

Heaps e Algoritmo de Kruskal

Ciro Cirne Trindade
(ciroct@unisantos.br)

José Luís Barboza Lobianco
(lobiancobr@yahoo.com.br)

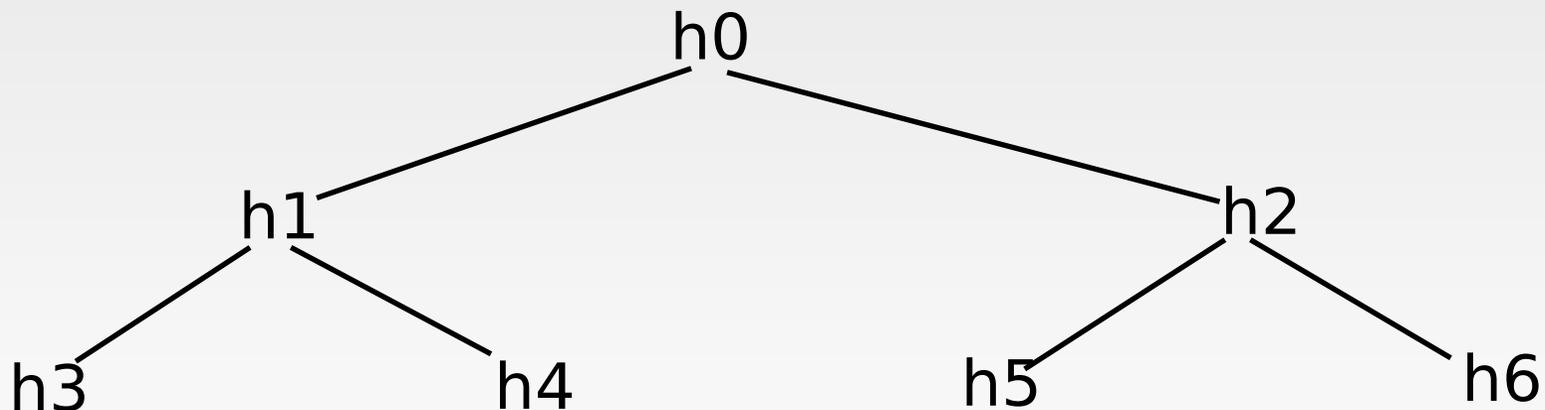
Evandro Luquini
(eluquini@yahoo.com.br)

- Um *heap* é a estrutura de dados mais eficiente para obter o valor máximo ou mínimo de um conjunto de objetos ($O(\lg(n))$)
- Um *heap* máximo é aquele a partir de onde se obtém sempre o maior valor
- Um *heap* mínimo é aquele a partir de onde se obtém sempre o menor valor

Heap Mínimo



- O **heap mínimo** é definido como sendo uma sequência de chaves h_E, h_{E+1}, \dots, h_D tais que
 - $h_i \leq h_{2i+1}$ e $h_i \leq h_{2i+2}$ para $i = E \dots D/2$
- Árvore binária representada por um vetor:



$$h_0 = \min(h_0, h_1, \dots, h_{n-1})$$

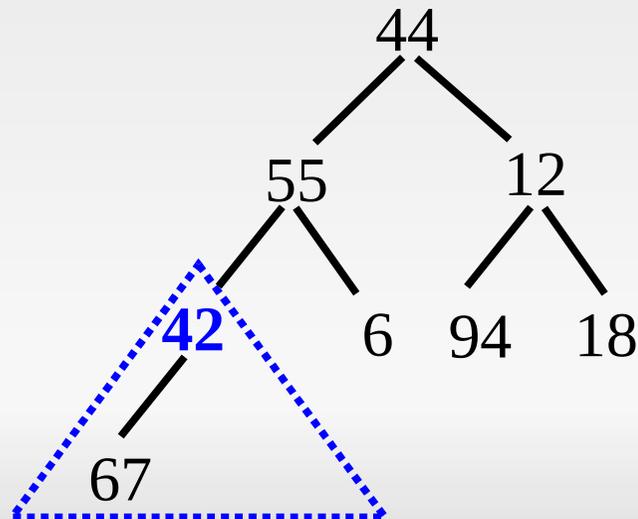
- Construção do *heap* mínimo
 - Dado um vetor $h_0 \dots h_{n-1}$, os elementos $h_m \dots h_{n-1}$ (com $m = n / 2$) já formam um “*heap*” uma vez que nenhum par de índices i, j é tal que $j = 2i+1$ (ou $j = 2i + 2$)
 - Estes elementos formam a linha inferior da árvore binária, entre os quais nenhuma relação de ordenação é exigida
 - O *heap* é, agora, estendido para a esquerda, sendo que um novo elemento é incluído a cada passo e posicionado apropriadamente por meio de uma operação de **escorregamento**

