

Caminhos Mínimos, Algoritmo de Dijkstra, Heap com Atualização

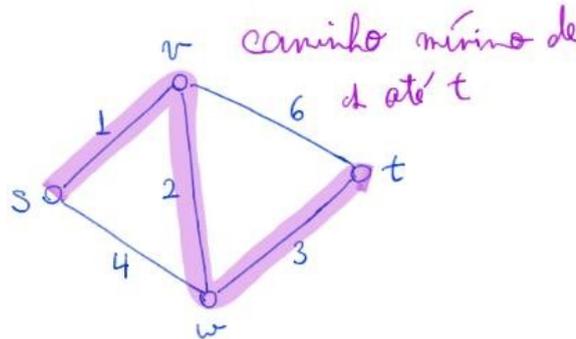
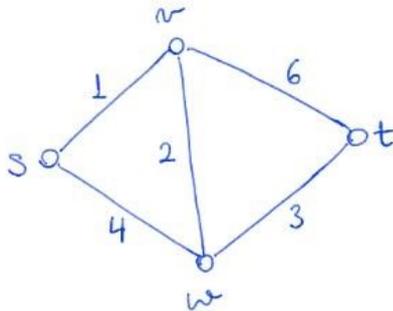
Vamos tratar do problema dos caminhos mínimos em grafos sem custos negativos.

- A entrada deste problema é um grafo $G = (V, E)$,
 - com custo $c(e) > 0$ em cada aresta e em E ,
 - e um vértice origem s .

Nosso objetivo é encontrar: o valor/custo do caminho mínimo de s

- até cada vértice v em V , i.e., a distância de s a v .
- Também gostaríamos que esses caminhos fossem devolvidos.

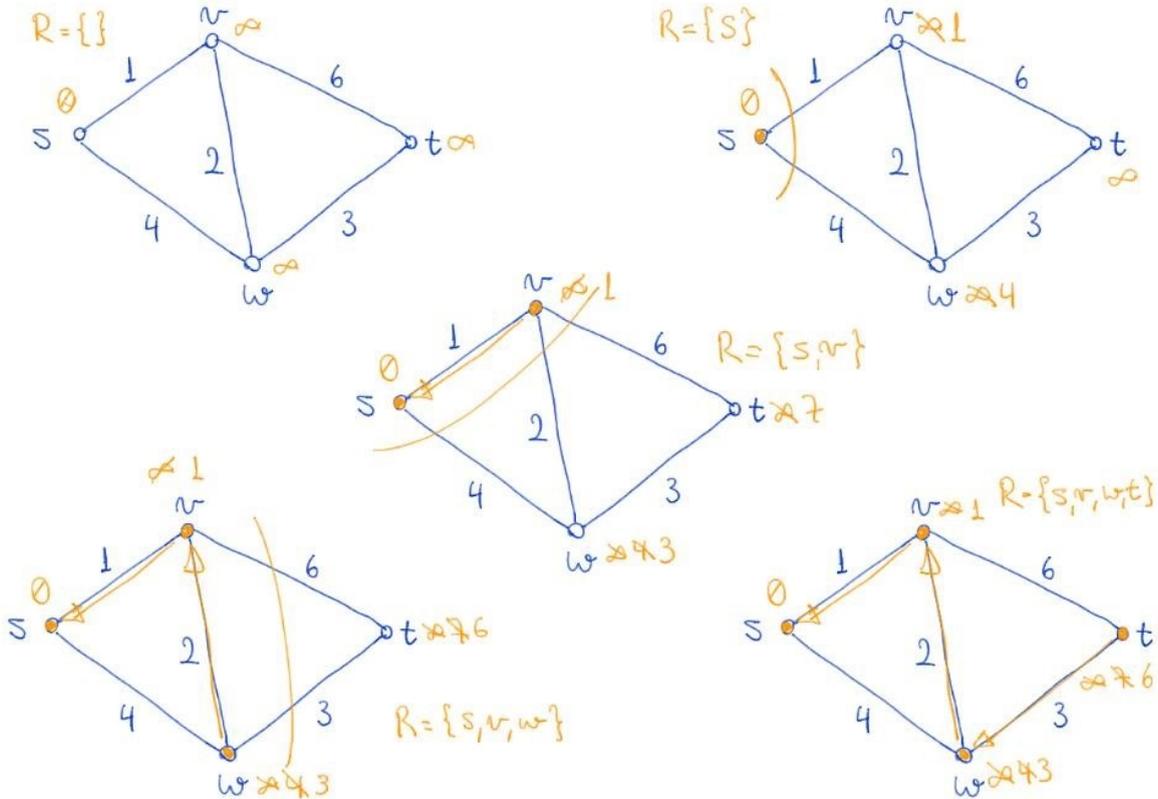
Exemplo de grafo com custos nas arestas: Caminho mínimo de s até t .



