

Algoritmo de Prim

Origem: Wikipédia, a enciclopédia livre.

Na ciência da computação o **algoritmo de Prim** é um algoritmo guloso (*greedy algorithm*) empregado para encontrar uma árvore geradora mínima (*minimal spanning tree*) num grafo conectado, valorado e não direcionado. Isso significa que o algoritmo encontra um subgrafo do grafo original no qual a soma total das arestas é minimizada e todos os vértices estão interligados. O algoritmo foi desenvolvido em 1930 pelo matemático Vojtěch Jarník e depois pelo cientista da computação Robert C. Prim em 1957 e redescoberto por Edsger Dijkstra em 1959.

Outros algoritmos conhecidos para encontrar árvores geradoras mínimas são o algoritmo de Kruskal e algoritmo de Boruvka. No entanto estes algoritmos podem ser empregados em grafos desconexos, enquanto o **algoritmo de Prim** precisa de um grafo conexo.

Índice

- 1 Descrição
 - 1.1 Algoritmo genérico
- 2 Pseudocódigo
- 3 Complexidade
- 4 Exemplo de execução
- 5 Implementações
 - 5.1 Implementação em Python
 - 5.2 Implementação em PHP
- 6 Referências
- 7 Bibliografia
- 8 Ligações Externas

Descrição

O **algoritmo de Prim** encontra uma árvore geradora mínima para um grafo desde que ele seja valorado e não direcionado. Por exemplo, se na *figura 1* os vértices deste grafo representassem cidades e as arestas fossem estradas de terra que interligassem estas cidades, como poderíamos determinar quais estradas asfaltar gastando a menor quantidade de asfalto possível para interligar todas as cidades. O **algoritmo de Prim** neste caso fornecerá uma resposta ótima para este problema que não necessariamente é única. A etapa *f*) da figura 1 demonstra como estas cidades devem ser conectadas com as arestas em negrito.

Algoritmo genérico

Um algoritmo genérico para o **algoritmo de Prim** é dado da seguinte forma:

*Escolha um vértice **S** para iniciar o subgrafo*
***enquanto** há vértices que não estão no subgrafo*
selecione uma aresta segura
insira a aresta segura e seu vértice no subgrafo

Pseudocódigo

```

prim(G) # G é grafo
# Escolhe qualquer vértice do grafo como vértice inicial/de par
s ← seleciona-um-elemento(vertices(G))

para todo v ∈ vertices(G)
    π[v] ← nulo
Q ← {(0, s)}
S ← ∅

enquanto Q ≠ ∅
    v ← extrair-mín(Q)
    S ← S ∪ {v}

    para cada u adjacente a v
        se u ∉ S e pesoDaAresta(π[u]→u) > pesoDaAresta(v→u)
            Q ← Q \ {(pesoDaAresta(π[u]→u), u)}
            Q ← Q ∪ {(pesoDaAresta(v→u), u)}
            π[u] ← v

retorna {(π[v], v) | v ∈ vertices(G) e π[v] ≠ nulo}

```

$\pi[v]$ indica o predecessor de v . Após o término do algoritmo, para cada v pertencente aos vértices de G , $\pi[v] \rightarrow v$ representa uma aresta selecionada para a árvore geradora mínima se $\pi[v] \neq \text{nulo}$.

O algoritmo retorna o conjunto dessas arestas, formado pelos pares $(\pi[v], v)$. Q é um conjunto de pares (peso, vértice). O método `extrair-mín(Q)` deve extrair o menor elemento de Q ; um par (a,b) é menor que um par (c,d) se $a < c$ ou se $a = c$ e $b < d$. S é um conjunto que armazena os vértices cujas adjacências já foram analisadas.