Heap Mínimo



- Construindo o *heap* mínimo
 - Inserindo o 42 no *heap*

44	55	12	42	6	94	18	67
0	1	2	3	4	5	6	7

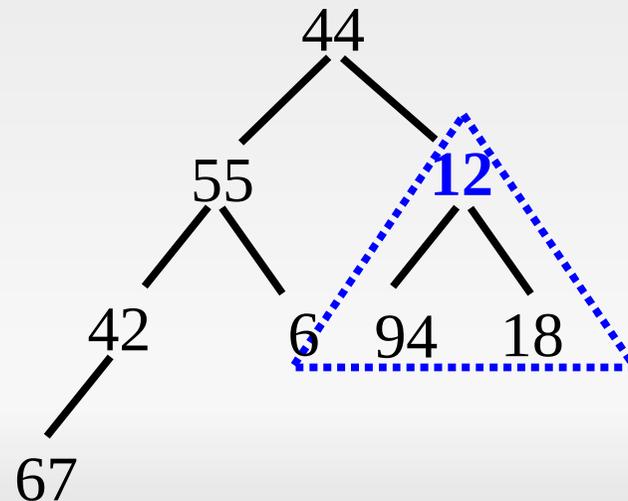


Heap Mínimo



- Construindo o *heap* mínimo
 - Inserindo o 12 no *heap*

44	55	12	42	6	94	18	67
0	1	2	3	4	5	6	7

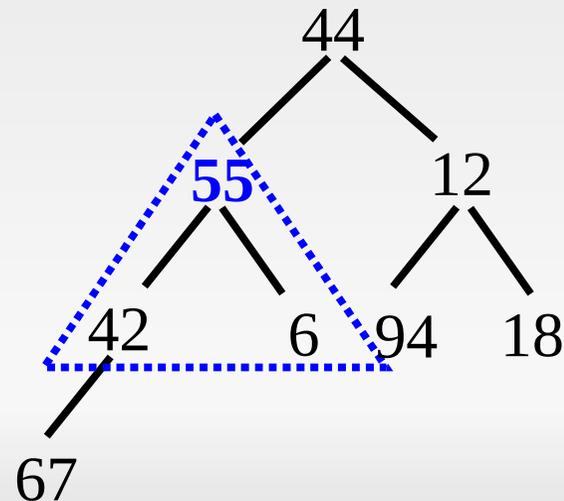


Heap Mínimo



- Construindo o *heap* mínimo
 - Inserindo o 55 no *heap*

44	55	12	42	6	94	18	67
0	1	2	3	4	5	6	7

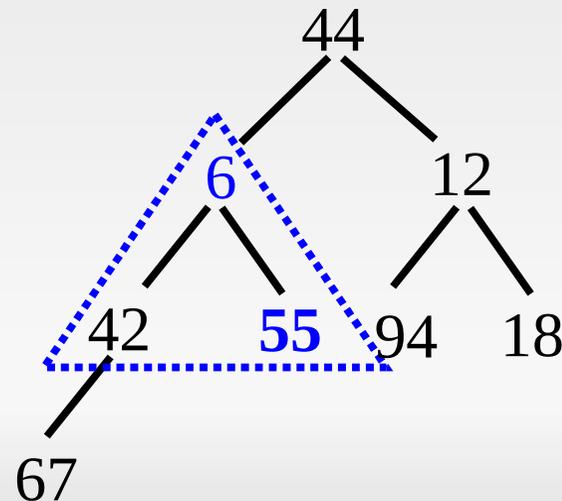


Heap Mínimo



- Construindo o *heap* mínimo
 - Inserindo o 55 no *heap*

44	6	12	42	55	94	18	67
0	1	2	3	4	5	6	7

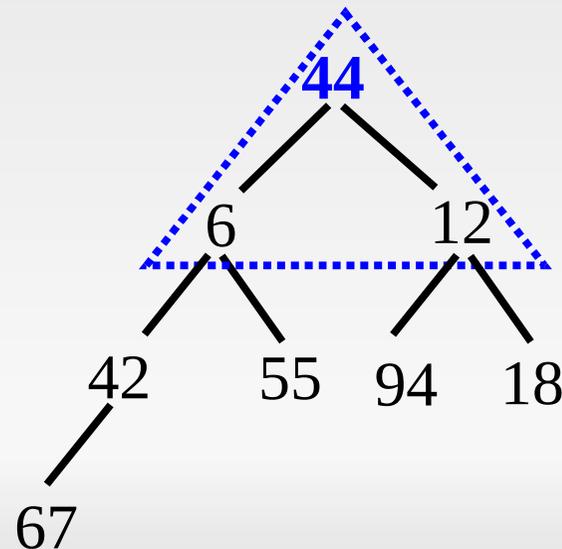


Heap Mínimo



- Construindo o *heap* mínimo
 - Inserindo o 44 no *heap*

44	6	12	42	55	94	18	67
0	1	2	3	4	5	6	7

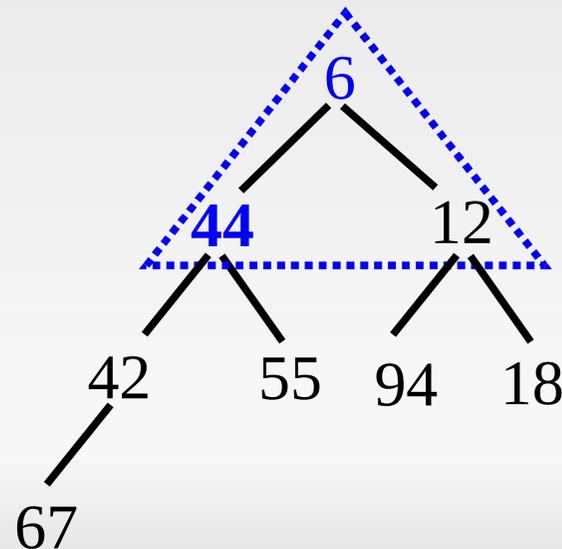


Heap Mínimo



- Construindo o *heap* mínimo
 - Inserindo o 44 no *heap*

6	44	12	42	55	94	18	67
0	1	2	3	4	5	6	7

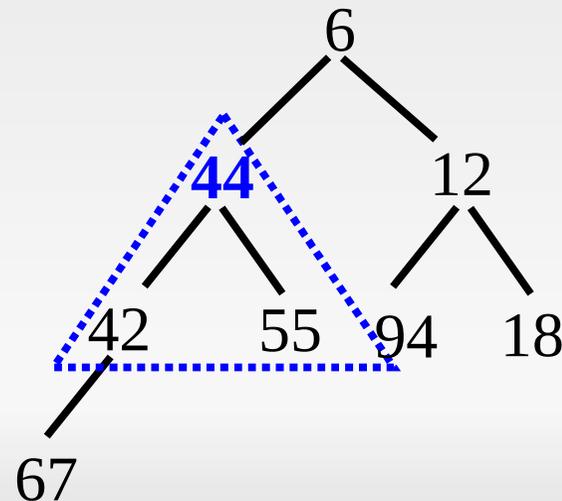


Heap Mínimo



- Construindo o *heap* mínimo
 - Inserindo o 44 no *heap*

6	44	12	42	55	94	18	67
0	1	2	3	4	5	6	7

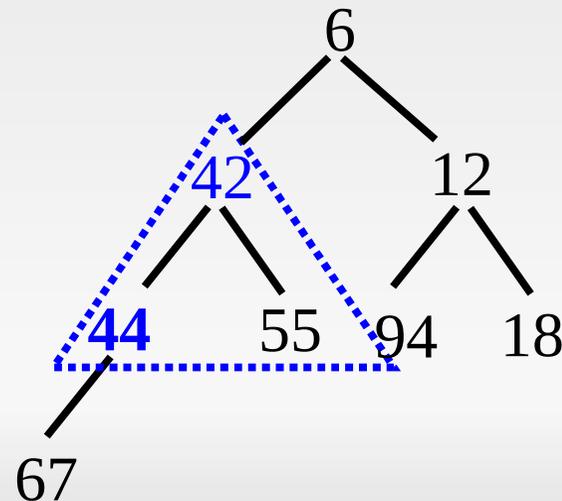


Heap Mínimo



- Construindo o *heap* mínimo
 - Inserindo o 44 no *heap*

6	42	12	44	55	94	18	67
0	1	2	3	4	5	6	7



Rotina para construção de um *heap* mínimo



```
void make_heap_min(int a[], int n) {
    int e = n/2, i, j, x, d = n-1; // índice do último
    while(e-- > 0) {
        i = e;
        j = 2*e+1; // índices do heap
        x = a[e]; // novo elemento
        if(j < d && a[j+1] < a[j]) j++;
        while(j <= d && a[j] < x) {
