ALGORITMO DE BORŮVKA

Disciplina Algoritmos e Estrutura de Dados 2 - [Semestre Letivo: 2022/1]	
Nome dos(as) acadêmicos(as) 1 – Iury Alexandre Alves Bo 2 – Luca Mascarenhas Plaster 3 – Maria Rafaela dos Anjos	**Números de matrícula** 1 – 202103735 2 – 202014610 3 – 202108525
Turma: INF0287C	**Professor(a):** Wanderley de Souza Alencar

TEMA: ALGORITMO DE BORŮVKA

I – OBJETIVO

O Algoritmo de Borůvka (ou Algoritmo de Sollin como também é conhecido) é um algoritmo para encontrar uma árvore geradora mínima em um grafo para o qual todos os pesos de arestas sejam distintos. Este algoritmo caracteriza-se pela divisão do grafo original em vários subgrafos para os quais é calculado a *Minimum Spanning Tree* (árvore geradora mínima). Ou seja, no fundo, pode ser considerada uma variação de algoritmos como os de Prim e Kruskal, que são grafos altamente recorrentes para buscar custos mínimos de suas árvores geradoras.

O funcionamento do algoritmo de Borůvka segue os seguintes passos:

1. Para cada vértice, escolher a sua aresta de menor custo. Deste passo poderão resultar um ou mais subgrafos.
2. Caso o passo 1 dê origem a grafos não conectados, considere cada subgrafo gerado no passo anterior como um vértice do grafo final. Então, repita o primeiro passo, encarando cada subgrafo como se fosse um vértice e olhando para as arestas entre esses subgrafos.

