Prova Didática

Grafos: Árvores Geradoras e Caminhos Mínimos, Análise de Complexidade

Gustavo E.A.P.A. Batista

25 de janeiro de 2005

Contextualização

- O aluno conhece as principais definições relacionadas a grafos, incluindo as definições de grafos orientados e não orientados, ponderados e conectados.
- O aluno conhece as principais formas de representação de grafos, tais como a matrizes e listas de adjacências.
- O aluno conhece o tipo abstrato de dados fila de prioridades e as implementações das operações de busca e remoção de elementos em O(log n).

Caminhos Mínimos

- Encontrar caminhos mínimos é um problema comum e importante no estudo de grafos.
- Uma possível aplicação seria encontrar rotas mínimas de vôo de uma companhia aérea.
- Inicialmente iremos considerar o problema de encontrar um caminho mínimo a partir de um único vértice.

Caminhos Mínimos de Origem Única

- Considere um grafo orientado ponderado $G = (V, E)$ em que cada aresta possui um rótulo não negativo associado que define o custo da aresta, e um dos vértices é especificado como *origem*.
- Nosso problema é determinar quais são os caminhos mais curtos do vértice origem para cada um dos demais vértices em V e os seus custos.
- O caminho *mais curto* ou *mínimo* é definido o caminho cuja soma dos custos dos vértices encontrados no caminho é mínima.

O Algoritmo de Dijkstra

- O algoritmo de Dijkstra mantém um conjunto S de vértices cujos pesos finais dos caminhos mais curtos desde a origem já foram determinados. Inicialmente S contém somente o vértice origem.
- O algoritmo de Dijkstra é um algoritmo “guloso”. A cada iteração, um vértice $w \in V - S$ cuja distancia ao vértice origem é tão pequena quanto possível é adicionado a S.
- Assumindo que todos os vértices possuem custos não negativos, sempre é possível encontrar um caminho mais curto do vértice origem a w que passa somente por vértices em S.

O Algoritmo de Dijkstra

- A cada iteração, um vetor D armazena o custo do caminho mais curto conhecido até o momento entre o vértice origem e os demais vértices do grafo. Para os vértices em S, D possui o caminho mais curto final.
- Quando todos os vértices estão em S, o algoritmo termina.

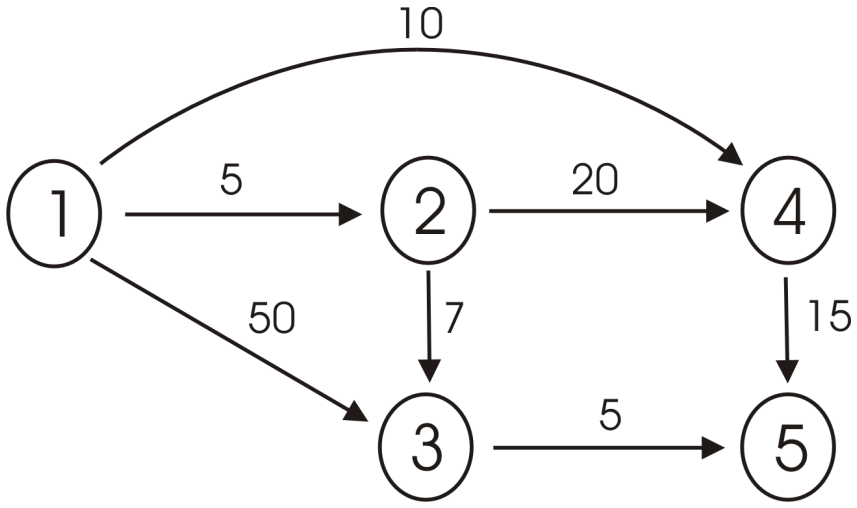
O Algoritmo de Dijkstra

Require: $G = (V, E)$, um grafo orientado ponderado
Require: C, uma matriz de custos associados aos vértices E
Ensure: D um vetor com as custos mínimos entre cada vértice em E e o vértice origem 1

1: $S \leftarrow \{1\}$
2: **for** $i \leftarrow 2$ to n **do**
3: $D[i] \leftarrow C[1, i]$
4: **end for**
5: **for** $i \leftarrow 2$ to n **do**
6: encontre um vértice $w \in V - S$ tal que $D[w]$ é mínimo
7: $S \leftarrow S \cup \{w\}$
8: **for all** $v \in V - S$ **do**
9: $D[v] \leftarrow$ min(D[v], D[w] + C[w,v])
10: **end for**
11: **end for**

O Algoritmo de Dijkstra

- Vejamos um exemplo



- Para simular a execução do algoritmo de Dijkstra, vamos anotar os valores das variáveis S, w e D.

O Algoritmo de Dijkstra

- Caso for necessário reconstruir o caminho mais curto entre o vértice origem e cada vértice, pode-se manter um vetor P de vértices, tal que $P[v]$ contém o vértice imediatamente anterior ao vértice v no caminho mais curto.
- Para isso, devemos realizar uma modificação no algoritmo anterior.