            a[i] = a[j];
            i = j;
            j = 2 * j + 1;
            if(j < d && a[j+1] < a[j]) j++;
        }
        a[i] = x;
    }
}
```

Rotina de remoção em um *heap* mínimo



```
int del_heap_min(int a[], int * n) {
    int i = 0, j = 1; // índices do heap
    int menor = a[0], x, d;
    a[0] = a[--(*n)]; // passa o último para a raiz
    x = a[0]; // novo elemento
    d = *n-1; // índice do último
    if(j < d && a[j+1] < a[j]) j++;
    while(j <= d && a[j] < x) {
        a[i] = a[j];
        i = j;
        j = 2 * j + 1;
        if(j < d && a[j+1] < a[j]) j++;
    }
    a[i] = x;
    return menor;
}
```

Rotina de inserção em um *heap* mínimo

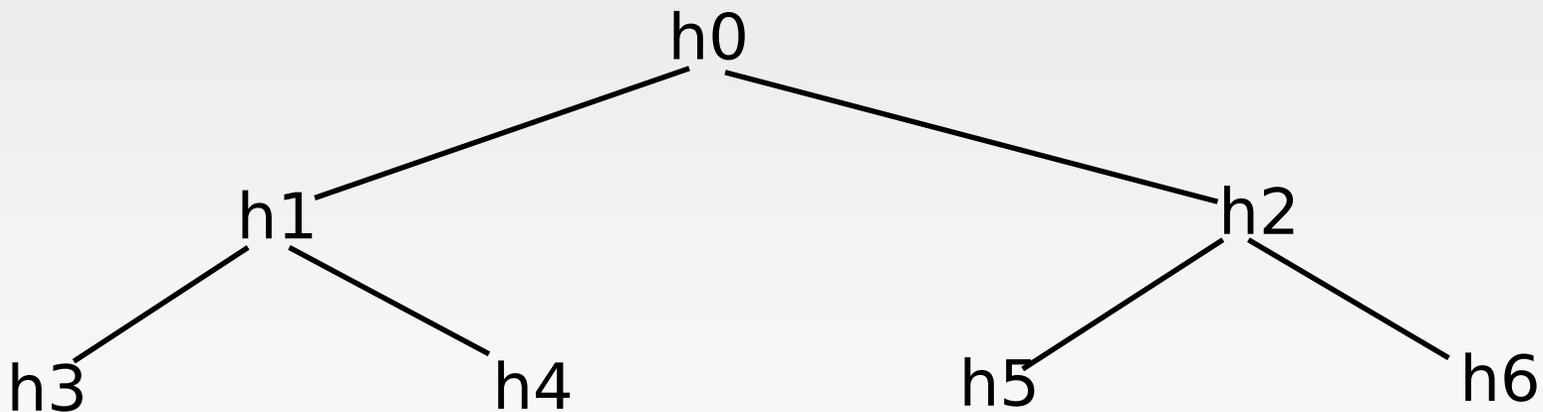


```
void ins_heap_min(int a[], int * n, int x) {
    int p, f;
    a[(*n)++] = x; // insere no final do heap
    p = *n-1; // índice do último
    f = p / 2 - !(p%2); // pai de p
    while (f >= 0 && a[f] > x) {
        a[p] = a[f];
        p = f;
        f = p/2 - !(p%2);
    }
    a[p] = x;
}
```

Heap Máximo



- O **heap máximo** é definido como sendo uma sequência de chaves h_E, h_{E+1}, \dots, h_D tais que
 - $h_i \geq h_{2i+1}$ e $h_i \geq h_{2i+2}$ para $i = E \dots D/2$
- Árvore binária representada por um vetor:



$$h_0 = \max(h_0, h_1, \dots, h_{n-1})$$

- Implemente as rotinas `make_heap_max`, `ins_heap_max` e `del_heap_max`.
- Problema Bolsa de Valores (Seletiva da Maratona de Programação da UFRJ – 2008). SPOJ: id 2841, código BOLSA. Dica: trate os valores como inteiros.