Algoritmo de Dijkstra

Vamos estudar um dos maiores clássicos da computação,

- o algoritmo para caminhos mínimos de Dijkstra, que é um algoritmo guloso.



Visto um exemplo, vamos escrever o pseudocódigo do algoritmo.

- 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 inicial a partir de s ,
- ou modificar levemente o algoritmo de Dijkstra. Quiz1: Como fazer isso?

Dijkstra(grafo $G=(V,E)$, custos c , vértice s):

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

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

$R = \{ \}$ // conjunto de vértices já visitados

enquanto $R \neq V$:

 // escolha gulosa do algoritmo de Dijkstra

 pegar o vértice v em $V \setminus R$ com menor valor de $\text{dist}[]$

 adicione v a R

 para toda aresta (v, w) com w em $V \setminus R$:

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

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

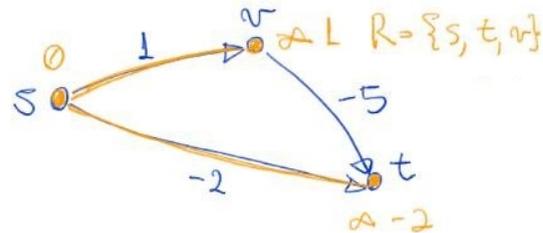
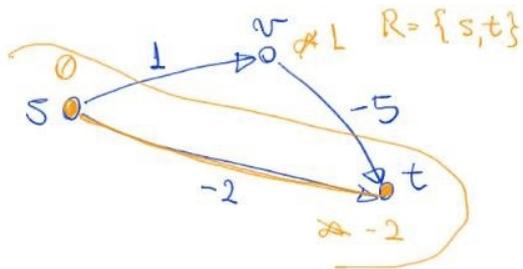
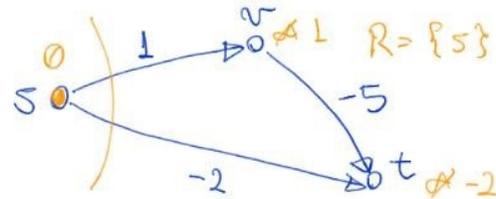
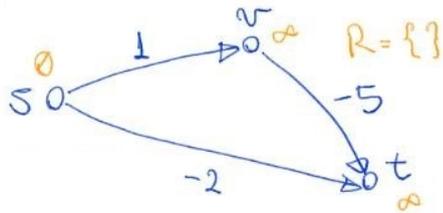
$\text{pred}[w] = v$

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.

Antes de provarmos a corretude do algoritmo,

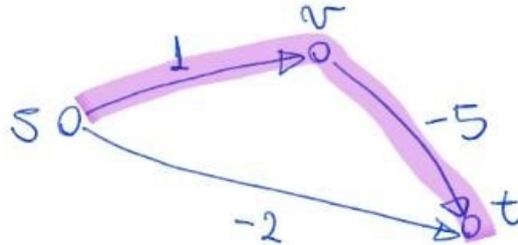
- vamos entender suas limitações, a fim de compreendê-lo melhor.
- Em particular, esse algoritmo pode não devolver a solução correta
 - quando as arestas apresentam custo negativo.



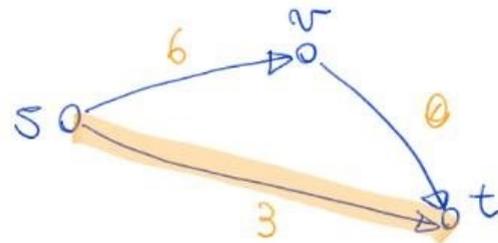
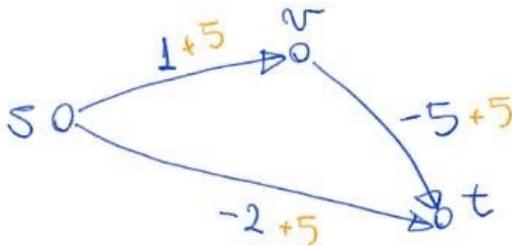
O algoritmo falha porque a escolha gulosa pode definir

- um valor para a distância de um vértice,
 - sem considerar um caminho menor.

