**LAPORAN PRAKTIKUM**

**STRUKTUR DATA DAN ALGORITMA**

**Judul: Algoritma Prim dan Algoritma Kruskal**

**DISUSUN OLEH**

**Ilham Nur Romdoni M0520038**

**PROGRAM STUDI INFORMATIKA**

**FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM**

**UNIVERSITAS SEBELAS MARET**

**2021**

# BAB I PENDAHULUAN

## 1.1 Tujuan praktikum

1. Praktikan dapat mengetahui informasi terkait algoritma Prim dan algoritma Kruskal.
2. Praktikan dapat menganalisis cara kerja dari algoritma Prim dan algoritma Kruskal.
3. Praktikan dapat mengetahui perbedaan dari algoritma Prim dan algoritma Kruskal.

## 1.2 Dasar teori

### A. Algoritma Prim

Algoritma Prim adalah sebuah algoritma dalam *graph theory* untuk mencari *minimum spanning tree* untuk sebuah *graph* berbobot yang saling terhubung. Ini berarti bahwa sebuah himpunan bagian dari *edge* yang membentuk suatu *tree* yang mengandung *node*, di mana bobot keseluruhan dari semua *edge* dalam *tree* diminimalisasikan. Bila graph tersebut tidak terhubung, maka *graph* itu hanya memiliki satu *minimum spanning tree* untuk satu dari komponen yang terhubung. Algoritma ini ditemukan pada 1930 oleh matematikawan Vojtěch Jarník dan kemudian secara terpisah oleh *computer scientist* Robert C. Prim pada 1957 dan ditemukan kembali oleh Dijkstra pada 1959. Karena itu algoritma ini sering dinamai algoritma DJP atau algoritma Jarnik.

Dengan struktur data *binary heap* sederhana, algoritma Prim dapat ditunjukkan berjalan dalam waktu O(Elog V), di mana E adalah jumlah cabang dan V adalah jumlah *node*. Dengan *Fibonacci heap*, hal ini dapat ditekan menjadi O(E + Vlog V), yang jauh lebih cepat bila grafiknya cukup padat sehingga E adalah Ω(Vlog V).

### B. Algoritma Kruskal

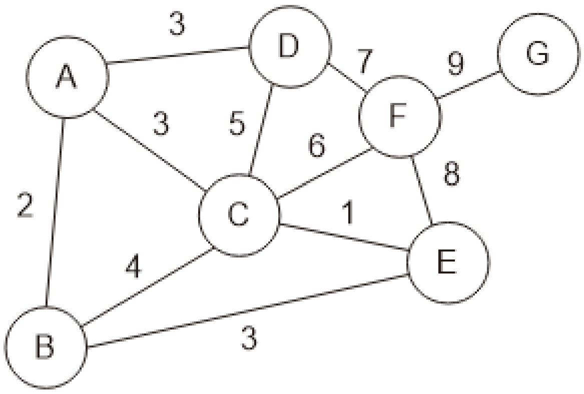
Algoritma Kruskal adalah sebuah algoritma dalam teori *graph* yang mencari sebuah *minimum spanning tree* untuk sebuah *graph* berbobot yang terhubung. Ini berarti menemukan *subset* dari *edge* yang membentuk sebuah *tree* yang mencakup setiap titik, di mana berat total dari semua *edge* di atas *tree* diminimalkan. Jika *graph* tidak terhubung, maka menemukan *minimum spanning forest* (*minimum spanning tree* untuk setiap komponen terhubung). Algoritma Kruskal juga tergolong algoritma rakus dalam *graph theory* yang digunakan untuk mencari *minimum spanning tree*. Algoritma ini pertama kali muncul pada tahun 1956 dalam sebuah tulisan yang ditulis oleh Joseph Kruskal. Algoritma ini pertama kali muncul dalam *Prosiding American Mathematical Society*, tahun 1956. Dasar pembentukan algoritma Kruskal berasal dari analogi *growing forest*. *Growing forest* maksudnya adalah untuk membentuk *minimum spanning tree* T dari *graph* G adalah dengan cara mengambil satu per satu *edge* dari *graph* G dan memasukkannya ke dalam *tree* yang telah terbentuk sebelumnya. Seiring dengan berjalannya iterasi untuk setiap *edge*, maka *forest* akan memiliki *tree* yang semakin sedikit. Oleh sebab itu, maka analogi ini disebut dengan *growing forest*. Algoritma Kruskal akan terus menambahkan semua *edge* ke dalam *forest* yang sesuai hingga akhirnya tidak akan ada lagi *forest* dengan, melainkan hanyalah *minimum spanning tree*.

