G3. Algoritmos sobre Grafos Pesados: Estratégia Greedy

[Codeboard de apoio a este módulo: https://codeboard.io/projects/10154]

Árvores Geradoras Mínimas / *Minimum Spanning Trees* [MST]

Seja $G = (V, E)$ um grafo *não-orientado, ligado* (i.e. todos os vértices são alcançáveis a partir de qualquer outro vértice, não havendo componentes separados), e com pesos.

Uma **árvore geradora** de G é um sub-grafo (V, T) acíclico e ligado de G.

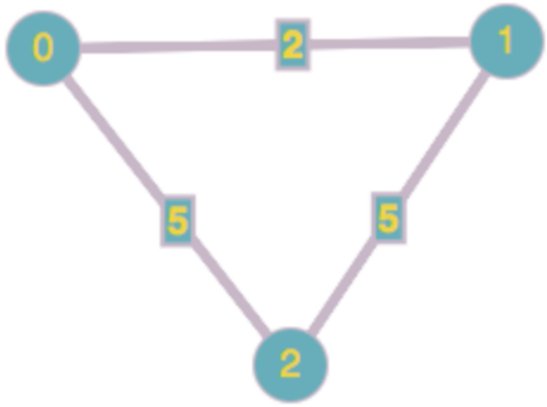
Note-se que, sendo sub-grafo acíclico de um grafo não-orientado, (V, T) é uma *árvore* que contém todos os vértices de G.

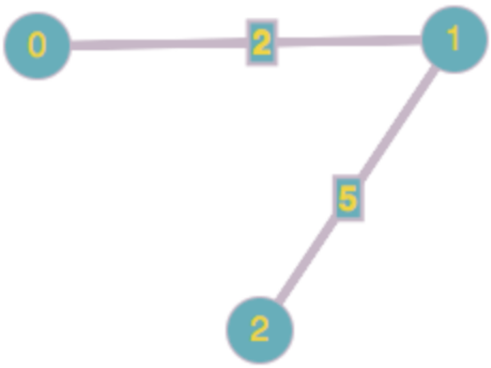
As **árvores geradoras mínimas** de G são aquelas para as quais o peso total $w(T) = \sum_{(u,v)\in T} w(u, v)$ é mínimo.

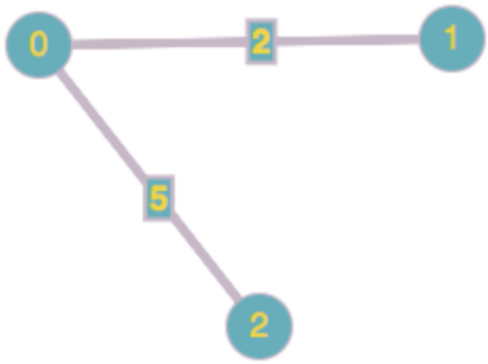
O problema da determinação de uma MST é pois um **problema de optimização**: trata-se de determinar um dos "melhores" objectos que satisfazem um conjunto de restrições (neste caso dadas pela definição de árvore geradora).

Exemplos

O grafo seguinte tem 3 árvores geradoras, das quais duas são mínimas, com peso 7.







> *No contexto de um grafo correspondente a uma rede de estradas, uma árvore geradora será uma sub-rede das estradas que permite deslocações entre todas as localidades sem redundância, i.e. existirá nesta árvore um único caminho entre cada par de localidades.*
> *Uma árvore geradora mínima será então uma sub-rede que permitirá ligar todas as localidades entre si, com comprimento total mínimo.*

Um outro exemplo será a ligação eléctrica de um número de pinos num circuito integrado. Cada fio liga um par de pinos; pretende-se minimizar a quantidade total de cobre utilizado nas ligações. Para isso constrói-se um grafo contendo as ligações possíveis entre pinos, com pesos correspondentes à quantidade de cobre utilizada em cada ligação. O peso de uma MST corresponde à quantidade mínima de cobre necessária para o conjunto de ligações.

Trata-se de um problema que ocorre em muitos contextos, nomeadamente como sub-problema de outros.

Sub-estrutura Óptima

Diz-se que um problema de optimização possui *sub-estrutura óptima* se uma solução óptima para o problema contém *soluções óptimas para sub-problemas* do problema original.

O problema de determinação de árvores geradoras mínimas possui sub-estrutura óptima, uma vez que cada sub-árvore de uma MST de um grafo G é seguramente uma MST de um sub-grafo de G.

Um problema com estas características pode ser reformulado da seguinte forma:

> *dada uma solução parcial (solução de um sub-problema), pretende-se estendê-la até obter uma solução total (solução do problema original).*

No caso do problema de determinação de MSTs:

> *dada uma MST de um sub-grafo de G, pretende-se estendê-la para obter uma MST de G*

A estratégia greedy para resolução de problemas com sub-estrutura óptima

A estratégia de resolução greedy ("gananciosa") é uma estratégia algorítmica para problemas com sub-estrutura óptima que se caracteriza da seguinte forma:

- É um método **top-down** (tal como a estratégia de divisão e conquista)
- Em cada passo, o algoritmo estende a solução actual efectuando uma escolha local
- Desta forma, o problema é reduzido a um problema mais pequeno, uma vez que a solução local cresce em cada passo, aproximando-se de uma solução para o problema inicial

Enquanto a estratégia de divisão e conquista gera problemas mais pequenos para resolver, e processa depois as respectivas soluções, a estratégia *greedy* transforma o problema num mais pequeno, estendendo localmente uma solução parcial.

A *prova de correcção* de um algoritmo baseado nesta estratégia terá que mostrar que o passo básico é correcto, ou seja:

> *se se parte de uma solução óptima de um sub-problema do problema original, então depois de se estender localmente esta solução teremos ainda uma solução óptima de um sub-problema do problema original.*

Algoritmo de Prim para Construção de Árvores Geradoras Mínimas

Este algoritmo pode ser visto como um algoritmo de travessia, usando uma estratégia alternativa (nem em profundidade nem em largura). Trata-se aqui de uma estratégia **greedy,** que faz uma escolha local guiada pelos pesos das arestas.

Em cada passo do algoritmo partir-se-á de uma MST (V', T') de um sub-grafo de G, e acrescentar-se-á um vértice e uma aresta a esta árvore, obtendo-se uma árvore (V'', T'') que será ainda uma MST de um subgrafo (maior) de G.

Estrutura geral do algoritmo

Considera-se em cada instante da execução o conjunto de vértices de G dividido em 3 conjuntos disjuntos:

1. os vértices da **árvore** de travessia construída até ao momento;
2. os vértices na **orla** (adjacentes aos da árvore);
3. os restantes vértices.

Em cada passo selecciona-se uma aresta (*com origem na **árvore** e destino na **orla***) para acrescentar à árvore. O vértice destino dessa aresta é também acrescentado. É esta a escolha local característica da estratégia *greedy.*

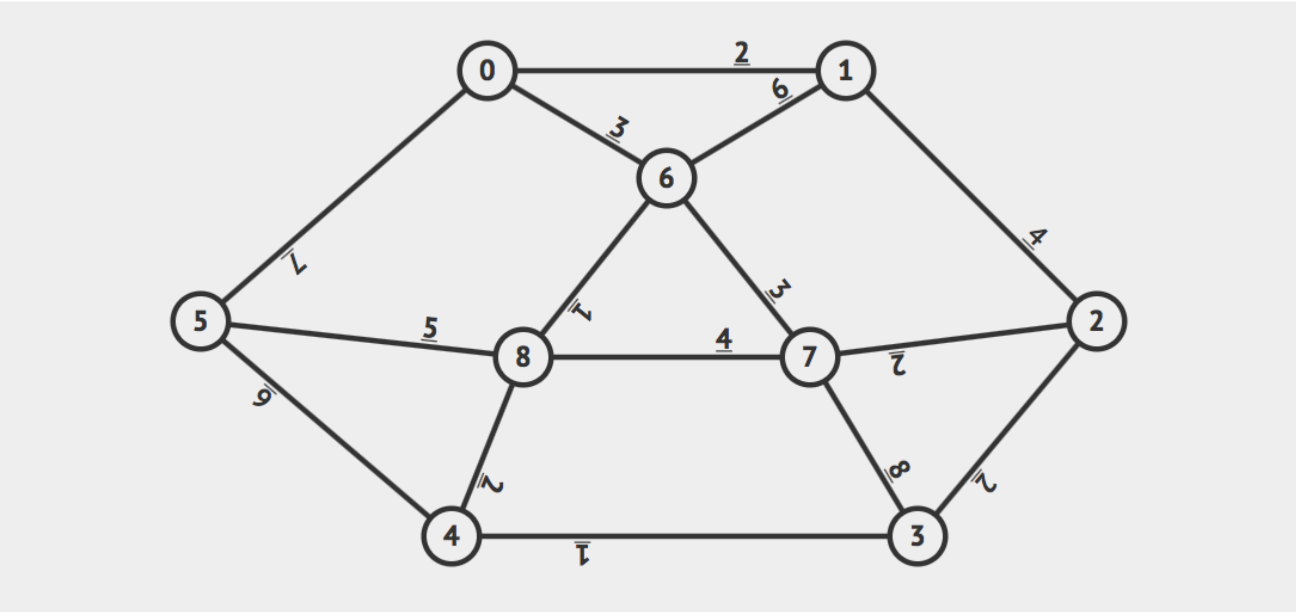
O algoritmo de Prim seleciona sempre o arco com **menor peso** nestas condições.

