

No pseudocódigo seguinte, para simplificar, vamos supor que

- todos os vértices do grafo são alcançáveis a partir da origem s .
- Se esse não for o caso, podemos focar nos vértices alcançáveis
 - realizando uma busca inicial a partir de s ,
- ou modificar levemente o algoritmo de Dijkstra.
 - Quiz1: Como?

Dijkstra ($G = (V, E), c, s$)

Para todo $v \in V$ faça $\text{dist}[v] = +\infty$ e $\text{pred}[v] = \text{NULL}$

$\text{dist}[s] = 0$

$R = \{\}$

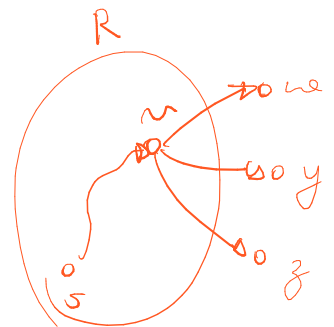
enquanto $R \neq V$

escolha $v \in V \setminus R$ que $\min. \text{dist}[v]$

$\Rightarrow$ adicione v a R

para cada arco (v, w) e/ $w \in V \setminus R$

$\Rightarrow$ se $\text{dist}[w] > \text{dist}[v] + c(v, w)$
 $\Rightarrow \text{dist}[w] = \text{dist}[v] + c(v, w)$
 $\text{pred}[w] = v$



Prova de corretude:

O algoritmo de Dijkstra mantém as seguintes propriedades invariantes

- no início de cada iteração do laço principal do algoritmo:

1- $\forall v \in R$ temos $\text{dist}[v]$ é o valor da distância de s a v

2- $\forall v \in V$ temos $\text{pred}[v]$ é o penúltimo vértice no caminho de s a v

3- $\forall v \in V \setminus R$ temos $\text{dist}[v]$ é o valor de um caminho mínimo de s a v que só usa vértices em R (exceto pelo destino v)

Faremos a prova por indução no número de iterações k .

H.I.: As propriedades 1, 2 e 3 do invariante valem no início da iteração k .

Caso base: no início da primeira iteração

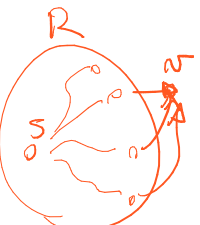
- 1 vale pois $R = \emptyset$

- 2 vale pois $\forall v \neq s$ $\nexists$ caminho e $\text{pred}[v] = \text{NULL}$
p/ s temos $\text{dist}[s] = 0$ um caminho
sem penúltimo vértice e $\text{pred}[s] = \text{NULL}$

- 3 vale pois $R = \emptyset$, assim $\nexists$ caminhos mínimos de

s a v e/ vértices em R , p/ $v \neq s$

p/ s temos $\text{dist}[s] = 0$ mas o caminho não usa vértices em R

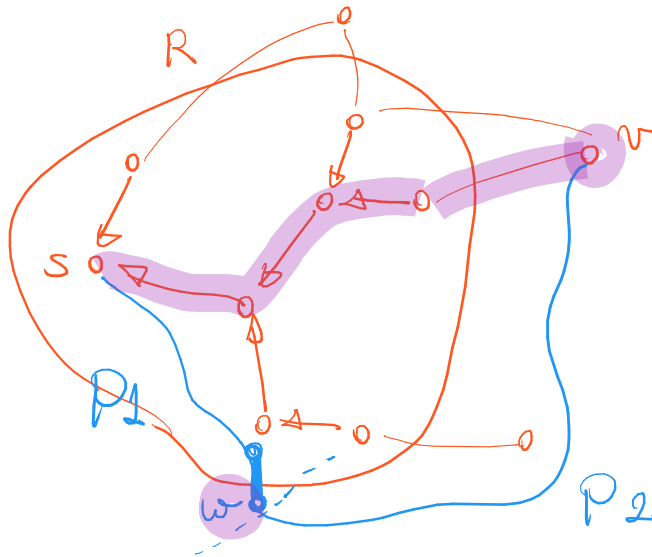


Passo: Seja k a iteração em que v é inserido em R . Vamos mostrar que

- as propriedades 1, 2 e 3 continuam valendo no início da iteração $k + 1$.

O vértice v é inserido por ser o vértice fora de R com menor valor para $\text{dist}[\]$.

- Pela propriedade 3 da H.I. temos que v é o vértice com menor caminho
 - cujos vértices intermediários estão em R .
- Vamos mostrar que este é um caminho mínimo de s até v .



Considere um caminho P qualquer de s até v e seja $c(P)$ o custo de P .