Algoritmo de Kruskal



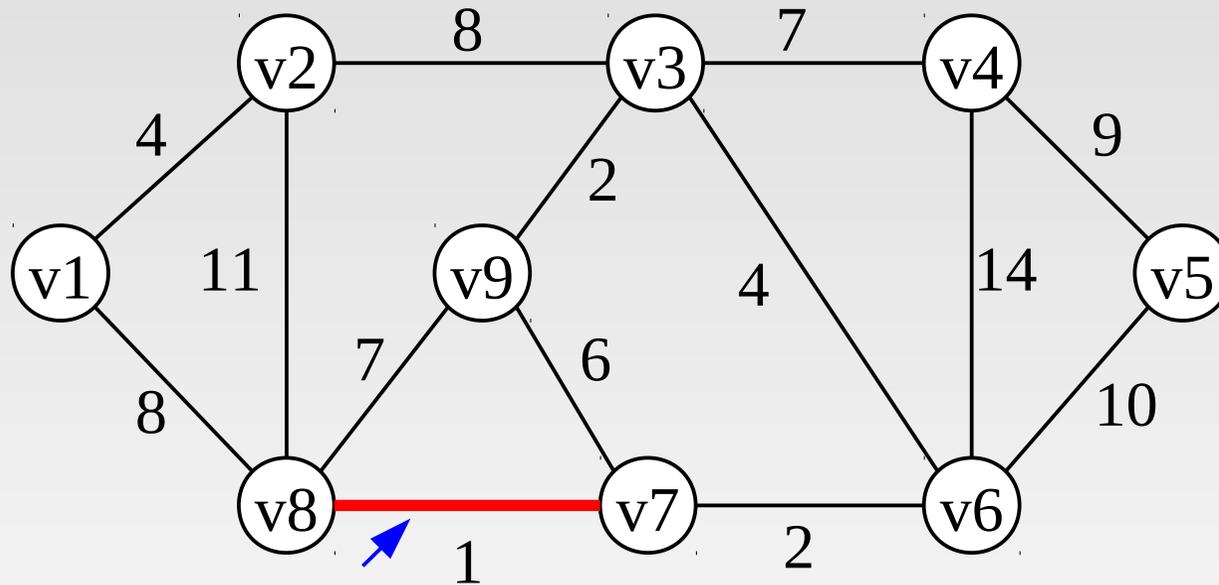
- Determina uma árvore geradora mínima para um grafo ponderado e **não orientado**
- Funciona para grafos desconexos
- Encontra uma aresta segura para adicionar à floresta crescente encontrando, entre todas as arestas que conectam duas árvores quaisquer na floresta, uma aresta (u,v) de peso mínimo
- Exige uma lista ordenada pelo peso de arestas

Algoritmo de Kruskal

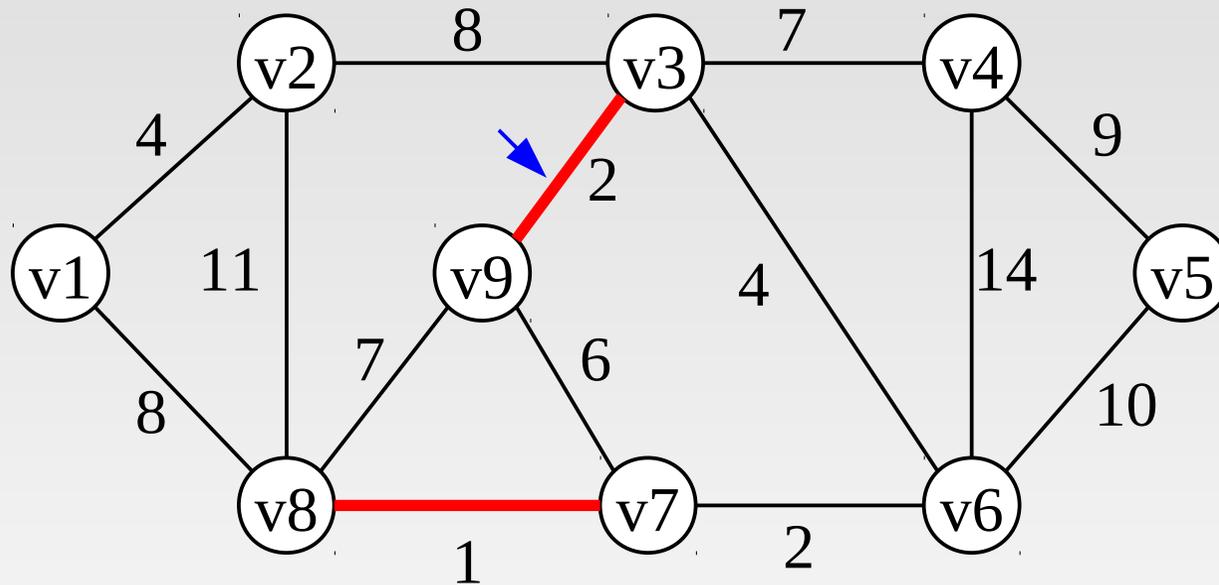


- Inicialmente o grafo é visto como uma floresta de $|V|$ árvores com apenas um nó (vértice)
- A cada passo do algoritmo escolhe-se a aresta de menor peso que una dois vértices de árvores diferentes
- Este processo se repete até que $|V|-1$ arestas tenham sido acrescentadas à árvore geradora mínima

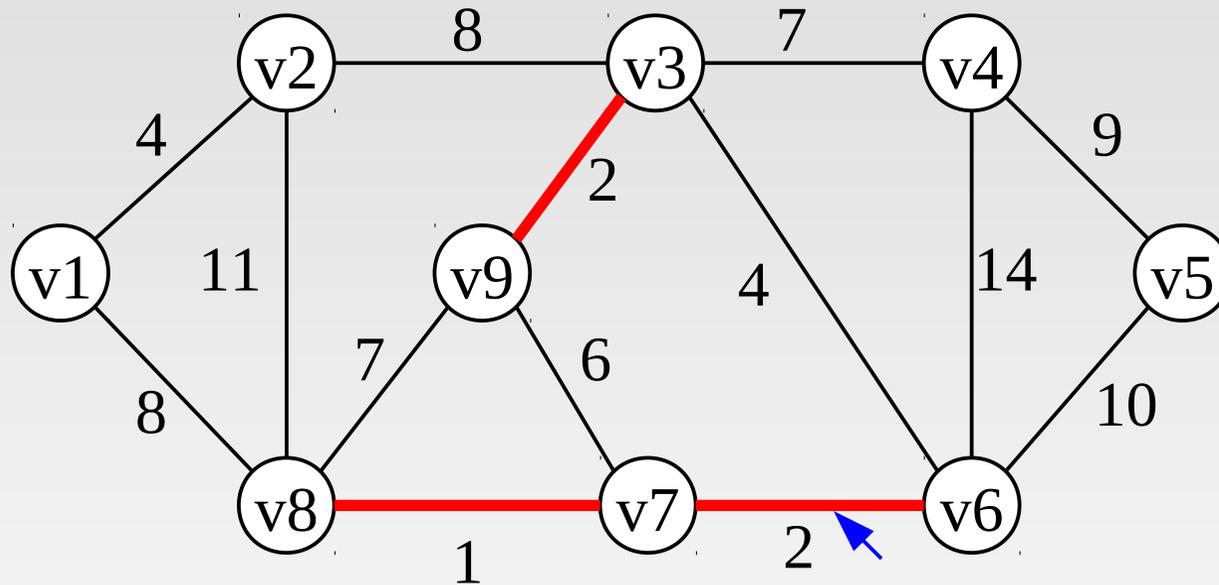
Exemplo da aplicação do algoritmo de Kruskal



