MCTA003-17 – Análise de Algoritmos
Centro de Matemática, Computação e Cognição
Universidade Federal do ABC
Profa. Carla Negri Lintzmayer

Aula 15 - Algoritmos de Prim e Dijkstra e Corretude do Kruskal

1 Problema da Árvore Geradora Mínima (MST)

O problema da árvore geradora mínima (MST, do inglês *minimum spanning tree*), é definido a seguir.

> Entrada: Grafo $G = (V, E)$ conexo e função c de custo sobre as arestas.
> Saída: Árvore $T \subseteq E$ (isto é, (V, T) não tem ciclos), que é geradora (isto é, (V, T) é conexo) e que tem custo mínimo, onde o custo de T, $c(T)$, é definido como $\sum_{e \in T} c(e)$.

Na Seção 1.1 veremos a corretude do algoritmo de Kruskal, que resolve esse problema e foi apresentado em sala de aula. Na Seção 1.2 veremos o algoritmo de Prim, que também resolve esse problema. Ambos executam em tempo $O(m \log n)$ sobre qualquer grafo $G = (V, E)$ com função c de custo nas arestas, onde $n = |V|$ e $m = |E|$. Os três resultados a seguir serão usados em ambas seções. Para isso, precisamos da seguinte definição.

Definição 1. *Um* corte $(S, V - S)$ *de um grafo $G = (V, E)$ é uma partição de V em dois conjuntos não vazios S e $V - S$ (isto é, alguns vértices de G estão em S e todos os outros estão no outro conjunto). Arestas que têm um extremo em cada conjunto são chamadas de* arestas do corte.

O lema a seguir relaciona ciclos e cortes.

Lema 1. *Dado um grafo $G = (V, E)$, suponha que um ciclo de G contém uma aresta que cruza um corte qualquer $(S, V - S)$. Então deve existir outra aresta do ciclo que também cruza o corte.*

Demonstração. Seja $C = (v_1, v_2, \ldots, v_k, v_1)$ o ciclo, ou seja, $v_i \in V$ para $1 \leq i \leq k$, $v_i v_{i+1} \in E$ para $1 \leq i < k$ e $v_k v_1 \in E$. Suponha, sem perda de generalidade, que $e = (v_1, v_2)$ é a aresta que cruza o corte $(S, V - S)$ e que $v_1 \in S$ e $v_2 \in V - S$. Como cada vértice do ciclo está ou em S ou em $V - S$, deve haver um v_j, com valor $j \geq 2$ mínimo tal que $v_2, v_3, \ldots, v_j$ é um caminho com vértices todos em $V - S$ e $v_{j+1} \notin V - S$. Mas então $v_{j+1} \in S$ e a aresta $v_j v_{j+1}$ é aresta do ciclo diferente de e que também cruza o corte. □

O resultado a seguir é direto desse lema.

Corolário 2. *Se e é a única aresta de um grafo $G = (V, E)$ que cruza um certo corte $(S, V - S)$, então ela não pertence a nenhum ciclo naquele grafo.*

O próximo teorema é o último resultado que precisamos.

Teorema 3. *Considere uma aresta e de um grafo $G = (V, E)$. Suponha que existe um corte $(S, V - S)$ tal que e é uma aresta de menor custo que o cruza. Então e pertence a alguma MST de G.*

Demonstração. Suponha por contradição que e não está em nenhuma MST de G. Seja T^* uma MST de G. Adicionando e a T^*, devemos ter um ciclo C criado, pois em T^* já existe um caminho conectando os extremos de e. Pelo Lema 1, existe outra aresta $f \in C$ tal que $f \in T^*$ e que cruza $(S, V - S)$ também. Então temos que $T' = T^* \cup \{e\} \setminus \{f\}$ é uma árvore geradora. Como $c(f) \geq c(e)$ (pois e é uma aresta de custo mínimo no corte e f também está no corte), temos que $c(T') = c(T^*) + c(e) - c(f) \leq c(T^*)$. Mas então T' também deve ser árvore geradora mínima e ela contém e. □

1.1 Algoritmo de Kruskal

Vimos em sala de aula o algoritmo de Kruskal, utilizando a estrutura de dados Union-Find, executa em tempo $O(m \log n)$ sobre um grafo $G = (V, E)$, onde $n = |V|$ e $m = |E|$. Veremos agora que esse algoritmo é correto, isto é, que ele de fato encontra uma árvore geradora de custo mínimo para qualquer grafo dado na entrada.