## 1.3 Peralatan/perangkat yang digunakan

1. Pc/ Laptop 1 Unit.
2. Microsoft Word.
3. CorelDraw.
4. Dev-C++.

# BAB II PEMBAHASAN

Mencari *minimum spanning tree* dari *graph* di bawah ini menggunakan algoritma Prim dan algoritma Kruskal.



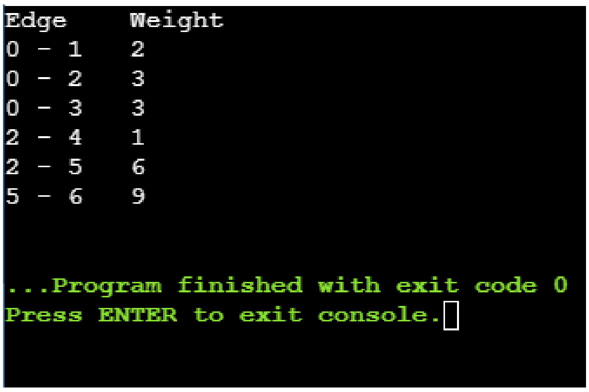

## 2.1 Algoritma Prim

### A. Source Code

```
#include <bits/stdc++.h>
using namespace std;

// Number of vertices in the graph
#define V 7

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V])
{
    cout<<"Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<" \n";
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
```

```
{
    // Array to store constructed MST
    int parent[V];

    // Key values used to pick minimum weight edge in cut
    int key[V];

    // To represent set of vertices included in MST
    bool mstSet[V];

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    // Make key 0 so that this vertex is picked as first vertex.
    key[0] = 0;
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++)
    {
        // Pick the minimum key vertex from the
        // set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of
        // the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not
        // yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent vertices of m
            // mstSet[v] is false for vertices not yet included in MST
            // Update the key only if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // print the constructed MST
    printMST(parent, graph);
}

// Driver code
int main()
{
    int graph[V][V] = { { 0, 2, 3, 3, 0, 0, 0},
                        { 2, 0, 4, 3, 0, 0, 0},
                        { 3, 4, 0, 5, 1, 6, 0},
                        { 3, 3, 5, 0, 0, 7, 0},
                        { 0, 0, 1, 0, 0, 8, 0},
                        { 0, 0, 6, 7, 8, 0, 9},
                        { 0, 0, 0, 0, 0, 9, 0} };


    // Print the solution
    primMST(graph);

    return 0;
}
```

Saat program di atas dijalankan akan memunculkan hasil sebagai berikut.

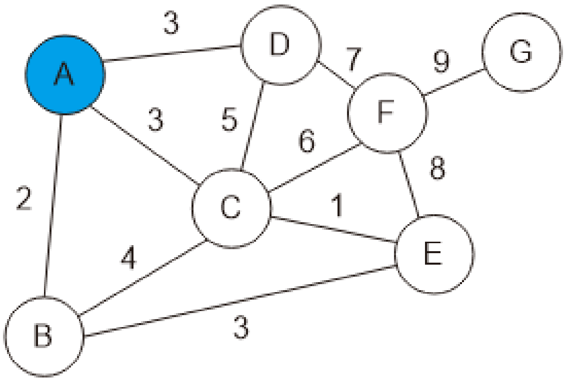


```
Edge    Weight
0 - 1   2
0 - 2   3
0 - 3   3
2 - 4   1
2 - 5   6
5 - 6   9

...Program finished with exit code 0
Press ENTER to exit console.
```