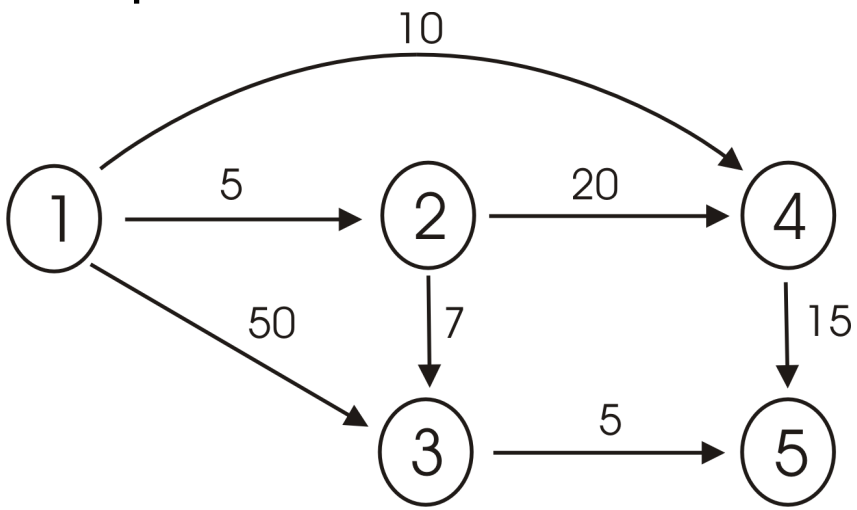
O Algoritmo de Dijkstra Modificado

Ensure: P, um vetor com os caminhos de custo mínimo entre a origem e os demais vértices

1: $S \leftarrow \{1\}$
2: **for** $i \leftarrow 2$ to n **do**
3: $D[i] \leftarrow C[1, i]$
4: $P[i] \leftarrow 1$
5: **end for**
6: **for** $i \leftarrow 1$ to $n - 1$ **do**
7: encontre um vértice $w \in V - S$ tal que $D[w]$ é mínimo
8: $S \leftarrow S \cup w$
9: **for all** $v \in V - S$ **do**
10: $D[v] \leftarrow$ min(D[v], D[w] + C[w,v])
11: **if** $D[w] + C[w, v] < D[v]$ **then**
12: $P[v] \leftarrow w$
13: **end if**
14: **end for**
15: **end for**

O Funcionamento do Algoritmo de Dijkstra

- Ao final da execução o vetor P possui o caminho para cada vértice que pode ser encontrado a partir do vértice origem.
- Para encontrar o caminho basta entre o vértice origem e o vértice v, basta iniciar em $P[v]$ e percorrer o vetor P em direção ao seu início, examinando qual é o predecessor de cada vértice.
- Para o grafo exemplo



o vetor P deve ter os valores $P[2] = 1$, $P[3] = 2$, $P[4] = 1$, $P[5] = 3$.

O Funcionamento do Algoritmo de Dijkstra

- Para entender o funcionamento do algoritmo de Dijkstra, é necessário verificar três fator importantes:
  1. Os vértices adicionados a S possuem o seu caminho mínimo definitivo e não precisam mais serem revistos;
  2. Não pode haver arestas com custo negativo, uma vez que esse fato faria com que fosse necessário rever os vértices em S
  3. D possui os caminhos mínimos conhecidos até o momento, as atualizações de D a cada novo vértice inserido em S se certifica disso.

Análise de Tempo de Execução do Algoritmo de Dijkstra

- Suponha que o algoritmo de Dijkstra opera em n vértices e e arestas.
- Se uma matriz de adjacências é utilizada para representar o grafo, o laço das linhas 8 e 9 requer $O(n)$, e esse laço é executado $n - 1$ vezes, fornecendo um tempo total de $O(n^2)$.

O Algoritmo de Dijkstra

Require: $G = (V, E)$, um grafo orientado ponderado
Require: C, uma matriz de custos associados aos vértices E
Ensure: D um vetor com as custos mínimos entre cada vértice em E e o vértice origem 1

1: $S \leftarrow \{1\}$
2: **for** $i \leftarrow 2$ to n **do**
3: $D[i] \leftarrow C[1, i]$
4: **end for**
5: **for** $i \leftarrow 2$ to n **do**
6: encontre um vértice $w \in V - S$ tal que $D[w]$ é mínimo
7: $S \leftarrow S \cup \{w\}$
8: **for all** $v \in V - S$ **do**
9: $D[v] \leftarrow$ min(D[v], D[w] + C[w,v])
10: **end for**
11: **end for**

Análise de Tempo de Execução do Algoritmo de Dijkstra

- Se uma lista de adjacências é utilizada, então pode-se encontrar diretamente os sucessores de w. Ainda, pode-se utilizar uma fila de prioridades para organizar os vértices em $V - S$.
- Uma fila de prioridades pode implementar uma operação de busca e remoção do novo vértice de menor custo w em $O(log\ n)$.
- O algoritmo de Dijkstra realiza em sua execução um total de e atualizações, cada uma a um custo de $O(log\ n)$. Portanto o tempo total dispendido é $O(e\ log\ n)$.
- Esse tempo de execução é consideravelmente melhor que $O(n^2)$ se e for bem menor que n^2, *i.e.* o grafo for esparso.

Caminhos Mínimos de Todos os Pares

- Suponha que um grafo orientado ponderado representa as possíveis rotas de uma companhia aérea conectando diversas cidades, nosso objetivo é construir uma tabela com os menores caminhos entre todas as cidades.
- Esse é um exemplo de problema que exige encontrar os caminhos mais curtos para todos os pares de vértices.
- Mais precisamente, dado um grafo $G = (V, E)$ no qual cada aresta (i, j) possui um custo não negativo $C[i, j]$, deseja-se encontrar para todos os pares ordenados (i, j) o caminho mínimo de i a j.