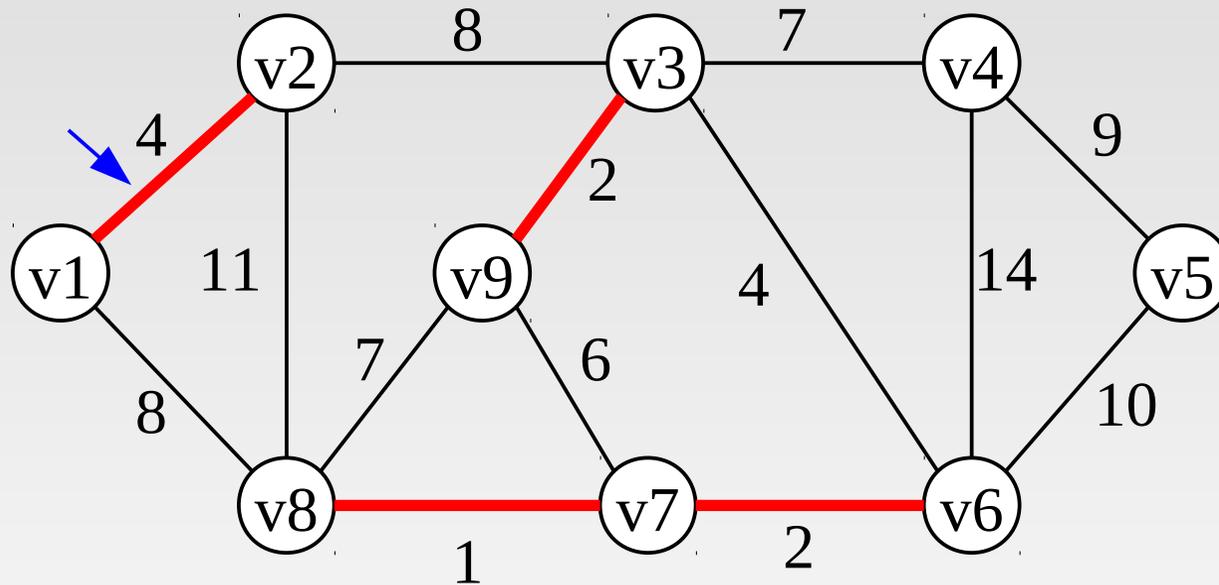
Exemplo da aplicação do algoritmo de Kruskal



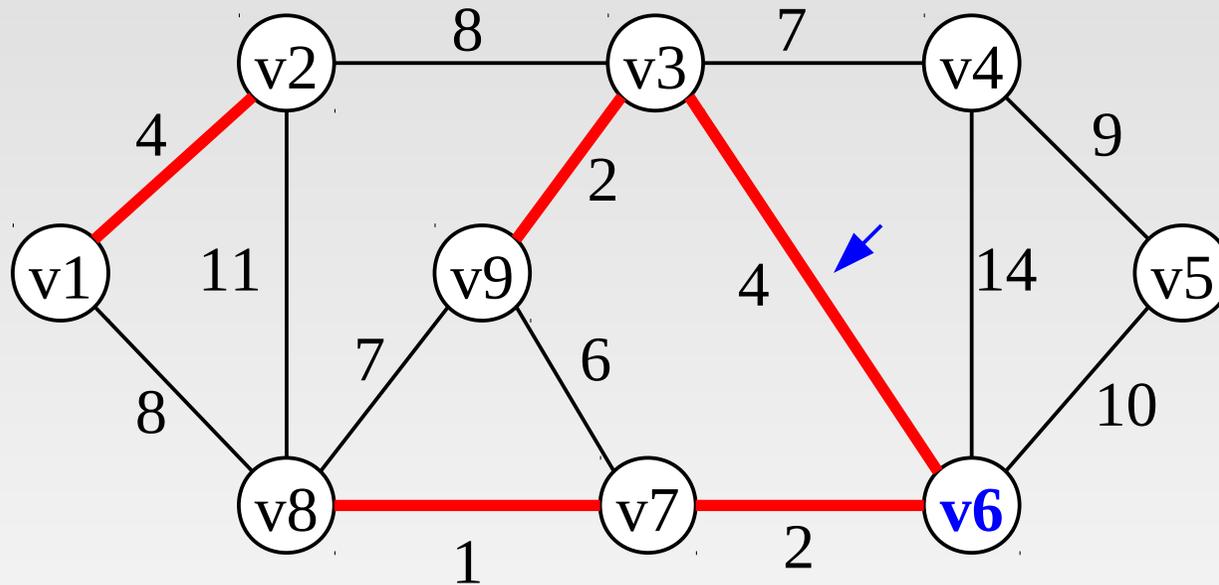
Exemplo da aplicação do algoritmo de Kruskal



