AED2 - Aula 27
Caminhos mínimos ponderados, grafos sem custos negativos e algoritmo de Dijkstra

Vamos continuar abordando o problema do caminho mínimo ponderado

- em grafos sem custos negativos.

Neste problema recebemos como entrada:

- Um grafo G = (V, E),

```
typedef struct noh Noh;
struct noh {
    int rotulo;
    int custo;
    Noh *prox;
};
```

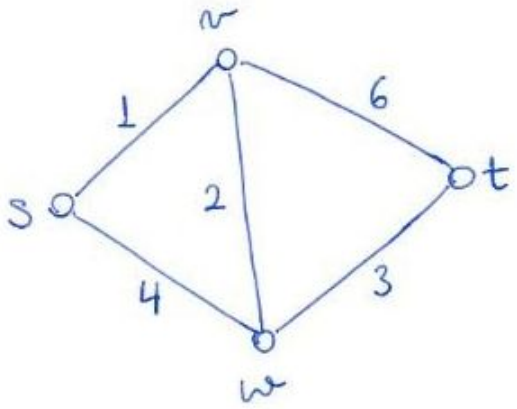
- com custo c(e) >= 0 em cada aresta e em E
- e um vértice origem s.

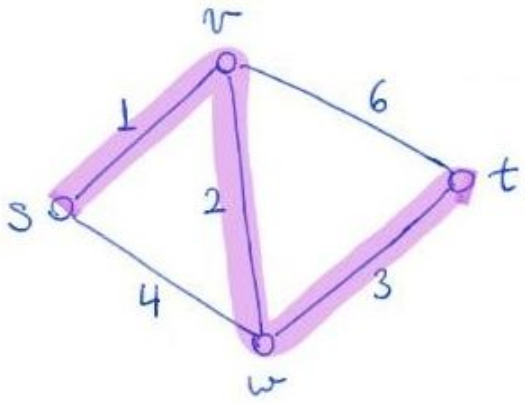
E queremos encontrar:

- O valor do caminho mínimo de s até cada vértice v em V,
    - i.e., a distância de s a v.
- Também gostaríamos que os caminhos mínimos fossem devolvidos.

Exemplo de grafo com custos nas arestas:

- Caminho mínimo de s até t.





Algoritmo de Dijkstra

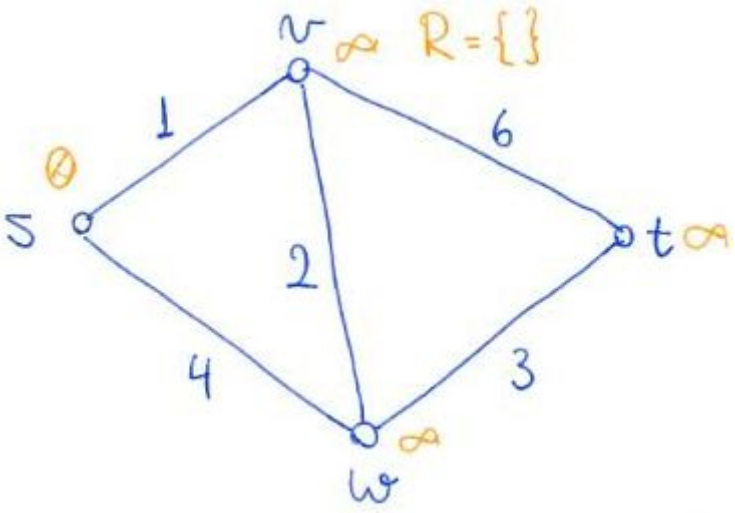
Vamos continuar a análise do algoritmo de Dijkstra para caminhos mínimos,

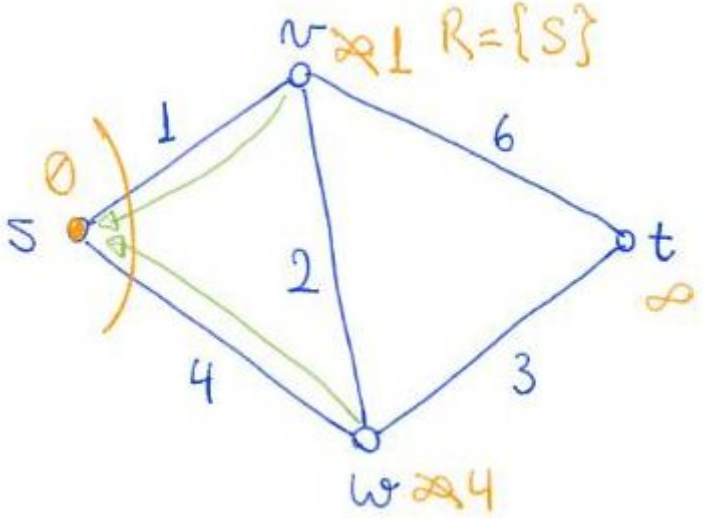
- que é um dos maiores clássicos da computação.