O algoritmo de Floyd

- Uma possível solução é utilizar o algoritmo de Dijkstra utilizando cada vértice como origem alternadamente.
- Uma solução mais direta é utilizar o algoritmo de Floyd. Assuma que os vértices em V estão numerados 1, 2, ..., n. O algoritmo de Floyd utiliza uma matriz A $n \times n$ para calcular e armazenar os tamanhos dos caminhos mais curtos.
- Inicialmente $A[i,j] = C[i,j]$ para todo $i \neq j$. Se não há uma aresta de i a j, é assumido que $A[i,j] = \infty$. Os elementos da diagonal são ajustados para 0.

O algoritmo de Floyd

- São realizadas n iterações sobre a matriz A. A cada iteração k, $A[i,j]$ armazena o menor caminho conhecido entre i e j que não passa por vértices com numeração acima de k.
- Na iteração k, a seguinte fórmula é utilizada para calcular A:

$$A_k[i,j] = min \begin{cases} A_{k-1}[i,j] \\ A_{k-1}[i,k] + A_{k-1}[k,j] \end{cases}$$

- Um detalhe é que $A_k[i,k] = A_{k-1}[i,k]$ e $A_k[k,j] = A_{k-1}[k,j]$. Portanto, é possível realizar todos os cálculos em um única cópia da matriz A.

O Algoritmo de Floyd

```
1:  for i ← 1 to n do
2:      for j ← 1 to n do
3:          A[i,j] ← C[i,j]
4:      end for
5:  end for
6:  for i ← 1 to n do
7:      A[i,i] ← 0
8:  end for
9:  for k ← 1 to n do
10:     for i ← 1 to n do
11:         for j ← 1 to n do
12:             if A[i,k] + A[k,j] < A[i,j] then
13:                 A[i,j] ← A[i,k] + A[k,j]
14:             end if
15:         end for
16:     end for
17: end for
```

Análise do Algoritmo de Floyd

- O algoritmo de Floyd é $O(n^3)$, uma vez que a sua implementação utiliza três laços **for** aninhados.
- Comparando o algoritmo de Floyd com o algoritmo de Dijkstra:
  - A versão que utiliza uma matriz de adjacência é $O(n^2)$, portanto para encontrar todos os caminhos mais curtos é necessário $O(n^3)$.
  - A versão que utiliza uma lista de adjacências requer $O(n \text{ e } log\ n)$ e portanto pode ser vantajosa para valores de e é muito menor que n^2. Dessa forma, o algoritmo de Dijkstra com lista de adjacências pode ser mais eficiente para grafos grandes e esparsos.
- É possível modificar o algoritmo de Floyd para informar os vértices contidos nos caminhos mais curtos, de forma similar ao feito no algoritmo de Dijkstra.

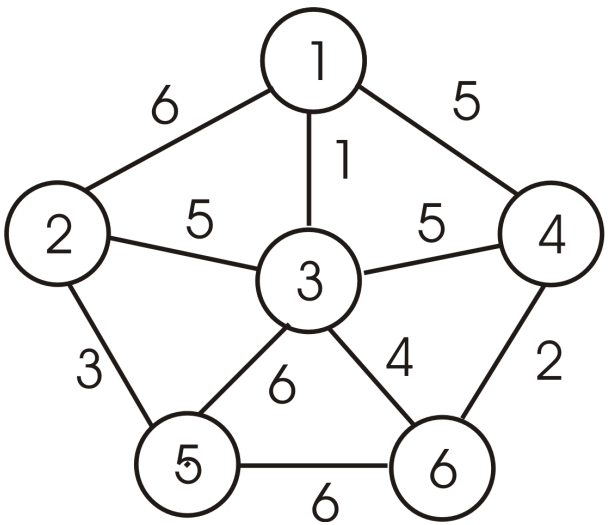
Árvores Geradoras

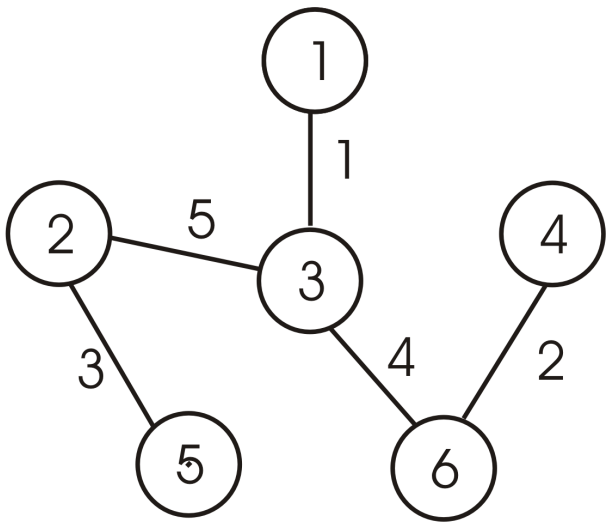
- Suponha que $G = (V, E)$ é um grafo conectado não orientado, no qual cada aresta $(i, j) \in E$ possui um custo $C[i, j]$.
- Uma árvore geradora de G é uma *árvore livre* (grafo conectado e acíclico) que conecta todos os vértices em V.
- O custo de uma árvore geradora é a soma dos custos das arestas na árvore.

Árvores Geradoras Mínimas

- Uma árvore geradora mínima é uma árvore geradora de um grafo G tal que seu custo é mínimo.
- Uma aplicação de árvores geradoras mínimas ocorre no projeto de redes de comunicação. Uma árvore geradora mínima representa uma rede de comunicação que conecta todas as cidades com custo mínimo.