- Em algum momento P tem que cruzar a fronteira entre
 - vértices que estão em R e vértices fora de R .
- Suponha que o primeiro vértice de P fora de R é w , e divida P em
 - P_1 (parte que vai de s a w) e P_2 (parte que vai de w a v).
- Pela propriedade 3 da H.I. temos que, $\text{dist}[w]$ é o menor caminho de s até w
 - que só usa vértices em R . Portanto, $\text{dist}[w] \leq c(P_1)$
- Como não temos arcos de custo negativo, $c(P_2) \geq 0$
- Assim, pela escolha de v como o vértice que minimiza $\text{dist}[\]$,
 - temos $\text{dist}[v] \leq \text{dist}[w] + 0 \leq c(P_1) + c(P_2) = c(P)$

Mostramos que qualquer caminho de s até v tem custo maior ou igual a $\text{dist}[v]$.

- Portanto, a propriedade 1 do invariante vale no início da iteração $k + 1$.

Agora vamos mostrar que as propriedades 2 e 3 continuam valendo.

- Para os vértices que não são destino de uma aresta com origem em v
 - nada muda e o resultado segue da H.I.
- Vamos considerar um vértice w em $V \setminus R$ tal que existe aresta (v, w) .
 - Pela H.I., $\text{dist}[w]$ e $\text{pred}[w]$ tinham os valores corretos
 - considerando os vértices em $R \setminus \{v\}$
- Assim, esses valores só devem ser alterados se existir
 - um caminho de custo menor para w , que passa por v
- Mas, se for esse o caso, durante a iteração k
 - o algoritmo faz $\text{dist}[w] \leftarrow \text{dist}[v] + c(v, w)$ e $\text{pred}[w] \leftarrow v$
- Portanto, $\text{dist}[w]$ corresponde ao custo do menor caminho
 - cujos vértices intermediários estão em R (propriedade 3)
- e $\text{pred}[w]$ aponta para o penúltimo vértice nesse caminho (propriedade 2)

Código de uma implementação básica do algoritmo de Dijkstra:

```
void Dijkstra(Grafo G, int origem, int *dist, int *pred) {
    int i, *R;
    int v, w, custo, tam_R, min_dist;
    Noh *p;
    // inicializando distâncias e predecessores
    for (i = 0; i < G->n; i++) {
        dist[i] = __INT_MAX__;
        pred[i] = -1;
    }
    - dist[origem] = 0;
    // inicializando conjunto de vértices "resolvidos" R
    - R = malloc(G->n * sizeof(int));
    for (i = 0; i < G->n; i++)
        R[i] = 0;
    - tam_R = 0;
    // enquanto não encontrar o caminho mínimo para todo vértice
    while (tam_R < G->n) {
        // buscando vértice v em V \ R que minimiza dist[v]
        min_dist = __INT_MAX__;
        v = -1;
        for (i = 0; i <= G->n; i++)
            if (R[i] == 0 && dist[i] < min_dist) {
                - v = i;
                - min_dist = dist[i];
            }
        // adicionando v a R e atualizando o conjunto R
        - R[v] = 1; tam_R++;
        // percorrendo lista com vizinhos de v
        for (p = G->A[v]; p != NULL; p = p->prox) {
            w = p->rotulo;
            custo = p->custo;
            // e atualizando as distâncias e predecessores quando for o caso
            if (R[w] == 0 && dist[w] > dist[v] + custo) {
                - dist[w] = dist[v] + custo;
                - pred[w] = v;
            }
        }
    }
    - free(R);
}
```

$O(n)$

$O(n)$

$O(n^2)$

$O(n)$

$O(\sum |S(v)|)$

$\sum |S(v)|$

$v \in V$

$\parallel$

$O(m)$

escolha gulosa

Eficiência: $O(n * n + m)$,

- pois o último laço realiza n iterações e em cada uma delas
 - o primeiro laço interno passa por todos os vértices
 - para escolher um vértice v ,
 - totalizando $n * n = n^2$ iterações,
 - enquanto o segundo laço interno
 - passa por todos os arcos do vértice v escolhido.
 - Com isso, ao longo do algoritmo, todo arco é visitado uma vez,
 - i.e., o segundo laço interno itera m vezes no total,
 - já que cada vértice é considerado
 - em apenas uma iteração do laço externo.
- Note que, como $m \leq n^2$ em grafos sem auto-laços e arestas múltiplas,
 - o algoritmo tem eficiência $O(n^2)$, 
 - independente do grafo ser denso ou esparso.