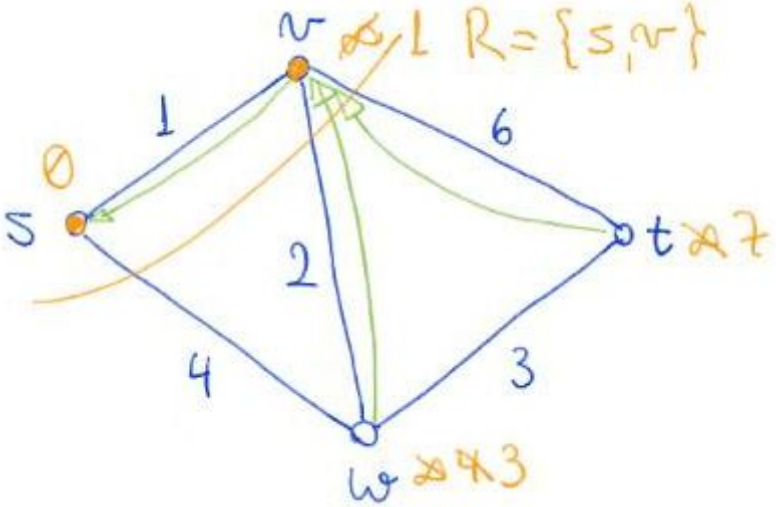
A seguir, para simplificar, vamos supor que todos os vértices do grafo
- são alcançáveis a partir do vértice s.
- Se esse não for o caso, podemos focar nos vértices alcançáveis
    - realizando uma busca a partir de s antes,
        - ou modificando levemente o algoritmo de Dijkstra.

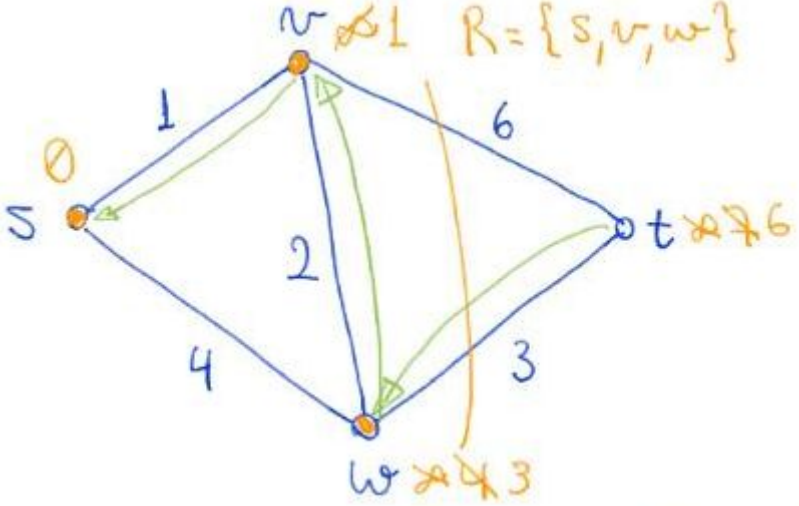
```
Dijkstra(grafo G=(V,E), custos c, vértice s) {
        para todo v \in V
                dist[v] = +\inf
                pred[v] = NULL
        dist[s] = 0
        R = {} // conjunto de vértices já visitados
        enquanto R != V
                // escolha gulosa do algoritmo de Dijkstra
                pegar o vértice v em V \ R com menor valor de dist[]
                adicione v a R
                para todo arco (v, w) com w em V \ R
                        se dist[w] > dist[v] + c(v, w)
                                dist[w] = dist[v] + c(v, w)
                                pred[w] = v
}
```