Exemplo de Árvores Geradoras Mínimas





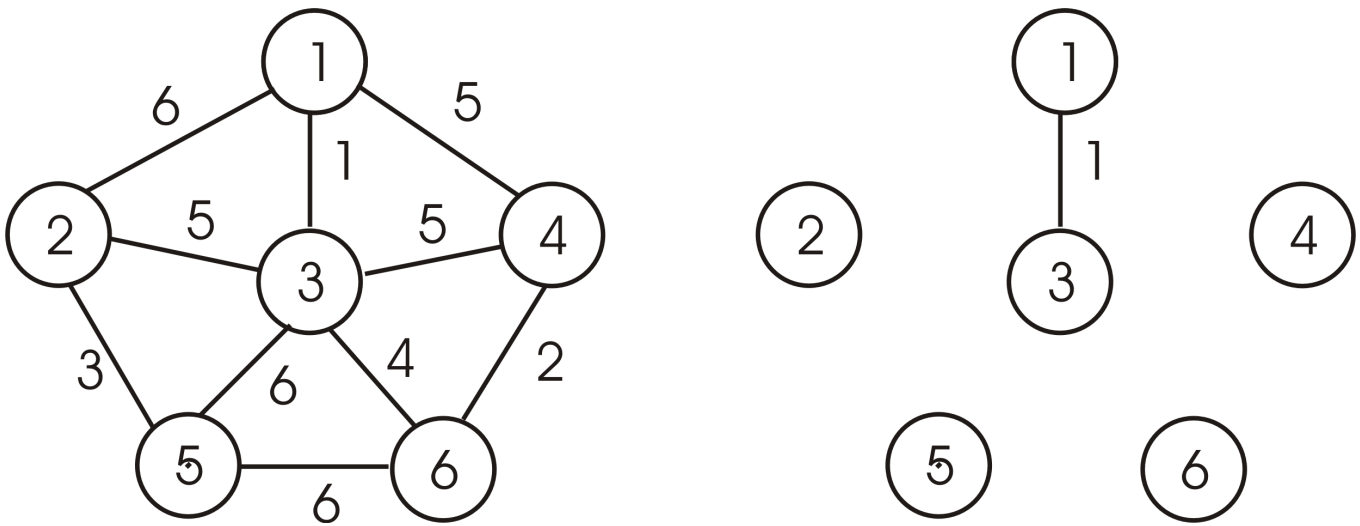
Propriedade das Árvores Geradoras Mínimas

- As árvores geradoras mínimas possuem uma propriedade importante. Essa propriedade é utilizada por diversos algoritmos para construir árvores geradoras mínimas.
- Seja $G = (V, E)$ um grafo conectado e ponderado. Seja U um subconjunto de vértices de V. Se (u, v) é uma aresta de custo mínimo tal que $u \in U$ e $v \in V - U$, então existe uma árvore geradora mínima que inclui (u, v) como uma aresta.

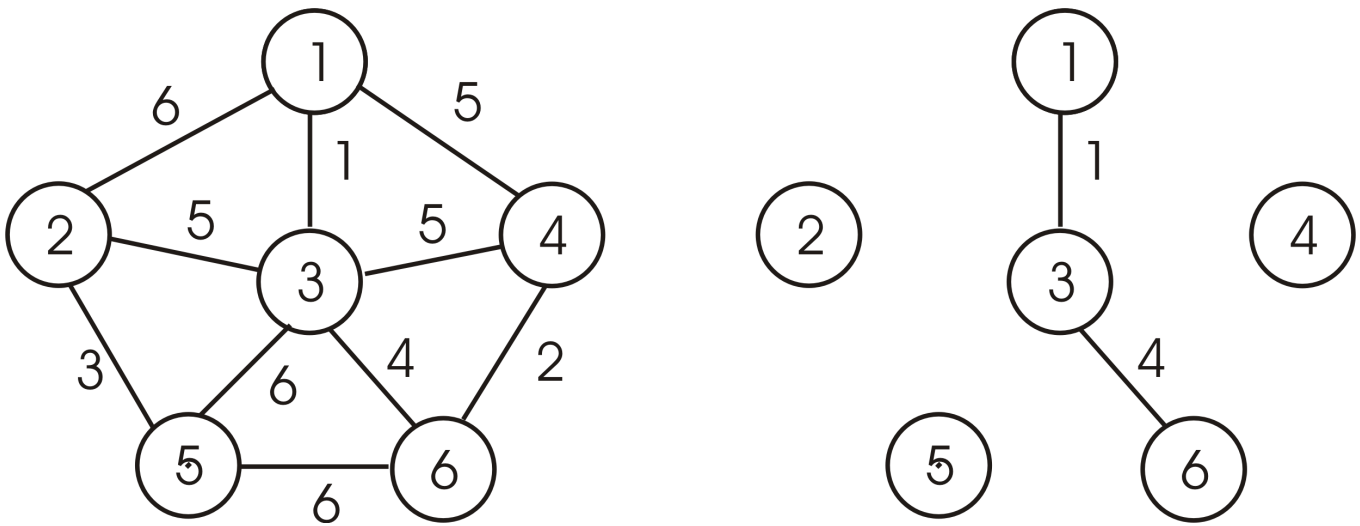
O Algoritmo de Prim

- Dado um grafo $G = (V, E)$, sendo $V = \{1, 2, \ldots, n\}$.
- O algoritmo de Prim começa com o conjunto $U = \{1\}$.
- A árvore geradora é formada um vértice por vez. A cada passo, o algoritmo encontra o vértice de menor custo (u, v) que conecta U e $V - U$. O vértice v é adicionado ao conjunto U.
- O processo é repetido até que $U = V$.

