Alunos: Vinícius Guimarães. RA: 802431
Vitor Enzo. RA: 802123

/ /

(1) • Make_Set (V)

A função Make_Set é responsável por criar uma árvore contendo um único vértice, que é V (Raiz). Será utilizado posteriormente para validar se uma aresta é segura ou não (Para decidir se será selecionada)

```
Make_Set(V) {
    V.p = V        // O pai o vértice V é o próprio vértice V (Raiz)
    V.rank = 0     // Altura da árvore
}
```

• Find_Set (V)

A função Find_Set é responsável por retornar qual árvore que V pertence.

```
Find_Set(V) {
    if V ≠ V.p          // Se o vértice atual for diferente do pai
        V.p = Find_Set(V.p)
    return V.p          // retorna a raiz da árvore
}
```

A função vai passando de pai em pai até encontrar a raiz da árvore.

• Union (U, V)

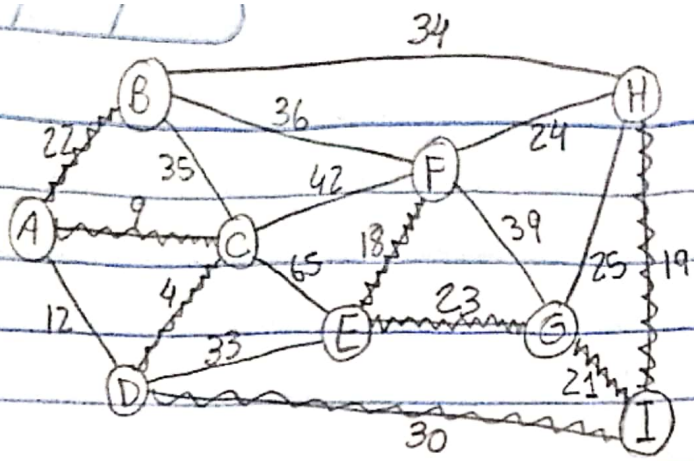
A função é responsável por juntar duas árvores (Árvore de U e árvore de V) gerando uma única árvore

```
Union(U,V) {
    Link(Find_Set(U), Find_Set(V))
}
```

// Como estamos executando Find_Set para U e V, então temos as árvores que U e V fazem parte. Assim, passamos a raiz dessas árvores para a função Link, que irá juntar as duas árvores pela raiz.

A função Link por sua vez, considera o atributo rank das duas raízes para saber qual será "pai" de qual e conseguir realizar a função

(2)



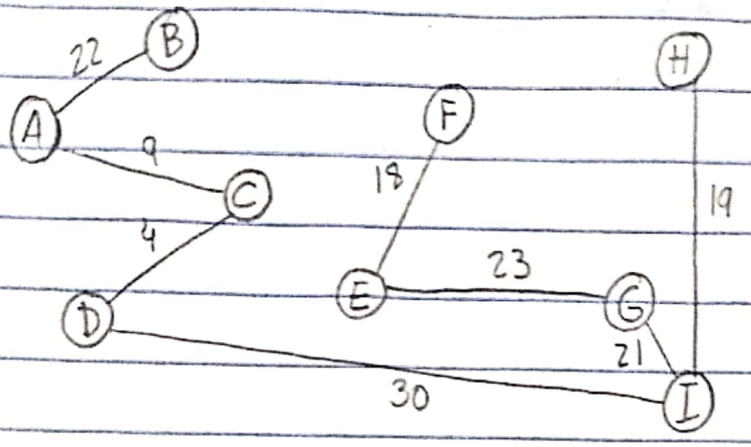
Lista das arestas ordenadas por peso

$E_W = [4, 9, 12, 18, 19, 21, 22, 23, 24, 25, 30, 33, 34, 35, 36, 42, 65]$

$E = \{(C,d), (a,c), (a,d), (E,F), (H,I), (G,I), (A,B), (E,6), (F,H), (6,H), (D,I), (D,E), (B,H), (B,C), (B,F), (C,F), (C,E)$