Teorema 4. *O algoritmo de Kruskal constrói uma MST para qualquer grafo $G = (V, E)$ conexo e função de custo c sobre as arestas.*

Demonstração. Seja T_i o conjunto de arestas na i-ésima iteração e seja T o conjunto de arestas devolvido no fim do algoritmo. Claramente, por construção, T não tem ciclos. Basta mostrar então que (V, T) é conexo e que $c(T)$ é mínimo.

Considere um corte qualquer $(S, V - S)$. Seja $e \in E$ a primeira aresta que cruza esse corte que é considerada pelo Kruskal e suponha que isso acontece na k-ésima iteração. Note que o Kruskal só se importa com as arestas $T_k \cup \{e\}$. Se ela é a primeira considerada nesse corte, então ela é sozinha no corte. Sendo sozinha, pelo Corolário 2, ela não cria ciclos em (V, T_k). Não criando ciclos, ela é adicionada ao conjunto que está sendo construído. Como escolhemos um corte qualquer, e para qualquer corte mostramos que existe uma aresta que o cruza, então (V, T) é conexo.

Por fim, seja $e = uv$ uma aresta de T que é adicionada na k-ésima iteração. Seja $S \subseteq V$ o componente do grafo (V, T_k) que contém u. Logo, S não contém v. Como e tem o menor custo em $(S, V - S)$ (pela ordem de escolha do algoritmo), então pelo Teorema 3 ela deve fazer parte de uma MST. Ou seja, o algoritmo apenas fez escolhas de arestas que estão em MST e, portanto, construiu uma MST. □

1.2 Algoritmo de Prim

A ideia do algoritmo de Prim é começar escolhendo um vértice arbitrário qualquer e repetidamente escolher um novo vértice que seja adjacente a algum dos vértices já escolhidos e cuja aresta de conexão entre ambos tenha o menor custo até que todos os vértices sejam escolhidos. Veja a Figura 1 para um exemplo de seu funcionamento.

A seguir apresentamos um pseudocódigo do algoritmo de Prim. Ele vai manter um conjunto X dos vértices já escolhidos e um conjunto T das arestas que formam a árvore. Toda vez que um novo vértice for escolhido, a aresta que tem menor custo que o conecta aos já escolhidos é adicionada a T.

1: **função** PRIM(G, c)
2: Seja $X = \{s\}$ onde s é um vértice arbitrário ▷ X vai manter o conjunto dos vértices escolhidos
3: $T \leftarrow \emptyset$ ▷ T vai manter as arestas da árvore
4: **enquanto** $X \neq V$ **faça**
5: Seja $e = uv$ uma aresta de menor custo onde $u \in X$ e $v \notin X$
6: $T \leftarrow T \cup \{e\}$
7: $X \leftarrow X \cup \{v\}$
8: **fim enquanto**
9: **devolve** T
10: **fim função**

Esse algoritmo é guloso porque a cada iteração escolhe a aresta de menor custo que conecta um vértice já escolhido a um vértice não escolhido.

Perceba que o algoritmo de Prim mantém a invariante que X é o conjunto de vértices cobertos pela árvore T construída até o momento. Note também a diferença entre Kruskal e Prim: em uma iteração qualquer, Kruskal pode conter vários componentes no grafo (V, T) enquanto que Prim contém apenas um. O teorema a seguir mostra que Prim está correto.

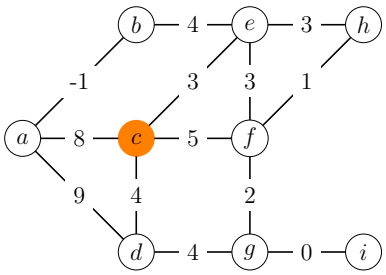
Teorema 5. *O algoritmo de Prim constrói uma MST para qualquer grafo $G = (V, E)$ conexo e função de custo c sobre as arestas.*

Demonstração. Note que o algoritmo termina: se esse não fosse o caso, haveria alguma iteração onde o corte $(X, V - X)$ seria vazio (não haveria escolha para e), o que significaria que G não é conexo, uma contradição. Então no fim temos de fato $X = V$.