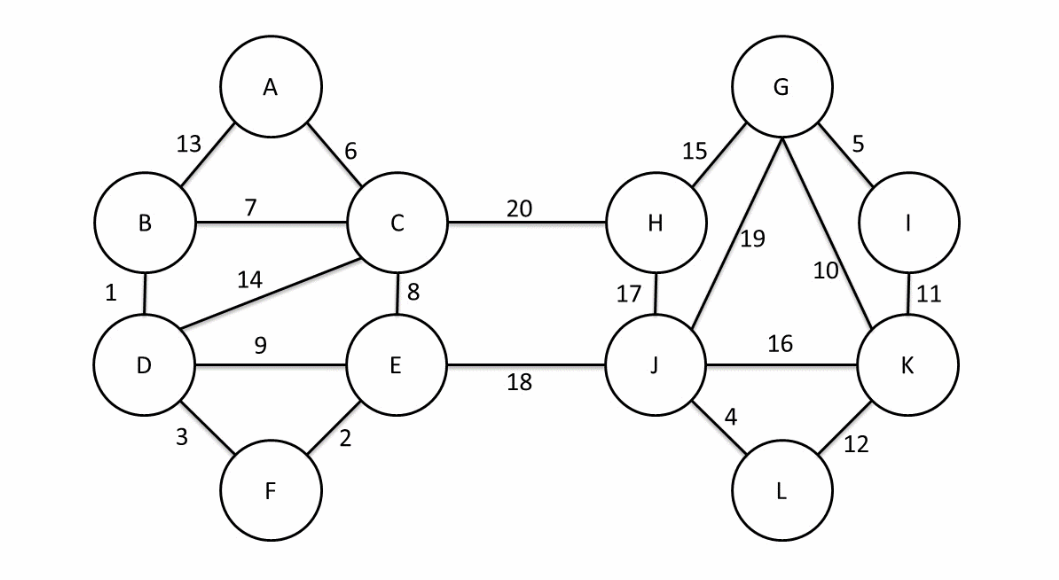
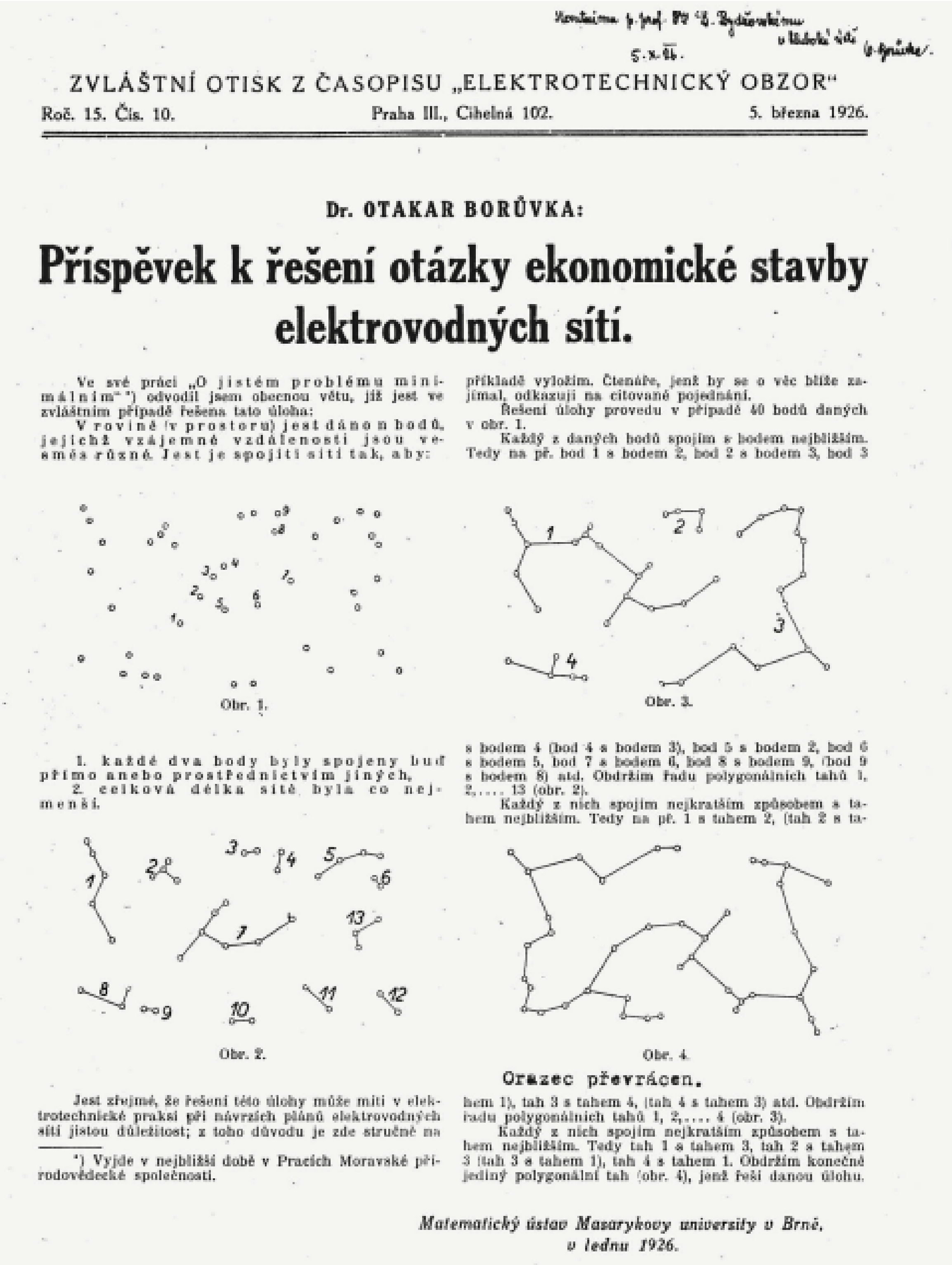


Figura 1: Exemplo prático do algoritmo de Borůvka

Outras curiosidades:

- O algoritmo de Borůvka é o algoritmo de árvore geradora mínima mais antigo descoberto, por Borůvka em 1926, muito antes de os computadores existirem. O algoritmo foi publicado como um método de construção de uma rede elétrica eficiente.

- O problema de mínima arborescência proposta por Borůvka foi registrado por um artigo de uma única página em 1926, tendo como em vista a contribuição para a solução de um problema econômico para a construção de uma rede de energia elétrica por meio de cabos.
- A complexidade de tempo do algoritmo de Borůvka é O(E *log*(V)), que é o mesmo que os algoritmos de Kruskal e Prim.
- O algoritmo de Borůvka é usado como um passo em um algoritmo aleatório mais rápido que funciona em tempo linear O(E). O algoritmo MST de tempo linear esperado é um algoritmo aleatório que calcula a floresta de extensão mínima de um grafo ponderado sem vértices isolados.