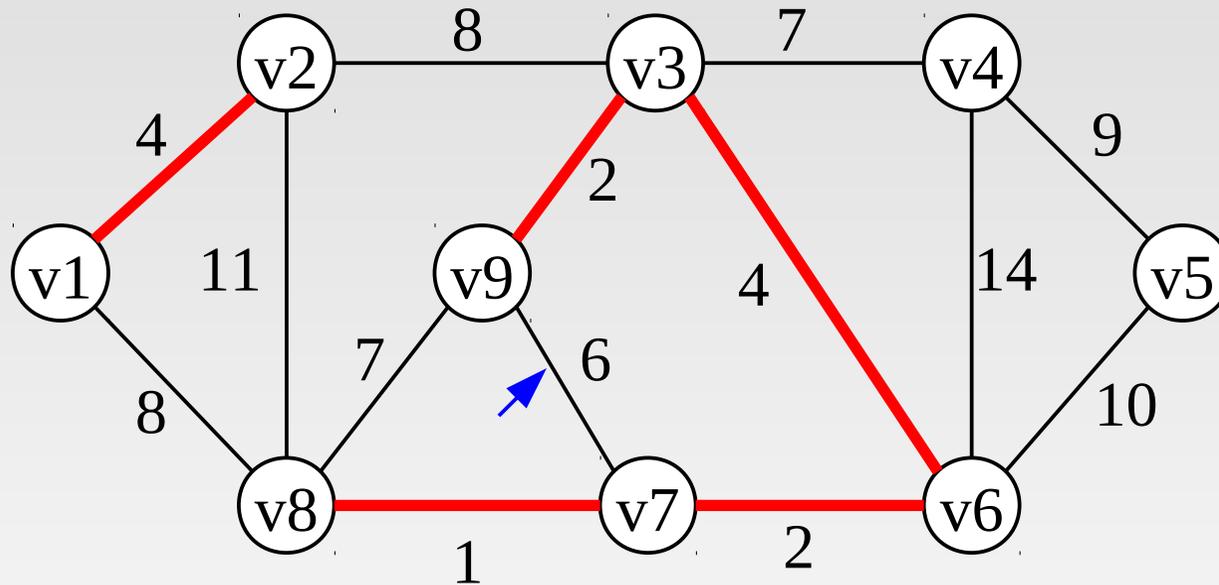
Exemplo da aplicação do algoritmo de Kruskal



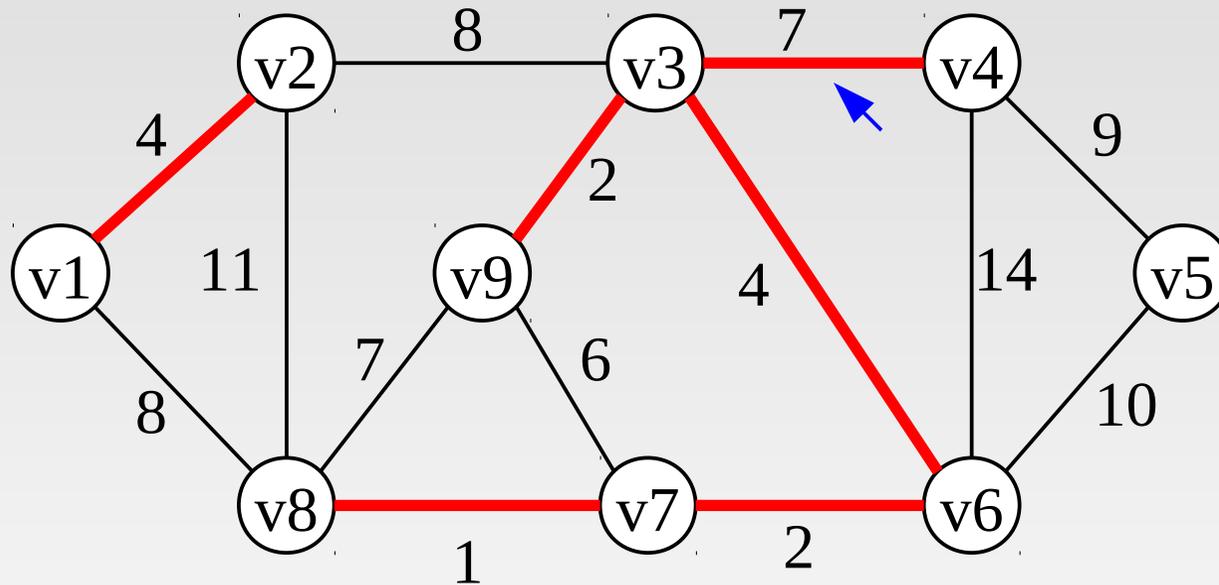
Exemplo da aplicação do algoritmo de Kruskal



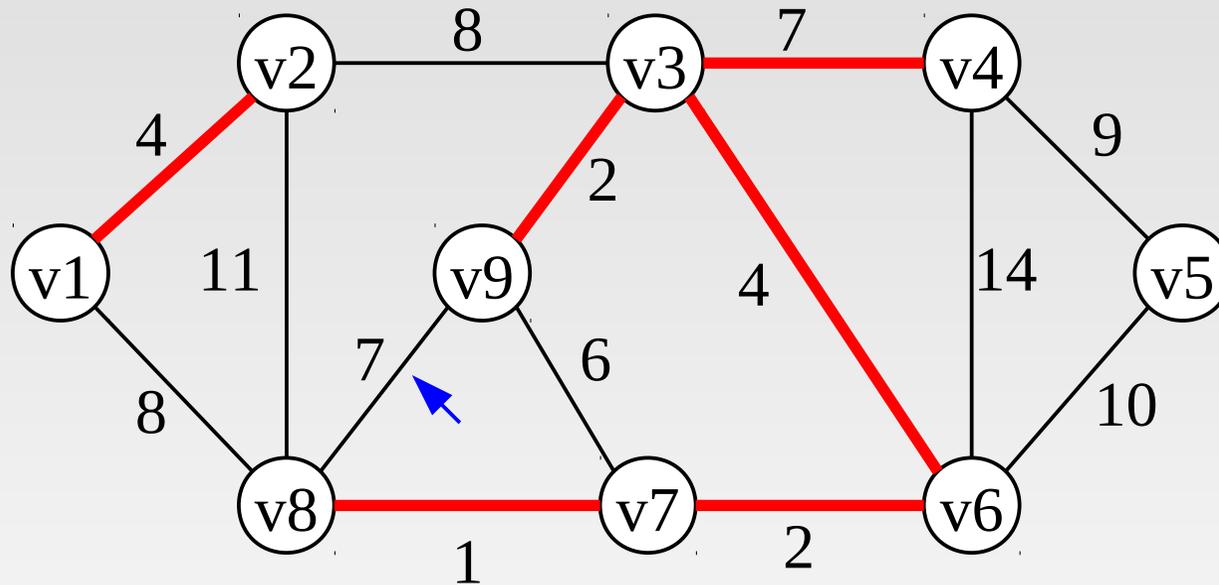
Exemplo da aplicação do algoritmo de Kruskal