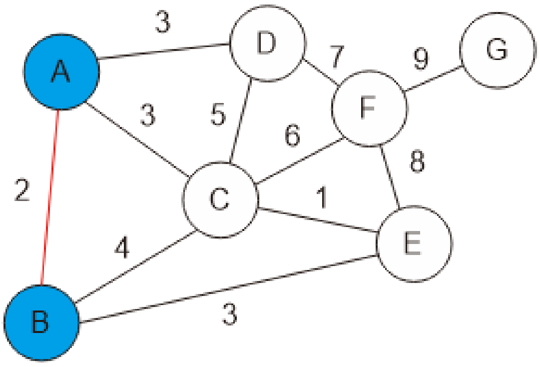
Pada program huruf A sampai G diganti menggunakan angka 0 sampai 6 di mana angka 0 berarti huruf A, angka 1 untuk huruf B, dan berurutan hingga angka 6 yang mewakili huruf G. Hasil dari program di atas menunjukkan bahwa *minimum spanning tree* dari *graph* adalah 2+3+3+1+6+9 =24 yang didapat dengan menjumlahkan semua bobot dari setiap *edge*. Urutan *edge* yang ditampilkan program sama dengan urutan langkah-langkah kerja algoritma Prim.

## B. Cara kerja

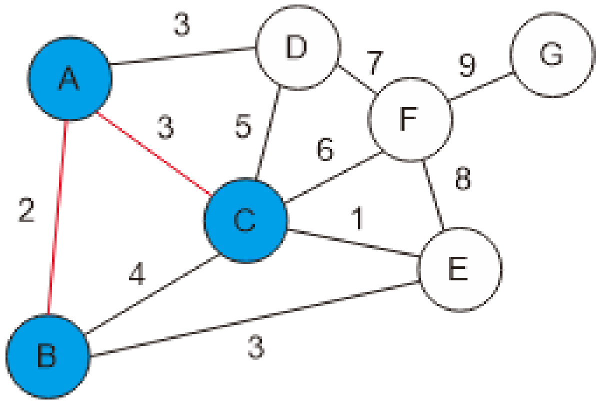
1. Memilih *vertex* yang diketahui, *vertex* tersebut adalah A.



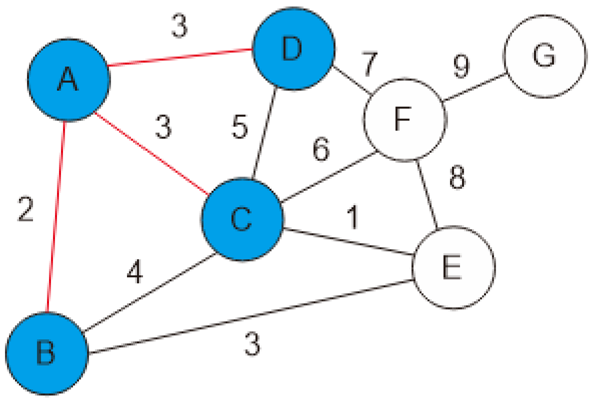

2. Menghubungkan *vertex* dengan lintasan yang paling kecil. Jika dilihat dari gambar di atas, dihubungkan ke B yang bernilai 2.