ZVLÁŠTNÍ OTISK Z ČASOPISU „ELEKTROTECHNICKÝ OBZOR"

Roč. 15. Čís. 10. Praha III., Cihelná 102. 5. března 1926.

Dr. OTAKAR BORŮVKA:

Příspěvek k řešení otázky ekonomické stavby elektrovodných sítí.

Ve své práci „O jistém problému minimálním"*) odvodil jsem obecnou větu, jíž jest ve zvláštním případě řešena tato úloha:

V rovině (v prostoru) jest dáno n bodů, jejichž vzájemné vzdálenosti jsou veskrze různé. Jest je spojiti sítí tak, aby:

Obr. 1.

1. každé dva body byly spojeny buď přímo anebo prostřednictvím jiných,
2. celková délka sítě byla co nejmenší.

Obr. 2.

Jest zřejmé, že řešení této úlohy může míti v elektrotechnické praksi při návrzích plánů elektrovodných sítí jistou důležitost; z toho důvodu je zde stručně na příkladě vyložím. Čtenáře, jenž by se o věc blíže zajímal, odkazuji na citované pojednání.

Řešení úlohy provedu v případě 40 bodů daných v obr. 1.

Každý z daných bodů spojím s bodem nejbližším. Tedy na př. bod 1 s bodem 2, bod 2 s bodem 3, bod 3

Obr. 3.

s bodem 4 (bod 4 s bodem 3), bod 5 s bodem 2, bod 6 s bodem 5, bod 7 s bodem 6, bod 8 s bodem 9, (bod 9 s bodem 8) atd. Obdržím řadu polygonálních tahů 1, 2, ... 13 (obr. 2).

Každý z nich spojím nejkratším způsobem s tahem nejbližším. Tedy na př. 1 s tahem 2, (tah 2 s tahem 1), tah 3 s tahem 4, (tah 4 s tahem 3) atd. Obdržím řadu polygonálních tahů 1, 2, ... 4 (obr. 3).

Obr. 4.

Obrazec převrácen.

Každý z nich spojím nejkratším způsobem s tahem nejbližším. Tedy tah 1 s tahem 3, tah 2 s tahem 3 (tah 3 s tahem 1), tah 4 s tahem 1. Obdržím konečně jediný polygonální tah (obr. 4), jenž řeší danou úlohu.

Matematický ústav Masarykovy university v Brně, v lednu 1926.

*) Vyjde v nejbližší době v Pracích Moravské přírodovědecké společnosti.

Figura 2: Manchete de pesquisa do Borůvka

II – ALGORITMO

1. Representação computacional do grafo:

```c
// estrutura de um vértice
struct vertice {
    int *arestas; // guarda as conexões, é a lista de adjacências, cada elemento do vetor é uma aresta do vértice com outro
    float *pesos; // se for grafo ponderado (as ligações possuem peso), cada elemento representa o peso de uma das arestas do vetor de arestas
    int grau; // quantas ligações existe no vértice
};

typedef struct vertice Vertice;

// estrutura de um grafo
struct grafo {
    int eh_ponderado; // informa se é ponderado ou não (as ligações possuirem peso)
    int n_vertices; // número de elementos (deve ser informado e fixo)
    int grau_max; // grau máximo: quantas arestas/ligações cada vértice/nó pode ter
    Vertice *vertices; // vetor de vertices, cada posição é um vértice (lista de adjacências)
};
```