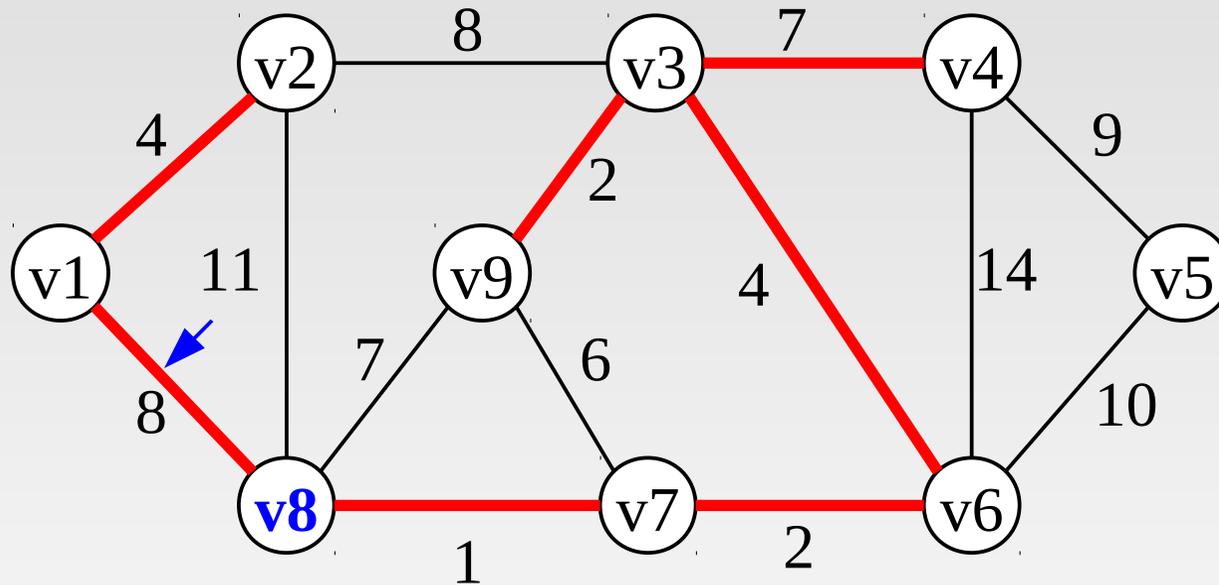
Exemplo da aplicação do algoritmo de Kruskal



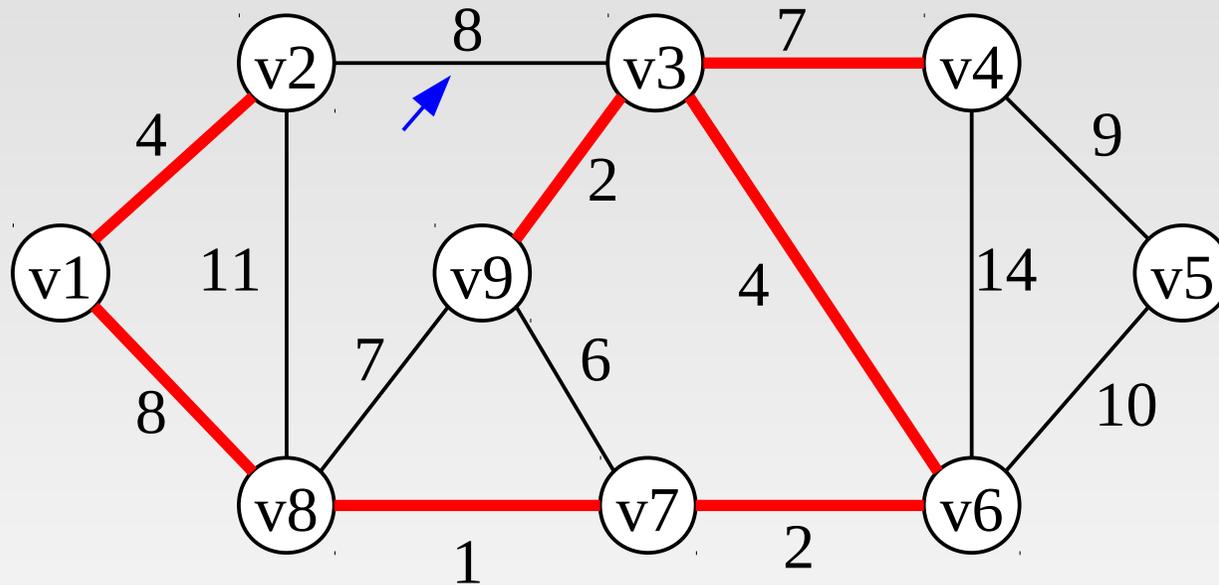
Exemplo da aplicação do algoritmo de Kruskal