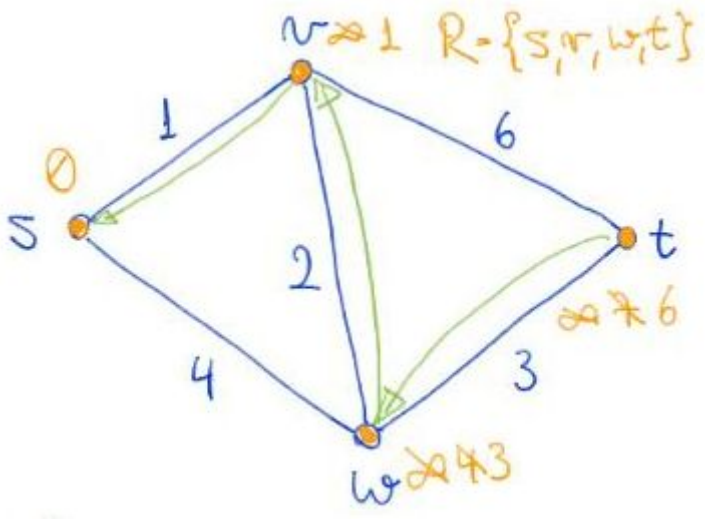
Exemplo de funcionamento do algoritmo:




v ∞ 1 R={s}
1
6
0
s
2
t
∞
4
3
w ∞ 4


v ∞ 1 R={s,v}
1
6
0
s
t ∞ 7
2
4
3
w ∞ 3


v ∞ 1 R={s,v,w}
1
6
0
s
t ∞ 6
2
4
3
w ∞ 3



Prova de corretude:

O algoritmo de Dijkstra mantém as seguintes propriedades invariantes

- no início de cada iteração do laço principal do algoritmo:
    1. para todo vértice v em R, o valor dist[v] corresponde
        - ao custo do caminho mínimo de s até v.
    2. para todo vértice v em V o vértice pred[v] corresponde
        - ao penúltimo vértice em um menor caminho de s até v
            - cujos vértices intermediários estão em R.
    3. para todo vértice v em V \ R o valor dist[v] corresponde ao custo
        - do menor caminho cujos vértices intermediários estão em R.

Faremos a prova por indução no número de iterações k.

Caso base: no início da primeira iteração R é vazio, portanto

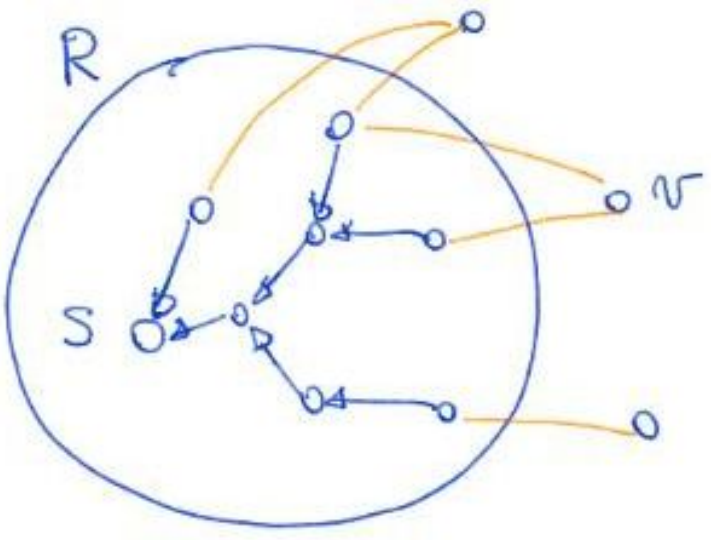
- as propriedades 1 e 2 valem trivialmente
- e a propriedade 3 vale porque todo vértice está em V \ R:
    - s é o único vértice com caminho sem vértices intermediários
        - e dist[s] = 0,
    - e os demais vértices não tem caminho sem vértices intermediários
        - e dist[ ] deles é +inf.

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

Passo: Considere a iteração k em que o vértice v é inserido em R.