Implementação avançada de Dijkstra e eficiência

A eficiência do algoritmo de Dijkstra depende fortemente

- da estrutura de dados que usamos para implementar as operações
 - de escolha do vértice com menor valor de $dist[]$.
- Como fazemos repetidas operações de remoção do mínimo de um conjunto,
 - a escolha natural é utilizar um heap de mínimo.

Sendo n o número de elementos armazenados, um heap de mínimo

- suporta as operações de remover o mínimo e de inserir em tempo $O(\log n)$.
- Também conseguimos construir um heap em tempo $O(n)$.
- Além disso, é possível atualizar o valor de elementos no meio do heap
 - em tempo $O(\log n)$, o que é particularmente relevante nesta aplicação.
- Como implementar essa atualização? Discutiremos isso mais adiante.

Dijkstra Com Heap ($G=(V,E)$, c , s)

para todo $v \in V$ faça $dist[v] = +\infty$ e $pred[v] = NULL$

$dist[s] = 0$

$H \leftarrow \text{constróiHeap}(V, dist)$

enquanto $H \neq \emptyset$

$v \leftarrow \text{removeMinHeap}(H)$

para cada arco (v, w)

se $dist[w] > dist[v] + c(v, w)$

$dist[w] \leftarrow dist[v] + c(v, w)$

$pred[w] \leftarrow v$

atualizaHeap($H, w, dist[w]$)

Quiz 2: por que pode remover a verificação de w já ter sido visitado?

Código do algoritmo de Dijkstra com heap,

- que usa um vetor auxiliar `pos_H`, que é indexado pelos rótulos dos vértices,
 - e armazena a posição de cada vértice no vetor do heap `H`.

```
void DijkstraComHeap(Grafo G, int origem, int *dist, int *pred) {
    int i, *pos_H, v, w, custo, tam_H;
    Elem *H, elem_aux;
    Noh *p;

    // inicializando distâncias e predecessores
    for (i = 0; i < G->n; i++) {
        dist[i] = __INT_MAX__;
        pred[i] = -1;
    }

    dist[origem] = 0;

    // criando um min heap em H com vetor de posições pos_H
    H = malloc(G->n * sizeof(Elem));
    pos_H = malloc(G->n * sizeof(int));

    for (i = 0; i < G->n; i++) {
        H[i].chave = dist[i]; // chave é a "distância" do vértice
        H[i].conteudo = i;    // conteúdo é o rótulo do vértice
        pos_H[i] = i; // vértice i está na posição i do heap H
    }

    troca(&H[0], &H[origem]); // coloca origem no início do heap H
    troca_pos(&pos_H[0], &pos_H[origem]); // atualiza posição

    tam_H = G->n;
    while (tam_H > 0) { // enquanto não visitar todo vértice
        // buscando vértice v que minimiza dist[v]
        tam_H = removeHeap(H, pos_H, tam_H, &elem_aux);
        v = elem_aux.conteudo;

        // percorrendo lista com vizinhos de v
        for (p = G->A[v]; p != NULL; p = p->prox) {
            w = p->rotulo;
            custo = p->custo;

            // e atualizando as distâncias e predecessores quando for o caso
            if (dist[w] > dist[v] + custo) {
                dist[w] = dist[v] + custo;
                pred[w] = v;
                elem_aux.chave = dist[w];
                elem_aux.conteudo = w;
                atualizaHeap(H, pos_H, elem_aux);
            }
        }
    }
}
```

$O(n)$

$O(n)$

$O(n \log n)$

$O(\log n)$

$O(n)$

$O(|\delta(v)|)$

$\sum_{v \in V} |\delta(v)|$

$O(m \log n)$

```

- free(H);
- free(pos_H);
}

```



Eficiência: $O((n + m) \log n) = O(m \log n)$ se o grafo for conexo,

- pois nesse caso o número de arestas supera o número de vértices.
- Essa eficiência decorre de, em cada iteração do algoritmo,
 - um vértice ser removido do heap, o que leva tempo $O(\log n)$.
- Assim, ao longo de toda a execução as remoções levam tempo $O(n \log n)$.
- Além disso, cada arco (v, w) é considerado uma vez,
 - quando seu vértice origem v é removido do heap.
- No caso de (v, w) fazer parte de um caminho mais curto para w ,
 - o valor $\text{dist}[w]$ deve ser atualizado no heap, o que custa $O(\log n)$.
- No total, ao longo de toda a execução, as atualizações custam $O(m \log n)$.