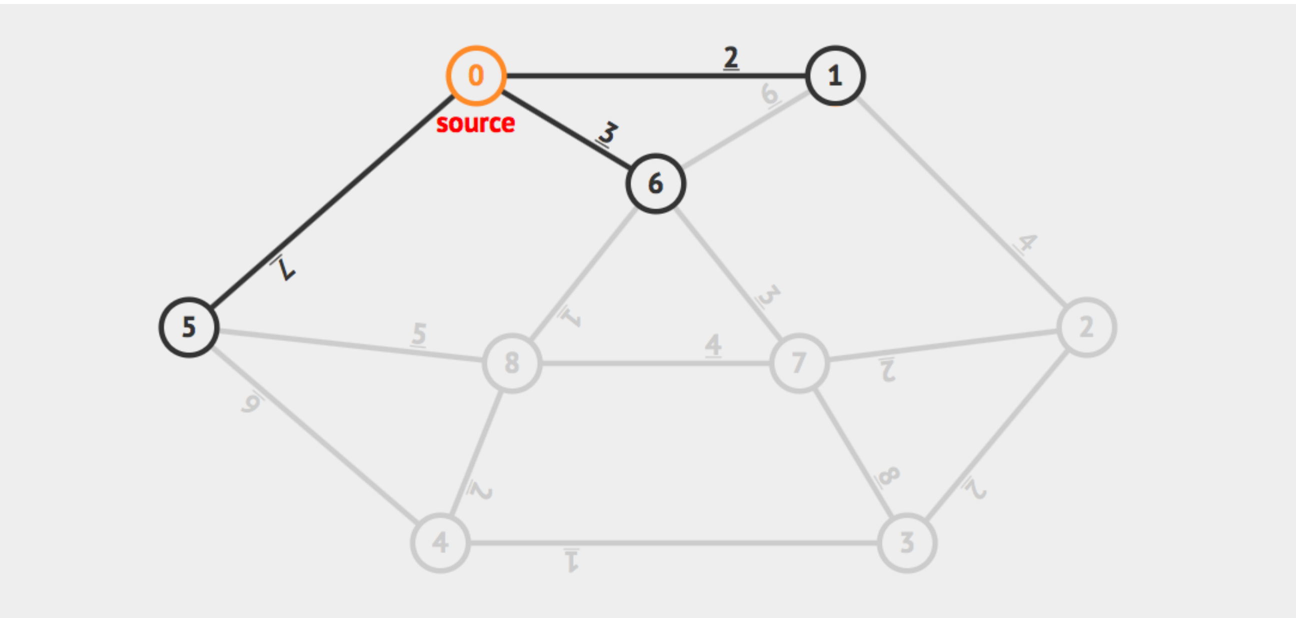
```
def mst_Prim(g):
  seleccionar vértice arbitrário x para inicio da árvore
  while (árvore não contém todos os vértices):
    actualizar orla em função do novo vértice x
    seleccionar aresta (u,v) de peso mínimo
```

```
                      entre vértices da árvore e da orla
    x = v
    acrescentar vértice x e aresta (u,x) à árvore
```

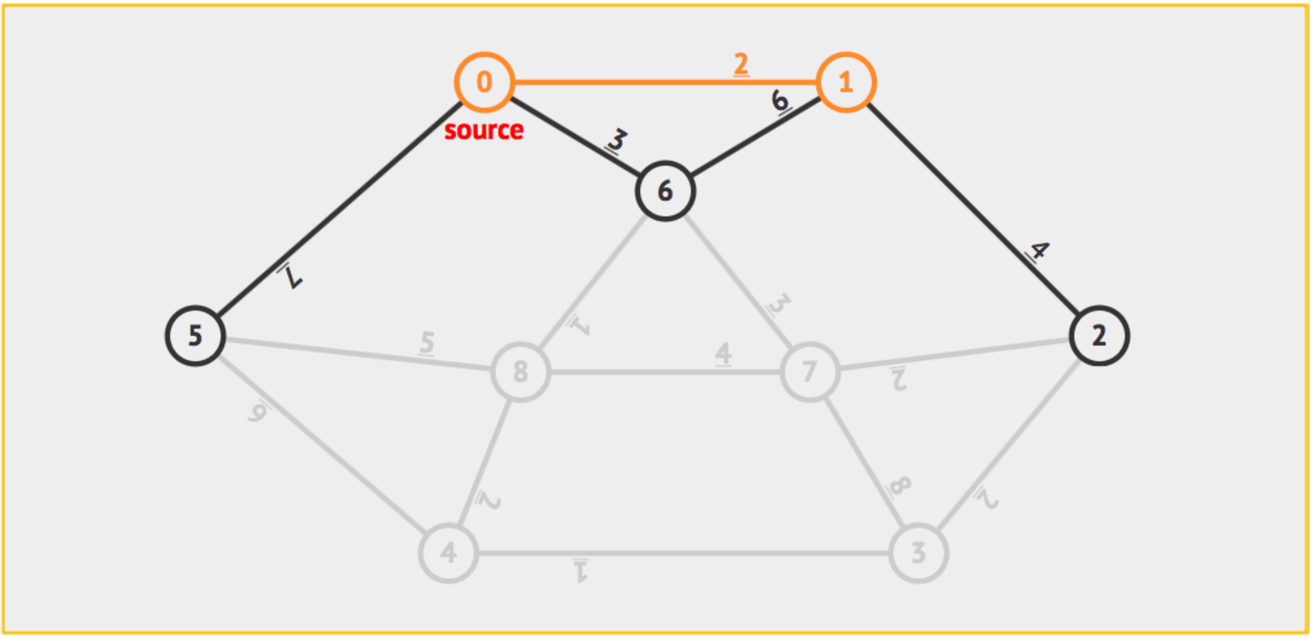
O exemplo seguinte ilustra a necessidade de se associar a cada vértice da orla o seu **arco candidato.** A selecção da aresta a acrescentar à árvore é feita escolhendo o **arco candidato de menor peso.**

Exemplo
[Animação realizada em https://visualgo.net/en/mst]



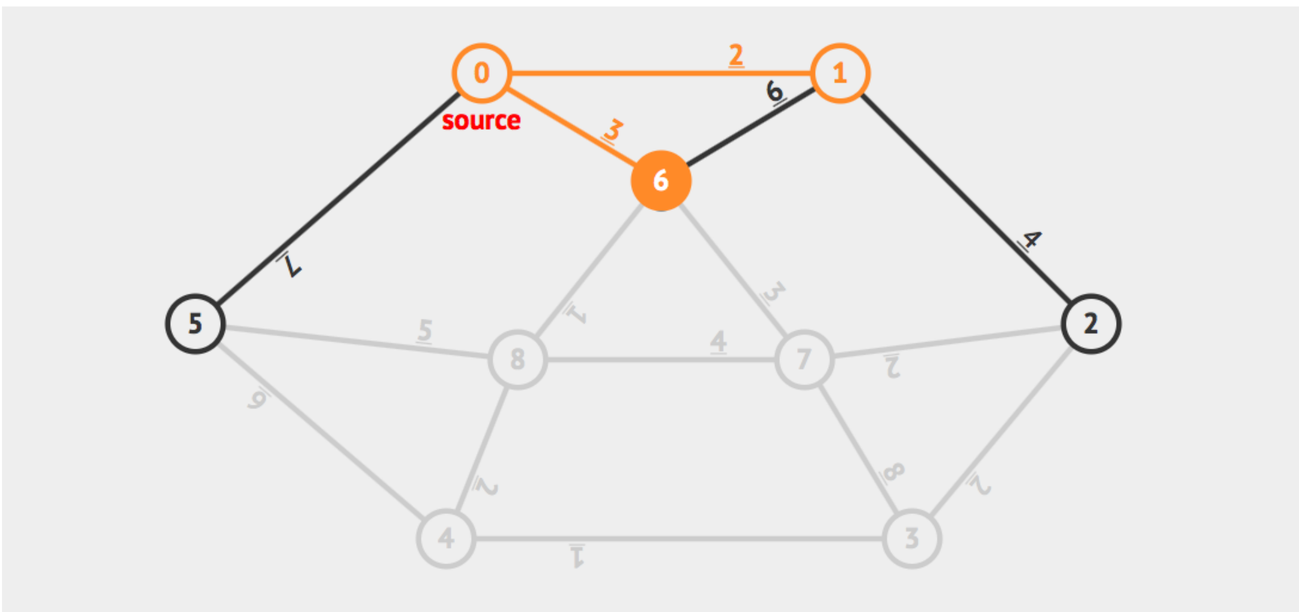


A orla contém neste momento os 3 vértices adjacentes à origem da árvore, 0. Será seleccionado o arco candidato (0, 1), uma vez que tem o menor peso:

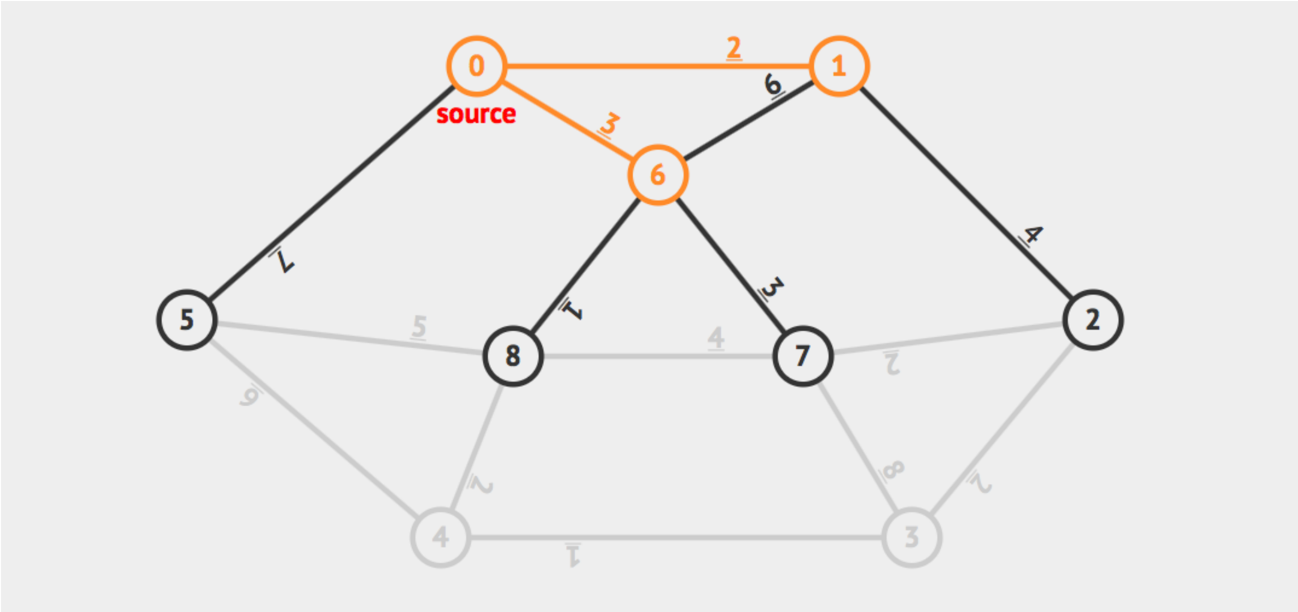


O vértice 2 foi acrescentado à orla, e existem agora dois arcos (0, 6) e (1, 6) que levam ao vértice 6. Mas o **arco candidato** deste vértice é único: naturalmente, (0, 6).

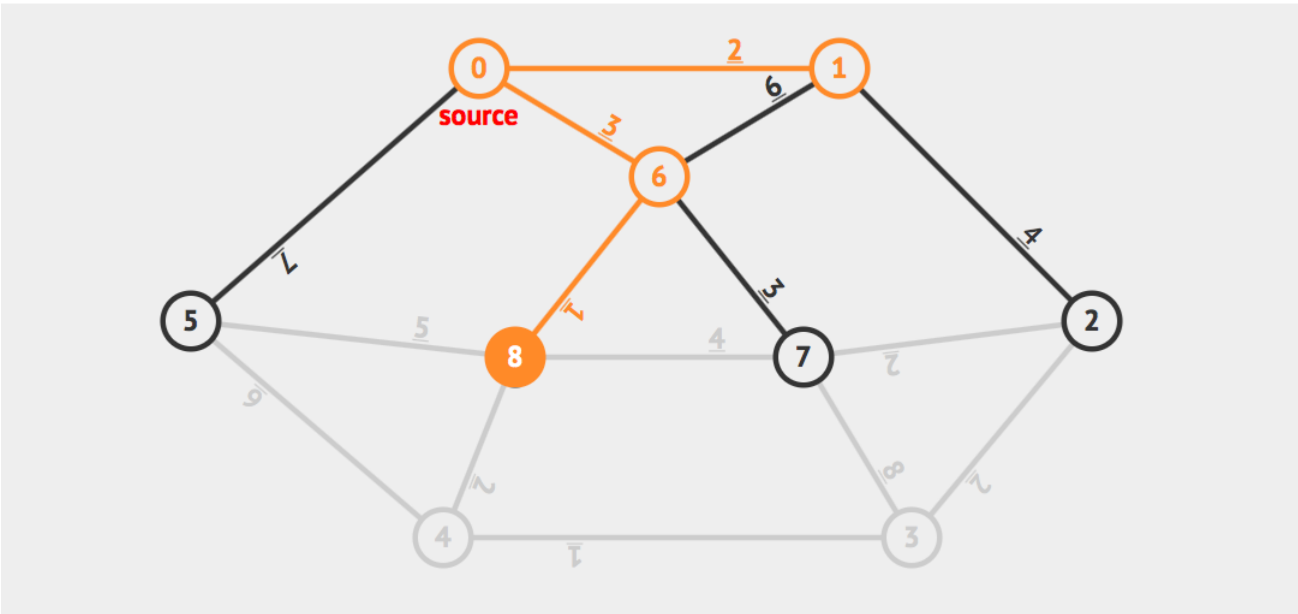
Durante a execução do algoritmo os arcos candidatos (a preto) fazem já parte da árvore de travessia construída. No instante acima, teremos `parent[6]==0`, e não `parent[6]==1`.



Depois de acrescentar um vértice à árvore há sempre que actualizar a orla:

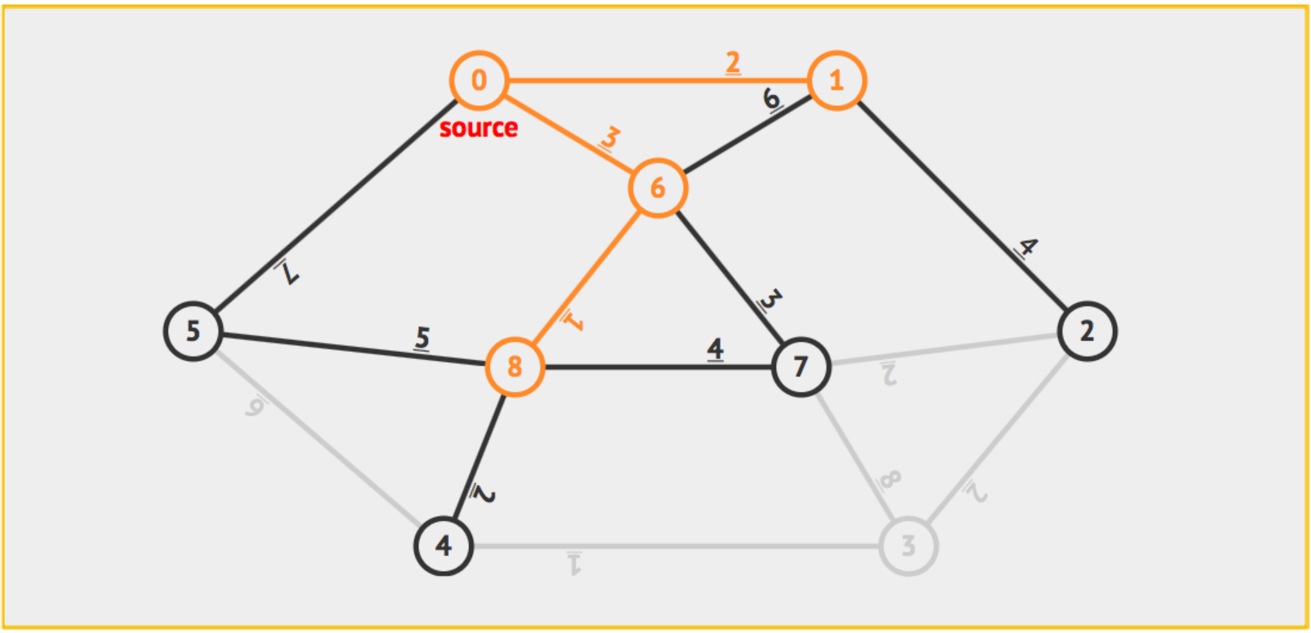


O próximo passo parece óbvio:

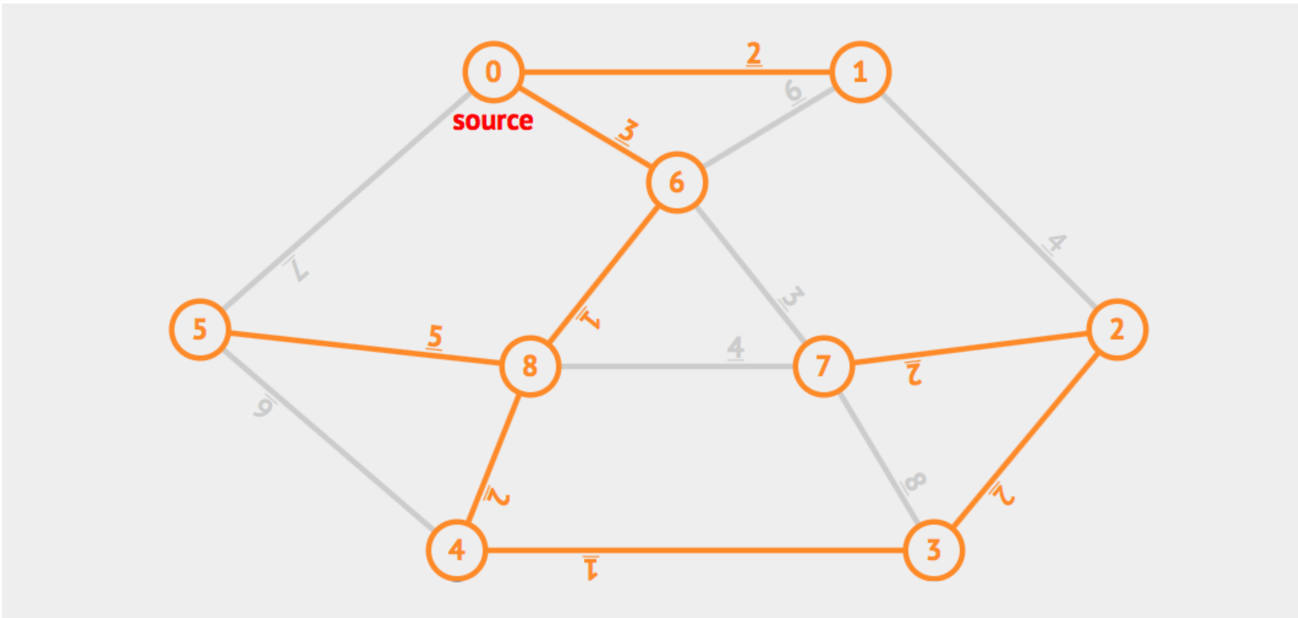


No entanto surge um fenómeno novo: ao actualizar a orla depois de acrescentar o vértice 8, surge um novo arco(8, 5) com destino no vértice 5, que **substitui o arco candidato anterior** (0, 5).

Os arcos candidatos fazem parte da árvore de travessia construída, no entanto, pela razão acima, poderão ser substituídos, não chegando nesse caso a pertencer à MST construída.



Continuando a aplicar este passo básico, obtemos a MST completa, com peso 18:



Algoritmo Detalhado em Python

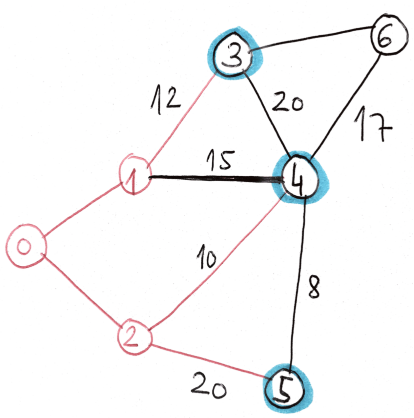
Nota: a representação de grafos em Python como dicionários permite uma sintaxe para acesso ao peso de uma aresta que parece matricial, mas não é: `g[x]` designa a "lista" de adjacências do vértice `x`, mas que é aqui também ela um dicionário da forma **vértice destino** $\mapsto$ **peso**, e não uma lista! Sendo assim `g[x][y]` é o peso da aresta (x, y)

Na implementação em Python representa-se por dicionários a árvore construída, a informação de status dos vértices, e a orla. Esta última em particular será representada por um dicionário cujo domínio são os vértices da orla, que são mapeados para o peso do respectivo arco candidato.

```
def mst_Prim(g):
    fringe = {}
    status = {}
    for i in g:
        status[i] = 'UNSEEN'
    edgeCount = 0

    x = 0;
    status[x] = 'INTREE'
    parent[x] = -1

    while (edgeCount < len(g)-1):
        for y in g[x]:
            wxy = g[x][y]            # weight of edge (x,y)
            if status[y] == 'UNSEEN':
                # add y to fringe with candidate edge (x,y)
                status[y] = 'FRINGE'
                parent[y] = x
                fringe[y] = wxy     # weight of candidate edge
            elif status[y] == 'FRINGE' and  wxy < fringe[y]:
                # replace candidate edge of y by (x,y)
                parent[y] = x
                fringe[y] = wxy     # weight of candidate edge

        # are we blocked? (non-connected graph)

        if len(fringe) == 0:
            return False

        # select next candidate edge and vertex;
```

```
        # remove them from fringe and add to tree
        x = min(fringe, key=lambda x:fringe[x])
        del fringe[x]
        status[x] = 'INTREE'
        edgeCount += 1

    return True
```

Vejamos detalhadamente um exemplo de execução do passo básico do algoritmo.



status[0] = status[1] = status[2] = INTREE
status[3] = status[4] = status[5] = FRINGE
status[6] = UNSEEN

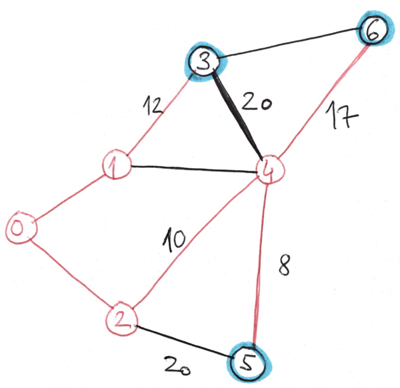
parent[1] = parent[2] = 0 (definitivos)
parent[3] = 1 and parent[4] = parent[5] = 2 (orla)

fringe[3] = 12 fringe[4] = 10 fringe[5] = 20

a aresta (1,4) não é um arco candidato!

Será seleccionado o vértice 4 para ser acrescentado à árvore, e serão percorridos os seus adjacentes:

- 3 já pertencia à orla, mas não haverá alteração do arco candidato (20 > 12)
- 5 já pertencia à orla, e haverá alteração do arco candidato (8 < 20)
- 6 não pertencia à orla; passará a pertencer, com arco candidato (4,6)



Alterações:
status[4] = INTREE
status[6] = FRINGE

parent[6] = 4 fringe[6] = 17

parent[5] = 4 fringe[5] = 8

Análise

O tempo de execução no pior caso (grafo ligado, existe uma árvore) está claramente em $\Omega(|V| + |E|)$, uma vez que a estrutura básica do algoritmo é a de uma travessia (percorre todas as listas de adjacências). No entanto, este algoritmo executa outras operações potencialmente mais pesadas:

1. Em cada iteração do ciclo `while`, i.e. $|V|$ vezes, é feita a selecção do arco candidato de menor peso, e removido da orla o respectivo vértice

2. Em cada iteração dos ciclos `for`, i.e. $|E|$ vezes, pode ser feita uma das seguintes coisas:
   a. é acrescentado um nó à orla, ou
   b. é alterado o peso do arco candidato de um nó da orla

O pior caso destas operações ocorrerá quando o primeiro vértice está ligado a todos os outros, tendo a orla sempre tamanho máximo.