2. Função - Algoritmo de Boruvka - Parte 1:

```c
void algoritmoBoruvka (Grafo *grafo, int *ordem) {

    // Requisito: Grafo fortemente conexo com arestas ponderadas estabelecidas entre todos eles, não pode haver grafos que
    // ligam a si mesmos ou grafos com duas arestas não direcionadas ou grafos com muitas arestas de mesmo peso.

    int **grupos = alocarGrupos(grafo->n_vertices); // alocando os grupos dinamicamente
    int *validos = alocarVetor_I(grafo->n_vertices); // alocando o vetor de inteiros dinamicamente

    inicializaGrupos(grupos, grafo->n_vertices); // colocando o primeiro elemento em cada grupo e -1 no resto das posições dos grupos

    inicializaValidos(validos,grafo->n_vertices); // colocando 1 em todas as posições do vetor de valores válidos

    inicializaOrdem(ordem, grafo->n_vertices); // colocando -1 em todas as posições do vetor de valores da ordem

    int cont = 1; // contador de quantos grupos já foram somados
    int i, j; // auxiliar
    int vmp; // vizinho mais próximo
    int posVMP; // posição do vizinho mais próximo
    int *ord; // guarda a ordem qual elemento eu preciso acessar para chegar em outro elemento

    // Calcular a ordem por meio da ideia de grupos de Boruvka
    while (cont < grafo->n_vertices) {
        for (i = 0; i < grafo->n_vertices; i++) {
            if (validos[i]) printf("\n\n\t%dº Estudo de Grupo\n\n", cont);
            if (!validos[i]) continue; // se não é um grupo válido, vai pra próxima iteração

            printf("\tGrupo sendo estudado:\n");
            imprimirGrupo(grupos[i]);

            // calculando a posição do vizinho mais próximo de i (é definido que existe um mais próximo)
            posVMP = VMPdoGrupo(grafo->vertices, grupos[i], ord);

            // calculando o vizinho mais próximo
            vmp = grafo->vertices[ord[0]].arestas[posVMP];

            // vmp encontrado no Vertice = ord[0];
            for (j = 0; j < grafo->n_vertices; j++) {
                if (!validos[j]) continue; // se não é um grupo válido, vai pra próxima iteração
                if (buscaVerticeNoGrupo(grupos[j],vmp) == 1) break; // se encontrar o elemento VMP no grupo[j]
            }
            printf("\tO VMP esta no grupo '%d', entao precisamos unir os dois grupos\n\n", j);
```

3. Algoritmo de Boruvka - Parte 2:

```c
            // unindo os dois grupos a partir do grupo do vizinho mais próximo
            // aumentar no grupo de menor índice
            if (i < j) {
                printf("\tUniao do grupo '%d' com o '%d':\n", i, j);
                unirGrupos(grupos[i],grupos[j]);
                imprimirGrupo(grupos[i]);

                printf("\tComo unimos G%d a outro grupo, G%d nao eh mais um grupo valido\n\n", j, j);
                validos[j] = 0; // não é mais um grupo válido
                ordem[vmp] = ord[0]; // deve ser acessado por ord[0], que é seu vizinho mais próximo
                if (ordem[ord[0]] == -1) { // se não houver um valor válido no vizinho mais próximo, então precisamos colocar um meio de chegar nele também
                    ordem[ord[0]] = vmp; // eve ser acessado por VMP, que é seu vizinho mais próximo
                }
            }
            else {
                printf("\tTemos sempre que unir em direcao ao grupo de menor indice, entao uniremos '%d' ao '%d':\n", j, i);
                unirGrupos(grupos[j], grupos[i]);
                imprimirGrupo(grupos[j]);

                printf("\tComo unimos G%d a outro grupo, G%d nao eh mais um grupo valido\n\n", i, i);
                validos[i] = 0; // invalidar o maior grupo

                // Como a direção acesso é sempre do G menor para o G maior
                // Temos que colocar que a ordem do elemento encontrado veio do seu vizinho mais próximo
                ordem[ord[0]] = vmp; // assim como o VMP vem dele, ele também vem do VMP, mas nesse caso, privilegiamos o VMP
                if (ordem[vmp] == -1) { // se não houver um valor válido no vizinho mais próximo, então precisamos colocar um meio de chegar nele também
                    ordem[vmp] = ord[0]; // deve ser acessado por ord[0], que é seu vizinho mais próximo
                }
            }

            printf("\tOrdem de visitacao ate agora:\n\n");
            imprimirOrdem(ordem, grafo->n_vertices);

            cont++; // mais um grupo somado
        }
        // printf("cont: %d\n", cont);
    }
    imprimirGrupo(grupos[0]);
    liberarGrupos(grupos,grafo->n_vertices);
}
```

4. Alocar e Liberar grupos:

```c
int **alocarGrupos(int tam) {
    int i;
    int **grupos = malloc(tam*sizeof(int *));
    if (!grupos) {
        printf("Erro! Memoria insuficiente!\n");
        return NULL;
    }
    for (int i = 0; i < tam+1; i++) {
        grupos[i] = malloc(sizeof(int)*(tam+1));
        if (!grupos[i]) {
            printf("Erro! Memoria insuficiente!\n");
            return NULL;
        }
    }
    return grupos;
}

void liberarGrupos(int **grupos, int tam) {
    int i;
    for (i = 0; i < tam; i++) {
        free(grupos[i]);
    }
    free(grupos);
}
```