Para a maior parte dos grafos essa é a melhor versão do algoritmo de Dijkstra,

- já que ela é quase linear no tamanho do grafo.
- A exceção são grafos particularmente densos, com $m = \Theta(n^2)$,
 - em que a complexidade do algoritmo é $O(m \log n) = O(n^2 \log n)$.
- Surpreendentemente, nesse caso conseguimos melhorar a eficiência
 - usando nossa implementação básica,
- que usa um vetor de tamanho n para armazenar as informações $\text{dist}[]$
 - e tem eficiência $O(n * n + m) = O(n^2)$.

Para a versão do algoritmo de Dijkstra com heap funcionar,

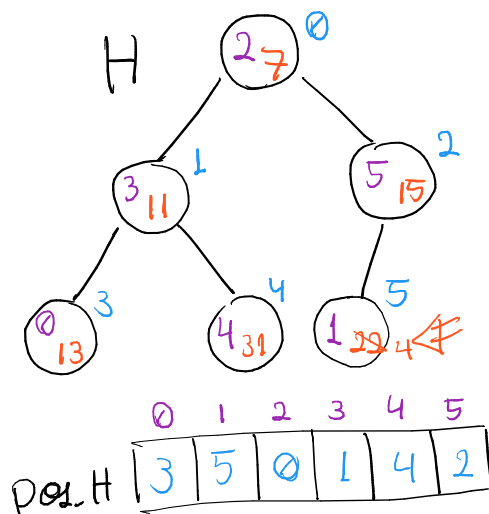
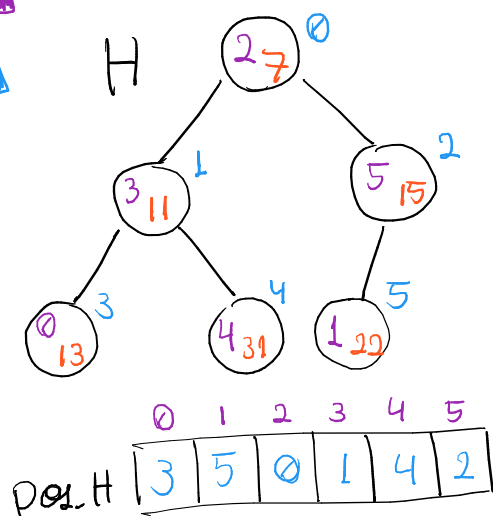
- é necessário implementar a função atualizaHeap.
- Para tanto é preciso modificar as funções sobeHeap e desceHeap.
- Isso porque, essas funções precisam manter atualizada,
 - num vetor auxiliar pos_H, a posição de cada elemento do heap H .

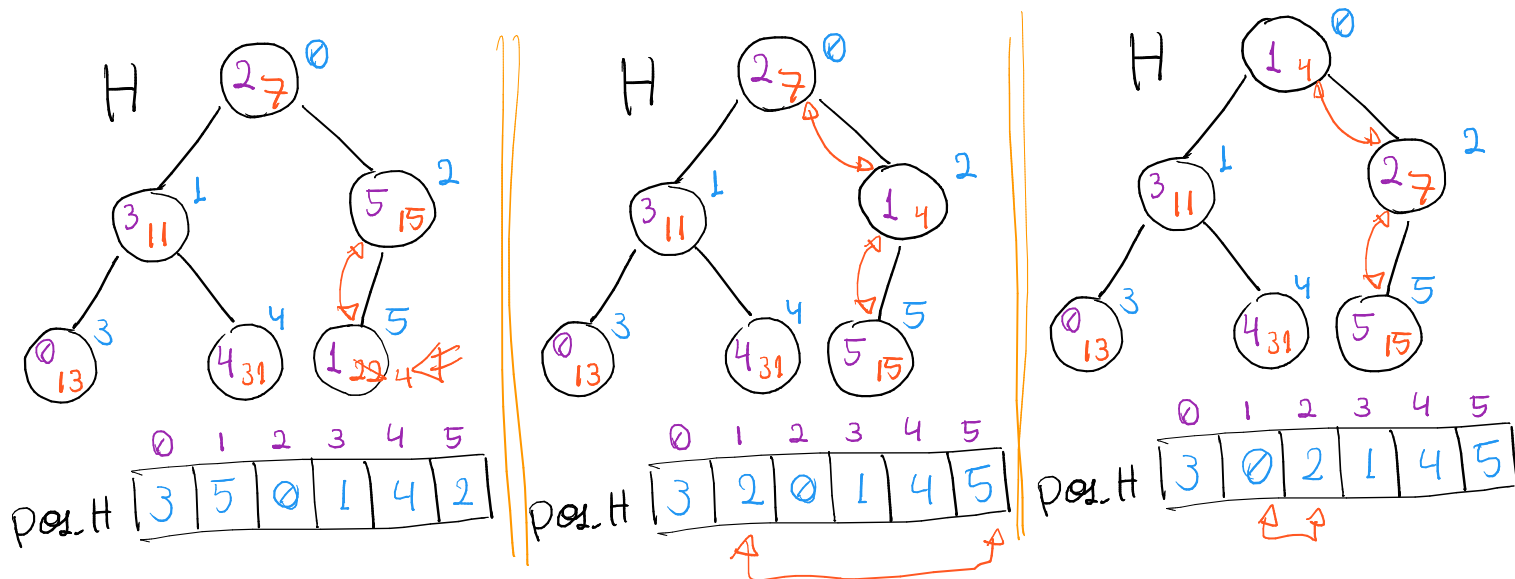
Exemplo de operações de um heap de mínimo,