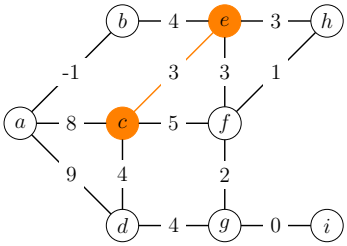
Seja T a árvore devolvida ao fim da execução. Por construção, T é geradora pois cobre X.

Note agora que T não tem ciclos: considere uma iteração onde e é escolhida para ser adicionada a T. Neste momento, todas as arestas de T têm extremos em X, então e é a primeira aresta a cruzar $(X, V - X)$ em (V, T) e, portanto, não participa de ciclos em T, pelo Corolário 2.

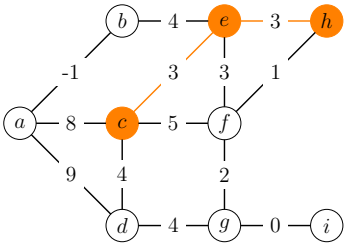
Resta mostrar que $c(T)$ é mínimo: cada aresta $e \in T$ é a menor do corte $(X, V - X)$ no



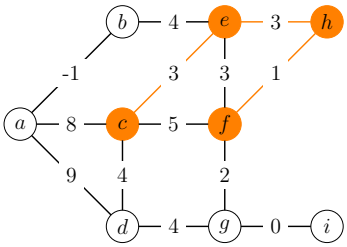
(a) Grafo G de entrada. O vértice c foi escolhido como inicial arbitrariamente.



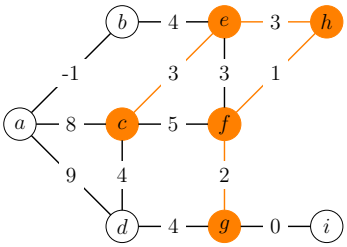
(b) Primeira iteração: escolhidos $= \{c\}$; aresta de menor custo que liga um não escolhido a um escolhido $= ce$.



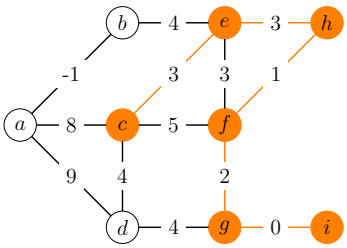
(c) Segunda iteração: escolhidos $= \{c, e\}$; aresta de menor custo que liga um não escolhido a um escolhido $= eh$.



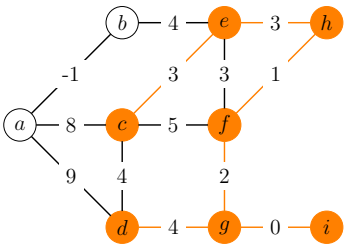
(d) Terceira iteração: escolhidos $= \{c, e, h\}$; aresta de menor custo que liga um não escolhido a um escolhido $= hf$.



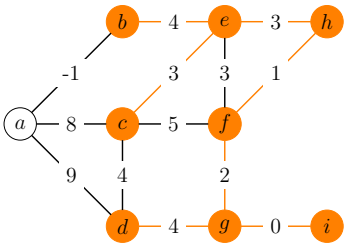
(e) Quarta iteração: escolhidos $= \{c, e, h, f\}$; aresta de menor custo que liga um não escolhido a um escolhido $= fg$.



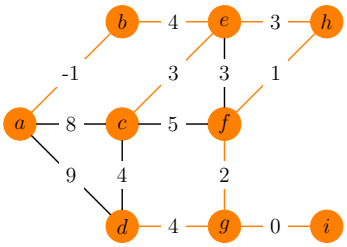
(f) Quinta iteração: escolhidos $= \{c, e, h, f, g\}$; aresta de menor custo que liga um não escolhido a um escolhido $= gi$.



(g) Sexta iteração: escolhidos $= \{c, e, h, f, g, i\}$; aresta de menor custo que liga um não escolhido a um escolhido $= gd$.



(h) Sétima iteração: escolhidos $= \{c, e, h, f, g, i, d\}$; aresta de menor custo que liga um não escolhido a um escolhido $= eb$.



(i) Oitava iteração: escolhidos $= \{c, e, h, f, g, i, d, b\}$; aresta de menor custo que liga um não escolhido a um escolhido $= ba$.