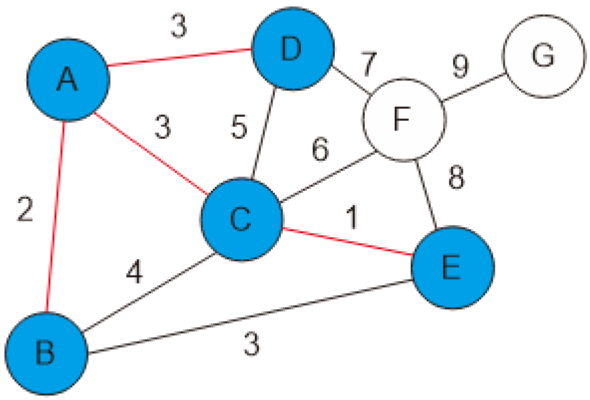
3. Menghubungkan *vertex* dengan lintasan yang paling kecil lainnya. Jika dilihat dari gambar, huruf C bernilai 3 Maka melewati lintasan C.



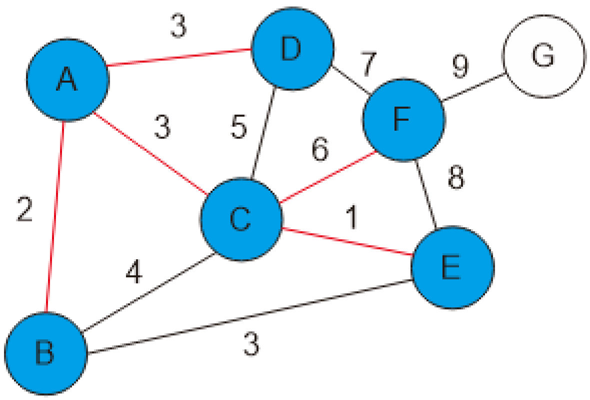

4. Menghubungkan *vertex* dengan lintasan yang paling kecil lainnya. Jika dilihat dari gambar, huruf D bernilai 3. Maka melewati lintasan D.



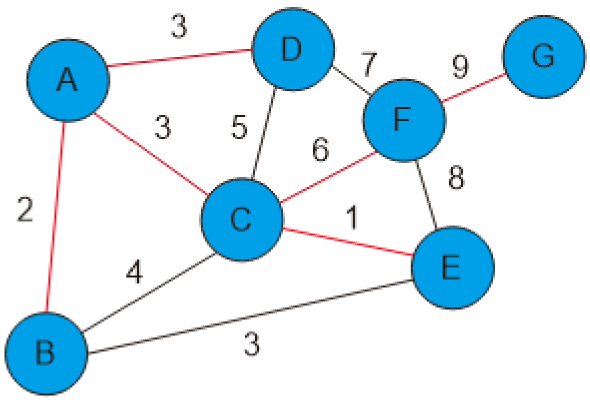

5. Karena *vertex* sudah terhubung dengan semua lintasan yang memungkinkan, langkah selanjutnya adalah menghubungkan lintasan dengan bobot paling kecil yang tidak menimbulkan *cycle*, yaitu lintasan C ke E yang bernilai 1.




6. Menghubungkan dengan sisa huruf yang belum terhubung yaitu F dan G. Sebelum ke G harus terhubung terlebih dahulu dengan huruf F. Lintasan dengan bobot paling kecil yaitu C ke F dengan nilai 6.




7. Menghubungkan dengan G yang bernilai 9. Pemilihan E yang bernilai 8 hanya akan menghasilkan *cycle*.




8. Maka dapat diketahui bahwa, nilai dari *minimum spanning tree* dengan menghitung semua bobot yaitu, 2 + 3 + 3 + 1 + 6 + 9 = 24.