- com conteúdo dos elementos independente do valor das chaves,
 - e atualizando o vetor auxiliar que guarda as posições no heap.

chave/prioridade
 conteúdo/rótulo
 posição no heap

atualize a chave do vértice 1 p/ valor 4





Código da estrutura de dados heap com operação de atualização

```
typedef struct elem {
```

```
    int chave; // vamos guardar dist aqui
```

```
    int conteudo; // vamos guardar o vértice aqui
```

```
} Elem;
```

```
#define PAI(i) (i - 1) / 2
```

```
#define FILHO_ESQ(i) (2 * i + 1)
```

```
#define FILHO_DIR(i) (2 * i + 2)
```

```
void troca(Elem *a, Elem *b);
```

```
void troca_pos(int *a, int *b);
```

```
// sobe o elemento em v[pos_elem_v] até restaurar a prop. do heap
```

```
void sobeHeap(Elem v[], int pos_v[], int pos_elem_v) {
```

```
    int f = pos_elem_v;
```

```
    while (f > 0 && v[PAI(f)].chave > v[f].chave) {
```

```
        troca(&v[f], &v[PAI(f)]);
```

```
        troca_pos(&pos_v[v[f].conteudo], &pos_v[v[PAI(f)].conteudo]);
```

```
        f = PAI(f);
```

```
    }
```

```
}
```

```
int insereHeap(Elem v[], int pos_v[], int m, Elem x) {
```

```
    v[m] = x;
```

```
    pos_v[x.conteudo] = m;
```

```
    sobeHeap(v, pos_v, m);
```

```
    return m + 1;
```

```
}
```

```
// desce o elemento em v[pos_elem_v] até restaurar a prop. do heap
void desceHeap(Elem v[], int pos_v[], int m, int pos_elem_v) {
    int p = pos_elem_v, f;
    - while (FILHO_ESQ(p) < m && (v[FILHO_ESQ(p)].chave < v[p].chave
    || (FILHO_DIR(p) < m && v[FILHO_DIR(p)].chave < v[p].chave))) {
        f = FILHO_ESQ(p);
        -> if (FILHO_DIR(p) < m && v[FILHO_DIR(p)].chave < v[f].chave)
            f = FILHO_DIR(p);
        - troca(&v[p], &v[f]);
        - troca_pos(&pos_v[v[p].conteudo], &pos_v[v[f].conteudo]);
        p = f;
    }
}
```

```
int removeHeap(Elem v[], int pos_v[], int m, Elem *px) {
    -> (*px) = v[0];
    - troca(&v[0], &v[m - 1]);
    - troca_pos(&pos_v[v[0].conteudo], &pos_v[v[m - 1].conteudo]);
    -> desceHeap(v, pos_v, m, 0);
    return m - 1;
}
```

```
void atualizaHeap(Elem v[], int pos_v[], Elem x) {
    int pos_x_v = pos_v[x.conteudo]; // pega a posição de x em v
    v[pos_x_v].chave = x.chave; // atualiza a chave de x em v
    -> sobeHeap(v, pos_v, pos_x_v); // Quiz3: por que mando subir e não
    descer? Em que situação mandaria descer?
}
```

Curiosidade:

- É possível implementar o algoritmo de Dijkstra com heap
 - sem usar a operação de atualização.
 - Quiz4: Como implementar essa versão?
- Dica: Inserir um mesmo vértice v várias vezes no heap.
 - Mais especificamente, cada vez que a distância de v é atualizada.
- Note que, isso não compromete o correto funcionamento do algoritmo,
 - pois sairá primeiro do heap a cópia do vértice com menor distância.
- No entanto, devemos modificar o algoritmo para
 - descartar vértices repetidos.