Figura 1: Exemplo de execução do Prim.

momento de sua adição. Então, pelo Teorema 3, T é MST. □

Com relação ao tempo de execução, podemos implementar o Prim diretamente da seguinte forma. A cada uma das $O(n)$ iterações, procure pela aresta de menor custo que tem um extremo em X e outro fora de X percorrendo todas as arestas do grafo, o que leva tempo $O(m)$, portanto. Então em tempo total $O(mn)$ temos uma MST.

Uma ideia melhor é usar a estrutura de dados heap, que fornece a operação de remover elementos de menor valor em tempo $O(\log k)$, onde k é a quantidade de elementos armazenados na estrutura.

Definição 2. Heaps *são estruturas de dados que conceitualmente podem ser vistas como uma árvore binária quase completa: todos os níveis estão cheios, exceto talvez pelo último, que é preenchido da esquerda para a direita. Em uma heap a seguinte propriedade é mantida: um nó tem chave menor do que a chave de seus dois filhos (se eles existirem). Sendo assim, o menor elemento de toda a estrutura deve estar obrigatoriamente na raiz.*

Uma heap de tamanho k é representada por um vetor de k posições: o elemento na posição i tem filho esquerdo na posição $2i$ (se $2i \leq k$), seu filho direito na posição $2i+1$ (se $2i+1 \leq k$), e seu pai na posição $\lfloor i/2 \rfloor$ (se $i > 1$). Assim, percorrendo o vetor da esquerda para a direita, vemos todos os elementos do nível x consecutivamente antes dos elementos do nível $x+1$.

- *A operação de remover o elemento de menor valor é feita da seguinte forma: trocamos o primeiro elemento (raiz) com o último (nó mais à direita do último nível) e restauramos a propriedade de heap fazendo trocas entre pais e filhos, começando da raiz e descendo pelo caminho onde as trocas vão sendo feitas, podendo chegar a uma folha (*Corrige-Descendo*).*

- *A operação de inserção de um novo elemento é feita da seguinte forma: insira-o à direita do último elemento (o mais à esquerda possível no último nível) e restaure a propriedade de heap fazendo trocas entre filhos e pais, começando da posição da inserção e subindo pelo caminho onde as trocas vão sendo feitas, podendo chegar á raiz (*Corrige-Subindo*).*

Assim, o tempo de execução de ambas é bem parecido e não passa de $O(\log k)$, que é o tamanho do maior caminho raiz-folha.

Para o Prim, vamos usar a heap para armazenar os vértices de $V - X$. Para cada $v \in V - X$, a prioridade de v será o custo da aresta de menor custo uv que tem $u \in X$. Se tal aresta não existir, a prioridade será ∞.

Na inicialização temos apenas $X = \{s\}$, então em tempo $O(m)$ calculamos os valores das prioridades dos vértices em $V - X$ e com $O(n \log n)$ operações inserimos cada um na heap. Então ao todo temos $O(m) + O(n \log n) = O(m \log n)$ (pois $m \geq n - 1$) na inicialização.

Se a heap mantiver a invariante de armazenar $V - X$ com as prioridades mencionadas, então remover o elemento de menor valor equivale a encontrar $v \notin X$ pertencente à aresta uv que

cruza $(X, V - X)$ e pode ser adicionado a X (e a aresta uv pode ser adicionada a T). Para manter essa invariante, note que a cada vértice v removido da heap é suficiente recalcular as prioridades dos vértices que são adjacentes a v: é de v que saem as únicas arestas que não estavam no corte antes e agora estão. Por isso, toda vez que um vértice v é adicionado a X, fazemos:

1: **para** cada $vw \in E$ **faça**
2: remova w da heap
3: faça p ser o valor min{prioridade atual de $w, c(vw)$}
4: insira w na heap com prioridade p
5: **fim para**

Note ainda que para esses passos serem eficientemente implementados, precisamos da habilidade de remover um item do meio da heap. Para que w seja removido da heap de forma eficiente, basta manter um vetor indexado por vértices que indique sua posição na heap.

Finalmente: temos $n-1$ inserções na heap durante o pré-processamento, $n-1$ extrações de mínimo (uma por iteração) e cada aresta dispara no máximo um combo "remoção/inserção" na heap (quando um de seus extremos vai parar em X). Totalizando, temos tempo total $O(m \log n)$ de execução.