k	E^-	E^+	e_k
		$\{(C,d)\}$	(c,d)
		$\{(A,c)\}$	(A,c)
	$\{(A,D)\}$	—	—
		$\{(E,F)\}$	(E,F)
		$\{(H,I)\}$	(H,I)
		$\{(G,I)\}$	(G,I)
		$\{(A,B)\}$	(A,B)
		$\{(E,6)\}$	$(E,6)$
	$\{(F,H)\}$	—	—
	$\{(6,H)\}$	—	—
		$\{(D,I)\}$	(D,I)
	$\{(D,E)\}$	—	—
	$\{(B,H)\}$	—	—
	$\{(B,C)\}$	—	—
	$\{(B,F)\}$	—	—
	$\{(C,F)\}$	—	—
	$\{(C,E)\}$	—	—

Assim, a MST obtida é:



O peso total da MST é: $4+9+18+19+21+22+23+30 =$

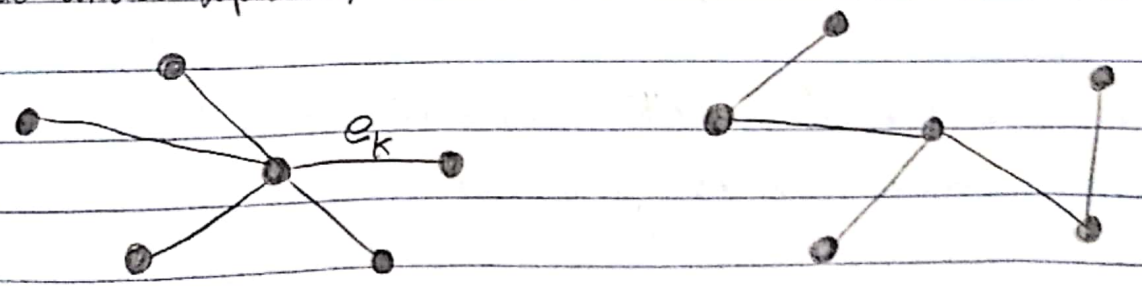
$= 146$

③ Realizando prova por contradição

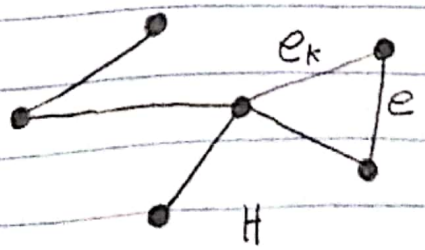
Considerando T como a árvore retornada por Kruskal

1 - Supondo $\exists S \neq T$ tal que $w(S) < w(T)$ (S também é uma árvore)

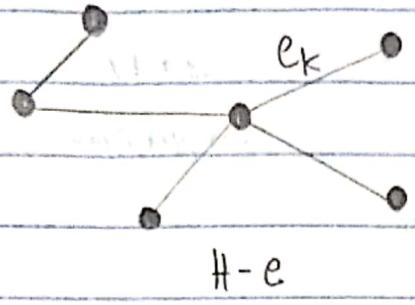
2 - Seja $e_K \in T$ a primeira aresta adicionada em T que não está em S (Uma vez que são árvores diferentes)



3 - Faça $H = (S + e_K)$. Observe que H não é mais uma árvore, pois contém ciclo

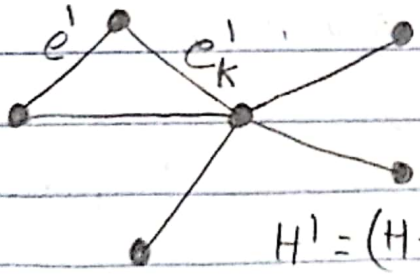


4- Note que no ciclo C, $\exists e \in S$ tal que $e \notin T$ (pois se não C existiria em T). O subgrafo H-e é conexo, possui n-1 arestas e define a árvore geradora de G



5- Porém, $w(e_k) \leq w(e)$ e assim $w(H-e) \leq w(S)$ (pois de acordo com Kruskal e_k vem antes de e na lista ordenada de arestas, já garantido pela ordenação)

6- Repetindo o processo usado para gerar H-e a partir de S é possível produzir uma sequência de árvores que se aproximam cada vez mais de T.