## 2.2 Algoritma Kruskal

### A. Source Code

```
#include<bits/stdc++.h>
using namespace std;

// Creating shortcut for an integer pair
typedef pair<int, int> iPair;

// Structure to represent a graph
struct Graph
{
    int V, E;
    vector< pair<int, iPair> > edges;

    // Constructor
    Graph(int V, int E)
    {
        this->V = V;
        this->E = E;
    }

    // Utility function to add an edge
```

```cpp
    void addEdge(int u, int v, int w)
    {
        edges.push_back({w, {u, v}});
    }

    // Function to find MST using Kruskal's
    // MST algorithm
    int kruskalMST();
};

// To represent Disjoint Sets
struct DisjointSets
{
    int *parent, *rnk;
    int n;

    // Constructor.
    DisjointSets(int n)
    {
        // Allocate memory
        this->n = n;
        parent = new int[n+1];
        rnk = new int[n+1];

        // Initially, all vertices are in
        // different sets and have rank 0.
        for (int i = 0; i <= n; i++)
        {
            rnk[i] = 0;

            //every element is parent of itself
            parent[i] = i;
        }
    }

    // Find the parent of a node 'u'
    // Path Compression
    int find(int u)
    {
        /* Make the parent of the nodes in the path

        from u--> parent[u] point to parent[u] */
        if (u != parent[u])
            parent[u] = find(parent[u]);
        return parent[u];
    }

    // Union by rank
    void merge(int x, int y)
    {
        x = find(x), y = find(y);

        /* Make tree with smaller height
        a subtree of the other tree */
        if (rnk[x] > rnk[y])
            parent[y] = x;
        else // If rnk[x] <= rnk[y]
            parent[x] = y;

        if (rnk[x] == rnk[y])
            rnk[y]++;
```

```
    }
};

/* Functions returns weight of the MST*/

int Graph::kruskalMST()
{
    int mst_wt = 0; // Initialize result

    // Sort edges in increasing order on basis of cost
    sort(edges.begin(), edges.end());

    // Create disjoint sets
    DisjointSets ds(V);

    // Iterate through all sorted edges
    vector< pair<int, iPair> >::iterator it;
    for (it=edges.begin(); it!=edges.end(); it++)
    {
        int u = it->second.first;
        int v = it->second.second;

        int set_u = ds.find(u);
        int set_v = ds.find(v);

        // Check if the selected edge is creating
        // a cycle or not (Cycle is created if u
        // and v belong to same set)
        if (set_u != set_v)
        {
            // Current edge will be in the MST
            // so print it
            cout << u << " - " << v << endl;

            // Update MST weight
            mst_wt += it->first;

            // Merge two sets
            ds.merge(set_u, set_v);
        }
    }

    return mst_wt;
}

// Driver program to test above functions
int main()
{
    /* Let us create above shown weighted
    and unidrected graph */
    int V = 7, E = 9;
    Graph g(V, E);

    // making above shown graph
    g.addEdge(1, 2, 2);
    g.addEdge(1, 3, 3);
    g.addEdge(1, 4, 3);
    g.addEdge(2, 3, 4);
    g.addEdge(2, 5, 3);
    g.addEdge(3, 4, 5);
```

```
    g.addEdge(3, 5, 1);
    g.addEdge(3, 6, 6);
    g.addEdge(4, 6, 7);
    g.addEdge(5, 6, 8);
    g.addEdge(6, 7, 9);

    cout << "Edges of MST are \n";
    int mst_wt = g.kruskalMST();

    cout << "\nWeight of MST is " << mst_wt;

    return 0;
}
```

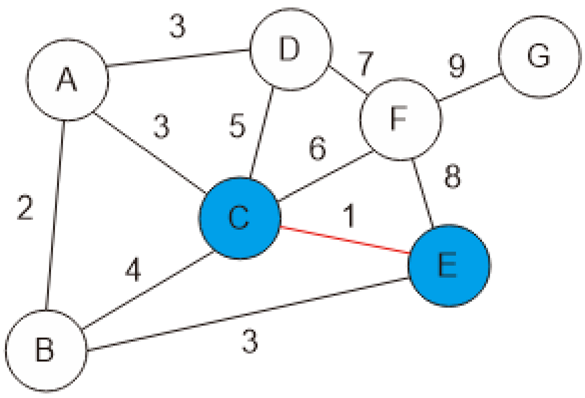
Saat program di atas dijalankan akan memunculkan hasil sebagai berikut.

```
Edges of MST are
3 - 5
1 - 2
1 - 3
1 - 4
3 - 6
6 - 7

Weight of MST is 24

...Program finished with exit code 0
Press ENTER to exit console.
```