2 Problema dos Caminhos Mínimos de Única Fonte

Considere um grafo $G = (V, E)$ e uma função de custo c sobre as arestas de G. Definimos o custo $c(P)$ de um caminho $P = (v_1, v_2, \ldots, v_k)$ como sendo a soma dos custos de suas arestas, isto é, $c(P) = \sum_{i=1}^{k} c(v_i v_{i+1})$. Dizemos que P é um *uv-caminho* se $v_1 = u$ e $v_k = v$. Se existir um uv-caminho em G, dizemos que v é *alcançável* a partir de u. As arestas de um uv-caminho são também ditas alcançáveis a partir de u. Denotamos, finalmente, por $dist(u, v)$ o custo de um uv-caminho de menor custo. Definimos $dist(u, v) = \infty$ se v não for alcançável a partir de u. Por conveniência, dizemos que um caminho de menor custo é um *caminho mínimo*.

O problema definido a seguir é o problema do Caminho Mínimo de Única Fonte.

> ENTRADA: Grafo $G = (V, E)$ conexo, função c de custo sobre as arestas e vértice $s \in V$.
> OBJETIVO: Calcular $dist(s, v)$ para todo $v \in V$.

Vamos considerar que o custo de cada uma das arestas é não negativo.

Existe uma forma bem simples de resolver esse problema: troque cada aresta e por $c(e)$ arestas de custo 1 cada e use BFS começando em s. Note que isso de fato resolve o problema, pois BFS calcula caminhos mínimos quando o custo do caminho é a quantidade de arestas do mesmo. No entanto, essa solução certamente não é a melhor possível e nem sabemos dizer

quanto tempo ela pode levar no pior caso, pois não podemos prever o tamanho desse grafo modificado (uma aresta pode ter custo 10000, ou 1000000).

Veremos um algoritmo clássico que resolve esse problema, chamado algoritmo de Dijkstra. A ideia do algoritmo de Dijkstra é começar escolhendo o vértice s e repetidamente escolher um novo vértice w que seja adjacente a algum dos vértices x já escolhidos e cujo custo do sx-caminho mais o custo da aresta xw seja mínimo até que todos os vértices sejam escolhidos.

A seguir apresentamos um pseudocódigo do algoritmo de Dijkstra. Ele vai manter um conjunto X dos vértices já escolhidos, um vetor d indexado por vértices que vai armazenar em $d[v]$ o custo do sv-caminho calculado e um vetor $pred$ também indexado por vértices que vai armazenar em $pred[v]$ o vértice que levou v a ser escolhido.

1: **função** DIJKSTRA(G, c, s)
2: Seja $X = \{s\}$
3: Faça $d[s] = 0$ e $d[v] = \infty$ para todo $v \in V - \{s\}$
4: Faça $pred[s] = NULL$
5: **enquanto** possível **faça**
6: Escolha aresta vw com $v \in X$ e $w \notin X$ tal que $score(vw) = d[v] + c(vw)$ é mínimo
7: Se tal aresta não existir, pare
8: Seja v^*w^* uma tal aresta
9: $X \leftarrow X \cup \{w^*\}$
10: $d[w^*] = d[v^*] + c(v^*w^*)$
11: $pred[w^*] = v^*$
12: **fim enquanto**
13: **fim função**

No algoritmo, definimos um score para cada aresta vw, que é o custo do caminho já calculado até v mais o custo da aresta vw. Esse algoritmo é guloso porque a cada iteração ele escolhe uma aresta que tenha score mínimo dentre todas as arestas que conectam um vértice já escolhido a um vértice não escolhido. Veja a Figura 2 para um exemplo de seu funcionamento.

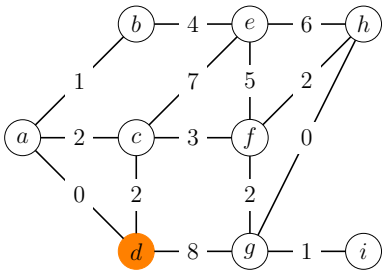
Podemos ver no exemplo simples da Figura 3 que Dijkstra de fato não funciona quando os custos das arestas são negativos.