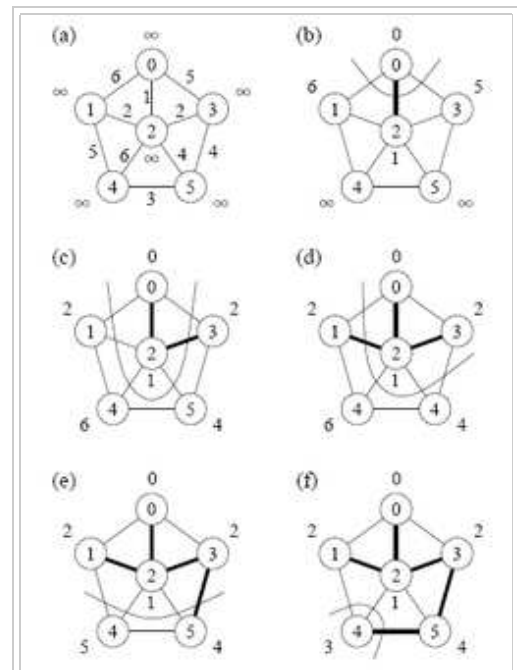


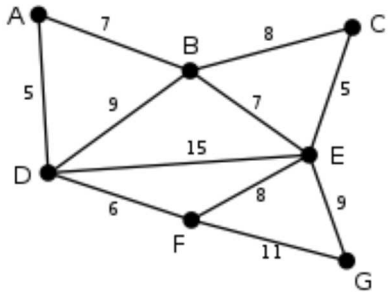
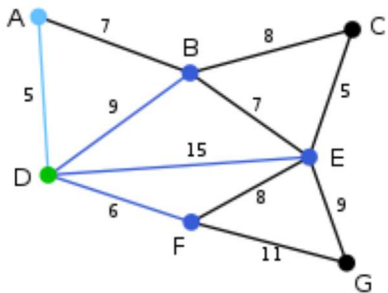
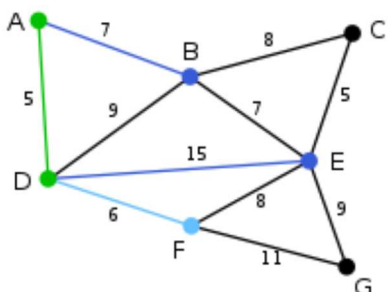
Figura 1: passo a passo da execução do **algoritmo de Prim** iniciado pelo vértice 0

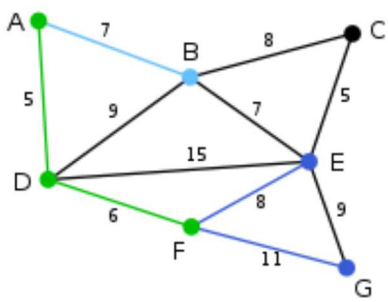
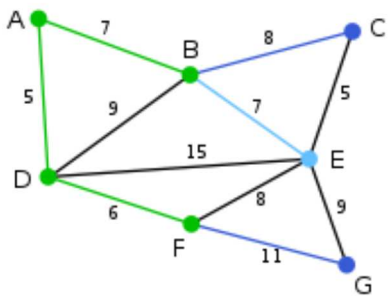
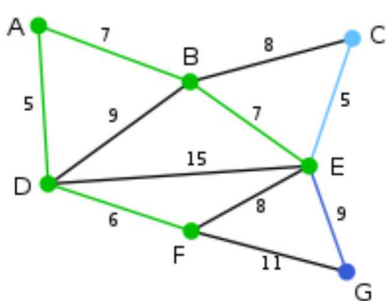
Complexidade

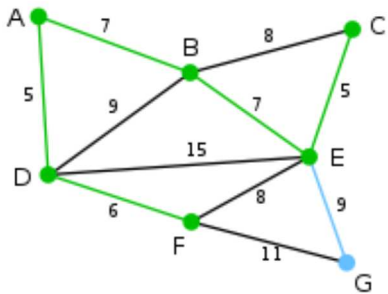
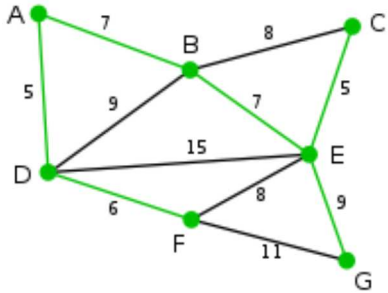
A complexidade do algoritmo de Prim pode mudar de acordo com a estrutura de dados utilizada para representar o grafo. As implementações mais comuns para um grafo são por listas de adjacência e por matrizes de adjacência e suas respectivas complexidades $O(|V|\log|A|)$ e $O(V^2)$ no pior caso.