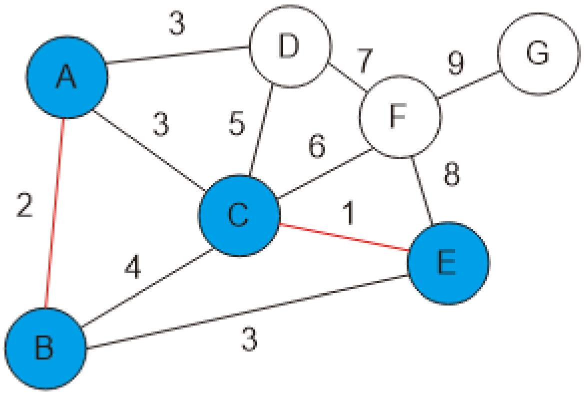
Pada program huruf A sampai G diganti menggunakan angka 1 sampai 7 di mana angka 1 berarti huruf A, angka 2 untuk huruf B, dan berurutan hingga angka 7 yang mewakili huruf G. Hasil dari program di atas menunjukkan bahwa *minimum spanning tree* dari *graph* adalah 24 yang didapat dengan menjumlahkan semua bobot dari setiap *edge*. Secara berurutan bobot dari *edge* adalah 1, 2, 3, 3, 6, dan 9. Urutan *edge* yang ditampilkan program sama dengan urutan langkah-langkah kerja algoritma Kruskal.

## B. Cara kerja

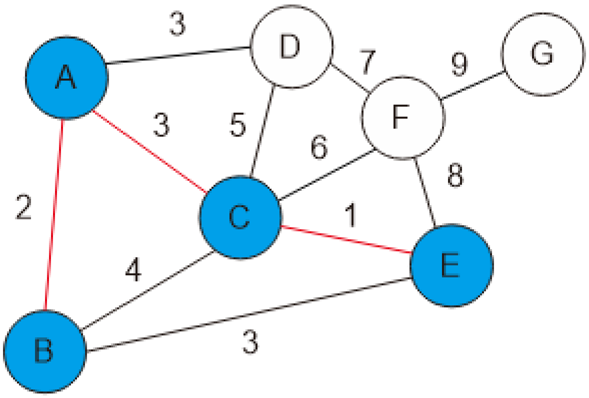
1. Memilih *edge* yang memiliki bobot paling kecil yakni C-E yang bernilai 1.



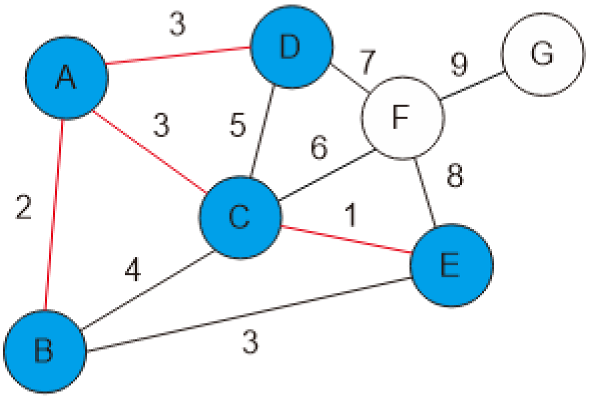
2. Memilih *edge* yang memiliki bobot paling kecil berikutnya, yakni A-B yang bernilai 2.