Teorema 6. *O algoritmo de Dijkstra calcula os valores dos sv-caminhos mínimos para qualquer grafo $G = (V, E)$ com custos não negativos nas arestas dados por uma função c, onde $s, v \in V$.*

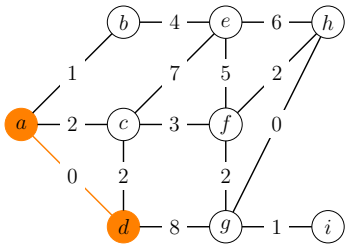
Demonstração. Vamos mostrar por indução no número de iterações que para todo vértice $v \in X$, $d[v] = dist(s, v)$.

Na primeira iteração, $X = \{s\}$ e de fato $d[s] = dist(s, s) = 0$.

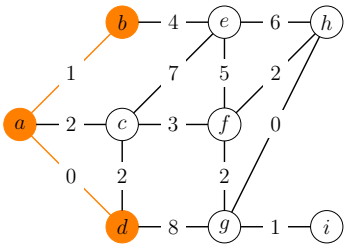
Suponha que a aresta v^*w^* é escolhida na iteração atual. Por hipótese de indução, $d[v^*] = dist(s, v^*)$. Nessa iteração iremos adicionar w^* a X e definir $d[w^*] = d[v^*] + c(v^*w^*) = dist(s, v^*) + c(v^*w^*)$.



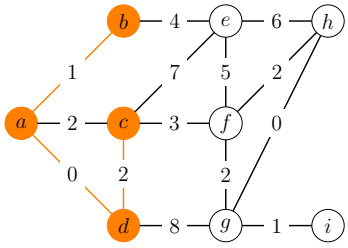
(a) Grafo G de entrada. O vértice d foi escolhido como inicial.



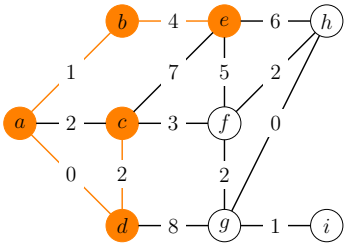
(b) Primeira iteração: $X = \{d\}$; aresta de menor score que liga um não escolhido a um escolhido = da.



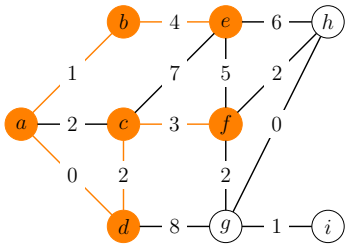
(c) Segunda iteração: $X = \{d, a\}$; aresta de menor score que liga um não escolhido a um escolhido = ab.



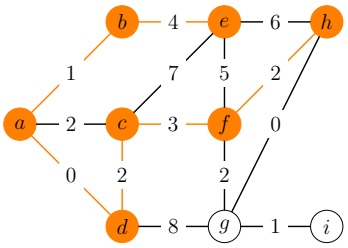
(d) Terceira iteração: $X = \{d, a, b\}$; aresta de menor score que liga um não escolhido a um escolhido = cd.



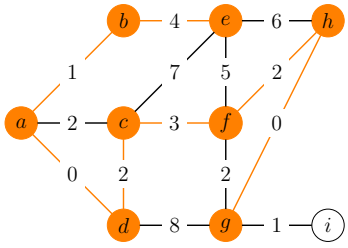
(e) Quarta iteração: $X = \{d, a, b, c\}$; aresta de menor score que liga um não escolhido a um escolhido = be.



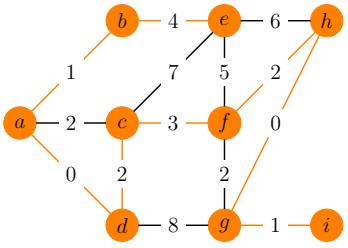
(f) Quinta iteração: $X = \{d, a, b, c, e\}$; aresta de menor score que liga um não escolhido a um escolhido = cf.



(g) Sexta iteração: $X = \{d, a, b, c, e, f\}$; aresta de menor score que liga um não escolhido a um escolhido = fh.



(h) Sétima iteração: $X = \{d, a, b, c, e, f, h\}$; aresta de menor custo que liga um não escolhido a um escolhido = hg.



(i) Oitava iteração: $X = \{d, a, b, c, e, f, h, g\}$; aresta de menor custo que liga um não escolhido a um escolhido = gi.