- Isso ocorre quando tal caminho possui um custo maior na sua parte inicial,
- que depois é reduzido por conta de arestas de custo negativo.
 - No nosso exemplo, o caminho $s \rightarrow v \rightarrow t$, que tem custo -4 ,
 - nunca é considerado pelo algoritmo.



- Poderíamos, ainda, pensar em reduzir o problema com arestas negativas
- para o problema sem essas arestas.
 - Uma tentativa seria somar o valor absoluto da aresta mais negativa
 - no comprimento de todas as arestas.
 - Isso certamente faria com que o grafo não tivesse mais arestas negativas.



No entanto, o caminho mínimo entre os vértices pode ser alterado,

- pois caminhos com números diferentes de arestas
 - seriam afetados de modo distinto.

Como regra geral, quando tentamos eliminar custos negativos,

- se as soluções do seu problema tem número variado de objetos
 - (no caso de caminhos mínimos os objetos são as arestas)
- então somar um mesmo valor no custo de cada objeto
 - afetará mais o custo de soluções com maior cardinalidade,
 - e menos o custo daquelas com menor cardinalidade.
- Assim, não há garantia de preservar a ordem relativa das soluções.
 - Por isso, esse procedimento não é recomendado.

Por outro lado, se todas as soluções do seu problema tem a mesma cardinalidade,

- i.e., o mesmo número de objetos,
- então somar um mesmo valor no custo de cada objeto
 - afetará homogeneamente o custo de todas as soluções.
- Neste caso, como a ordem relativa dos custos das soluções é preservada,
 - o procedimento é seguro.
- Este é o caso, por exemplo, do problema da árvore geradora mínima,
 - um problema central em otimização combinatória e grafos
 - que veremos nas próximas aulas.

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 $\text{dist}[v]$ corresponde
 - ao custo do caminho mínimo de s até v .
 2. para todo vértice v em V o vértice $\text{pred}[v]$ corresponde
 - ao penúltimo vértice em um menor caminho de s até v
 - cujos vértices intermediários estão em R .
 3. para todo vértice v em $V \setminus R$ o valor $\text{dist}[v]$ corresponde ao custo
 - do menor caminho cujos vértices intermediários estão em R .

Faremos a prova por indução no número de iterações k .

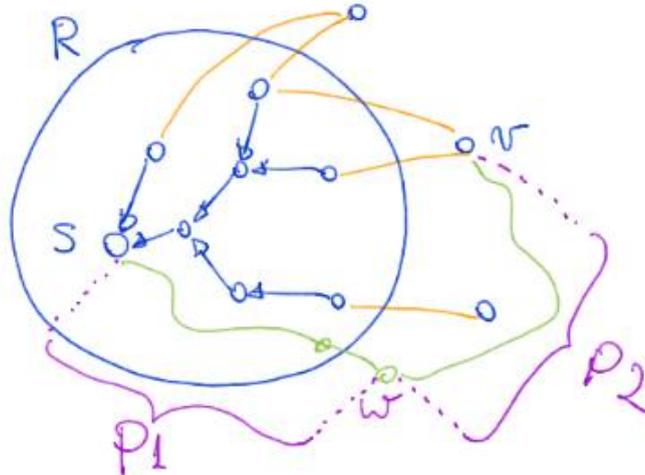
- H.I.: As propriedades 1, 2 e 3 do invariante valem no início da iteração k .

Caso base: no início da primeira iteração

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

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

- as propriedades 1, 2 e 3 valem 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, $c(P1) \geq \text{dist}[w]$.
- Como não temos arcos de custo negativo, $c(P2) \geq 0$.
- Assim, pela escolha de v como o vértice que minimiza $\text{dist}[\]$,
 - temos $c(P) = c(P1) + c(P2) \geq \text{dist}[w] + 0 \geq \text{dist}[v]$.

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 (prop. 3)
- e $\text{pred}[w]$ aponta para o penúltimo vértice nesse caminho (prop. 2).

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 $\text{dist}[\]$.
- Como fazemos repetidas operações de remoção do mínimo de um conjunto,
 - a escolha natural é utilizar um heap de mínimo.