$S \to (H-e) \to (H'-e') \to (H''-e'') \to \ldots \to T$

de forma que $w(S) \geq w(H-e) \geq w(H'-e') \geq \ldots \geq w(T)$
(contradição a suposição inicial)

Assim, não existe árvore com peso menor do que T, mostrando que T tem peso mínimo (Não é possível S ter peso menor que T)

(14) Unificando as complexidades:

- Make_Set(V) é O(1). Como é executada n vezes, então custo total será O(n)
- Ordenação = Realizada com merge sort ou quicksort, que são O(m Log m), onde m é o número de arestas
- Para a função Find_set(V), iremos depender da altura da árvore. No caso em que V é um nó folha da árvore binária, h = Log n e portanto a complexidade da função é O(Log n). Caso a árvore não for binária, muda-se apenas a base do Logaritmo
- Union(u, v) faz uso de Find_Set e também é O(Log n)
- Como Find_set(v) é executada duas vezes para cada aresta (Uma para cada extremidade: (u, v)), então o custo total é 2m O(Log n), que é O(m Log n)
- Union(u, v) é executada somente quando uma aresta é adicionada na árvore. Como uma árvore tem m = n − 1 arestas, a complexidade é O(n Log n)

Assim, o custo total de Kruskal é:

$$C = O(n) + O(m \log n) + O(m \log n) + O(n \log n)$$

Observa-se que O(m Log n) é o termo dominante. Como $m < n^2$, temos que

$$\log m < \log n^2 < 2 \log n = O(\log n)$$

Portanto, a complexidade do algoritmo de Kruskal é O(m Log n)

⑤ Dado o algoritmo de Prim:

```
MST_Prim(G, w, r) {
    for each v ∈ V {
        λ(v) = ∞
        π(v) = Nil
    }
    λ(r) = 0
    Q = ∅
    for each v ∈ V
        Insert(Q, v)
    while Q ≠ ∅ {
        u = ExtractMin(Q)
        S = S ∪ {u}
        for each v in N(u) {
            if v ∈ Q and w(u,v) < λ(v) {
                λ(v) = w(u,v)
                π(v) = u
                DecreaseKey(Q, v, w(u,v))
            }
        }
    }
}
```

Explicação do código

Inicialmente, para cada vértice do grafo G, é armazenado o custo para chegar nesse vértice como sendo infinito (∞) e também armazena Nil como antecessor do vértice v.

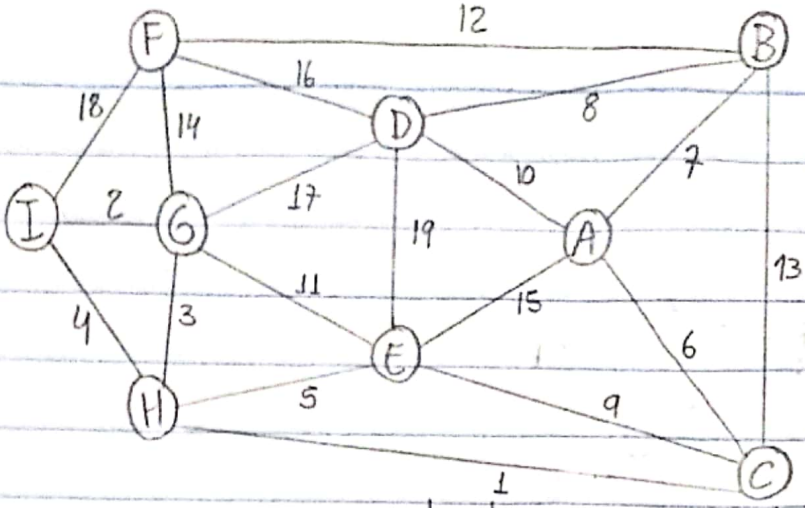
Para cada vértice do grafo G, iremos adicionar um vértice dentro da fila de prioridades, onde a maior prioridade é determinada pelo menor λ(v)