Tabela de *hash*
Se a orla for implementada como no código Python acima, por um dicionário (tabela de *hash* sobre um vector dinâmico), as operações de inserção/alteração do peso são executadas em tempo (tendencialmente) constante, mas a operação de selecção do mínimo obriga a percorrer todos os vértices da orla, sendo executada em tempo linear.
O pior caso ocorrerá quando o primeiro vértice está ligado a todos os outros, tendo a orla sempre tamanho máximo, e teremos $T(N) = \mathcal{O}(|V|^2 + |E|)$.

Fila com Prioridades / *Min-heap*
A orla pode ainda ser implementada por uma *Fila* de vértices, tendo os pesos dos arcos candidatos como *prioridades*. Neste caso as operações de inserção, alteração de peso, e remoção do mínimo serão todas executadas em tempo logarítmico, pelo que teremos $T(N) = \mathcal{O}(|V|.\log|V| + |E|.\log|V|)$

Lista ligada
Uma alternativa semelhante em termos de performance é implementar a orla como uma lista ligada (*não-ordenada*) contendo os vértices. A selecção/remoção de mínimo exigirá no pior caso tempo linear, com a inserção e alteração de peso (mantidos num array) feitas em tempo constante, pelo que teremos ainda $T(N) = \mathcal{O}(|V|^2 + |E|)$.

EXERCÍCIO
[https://codeboard.io/projects/10154]
Com base nas definições fornecidas no projecto Codeboard, implemente em C o algoritmo de Prim, escolhendo para a orla uma das estruturas sugeridas acima.

Caminhos Mais Curtos

O problema da determinação de caminhos mais curtos num grafo pesado é uma generalização do mesmo problema em grafos sem pesos. Pretende-se agora minimizar não o número de arestas dos caminhos, mas sim o *peso* (comprimento, na analogia geográfica) desses caminhos.

A definição de peso de um caminho $P = v_0, v_1, \ldots v_k$ é a esperada:

$$w(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

Algumas notas:

- O problema está intimamente ligado com o do cálculo da *distância* entre dois vértices, que é precisamente definida como o peso do caminho mais curto.
- O problema que consideraremos aqui é o da construção de todos os caminhos mais curtos com origem num determinado vértice e destino em todos os vértices alcançáveis a partir dele, conhecido por ***single-source sortest paths***.
- O problema simétrico, ***single-destination sortest paths,*** pode ser reduzido a este sobre o grafo simétrico do original.
- O problema ***single-pair sortest paths*** (caminho mais curto ponto a ponto) é naturalmente um problema aparentemente mais simples, mas não existe um algoritmo específico: resolve-se utilizando um algoritmo de ***single-source***

sortest paths (podendo a sua execução ser interrompida antecipadamente).

Algoritmo de Dijkstra para o problema *single-source sortest paths*

Trata-se de um algoritmo extremamente parecido com o de Prim, igualmente baseado numa travessia guiada por uma estratégia específica, e com o mesmo tempo assimptótico de execução.

A árvore de travessia construída, com origem num vértice s, contém todos os caminhos mais curtos com origem em s e destino em vértices alcançáveis a partir de s.

As alterações a introduzir são as seguintes:

- À semelhança do que acontece no algoritmo de travessia em largura, utiliza-se um vector dist[] para armazenar a distância desde a origem até aos vértices da árvore de travessia
  - $\text{dist}[y] = \delta(s, y)$ se y pertence à árvore de travessia

  neste caso o valor $\text{dist}[y]$ não se alterará mais durante a execução do algoritmo, contém já o valor real da distância entre s e y

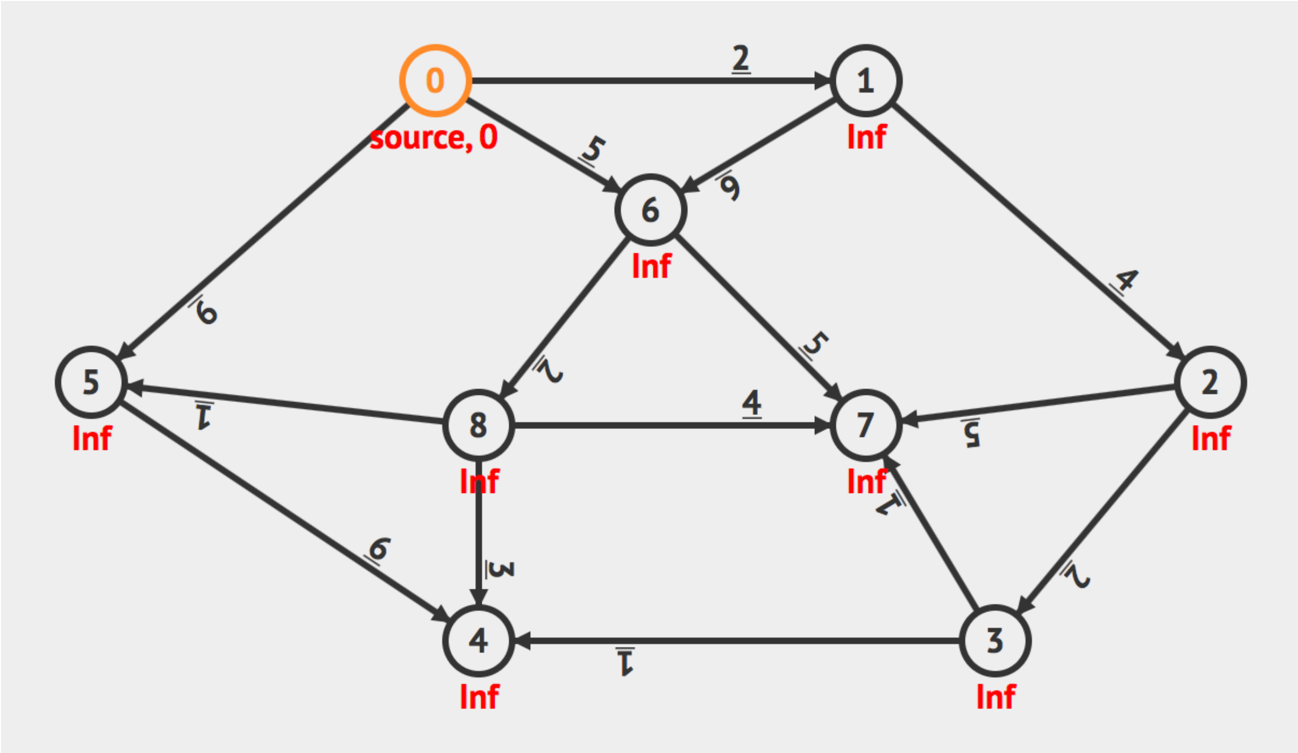
- Mas tal como acontece com o vector parent[], o vector dist[] estará definido também para os vértices da orla:
  - $\text{dist}[z] = \delta(s, y) + w(y, z)$ se z está na orla e (y, z) é o seu arco candidato

  neste caso esta informação é provisória e poderá ser modificada caso se altere o arco candidato