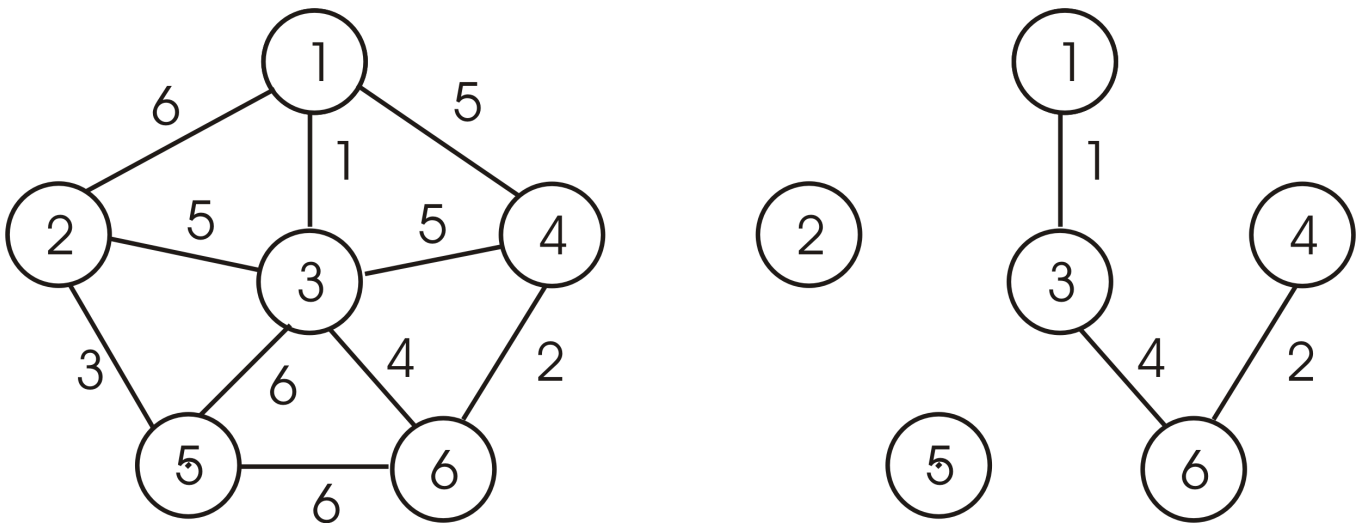
Exemplo Execução do Algoritmo de Prim



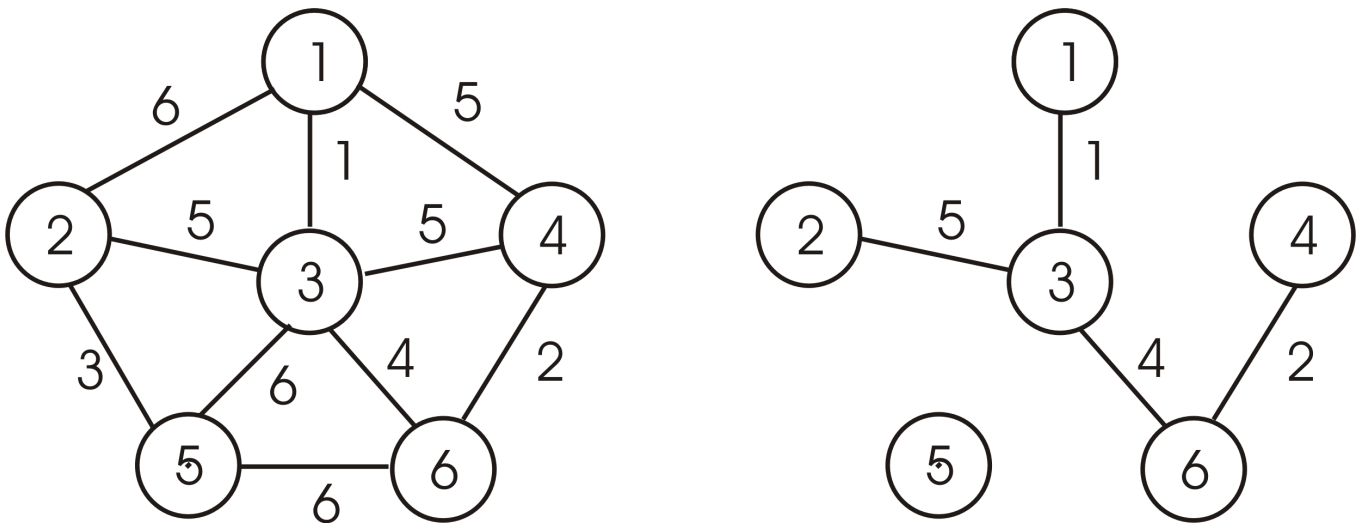
Exemplo Execução do Algoritmo de Prim



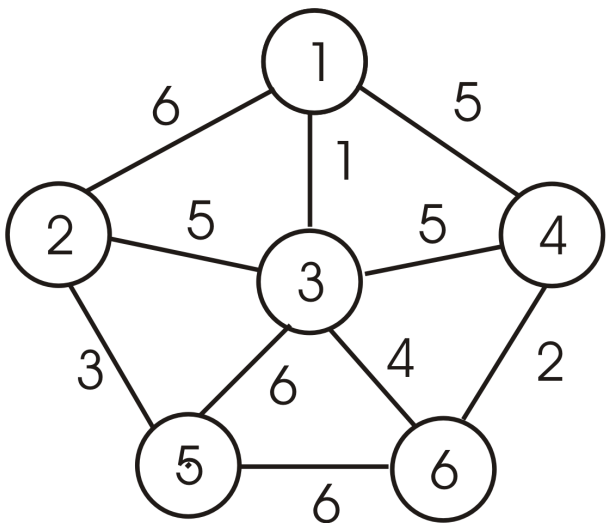
Exemplo Execução do Algoritmo de Prim

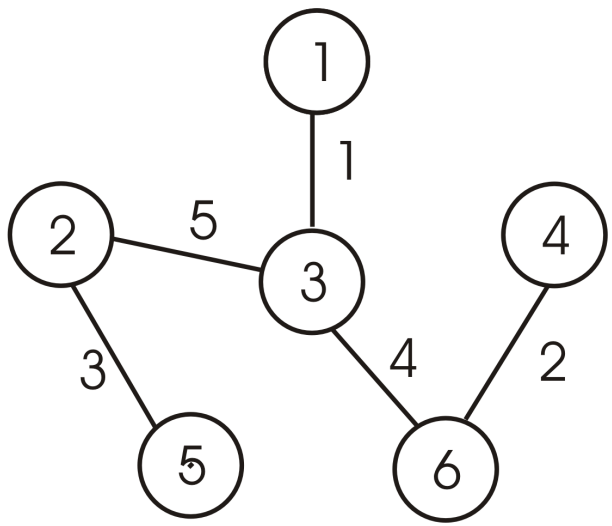


Exemplo Execução do Algoritmo de Prim



Exemplo Execução do Algoritmo de Prim





O Algoritmo de Prim

Require: $G = (V, E)$, um grafo ponderado
Ensure: T, um conjunto de arestas tal que T e V formam uma árvore geradora mínima de G

1: $T \leftarrow \varnothing$
2: $U \leftarrow \{1\}$
3: **while** $U \neq V$ **do**
4: seja (u, v) a aresta de custo mínimo tal que $u \in U$ e $v \in V - U$
5: $T \leftarrow T \cup \{(u, v)\}$
6: $U \leftarrow U \cup \{v\}$
7: **end while**

Análise do Algoritmo de Prim

- Para analisar a eficiência do algoritmo do algoritmo de Prim é necessário definir como será feita a seleção da aresta (u, v).
- É possível realizar uma implementação utilizando dois vetores. O primeiro $PROX[i]$ fornece o vértice em U que é atualmente o mais próximo ao vértice i de $V - U$. O segundo vetor $MC[i]$ fornece o custo da aresta $(i, PROX[i])$.
- A operação de encontrar (u, v) pode ser realizada em $O(n)$ percorrendo o vetor MC.
- É necessário também atualizar os vetores $PROX$ e MC. Essa operação também pode ser implementada em $O(n)$.
- Portanto, o algoritmo de Prim é $O(n^2)$.

O Algoritmo de Kruskal

- O algoritmo de Kruskal é também um algoritmo bastante popular para encontrar uma árvore geradora mínima.
- O desempenho do algoritmo de Kruskal é $O(e \log e)$, no qual e é o número de arestas.
- Se e é bem menor do que n^2, o algoritmo de Kruskal é assintóticamente superior ao algoritmo de Prim.
- Entretanto se e é próximo de n^2, o algoritmo de Prim pode ser preferível.