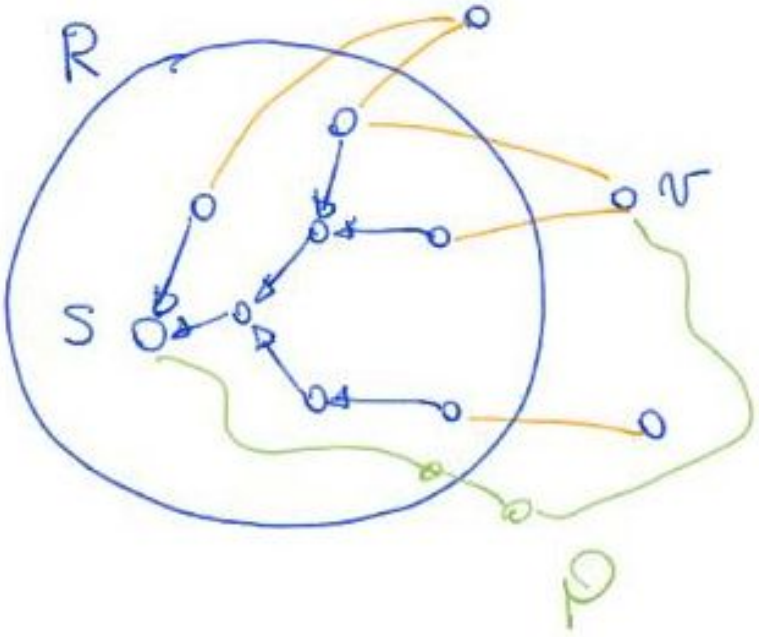
- Queremos mostrar que as propriedades 1, 2 e 3 continuam valendo
    - no início da iteração k + 1.

O vértice v é inserido por ser o vértice fora de R com menor valor para dist[ ].



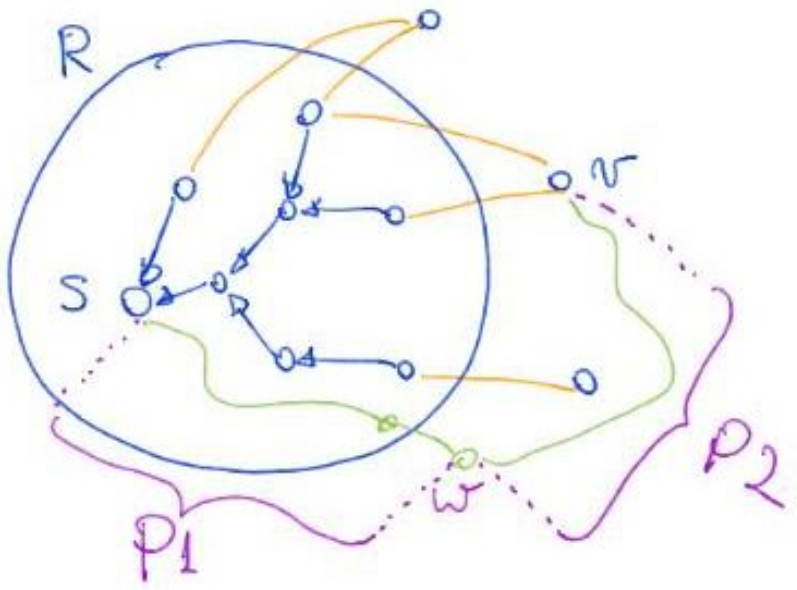
Pela propriedade 3 da H.I. temos que v é o vértice com menor caminho

- 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
    - P1 (parte que vai de s a w)
    - e P2 (parte que vai de w a v).



- Pela propriedade 3 da H.I. temos que,
    - dist[w] é o menor caminho de s até w que só usa vértices em R.
        - Portanto, c(P1) >= dist[w].
- Como não temos arcos de custo negativo,
    - c(P2) >= 0.
- Assim, c(P) = c(P1) + c(P2)
  >= dist[w] + 0 >= dist[v],
    - pela escolha de v como o vértice que minimiza dist[ ].

Como mostramos que o custo de um caminho P qualquer de s até v

- tem custo maior ou igual a dist[v],
    - concluímos que dist[v] corresponde ao custo
        - de um caminho mínimo de s até v.
- Ou seja, a propriedade 1 do invariante se mantém no início da iteração k + 1,
    - depois que v é adicionado a R.

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

- Para os vértices cujos valores dist[ ] e pred[ ]
    - não mudaram ao longo da iteração k
        - o resultado segue da H.I..
- Vamos considerar um vértice z para o qual
    - os valores dist[z] e pred[z] mudaram na iteração k.
- Neste caso dist[z] = dist[v] + c(v,z) é menor que o valor antigo de dist[z],
    - já que essa é a condição do algoritmo para atualizar o valor de dist[z].
- Portanto, dist[z] corresponde ao custo do menor caminho
    - cujos vértices intermediários estão em R,
        - já que agora v está em R
        - e o antigo dist[z] respeitava a propriedade 3 da H.I..