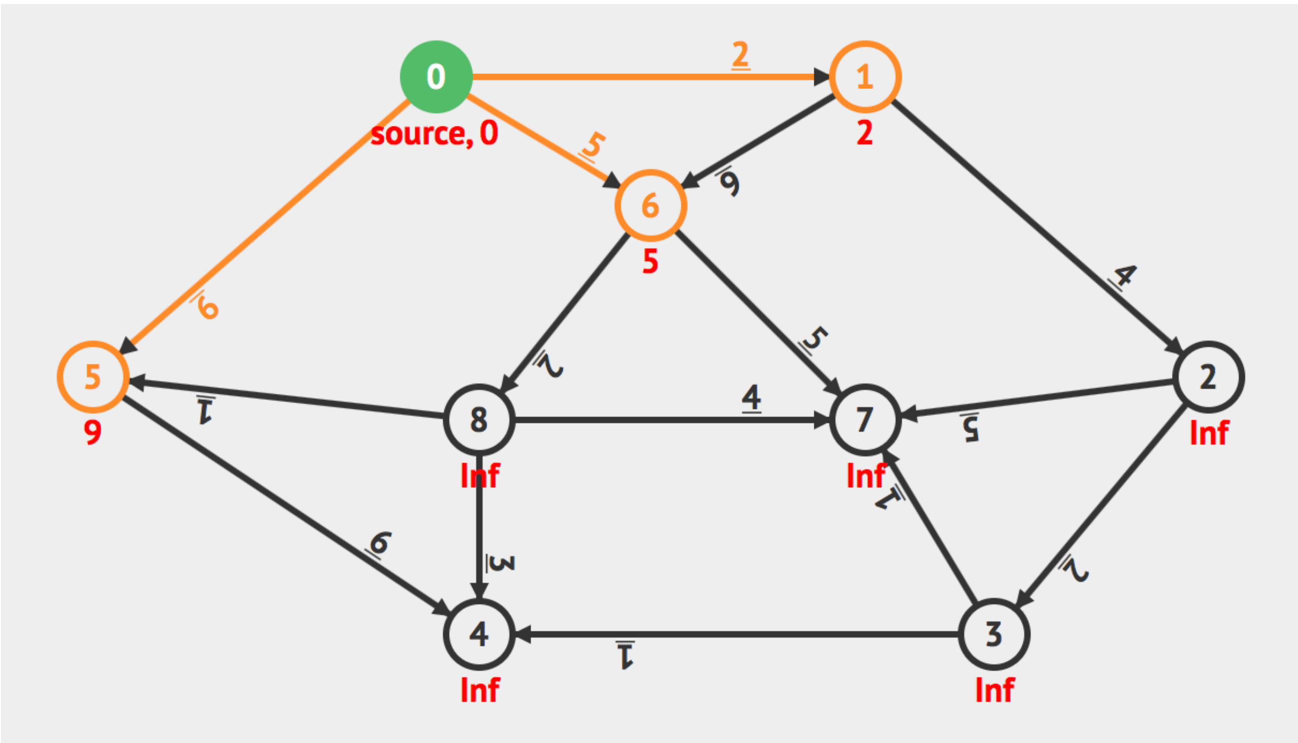
- O critério de selecção do vértice da orla a acrescentar à árvore em casa passo é simplesmente a minimização do valor $\text{dist}[z]$, ou seja, *acrescenta-se o vértice cuja distância desde a origem é a menor*.

Exemplo

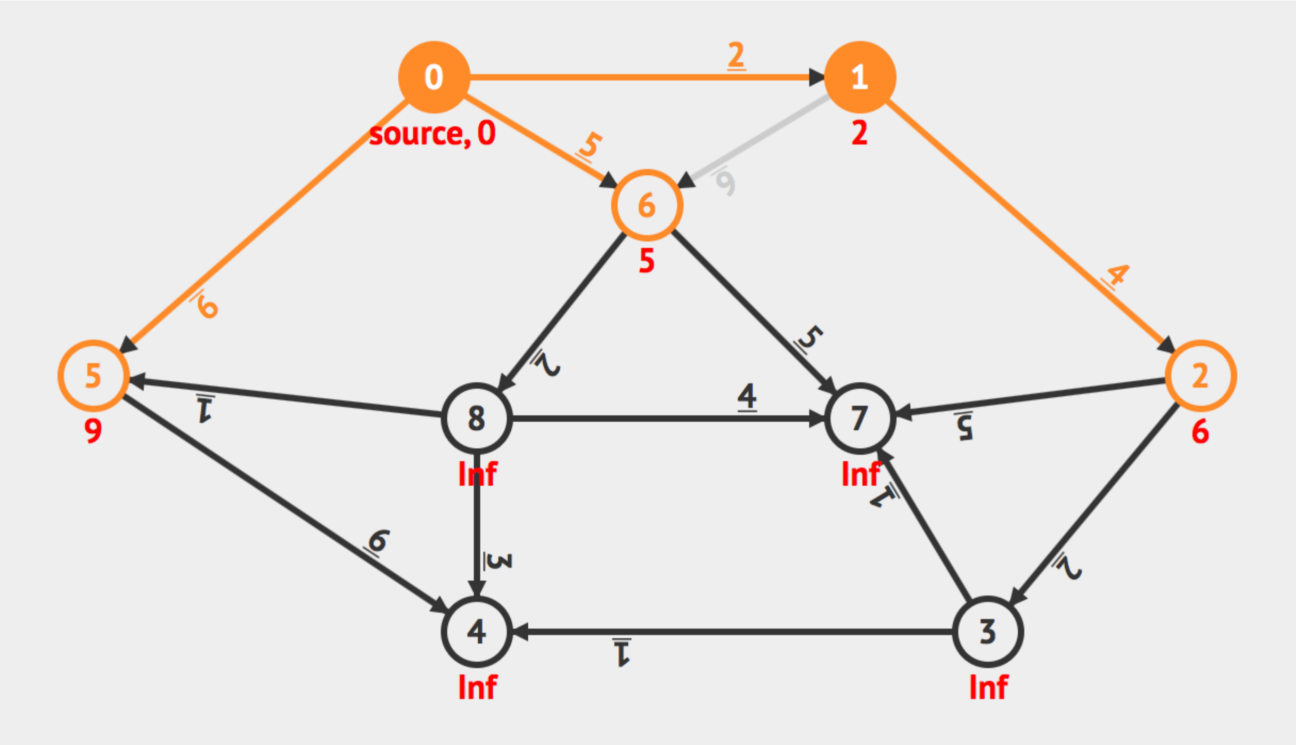
[Animação construída em https://visualgo.net/en/sssp]



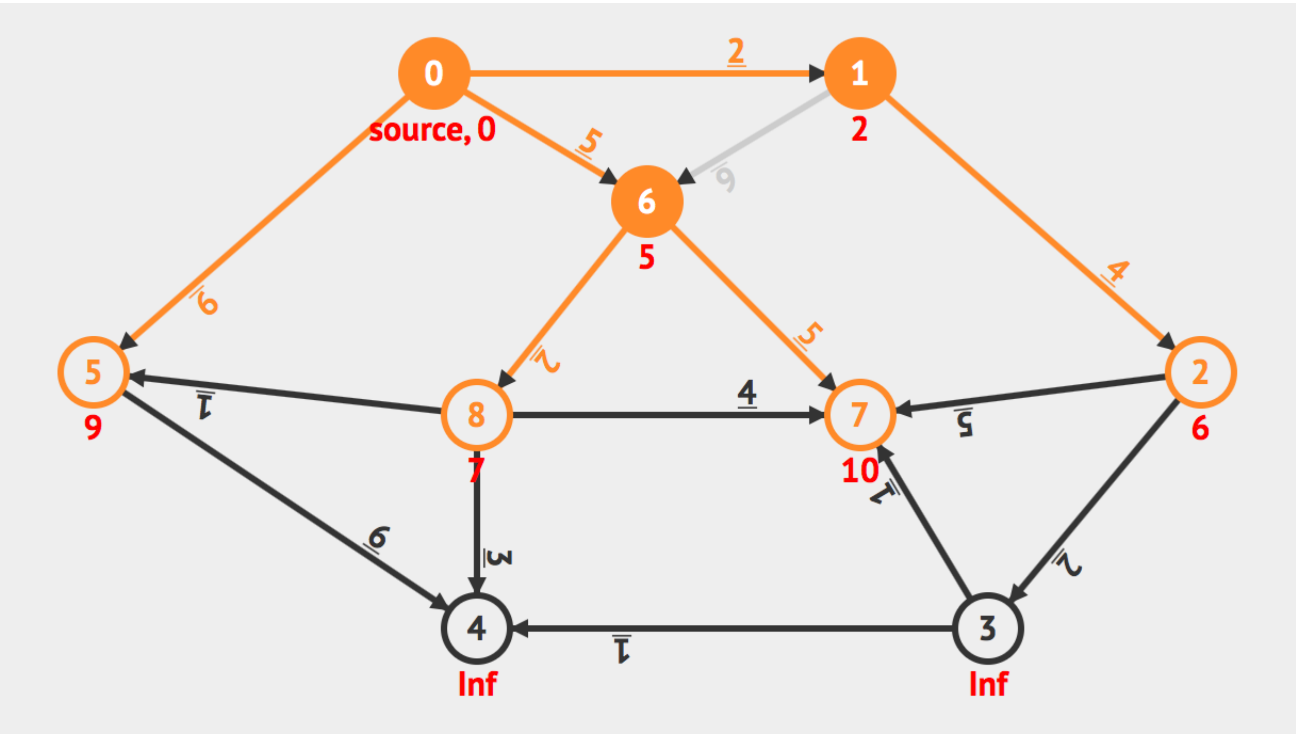
Anotaremos (a vermelho) nos vértices da árvore e da orla a distância desde a raíz até cada um deles. Para os vértices da orla estes valores não são definitivos, podendo ser alterados se houver mudança do arco candidato.