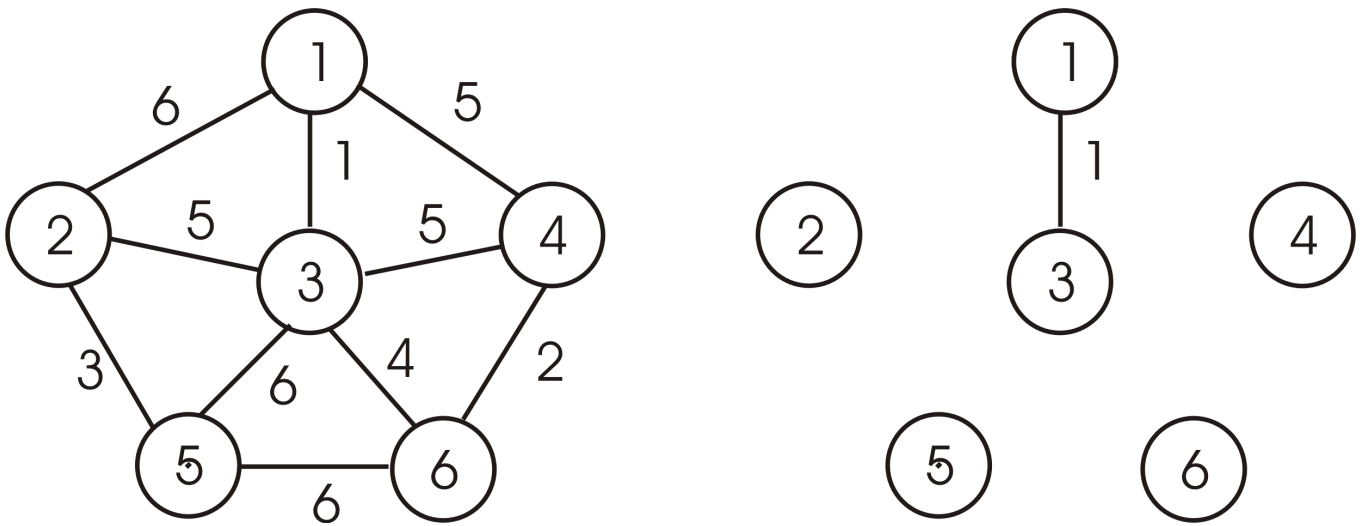
O Algoritmo de Kruskal

- Dado o grafo $G = (V, E)$ ponderado, sendo $V = \{1, 2, \ldots, n\}$.
- Inicia-se com um grafo $T = (V, \emptyset)$, que consiste nos n vértices de G, mas sem nenhuma aresta.
- Pode-se entender cada vértice como sendo um componente conectado a ele próprio.

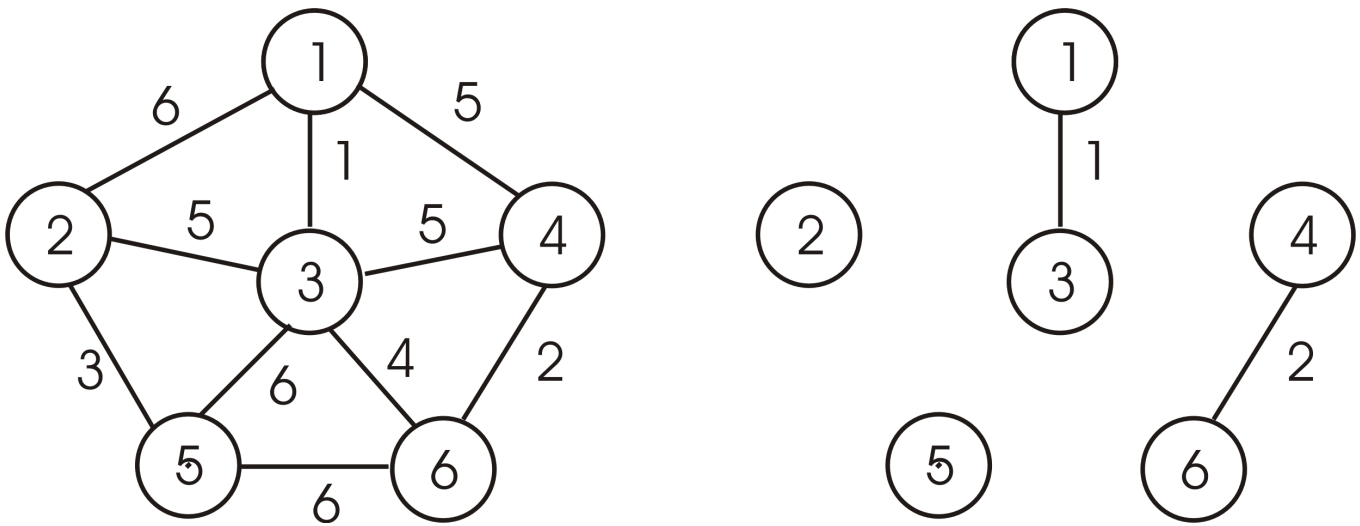
O Algoritmo de Kruskal

- Para construir progressivamente componentes maiores, as arestas em E são examinadas em ordem de custo crescente. Se uma aresta conecta dois vértices, em dois componentes conectados diferentes, então a aresta é adicionada a T.
- Se a aresta conecta dois vértices no mesmo componente, então a aresta é descartada pois causaria um ciclo.
- Quando todos os vértices estão em um único componente, T é uma árvore geradora mínima de G.

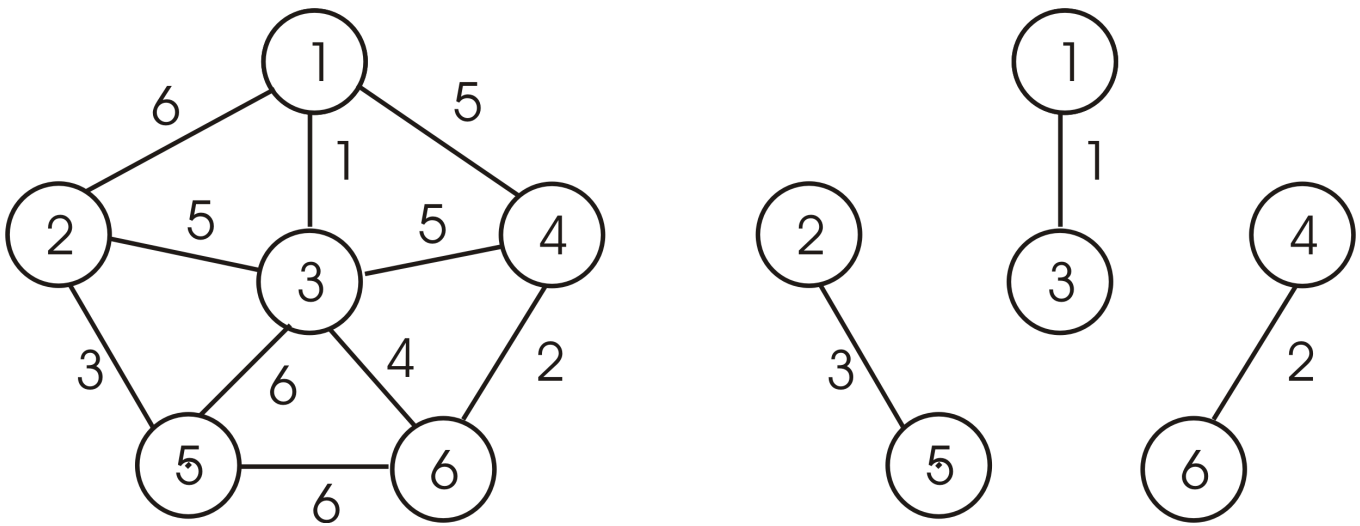
Exemplo Execução do Algoritmo de Kruskal