5. Inicializações:

```c
void inicializaGrupos(int **grupos, int tam) {
    int i, j;
    for (i = 0; i < tam; i++) {
        for (j = 0; j < tam+1; j++) {
            if (j == 0) grupos[i][j] = i;
            else grupos[i][j] = -1;
        }
    }
}

void inicializaValidos(int *validos, int tam) {
    int i;
    for (i = 0; i < tam; i++) validos[i] = 1;
}

void inicializaOrdem(int *ordem, int tam) {
    int i;
    for (i = 0; i < tam; i++) ordem[i] = -1;
}
```

6. Impressões:

```c
void imprimirGrupo(int *grupo) {
    int tam = tamanhoGrupo(grupo);
    int i;
    printf("\t{[%d]", grupo[0]);
    for (i = 1; i < tam; i++) {
        printf(",[%d]", grupo[i]);
    }
    printf("}\n\n");
}

void imprimirOrdem(int *ordem, int tam) {
    int i;
    for (i = 0; i < tam; i++) {
        if (i == 0) printf("\n\tVertice -> -", i);
        else printf(" -", i);
    }
    printf("\n\t");
    i = 0;
    printf(" Acesso -> -", ordem[i]);
    i++;
    for (; i < tam; i++) {
        printf(" -", ordem[i]);
    }
    printf("\n");
}
```

7. Tamanho do grupo:

```c
int tamanhoGrupo(int *grupo) {
    int tam;
    // encontrar posição do último elemento de G1
    for (tam = 0; grupo[tam] != -1; tam++);
    // printf("\nTamanho do grupo: %d\n",tam); // Debug verificação
    return tam;
}
```

8. Vizinho mais próximo do grupo:

```c
int VMPdoGrupo (Vertice *V, int *grupo, int *ord) {

    int tam = tamanhoGrupo(grupo); // tamanho do grupo para verificarmos cada vértice do grupo
    int i;
    int posVP; // vizinho mais próximo do vertice
    int posVMP; // vizinho mais próximo do grupo
    int check = 1;

    // Veja bem, cada elemento do meu grupo representa um VÉRTICE V[grupo[i]] do grafo
    // Cada posVP representa a posição do vizinho mais próximo
    // O i é o auxiliar que eu uso para chegar em um valor possível dentro dos meus valores do grupo
    for (i = 0; i < tam; i++) {
        printf("\tAvaliando o vertice '%d'\n\n", grupo[i]);
        posVP = VMPdoVertice(V[grupo[i]], grupo); // calculando o vizinho mais próximo de i (é definido que existe um mais próximo)

        if (posVP == -1) {
            continue; // significa que valor não há valores válidos para este vértice mais
        }

        // Tendo o vetor de vizinhos mais próximos, precisamos saber qual é o vizinho mais próximo definitivo, o menor peso
        if (check == 1) { // primeiro valor válido (pode não estar em i == 0)
            posVMP = posVP; // se for o primeiro, damos valor válido para entrar na próxima iteração
            // o valor que eu peguei foi encontrado graças ao Vert. = grupo[i], preciso guardar ele para saber a ordem
            ord[0] = grupo[i];
            check = 0; // não é mais o primeiro, devo verificar se é menor que o valor que está guardado
        }
        else {
            if (V[grupo[i]].pesos[posVP] < V[ord[0]].pesos[posVMP]) {
                posVMP = posVP;
                // o valor que eu peguei foi encontrado graças ao V[grupo[i]], preciso guardar ele para saber a ordem
                ord[0] = grupo[i];
            }
        }
    }
    printf("\tO vizinho mais proximo definitivo foi encontrado no vertice '%d' ele eh o vizinho de posicao '%d'\n", ord[0], posVMP);
    printf("\tEssa posicao armazena o vertice '%d' de peso %.2f\n\n", V[ord[0]].arestas[posVMP], V[ord[0]].pesos[posVMP]);

    return posVMP; // retorno da posição do vizinho mais próximo entre os elementos do meu grupo
}
```