→ Remove da fila de prioridades o nó u que contém o menor λ(u)

Para cada vizinho do nó u, se esse vizinho ainda estiver na fila de prioridades e o custo para ir de u até esse vizinho for menor do que o custo desse vizinho V, então atualiza λ(v) para ser o novo custo w(u,v), coloca o antecessor de V como sendo u e atualiza o custo de V na fila de prioridades.

• Como pode-se observar, o algoritmo de Prim é guloso pois a cada execução do while, pega-se o vértice que contém a maior prioridade (menor custo de entrada), para que depois seja possível calcular um novo custo / predecessor para os vizinhos do nó escolhido.

6



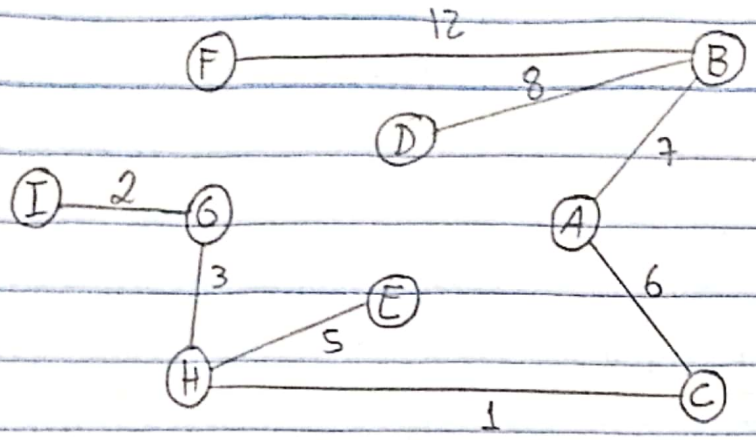
	A	B	C	D	E	F	G	H	I
λ_0	∞	∞	∞	∞	∞	∞	∞	∞	[0]
λ_1	∞	∞	∞	∞	∞	18	[2]	4	
λ_2	∞	∞	∞	17	11	14	[3]		
λ_3	∞	∞	[1]	17	5	14			
λ_4	6	13		17	[5]	14			
λ_5	[6]	13		17		14			
λ_6		[7]		10		14			
λ_7			[8]			12			
λ_8						[12]			
λ_9									

U	V' = {V∈N(U) ∧ V∈Ū}	λ(V)	π(V)
I	{F, G, H}	$\lambda(F) = \min\{\infty, 18\} = 18$	$\pi(F) = I$
		$\lambda(G) = \min\{\infty, 2\} = 2$	$\pi(G) = I$
		$\lambda(H) = \min\{\infty, 4\} = 4$	$\pi(H) = I$
G	{D, E, F, H}	$\lambda(D) = \min\{\infty, 17\} = 17$	$\pi(D) = G$
		$\lambda(E) = \min\{\infty, 11\} = 11$	$\pi(E) = G$
		$\lambda(F) = \min\{18, 14\} = 14$	$\pi(F) = G$
		$\lambda(H) = \min\{4, 3\} = 3$	$\pi(H) = G$
H	{C, E}	$\lambda(C) = \min\{\infty, 1\} = 1$	$\pi(C) = H$
		$\lambda(E) = \min\{11, 5\} = 5$	$\pi(E) = H$
C	{A, B, E}	$\lambda(A) = \min\{\infty, 6\} = 6$	$\pi(A) = C$
		$\lambda(B) = \min\{\infty, 13\} = 13$	$\pi(B) = C$
		$\lambda(E) = \min\{5, 9\} = 5$	—
E	{A, D}	$\lambda(A) = \min\{6, 15\} = 6$	—
		$\lambda(D) = \min\{17, 19\} = 17$	—
A	{B, D}	$\lambda(B) = \min\{13, 7\} = 7$	$\pi(B) = A$
		$\lambda(D) = \min\{17, 10\} = 10$	$\pi(D) = A$
B	{D, F}	$\lambda(D) = \min\{10, 8\} = 8$	$\pi(D) = B$
		$\lambda(F) = \min\{14, 12\} = 12$	$\pi(F) = B$
D	{F}	$\lambda(F) = \min\{12, 16\} = 12$	—
F	—	—	—