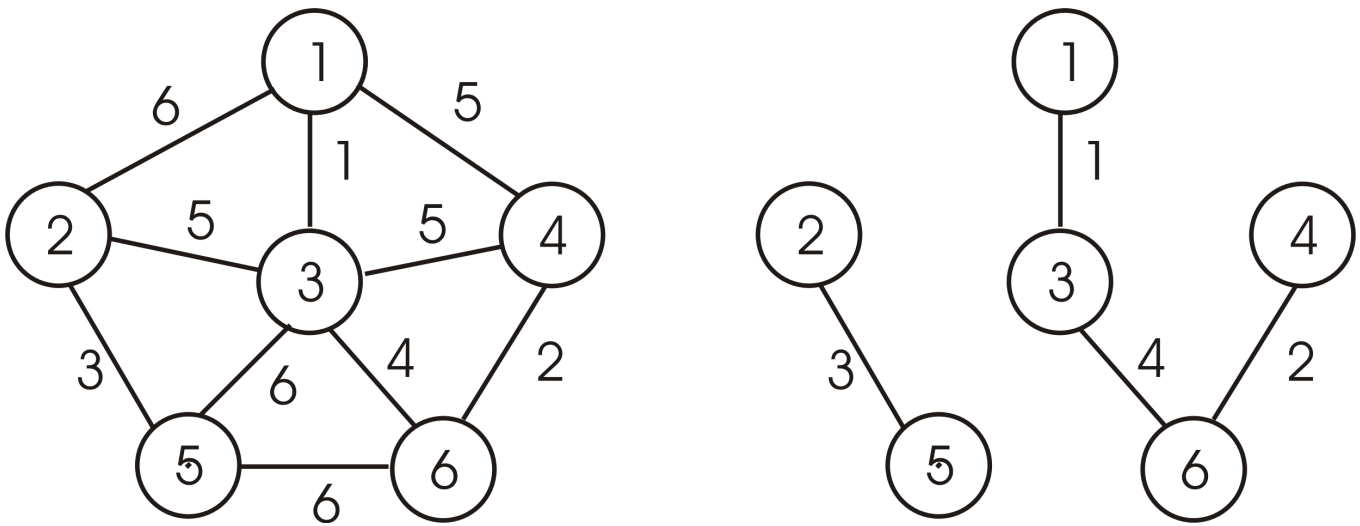
Exemplo Execução do Algoritmo de Kruskal



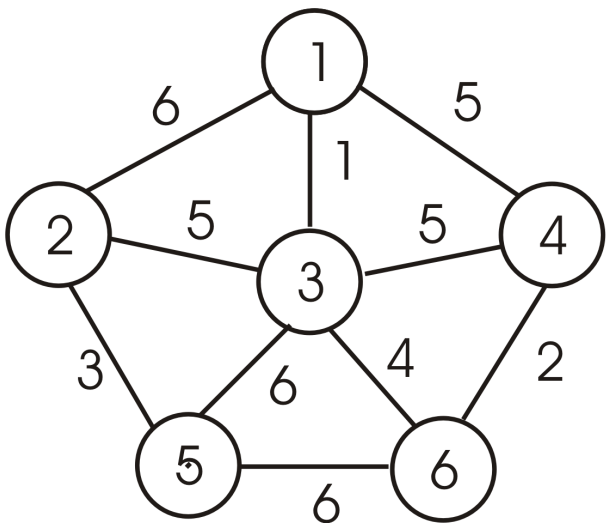
Exemplo Execução do Algoritmo de Kruskal

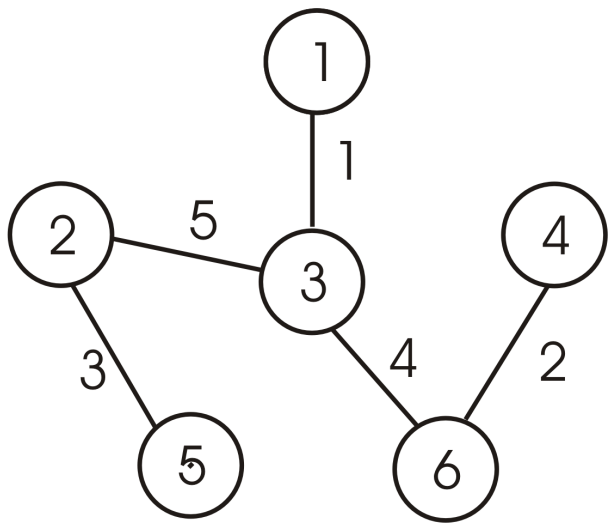


Exemplo Execução do Algoritmo de Kruskal



Exemplo Execução do Algoritmo de Kruskal





Referências

()

Aho, A. V., J. E. Hopcroft, and J. D. Ullman (1983).
Data Structures and Algorithms.
Addison-Wesley.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein (2002).
Algoritmos: Teoria e Prática (2 ed.).
Editora Campus.