Exemplo de execução

Repare neste exemplo de execução do algoritmo como as arestas são escolhidas para entrar no subgrafo. O conjunto $V \setminus U$ são os vértices que ainda não entraram no subgrafo, o conjunto U são os vértices que já estão no subgrafo, as **arestas possíveis** é uma lista de arestas que poderiam ser incluídas no subgrafo, pois conectam vértices contidos no subgrafo com os que ainda não estão e as **arestas incluídas** são aquelas que já estão no subgrafo. Dessa maneira e segundo o algoritmo genérico dado acima, para escolhermos uma aresta segura devemos observar o conjunto de arestas possíveis e selecionar aquelas que não formam ciclos com o subgrafo até então formado e cujo peso é o mínimo possível naquele momento. Se uma aresta apresentar todos estes quesitos podemos considerá-la uma aresta segura.

Imagem	Arestas incluídas no subgrafo	U	Arestas possíveis	$V \setminus U$	Descrição
	$\{\}$	$\{\}$		$\{A,B,C,D,E,F,G\}$	Este é o grafo original. Os números próximos das arestas significam o seu peso.
	$\{DA\}$	$\{D\}$	$\{D,A\} = 5V$ $\{D,B\}=9$ $\{D,E\}=15$ $\{D,F\}=6$	$\{A,B,C,E,F,G\}$	O vértice D foi escolhido como ponto inicial do algoritmo. Vértices A , B , E e F estão conectados com D através de uma única aresta. A é o vértice mais próximo de D e, portanto a aresta AD será escolhida para formar o subgrafo.
	$\{DA, DF\}$	$\{A,D\}$	$\{D,B\}=9$ $\{D,E\}=15$ $\{D,F\}=6V$ $\{A,B\}=7$	$\{B,C,E,F,G\}$	O próximo vértice escolhido é o mais próximo de D ou A . B está a uma distância 9 de D , E numa distância 15 e F numa distância 6. E A está a uma

					distância de 7 de B . Logo devemos escolher a aresta DF , pois é o menor peso.
	{DA, DF, AB}	{A,D,F}	$\{D,B\}=9$ $\{D,E\}=15$ $\{A,B\}=7V$ $\{F,E\}=8$ $\{F,G\}=11$	{B,C,E,G}	Agora devemos escolher o vértice mais próximo dos vértices A , D ou F . A aresta em questão é a aresta AB .
	{DA, DF, AB, BE}	{A,B,D,F}	$\{B,C\}=8$ $\{B,E\}=7V$ $\{D,B\}=9$ ciclo $\{D,E\}=15$ $\{F,E\}=8$ $\{F,G\}=11$	{C,E,G}	Agora podemos escolher entre os vértices C , E , e G . C está a uma distância de 8 de B , E está a uma distância 7 de B e G está a 11 de F . E é o mais próximo do subgrafo e, portanto escolhemos a aresta BE .
	{DA, DF, AB, BE, EC}	{A,B,D,E,F}	$\{B,C\}=8$ $\{D,B\}=9$ ciclo $\{D,E\}=15$ ciclo $\{E,C\}=5V$ $\{E,G\}=9$ $\{F,E\}=8$ ciclo $\{F,G\}=11$	{C,G}	Restam somente os vértices C e G . C está a uma distância 5 de E e de G a E 9. C é escolhido, então a aresta EC entra no

					subgrafo construído.
	{DA, DF, AB, BE, EC, EG}	{A,B,C,D,E,F}	{B,C}=8ciclo {D,B}=9ciclo {D,E}=15ciclo {E,G}=9V {F,E}=8ciclo {F,G}=11	{G}	Agora só resta o vértice G . Ele está a uma distância de 11 de F , e 9 de E . E é o mais próximo, então G entra no subgrafo conectado pela aresta EG .
	{DA, DF, AB, BE, EC, EG}	{A,B,C,D,E,F,G}	{B,C}=8 ciclo {D,B}=9 ciclo {D,E}=15 ciclo {F,E}=8 ciclo {F,G}=11 ciclo	{ }	Aqui está o fim do algoritmo e o subgrafo formado pelas arestas em verde representam a árvore geradora mínima. Nesse caso esta árvore apresenta a soma de todas as suas arestas o número 39.