    - Ou seja, a propriedade 3 continua valendo.
- Além disso, a propriedade 2 continua valendo porque pred[z] passa a ser v.

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
        p = G->A[v];
        while (p != NULL) {
            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;
            }
            p = p->prox;
        }
    }
    free(R);
}
```

Eficiência: O(n * n + m),

- pois o while 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 <= 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[ ]
    - (e também na qual iremos atualizar os valores de dist[ ]
        - conforme encontramos novas arestas).
- A escolha tradicional neste caso, já que fazemos
    - sucessivas operações de remoção do mínimo de um conjunto,
        - é utilizar um heap de mínimo.

Um heap de mínimo suporta as operações

- de remover o mínimo e de inserir em tempo O(log n).
- Também conseguimos construir um heap com n elementos
  - 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 operação de atualização?
  - Discutiremos isso mais adiante.

```
DijkstraComHeap(grafo G=(V,E), custos c, vértice s) {
        para todo v \in V
                dist[v] = +\inf
                pred[v] = NULL
        dist[s] = 0
        H = constroiHeap(V, dist)
        enquanto H != { }
                // escolha gulosa do algoritmo de Dijkstra
                v = removeMinHeap(H)
                para todo arco (v, w)
                        se dist[w] > dist[v] + c(v, w)
                                dist[w] = dist[v] + c(v, w)
                                pred[w] = v
                                atualizaHeap(H, w, dist[w])
}
```

Se implementarmos o algoritmo de Dijkstra usando um heap de mínimo

- ele terá eficiência de tempo de pior caso O(m log n),
  - supondo que o grafo seja conexo, pois nesse caso
    - o número de arestas supera o número de vértices.
- Essa eficiência vem do fato que
  - em cada iteração do algoritmo um vértice é removido do heap.
- Assim, ao longo de toda a execução
  - as remoções levam tempo O(n log n).
- Além disso, cada arco é considerado uma vez,
  - quando seu vértice origem é removido do heap.
- No caso deste arco fazer parte de um caminho mais curto
  - do que o já encontrado para seu o vértice destino
    - o valor dist[ ] do vértice destino tem que ser atualizado no heap,
      - o que custa O(log n).
- No total, ao longo de toda a execução do algoritmo,
  - essas operações de atualização custam O(m log n).

Para a maior parte dos grafos essa é a melhor escolha

- para se implementar o algoritmo de Dijsktra,
    - já que ela é quase linear no tamanho do grafo.
- No entanto, existe uma exceção.