Figura 2: Exemplo de execução do Dijkstra.

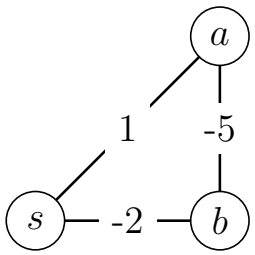


Figura 3: Grafo com pesos negativos nas arestas. Dijstra calcula $d[b] = -2$ quando na verdade $dist(s, b) = -4$.

Seja P qualquer outro sw^*-caminho que não seja o construído pelo algoritmo. Note que, como $s \in X$ e $w^* \notin X$, deve existir uma aresta yz de P tal que $y \in X$ e $z \notin X$. Seja yz a primeira aresta de P dessa forma. Note que: (i) o subcaminho de P entre z e w^* tem custo maior ou igual a zero, (ii) a aresta yz tem custo $c(yz) \geq 0$ e (iii) o subcaminho de P entre s e y tem custo pelo menos $dist(s, y)$. Então o custo de P é pelo menos $dist(s, y) + c(yz)$, ou seja, o score da aresta yz. Pelo critério guloso, Dijkstra escolheu v^*w^* por ela ser a de menor score. Então, em particular, $dist(s, v^*) + c(v^*w^*) \leq dist(s, y) + c(yz)$. Então o caminho escolhido por Dijkstra para w^* deve ser mínimo (pois não é pior que nenhum outro sw^*-caminho). □

Note agora que é possível implementar Dijkstra facilmente em tempo $O(mn)$, onde $m = |E|$ e $n = |V|$: o laço principal executa no máximo $n-1$ vezes (escolhendo um vértice para adicionar a X por iteração) e em cada iteração podemos percorrer todas as arestas para verificar as que tem um extremo em X e outro não, mantendo a de menor score. No entanto, note que o que o algoritmo faz repetidamente é procurar algo de valor mínimo. Assim, podemos usar a estrutura de dados heap para melhorar a complexidade de tempo, pois ela fornece esse tipo de operação em tempo $O(\log k)$ quando existem k elementos armazenados nela.

Veja uma definição/lembrete de heap na Definição 2.

Para o Dijkstra, vamos usar heap para manter os vértices de $V - X$. Para cada $v \in V - X$, a prioridade de v será dada por $\min_{w \in X}\{d[w] + c(wv)\}$ se existir $w \in X$ tal que $wv \in E$ ou será ∞, caso contrário. Assim, se a manutenção for feita corretamente, extrair o menor elemento da heap fornece exatamente o vértice w^* que deve ser adicionado a X.

Na inicialização temos $X = \{s\}$, então em tampo $O(m)$ calculamos as prioridades dos vértices em $V - X$ e com $O(n \log n)$ operações inserimos cada um na heap. Então ao todo temos tempo $O(m) + O(n \log n) = O(m \log n)$ (pois $m \geq n - 1$).

Para manter a heap corretamente, note que as únicas arestas que cruzam o corte $(X \cup \{w^*\}, V - (X \cup \{w^*\}))$ e não cruzavam o corte $(X, V - X)$ são as arestas que saem de w^*. Assim, quando extrairmos w^* da heap, fazemos as seguintes operações:

1: **para** cada $w^*x \in E$ **faça**
2: remova x da heap
3: recalcule p, a nova prioridade de x (scores das arestas que saem de x)
4: insira x na heap com prioridade p

5: **fim para**

Note ainda que para esses passos serem eficientemente implementados, precisamos da habilidade de remover um item do meio da heap. Para isso, basta manter um vetor indexado por vértices que indica em qual posição da heap um vetor está.

Então temos no máximo $n - 1$ operações de remoção de mínimo que levam tempo total $O(n \log n)$ e, para cada aresta do grafo, fazemos no máximo uma remoção e uma inserção na heap, quando seu extremo é adicionado a X, levando tempo total $O(m \log n)$.

Com o vetor *pred* computado, podemos facilmente encontrar um sv-caminho (os vértices que o constituem) que tem custo $dist(s, v)$:

1: **função** CAMINHO(*pred*, s, v)
2: **se** $s \neq v$ **então**
3: CAMINHO(*pred*, s, *pred*[v])
4: **fim se**
5: imprima v
6: **fim função**