9. Vizinho mais próximo do vértice:

```c
int VMPdoVertice (Vertice V, int *grupo) {

    // Posição do vizinho mais próximo, também preciso acessar o peso dessa posição
    // Por isso não guardo o valor da posição
    int posVMP = -1;

    int i;
    for (i = 0; i < V.grau; i++) {
        printf("\t\tVendo se o vizinho[%d]: %d de peso: %.2f é o vizinho mais proximo desse vertice\n\n", i, V.arestas[i], V.pesos[i]);
        if (posVMP == -1) { // se for o primeiro valor
            if (buscaVerticeNoGrupo(grupo,V.arestas[i]) == FALSE) posVMP = i;
            else printf("\t\tO vizinho ja pertence ao grupo, nao deve ser considerado\n\n");
        }
        if (posVMP != -1) {
            if (buscaVerticeNoGrupo(grupo,V.arestas[i]) == FALSE && V.pesos[i] < V.pesos[posVMP]) {
                // printf("Vertice[%d]: %d, Peso: %f\n\n", posVMP, V.arestas[posVMP], V.pesos[posVMP]);
                posVMP = i; // se encontrar um vizinho mais próximo VÁLIDO
            }
        }
    }
    if (posVMP != -1) printf("\t\tO vizinho[%d]: %d, de peso: %.2f eh um candidato de ser o vizinho mais proximo do grupo\n\n", posVMP, V.arestas[posVMP], V.pesos[posVMP]);
    else printf("\t\tO vertice nao possui candidatos a serem considerados, todos seus vizinhos fazem parte do grupo\n\n");
    return posVMP; // retorno o valor do vizinho mais próximo desse vértice
}
```

10. Buscar vértice no grupo:

```c
int buscaVerticeNoGrupo(int *grupo, int vertice) {
    int tam = tamanhoGrupo(grupo);
    int i;
    for (i = 0; i < tam; i++) {
        if (vertice == grupo[i]) {
            // se encontrar o vértice neste grupo, é porque o vértice está nesse grupo
            return TRUE;
        }
    }
    return FALSE;
}
```

11. Unir grupos:

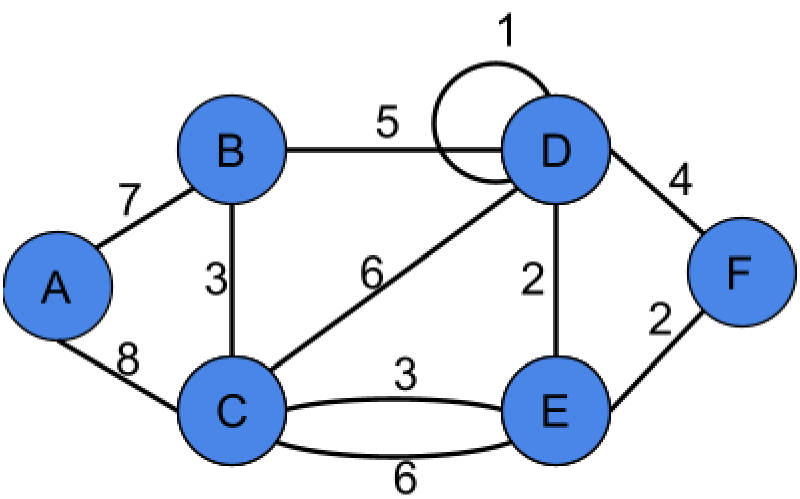
```c
void unirGrupos(int *G1, int *G2) {
    int i, j;
    i = tamanhoGrupo(G1);
    // adicionar elementos de G2 em G1 a partir do i encontrado
    for (j = 0; G2[j] != -1; j++, i++) G1[i] = G2[j];
}
```