    - Em grafos particularmente densos, temos que m = O(n^2).

Nestes grafos, a complexidade do algoritmo será O(m log n) = O(n^2 log n)

- Será que neste caso conseguimos fazer melhor?
    - A resposta é sim.
- Surpreendentemente, neste caso conseguimos
    - melhorar a eficiência de pior caso do algoritmo
        - usando uma estrutura de dados mais simples no lugar do heap.
- De fato, nossa implementação básica, que usa apenas
    - um vetor de tamanho n para armazenar as informações dist[ ],
- tem eficiência O(n * n + m) = O(n^2).

Código de uma implementação com heap do algoritmo de Dijkstra,

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

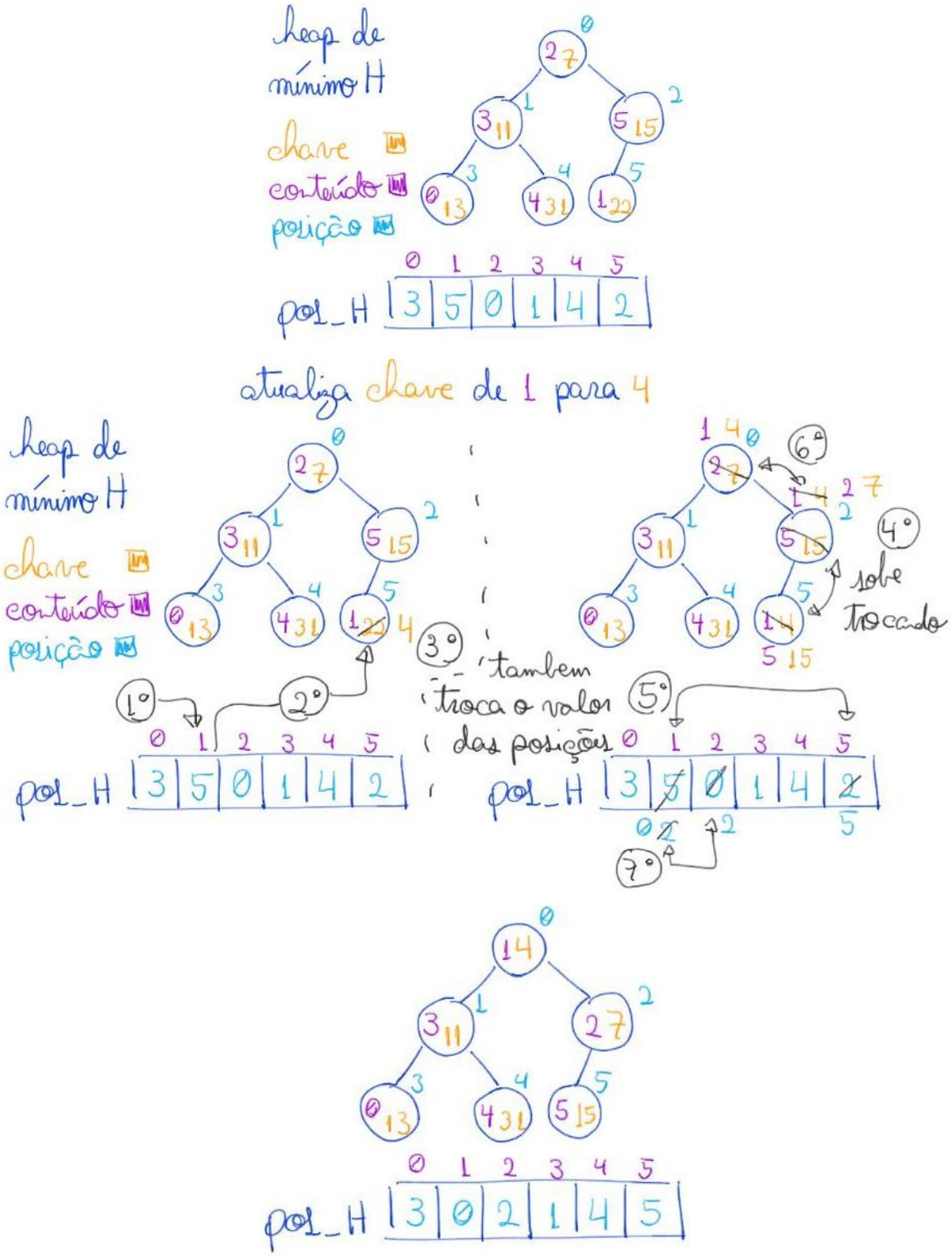
```c
void DijkstraComHeap(Grafo G, int origem, int *dist, int *pred) {
    int i, *pos_H;
    Elem *H, elem_aux;
    int v, w, custo, tam_H;
    Noh *p;
    // inicializando distancias e predecessores
    for (i = 0; i < G->n; i++) {
        dist[i] = __INT_MAX__;
        pred[i] = -1;
    }
    dist[origem] = 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;    // conteudo é o rótulo do vértice
        pos_H[i] = i; // vértice i começa 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 dos
vértices
    tam_H = G->n;
    // enquanto não encontrar o caminho mínimo para todo vértice
    while (tam_H > 0) {
        // 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
        p = G->A[v];
        while (p != NULL) {
            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);
            }
            p = p->prox;
        }
    }
    free(H);
    free(pos_H);
}
```

- Note que é necessário implementar a função atualizaHeap, e que
    - para tanto é necessário modificar as funções sobeHeap e desceHeap.
- Isso porque, essas funções precisam manter atualizada,
    - num vetor auxiliar pos_H,
        - a posição em que está cada elemento no heap H.

Código das 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.




```
typedef struct elem {
    int chave; // vamos guardar dist aqui
    int conteudo; // vamos guardar o vértice aqui
} Elem;
```

```c
#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) {
    Elem aux = *a;
    *a = *b;
    *b = aux;
}

void troca_pos(int *a, int *b) {
    int aux = *a;
    *a = *b;
    *b = aux;
}

// 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;
    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);// por que mando subir e não descer?
}
```

Curiosidade:

- É possível implementar o algoritmo de Dijkstra usando heap
    - sem usar operação de atualização
        - e vetor auxiliar com posições dos vértices no heap.
    - Como?
- 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/prioridade.