Implementações

Implementação em Python

A implementação a seguir usa uma lista de adjacência para representar o grafo. A complexidade de tempo é $O(|V| + |A|\log|V|)$. Uma função adicional, `primDesconexo`, resolve o problema para grafos desconexos, sem alterar a complexidade de tempo do algoritmo.

```
# Implementacao do algoritmo de Prim  $O(E \log V)$  em Python
# Note que a unica funcao que representa a implementacao do algoritmo eh a funcao prim(graph,Vi=0,edge=[],vj
# A funcao add_edge eh apenas auxiliar, e a funcao primDesconexo(graph) eh um adicional, e nao costuma seque
# implementada para o algoritmo de Prim (pois no caso de um grafo ser desconexo, Kruskal eh a solucao ideal,
```

```

from heapq import heappop, heappush

MAXV = 1000 # numero de vertices no grafo
graph = [[] for x in xrange(MAXV)]
def add_edge(v, u, w):
    graph[v].append((u,w))
    graph[u].append((v,w)) # considera que o grafo eh nao direcionado

# Se o grafo for totalmente conectado, Vi pode receber qualquer vertice sem diferenca no peso total da arvore
# Se o grafo for desconexo, apenas a parte conectada a Vi tera sua arvore geradora minima calculada
# O retorno eh uma lista de tuplas edge[v]=(w,u), que representa, para cada v, a aresta u->v com peso w, usa
# conectar a sub-arvore de v a sub-arvore de u na arvore geradora minima
def prim(graph, Vi=0, edge=[], vis=[]):
    # edge[v] = (pesoDaAresta(u->v), u)
    # Se edge[] ou vis[] nao tiverem sido gerados ainda, geramos. Geralmente esta condicao nao existe, e am
    # sao geradas dentro do proprio prim; porem, para manter o primDesconexo em O(V + E log V), permitimos q
    # passadas pelos parametros da funcao.
    if edge == []:
        edge = [(-1,-1)] * len(graph)
    if vis == []:
        vis = [False] * len(graph)

    edge[Vi] = (0,-1)
    heap = [(0,Vi)]

    while True:
        v = -1
        while len(heap) > 0 and (v < 0 or vis[v]):
            v = heappop(heap)[1]

        if v < 0 or edge[v][0] < 0:
            break
        vis[v] = True

        for (u, w) in graph[v]:
            if edge[u][0] < 0 or edge[u][0] > w:
                edge[u] = (w, v)
                heappush(heap, (edge[u][0],u))

    return edge

# Se o grafo for desconexo, pode-se usar:
def primDesconexo(graph):
    edge = [(-1,-1)] * len(graph)
    vis = [False] * len(graph)
    for i in xrange(len(graph)):
        if edge[i][0] == -1:
            prim(graph, i, edge, vis)
    return edge

```

Implementação em PHP

```

$origem = array( 1 => 1,1,2,2,2,3,4,4,5);
$destino = array( 1 => 2,3,3,4,5,5,6,5,6);
$custo = array( 1 => 1,3,1,2,3,2,3,-3,2);
$nos = 6;
$narcos = 9;

// Define o infinito como sendo a soma de todos os custos
$infinito = array_sum($custo);

// Imprimindo origem destino e custo
echo utf8_decode("Grafo:<br>");

for($i =1 ; $i <= count($origem) ; $i++) {
    echo utf8_decode("$origem[$i] $destino[$i] $custo[$i]<br>");
}

// ----- Passo inicial

// Seta os valores de T
for($i =1 ; $i <= 6 ; $i++) {
    if($i == 1) {
        $t[$i] = $i;
    }
}

