum spanning tree. In problems involving minimum spanning trees where you have to output the tree itself (and not just the minimum cost), it either puts more constraint so the answer is unique, or simply asks for any minimum spanning tree.

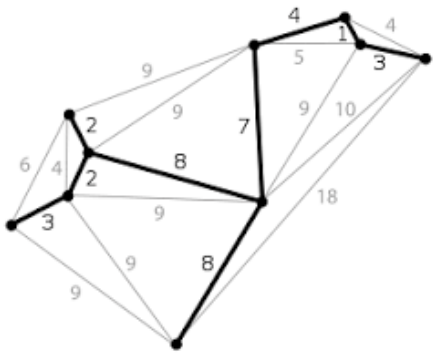

Figure 9: MST of the graph. It spans all nodes of the graph and it is connected.

To find the minimum spanning tree of a graph, we will introduce two algorithms. The first one called Prim's algorithm, which is similar to Dijkstra's algorithm. Another algorithm is Kruskal agorithm, which makes use of the disjoint set data structure. Let's discover each one of them in detail!

6.2 Prim Algorithm

Prim algorithm is very similar to Dijkstra's shortest path algorithm. In this algorithm we have a set S which represents the explored nodes and again we can maintain a priority queue data structure the closest node in $V - S$. It is a greedy algorithm just like Dijkstra's shortest path algorithm.

G = (V,E) V set of all nodes
T = {} result, edges of MST
S = 1 ; explored nodes
while S /= V **do**
  let (u, v) be the lowest cost edge such that u in S and v in V - S;
  T = T U {(u, v)}
  S = S U {v}
end

Algorithm 1: Prim Algorithm in Pseudo code, what is the problem here?

There is a problem with this implementation, it assumes that the graph is connected. If the graph is not connected this algorithm will be stuck on loop. There is a good visualization for Prim algorithm at [10]. If we use priority queue complexity would be $O(ElogV)$

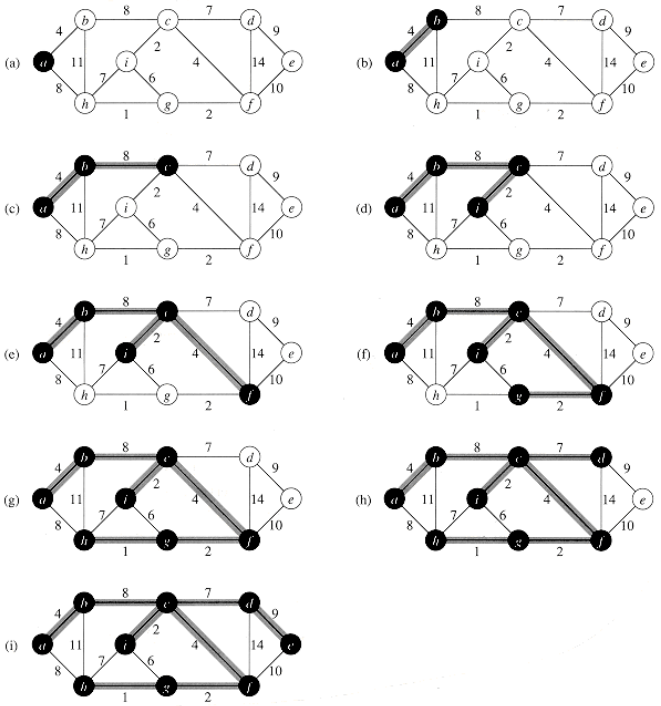

Figure 10: Example of how Prim Algorithm constructs the MST

6.3 Kruskal Algorithm

In Prim algorithm we started with a specific node and then proceeded with choosing the closest neighbor node to our current graph. In Kruskal algorithm, we follow a different strategy; we start building our MST by choosing one edge at a time, and link our (intially separated) nodes together until we connect all of the graph.

To achieve this task, we will start with having all the nodes separated each in a group. In addition, we will have the list of edges from the original graph sorted based on their cost. At each step, we will:

1. Pick the smallest available edge (that is not taken yet)
2. Link the nodes it connects together, by merging their group into one unified group
3. Add the cost of the edge to our answer

However, you may realize in some cases the link we add will connect two nodes from the same group (because they were grouped before by other taken edges), hence violating the spanning tree condition (Acyclic) and more importantly introducing unnecessary edges that adds more cost to the answer. So to solve this problem, we will only add the edges as long as they connect two currently (at the time of processing this edge) separated nodes that belong to different groups, hence completing the algorithm.