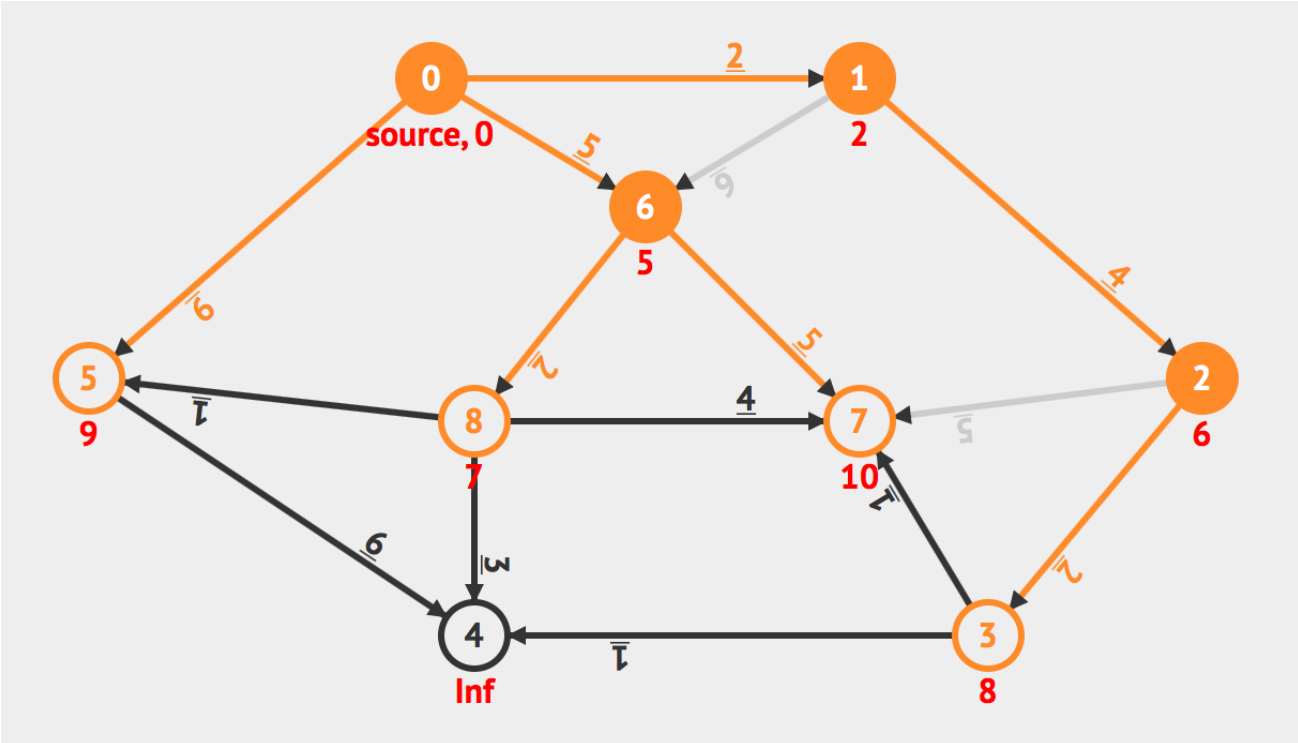


Será escolhido o vértice 1. Note-se que apesar de haver duas arestas que levam ao vértice 6, não haverá alteração do arco candidato, uma vez que 2+6 > 5.

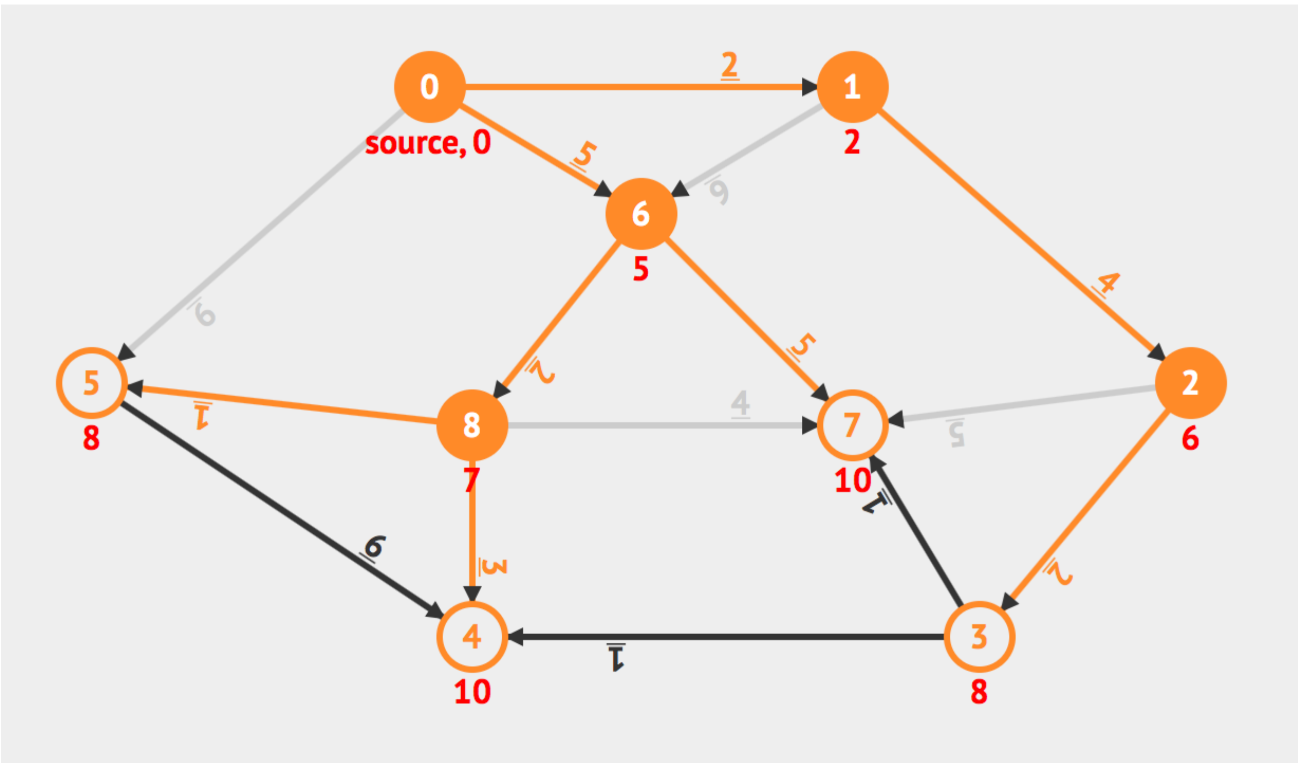


Note-se que, ao acrescentar vértices à orla, a distância desde a raíz é calculada somando a distância da origem ao pai com o peso das arestas acrescentadas à árvore.
No próximo passo, a distância até 8 será 5+2=7, e até 7 será 5+5=10.