```

```

    } else {
        $t[$i] = "nulo";
    }
}

// Seta os valores de V
for($i =1 ; $i <= 6 ; $i++) {
    if($i == 1) {
        $v[$i] = "nulo";
    } else {
        $v[$i] = $i;
    }
}

echo utf8_decode("Início");
echo utf8_decode("<br> T: ");
print_r($t);
echo utf8_decode("<br> V: ");
print_r($v);
echo utf8_decode("<br>");

// ----- Fim do passo inicial
$total_nos = count($origem);
for($x =1 ; $x <= ($nos-1) ; $x++) {
    // Verifica origem -> destino
    $minimo1 = $infinito;
    for($i =1 ; $i <= $narcos ; $i++) {
        for($j =1 ; $j <= $nos ; $j++) {
            if($origem[$i] == $t[$j]) {
                for($k =1 ; $k <= $nos ; $k++) {
                    if($destino[$i] == $v[$k]) {
                        if($custo[$i] < $minimo1) {
                            $minimo1 = $custo[$i];
                            $aux1 = $i;
                        }
                    }
                }
            }
        }
    }

    // Verifica destino -> origem
    $minimo2 = $infinito;
    for($i =1 ; $i <= $narcos ; $i++) {
        for($j =1 ; $j <= $nos ; $j++) {
            if($destino[$i] == $t[$j]) {
                for($k =1 ; $k <= $nos ; $k++) {
                    if($origem[$i] == $v[$k]) {
                        if($custo[$i] < $minimo2) {
                            $minimo2 = $custo[$i];
                            $aux2 = $i;
                        }
                    }
                }
            }
        }
    }

    if($minimo2 < $minimo1) {
        $cont = 1;
        $minimo = $minimo1;
        $aux = $aux1;
        echo utf8_decode("<br> Aresta ($origem[$aux],$destino[$aux]) escolhida de custo $custo[$aux]");
    } else {
        $minimo = $minimo2;
        $aux = $aux2;
        echo utf8_decode("<br> Aresta ($destino[$aux],$origem[$aux]) escolhida de custo $custo[$aux]");
        $cont = 2;
    }

    if($cont == 1) {
        $t[$destino[$aux]] = $destino[$aux];
        $v[$destino[$aux]] = "nulo";
    } else {
        $t[$origem[$aux]] = $origem[$aux];
        $v[$origem[$aux]] = "nulo";
    }

    echo utf8_decode("<br> ".$x."ª iteração");
    echo utf8_decode("<br> T: ");
    print_r($t);
    echo utf8_decode("<br> V: ");

```

```
print_r($v);}
```

Referências

Bibliografia

- CORMEN, Thomas; Stein, Clifford. *Introduction to Algorithms* (em inglês). 2 ed. [S.l.]: MIT Press and McGraw-Hill, 2001. Capítulo: 23, ISBN 0-262-03293-7

Ligações Externas

- Algoritmo de Prim (<http://www.mincel.com/java/prim.html>)
- Exemplo animado de um algoritmo de Prim (<http://students.ceid.upatras.gr/~papagel/project/prim.htm>)
- Demonstração em Python de uma árvore mínima (<http://people.csail.mit.edu/rivest/programs.html>)
- Implementação em Java do algoritmo de Prim (<http://code.google.com/p/annas/>)
- Implementação em C# do algoritmo de Prim (<http://code.google.com/p/ngenerics/>)

Obtida de "http://pt.wikipedia.org/w/index.php?title=Algoritmo_de_Prim&oldid=36757162"

Categoria: Algoritmos de grafos

-
- Esta página foi modificada pela última vez à(s) 03h46min de 26 de agosto de 2013.
 - Este texto é disponibilizado nos termos da licença Atribuição-Partilha nos Mesmos Termos 3.0 não Adaptada (CC BY-SA 3.0); pode estar sujeito a condições adicionais. Consulte as condições de uso para mais detalhes.