Assim, temos:

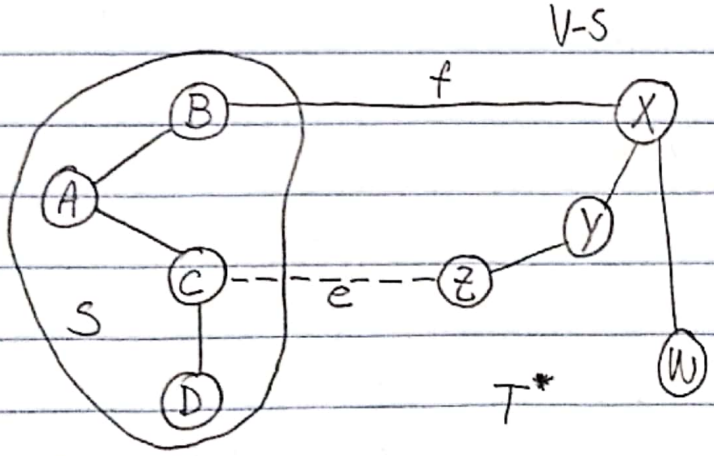
	A	B	C	D	E	F	G	H	I
Antecessor:	C	A	H	B	H	B	I	G	–

MST:



Assim, o peso total da MST é: 1+2+3+5+6+7+8+12 = 44

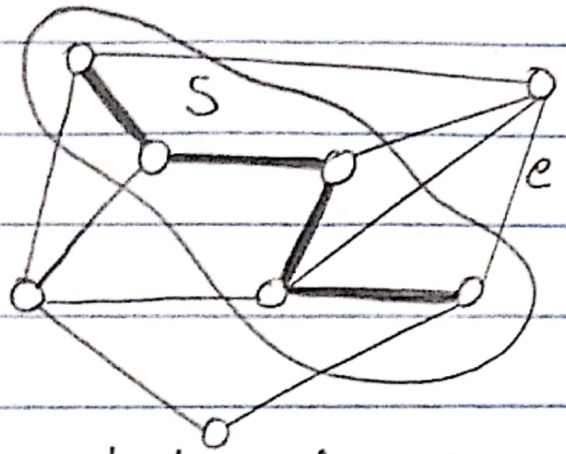
(7) Considerando $G = (V, E, w)$ um grafo, S sendo um subconjunto qualquer de V e $e \in E$ a aresta de menor custo com exatamente uma extremidade em S. Então, a MST de G contém e.



Realizando prova por contradição: Sendo T^* uma MST de G, suponha que $e \in T^*$. Ao adicionar e em T^*, vê-se um único ciclo C. Para que $T^* - e$ fosse uma MST, tem que haver alguma outra aresta f com apenas uma extremidade em S (Pois se não seria descartada). Então $T = T^* + e - f$ também é uma árvore geradora. Como, $w(e) < w(f)$, então T tem peso mínimo. Portanto, T^* não se pode ser MST de G.

8) Provar a otimalidade (corretude) do algoritmo de prim

Sendo S o subconjunto de vértices de G na árvore T (definido pelo algoritmo), o algoritmo de prim, durante sua iteração, vai adicionar em T a cada passo, a aresta de menor custo com apenas um vértice extremidade em S. Portanto, pela propriedade do corte, como está sendo considerada toda vez a aresta de menor custo, toda aresta adicionada pertence a MST de G.



Exemplo de grafo e representação da aresta de menor custo