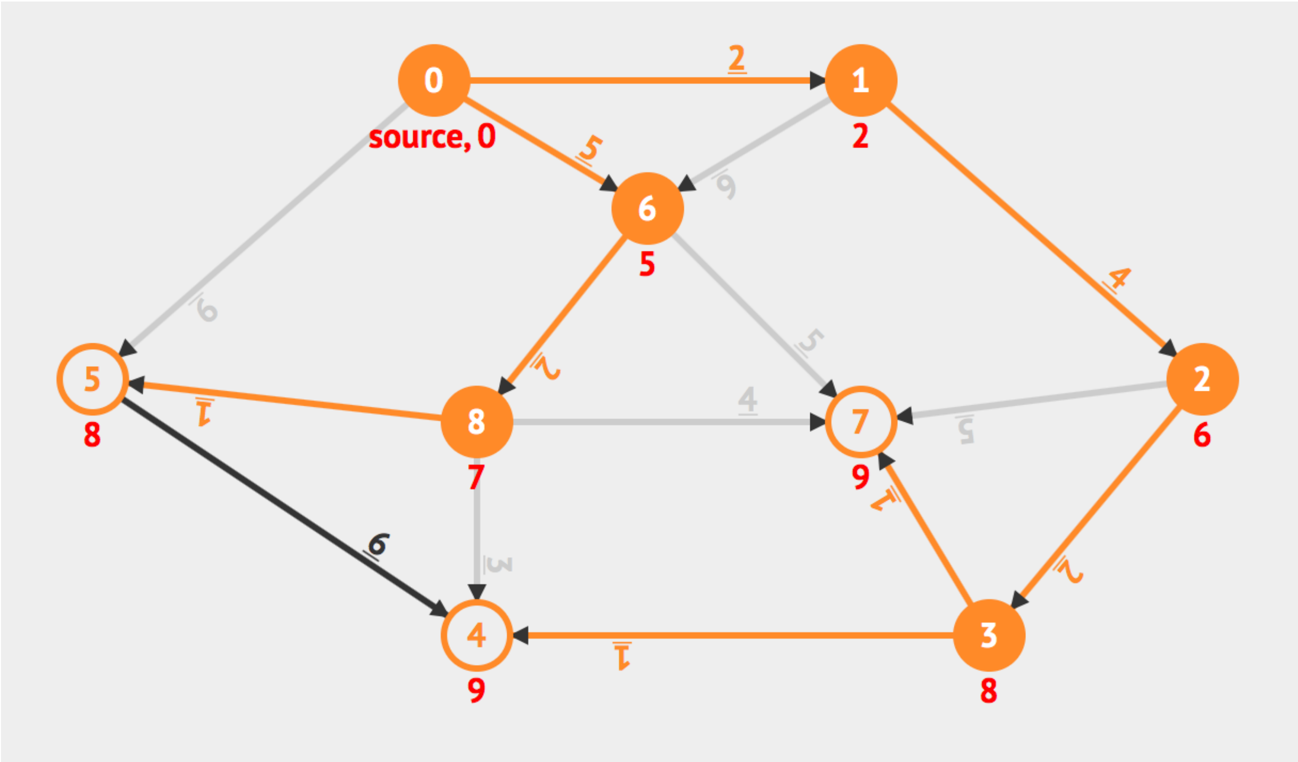




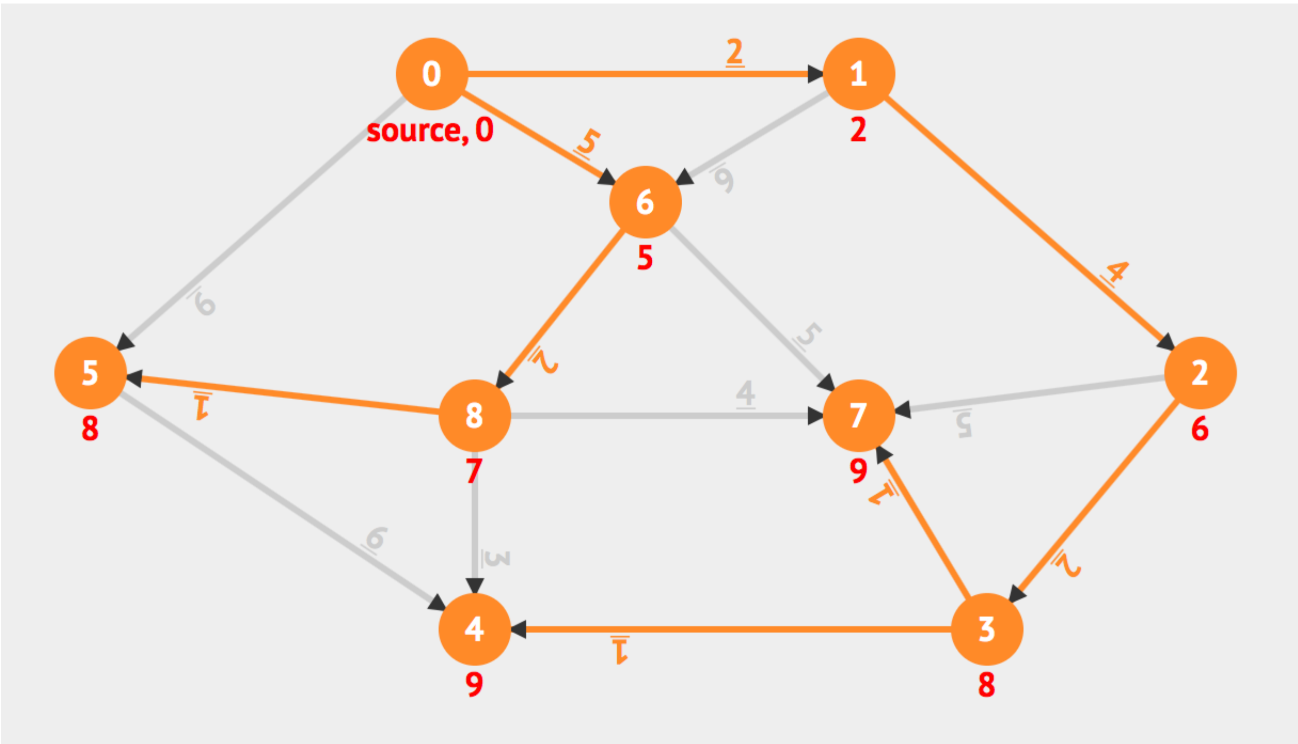
No próximo passo acrescentaremos o vértice 8 à árvore, o que levará a uma alteração do arco candidato de 5, uma vez que 7+1 < 9. Também a distância desde a raiz, anotada no vértice, será alterada.



No próximo passo haverá nova alteração de arco candidato e distância, desta vez do vértice 7.



Os restantes passos não trazem novidades. A árvore de travessia construída contém todos os caminhos mais curtos com origem no vértice 0.



Algoritmo Detalhado em Python

```
def sp_Dijkstra(g, s):
    fringe = {}
    status = {}
    ## NEW IN DIJKSTRA
    dist = {}
    for i in g:
        status[i] = 'UNSEEN'
    edgeCount = 0

    x = s;
    status[x] = 'INTREE'
    parent[x] = -1
    ## NEW IN DIJKSTRA
    dist[x] = 0

    while (edgeCount < len(g)-1):
        for y in g[x]:
            wxy = g[x][y]
            if status[y] == 'UNSEEN':
                # add y to fringe with candidate edge (x,y)
                status[y] = 'FRINGE'
                parent[y] = x
                fringe[y] = wxy
                ## NEW IN DIJKSTRA
                dist[y] = dist[x] + wxy
            ## MODIFIED FOR DIJKSTRA
            elif (status[y] == 'FRINGE'
            and  dist[x] + wxy < dist[y]):
                # replace candidate edge of y by (x,y)
                parent[y] = x
                fringe[y] = wxy
                ## NEW IN DIJKSTRA ##
```

```
                dist[y] = dist[x] + wxy

        # are we blocked? (non-connected graph)

        if len(fringe) == 0:
            return False

        # select next candidate edge and vertex
        # remove them from fringe and add to tree
        ## MODIFIED FOR DIJKSTRA
        x = min(fringe, key=lambda x:dist[x])
        del fringe[x]
        status[x] = 'INTREE'
        edgeCount += 1

    return True
```

Código Python para teste dos algoritmos

Prim:

```
graph = {
    0: { 1: 2, 5: 7, 6: 3 },
    1: { 0: 2, 2: 4, 6: 6 },
    2: { 1: 4, 3: 2, 7: 2 },
    3: { 2: 2, 4: 1, 7: 8 },
    4: { 3: 1, 5: 6, 8: 2 },
    5: { 0: 7, 4: 6, 8: 5 },
    6: { 0: 3, 1: 6, 7: 3, 8:1 },
    7: { 2: 2, 3: 8, 6: 3, 8:4 },
    8: { 4: 2, 5: 5, 6: 1, 7:4 }
    }
```

```python


def printGraph(g):
    for i in g:
        print"[%s]"%(i),
        for j in sorted(g[i].keys()):
            print "-> (%s, %s)"%(j,g[i][j]),
        print


printGraph(graph)
print

parent = {}
ok = mst_Prim(graph)
if ok:
    sum = 0;
    print "\nMST:\n",
    for i in graph:
        if parent[i]>=0 :
            print"%1s--%1s"%(parent[i], i)
            sum += graph[i][parent[i]]
    print "Total weight = ", sum
else:
    print "UNCONECTED GRAPH, CANNOT BUILD MST!"

```

Dijkstra:

```python
graph = {
    0: { 1: 2, 5: 9, 6: 5 },
    1: { 2: 4, 6: 6 },
    2: { 3: 2, 7: 5 },
```

```
    3: { 4: 1, 7: 1 },
    4: { },
    5: { 4: 6 },
    6: { 7: 5, 8: 2 },
    7: { },
    8: { 4: 3, 5: 1 }
    }


def printGraph(g):
    for i in g:
        print"[%s]"%(i),
        for j in sorted(g[i].keys()):
            print "-> (%s, %s)"%(j,g[i][j]),
        print


printGraph(graph)
print

parent = {}
ok = sp_Dijkstra(graph, 0)
if ok:
    sum = 0;
    print "\nShortest Paths Tree:\n",
    for i in graph:
        if parent[i]>=0 :
            print"%1s--%1s"%(parent[i], i)
else:
    print "UNCONNECTED GRAPH, CANNOT REACH ALL VERTICES!"

```