The optimality of Kruskal algorithm comes from the fact that we are taking from a sorted list of edges. For more rigorous proof please refer to [11].

So how can we effectively merge the group of nodes and check that which group each node belong? We can utilize disjoint set data structure which will help us to make union and find operations in an amortized constant $O(1)$ time.

```
typedef pair<int,pair<int,int>> edge;
// represent edge as triplet (w,u,v)
// w is weigth, u and v verticies.
// edge.first is weigth edge.second.first -> u, edge.second.second -> v
typedef vector<edge> weigthed_graph;

/*union - find data structure utilities */
const int maxN = 3005;
int parent[maxN];
int ssize[maxN];
void make_set(int v);
int find_set(int v);
void union_sets(int a, int b);
void init_union_find();

/*Code that finds edges in MST */
```

```
void kruskal(vector<edge> &edgeList ){
    vector<edge> mst;
    init_union_find();
    sort(edgeList.begin(),edgeList.end(), \
        [](const auto &a, const auto  &b) { return a.first< b.first;});
    //well this weird syntax is lambda function
    // for sorting pairs to respect their first element.
    for( auto e: edgeList){
        if( find_set(e.second.first )!= find_set(e.second.second)){
            mst.push_back(e);
            union_sets(e.second.first, e.second.second);
        }
    }
}
```

To calculate the time complexity, observe how we first sorted the edges, this takes $O(ElogE)$. In addition we pass through the edges one by one, and each time we check which group the two nodes of the edge belongs to, and in some cases merge the two groups. So in the worst case we will assume that both operations (finding and merging) happens, but since the disjoint data structure guarantee $O(1)$ amortized time for both operations, we end up with $O(E)$ amortized time of processing the nodes.

So in total we have $O(ElogE)$ from sorting edges and $O(E)$ from processing them, those results in a total of $O(ElogE)$ (if you don't understand why please refer to the first bundle where we discuss time complexity).

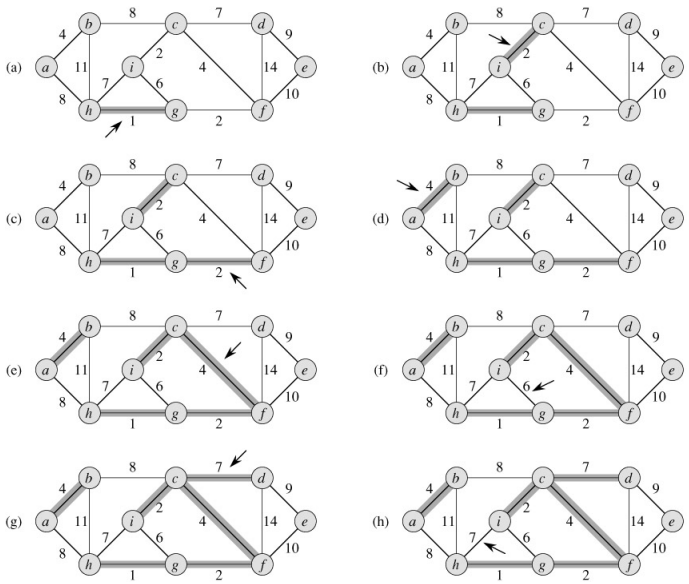

Figure 11: Example of how Kruskal Algorithm constructs the MST

References

[1] Bipartite Graph. Wikipedia, the free online encyclopedia. Retrieved January 5, 2019

[2] Directed Acyclic Graph. Wikipedia, the free online encyclopedia. Retrieved January 5, 2019

[3] Topological Sort. Wikipedia, the free online encyclopedia. Retrieved January 5, 2019

[4] Algorithm Design, Kleinberg, Tardos

[5] PyTorch AutoGrad website Depp Learning Framework

[6] Topological sort. Geeksforgeeks website. Retrieved January 5, 2019

[7] Shortest Path. Wikipedia, the free online encyclopedia. Retrieved January 5, 2019

[8] Floyd Warshall. Geeksforgeeks website. Retrieved January 5, 2019

[9] Bellman Ford Wikipedia, the free online encyclopedia. Retrieved January 5, 2019

[10] Prim Algorithm. Wikipedia, the free online encyclopedia. Retrieved January 5, 2019

[11] Kruskal Algorithm. Wikipedia, the free online encyclopedia. Retrieved January 5, 2019