12. Calcular distância

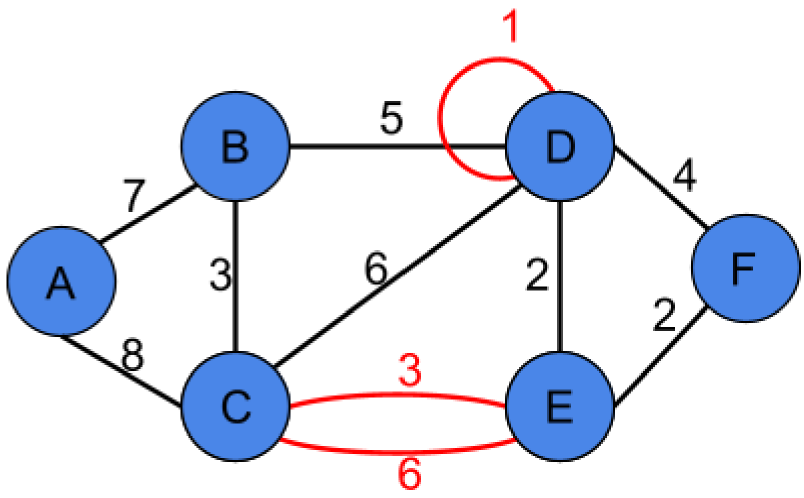
```c
void calcularDist(Grafo *grafo, int *ordem, float *dist) {
    int i, j;
    float md; // menor distância

    md = 0;

    for (i = 1; i < grafo->n_vertices; i++) {
        for (j = 0; j < grafo->n_vertices; j++) {
            if (ordem[i] == grafo->vertices[i].arestas[j]) break; // encontrando a posição do vertice vizinho 'ordem[i]' em V[i]
        }
        dist[i] = grafo->vertices[i].pesos[j];
    }

    imprimirDist(dist,grafo->n_vertices);
    printf("\n\tCalculando arvore geradora de custo minimo (minimum spanning tree)\n");
    printf("\tPartindo do vertice 0, a distancia total para percorrer todos os vertices, eh: ");
    for (i = 0; i < grafo->n_vertices; i++) md += dist[i];
    printf("%.2f\n\n", md);
}

void imprimirDist(float *dist, int tam) {
    int i;
    for (i = 0; i < tam; i++) {
        if (i == 0) printf("\n\t Vertices -> M", i);
        else printf(" M", i);
    }
    printf("\n\t");
    i = 0;
    printf("Distancia -> %.2f", dist[i]);
    i++;
    for (; i < tam; i++) {
        printf(" %.2f", dist[i]);
    }
    printf("\n");
}
```

III – EXEMPLO

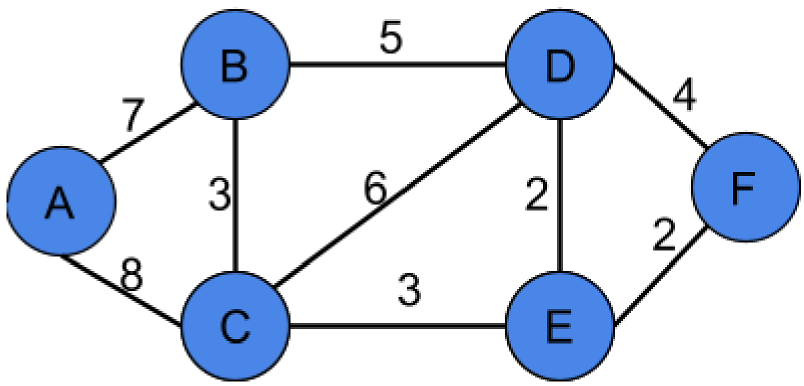
Vamos utilizar os passos apresentados no tópico I deste documento para encontrar o menor caminho que percorra todos os vértices do grafo abaixo. Observe que há arestas com o mesmo custo.



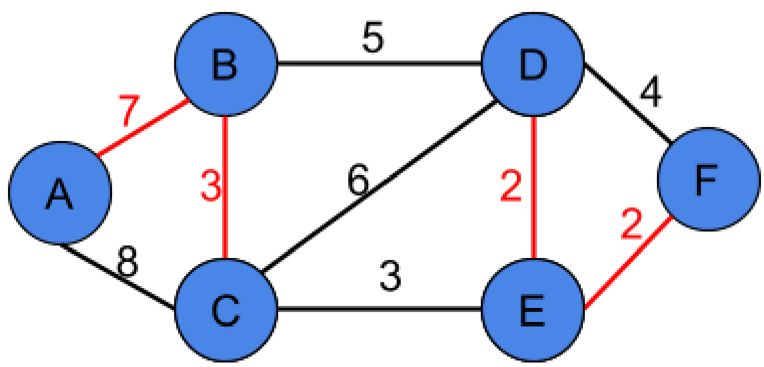
Observação: Antes de tudo devemos observar algumas particularidades do grafo, como arestas paralelas e ciclos.