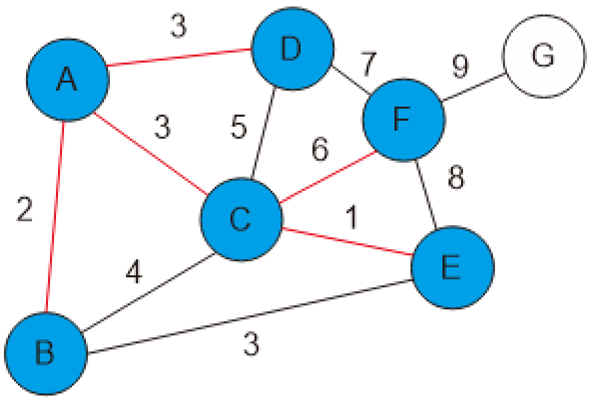

3. Memilih *edge* yang memiliki bobot paling kecil berikutnya yang menghubungkan dua lintasan, yakni jalur A-C dan B-E yang nilainya sama-sama 3. Secara arbiter jalur yang akan dipilih adalah A-C.



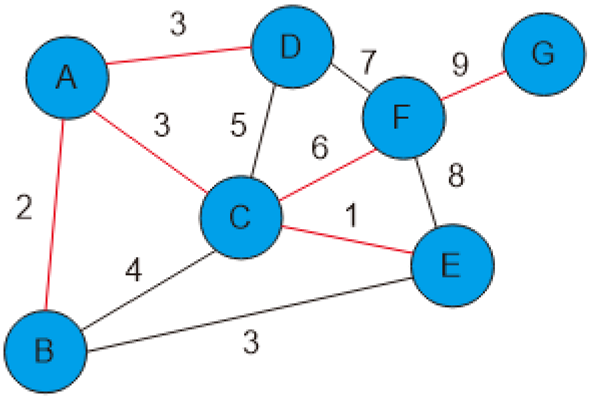

4. Memilih *edge* yang memiliki bobot paling kecil berikutnya, yakni A-D yang bernilai 3.




5. Memilih *edge* yang memiliki bobot paling kecil selain jalur A-C dan B-E, Satu-satunya jalur dengan bobot paling kecil tanpa menimbulkan *cycle* adalah jalur C-F yang bernilai 6.




6. Menghubungkan jalur F-G yang bernilai 9.




7. Maka dapat diketahui bahwa, nilai dari *minimum spanning tree* dengan menghitung semua bobot yaitu, 1 + 2 + 3 + 3 + 6 + 9 = 24.

# BAB III PENUTUP

## 3.1 Kesimpulan

Dari penghitungan *minimum spanning tree* di atas dapat diketahui perbedaan dari algoritma Prim dan algoritma Kruskal. Perbedaan tersebut ditampilkan pada tabel di bawah ini.

| No | Algoritma Prim | Algoritma Kruskal |
|---|---|---|
| 1 | Penyelesaian masalah dimulai dari *node* | Penyelesaian masalah dimulai dari *edge* |
| 2 | Terbentang dari *node* satu ke *node* lainnya | Memilih posisi *edge* dengan tidak berdasar pada langkah terakhir |
| 3 | *Graph* penyelesaian harus berhubungan | *Graph* penyelesaian dapat terputus |
| 4 | Kompleksitas waktu = O ($V^2$) | Kompleksitas waktu = O (log V) |

## 3.2 Referensi


Wikipedia. 2021. “Algoritma Prim”, https://id.wikipedia.org/wiki/Algoritma_Prim, diakses pada 19 Juni 2021 pukul 20.55.

Purbadinata, Rizal Aji. 2021. “Week 11 – Graph 2”, https://classroom.google.com/u/1/c/MzExMjIzMTIxODA4/m/MzYxMjYwMDY1MDU0/details, diakses pada 15 Juni 2021 pukul 16.50.

Mustofa, Zaenal. 2017. “Makalah Algoritma Kruskal”,www.slideshare.net/zaenalmustofa54943/makalah-algoritma-kruskal, diakses pada 19 Juni 2021 pukul 22.36.