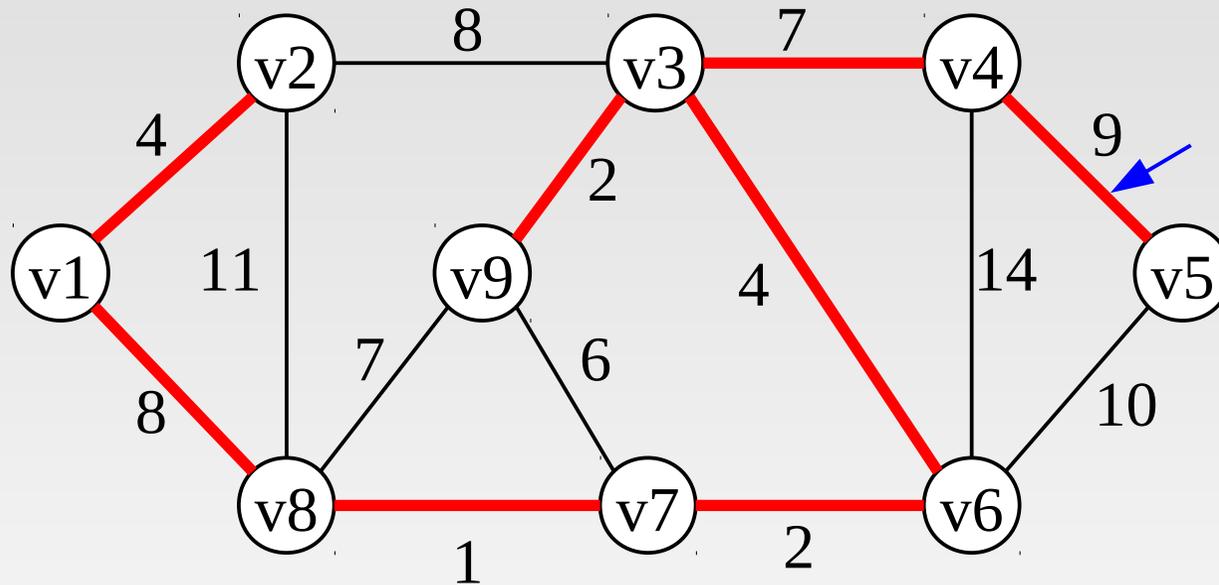
Exemplo da aplicação do algoritmo de Kruskal



Exemplo da aplicação do algoritmo de Kruskal



Exemplo da aplicação do algoritmo de Kruskal



Implementação do Algoritmo de Kruskal



- Vamos considerar a seguinte estrutura de dados para representar um grafo

```
#define MAXV 500 // número máximo de vértices
#define MAXA MAXV*(MAXV-1)/2 //número máximo de arestas

typedef struct {
    int v; // vértice
    int w; // vértice
    int peso; // peso da aresta
} aresta;

typedef struct {
    aresta arestas[MAXA]; // lista de arestas
    int nvertices; // número de vértices
    int narestas; // número de arestas
} grafo;
```

Implementação do Algoritmo de Kruskal



- Entrada de dados

```
grafo g;  
aresta mst[MAXA];  
int i,num_arestas,cst;  
scanf("%d %d",&g.nvertices,&g.narestas);  
while(g.nvertices != 0) {  
    for(i = 0; i < g.narestas; i++) {  
        scanf("%d %d %d", &g.arestas[i].v,  
            &g.arestas[i].w,&g.arestas[i].peso);  
        g.arestas[i].v--;  
        g.arestas[i].w--;  
    }  
    num_arestas = kruskal(&g,mst);  
    for(cst = i = 0; i < num_arestas; i++)  
        cst += mst[i].peso;  
    printf("%d\n",cst);  
    scanf("%d %d",&g.nvertices,&g.narestas);  
}
```

Implementação do Algoritmo de Kruskal



■ Funções auxiliares

```
/* função usada para a ordenação das aresta do grafo */
int compara(const void * a, const void * b) {
    aresta * x = (aresta *) a;
    aresta * y = (aresta *) b;
    if(x->peso < y->peso) return -1;
    if(x->peso > y->peso) return 1;
    return 0;
}

/* inicialmente cada vértice é uma árvore da floresta */
void init_floresta(int pai[], int tam[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        pai[i] = i; // pai do nó i
        tam[i] = 1; // tamanho da árvore com raiz em i
    }
}
```

Implementação do Algoritmo de Kruskal



■ Funções auxiliares (cont.)

```
/* A função mesma_arvore devolve true se os vértices v e
 * w estiverem na mesma árvore (e portanto na mesma
 * componente da floresta mst[0..k-1]). */
bool mesma_arvore(int v, int w, int pai[]) {
    return (raiz(v, pai) == raiz(w, pai));
}
```

```
/* A função raiz devolve a raiz da árvore que contém o
 * vértice x */
int raiz(int x, int pai[]) {
    int i = x;
    while (i != pai[i])
        i = pai[i];
    return i;
}
```

Implementação do Algoritmo de Kruskal



- Funções auxiliares (cont.)

```
/* A função une_arvores faz a união das árvores que
 * contêm os vértices v e w. */
void une_arvores(int v, int w, int pai[], int tam[]) {
    int i = raiz(v, pai), j = raiz(w, pai);
    if (i == j) return;
    if (tam[i] < tam[j]) {
        pai[i] = j;
        tam[j] += tam[i];
    }
    else {
        pai[j] = i;
        tam[i] += tam[j];
    }
}
```

Implementação do Algoritmo de Kruskal



- Algoritmo de Kruskal

```
/* função que recebe um grafo g, calcula uma MST em cada
 * componente de g, armazena as arestas das MSTs no vetor
 * mst[0..k-1] e devolve k. */
int kruskal(grafo * g, aresta mst[]) {
    static int pai[MAXV], tam[MAXV];
    int i, k, a = g->narestas;
    qsort(g->arestas, a, sizeof(aresta), compara);
    init_floresta(pai, tam, g->nvertices);
    for (i = k = 0; i < a && k < g->nvertices-1; i++)
        if (!mesma_arvore(g->arestas[i].v, g->arestas[i].w,
pai)) {
            une_arvores(g->arestas[i].v, g->arestas[i].w,
pai, tam);
            mst[k++] = g->arestas[i];
        }
    return k;
}
```

- Substitua a lista ordenada de aresta do algoritmo de Kruskal por um *heap* mínimo de arestas
- Problema Sistema Cipoviário (Seletiva da Maratona de Programação do IME-USP – 2007). SPOJ: id 1746, código CIPO.

Referências



- CORMEN, T.H.; LEISERSON, C.E.; RIVEST, R.L.; STEIN, C.. *Algoritmos: teoria e prática*. Tradução da 2ª edição americana, Editora Campus, 2002.
- FEOFILOFF, Paulo. *Algoritmo de Kruskal*. Disponível em http://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/kruskal.html.