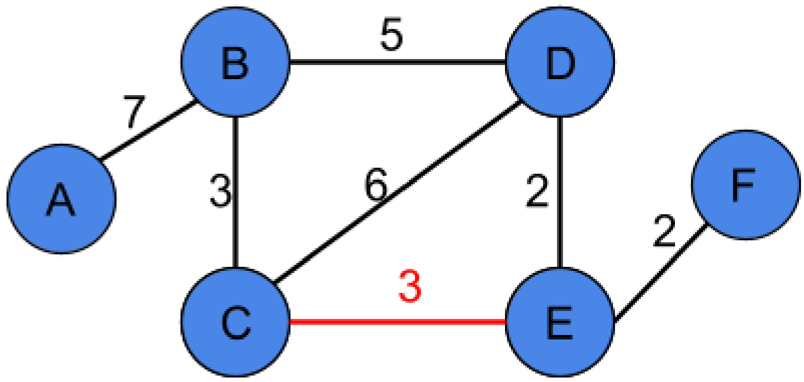
Logo, removeremos o ciclo ligado ao vértice D e a aresta de maior custo (6) que liga C e E, já que são paralelas.



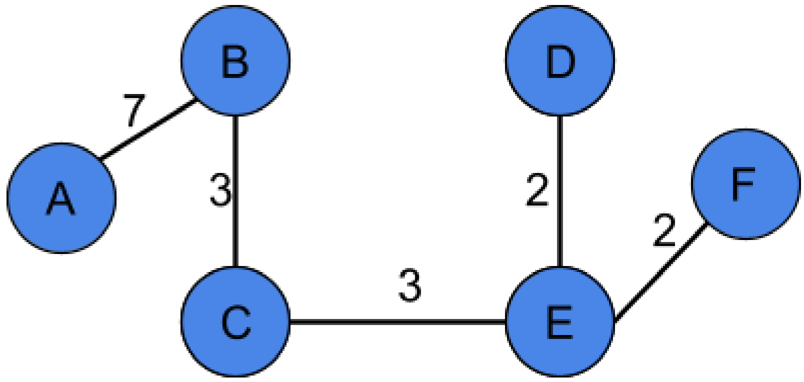
Agora, mantemos a aresta de menor custo referente a cada vértice,mesmo que ela também seja a de menor custo de outro vértice, e desconsideramos as outras.
A aresta de menor custo do vértice A é AB= 7;
A aresta de menor custo do vértice B é BC= 3;
A aresta de menor custo do vértice C também é BC= 3;
A aresta de menor custo do vértice D é DE= 2;
A aresta de menor custo do vértice E também é DE= 2;
A aresta de menor custo do vértice F é EF= 2;



Dessa forma, devemos observar que se o grafo estiver subdividido, deve-se considerar a aresta de menor custo que une os subgrafos. Logo, CE= 3.



Devem existir (n-1) arestas, sendo n o número de vértices. Assim, 6-1=5, que corresponde ao número de arestas do nosso caminho mínimo.



Solução final: 7+3+3+2+2= 17
Supondo que a distância entre um vértice e outro é dada em metros, 17m seria a distância mínima para passar por todos os vértices.

IV – BIBLIOGRAFIA

ALIESERAJ, **Animation of Boruvka's algorithm** via Wikipedia, 2012. Disponível em: <https://en.wikipedia.org/wiki/Borůvka%27s_algorithm#/media/File:Boruvka's_algorithm_(Sollin's_algorithm)_Anim.gif>;

INDUSTRIAL21. **Algoritmo de Boruvka: Teoria dos Grafos (Exemplo Prático)**. Youtube, 2021. Disponível em: <https://www.youtube.com/watch?v=3cjmO2GCyhU>;

O Boruvka, **Príspevek k reseníotázky ekonomické stavby elektrovodných sítí** (Contribution to the solution of a problem of economical construction of electrical networks), Elektronický obzor 15 (1926), 153–154;

Pesquisa Operacional. **Algoritmo de Boruvka: Exemplo Numérico**. Youtube, 2020. Disponível em: <https://www.youtube.com/watch?v=cvxnF1VqVx4>.