Pseudocódigo com heap:

DijkstraComHeap(grafo $G=(V,E)$, custos c , vértice s):

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

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

$H = \text{constroiHeap}(V, \text{dist})$

enquanto $H \neq \{ \}$:

 // escolha gulosa do algoritmo de Dijkstra

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

 para todo arco (v, w) : // Quiz2: por que pude remover uma condição?

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

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

$\text{pred}[w] = v$

$\text{atualizaHeap}(H, w, \text{dist}[w])$

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 $O(m \log n)$.

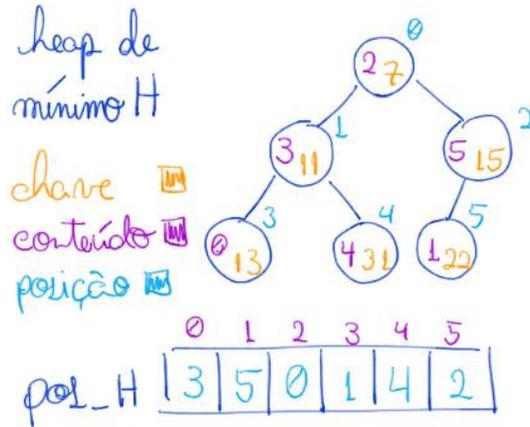
Para a maior parte dos grafos essa é a melhor versão do algoritmo de Dijkstra,

- já que ela é quase linear no tamanho do grafo.
- A exceção são grafos particularmente densos, com $m = O(n^2)$,
 - 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 $O(n * n + m) = O(n^2)$.

Seja 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? Para implementar a função
 - `atualizaHeap`, 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 .

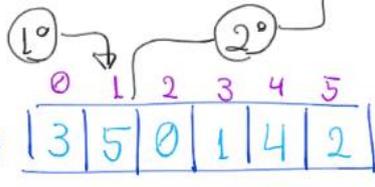
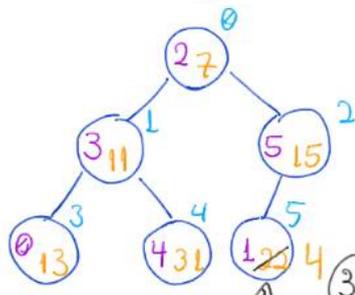
Exemplo de operações de um heap de mínimo, atualizando o vetor auxiliar.



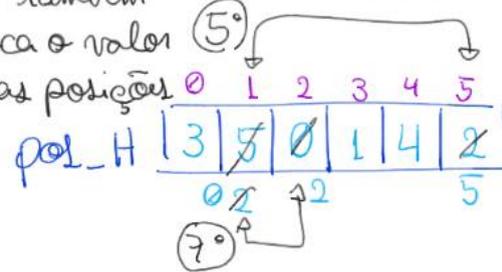
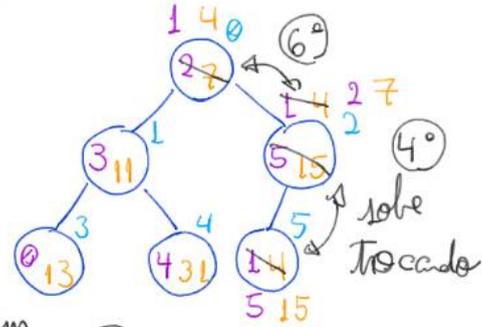
atualiza chave de 1 para 4

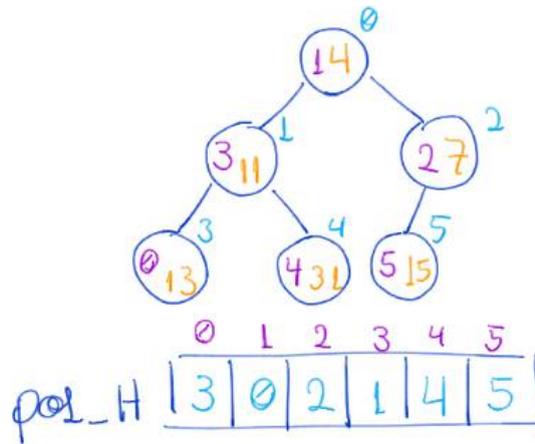
heap de
mínimo H

chave \square
conteúdo \square
posição \square



3^o - também
troca o valor
das posições





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.