Graph notes

PDF: Grafos.pdf
Data: 30-11-2022
Tags: #SoftwareEngineering #ALGC #C

Other notes

- Graphs
  - Adjacency Matrix implementation in C
  - Graph Theory
    - Sorting Graphs (Adjacency List)
      - Adjacency List Implementation in C

"/home/rudrigu/obsidian/Collection of Notes/Universidade/UCs/1º Semestre/Fundamentos de Comunicação de Dados/Other notes/Análise de Sinais.md"

Procura e Travessias

Dado um grafo $G = (V, E)$, um caminho em G é uma sequência de vértices

$$[v_1, v_2, \ldots, v_n]$$

em que existe uma aresta entre cada par de vértices consecutivos, i.e., $(v_i, v_{i+1}) \in E$.

Uma classe importante de problemas sobre grafos consiste em determinar se dados dois vértices o e d existe um caminho com origem o e d. Neste caso dizemos que o vértice d é **alcançável** a partir de o.

Este problema pode ser visto como um caso particular de determinar os vértices que são alcançáveis a partir de um dado vértice. Este problema é normalmente conhecido como uma **travessia** do grafo (a partir de um destino).

Depth first

A definição de caminho entre dois vértices dada acima tem uma formulação recursiva que facilmente se traduz num processo de determinação de caminhos:
Uma sequência de vértices é um caminho no grafo sse

1. $n = 1$, i.e., é uma sequência singular e por isso trata-se de uma caminho vazio (sem arestas);
2. $(v_1, v_2 \in E)$ e além disso $[v_2, \ldots, v_n]$ é um caminho no grafo g (de v_2 a v_n).

```
int procura (Graph g, int o, int d) {
        int found = 0;
        EList it;
        if (o == d) found = 1;
        else
                for (it = g[o]; it != NULL && !found; it = it->next)
                        found = procura(g, it->dest, d);
        return found;
}
```

No entanto, esta definição tem um problema: num grafo **cíclico** é possível que esta função não termine. Isto porque não temos nenhuma maneira de determinar se um determinado vértice já foi incluído no caminho.

Uma forma de contornar este problema passa por adicionar aos argumentos da função a informação sobre os vértices que já foram incluídos no caminho.

Vamos então usar um vetor adicional, passado como argumento, que marca todos os vértices já incluídos na procura. Esse array terá de ser inicializado (uma única vez) e por isso devemos definir a função em dois passos: um (recursivo) que faz a procura ao longo do grafo e um outro, anterior, que começa por inicializar o referido array.

```
int procura (Graph g, int o, int d) {
        int visitados[NV];
        for (int i = 0; i < NV; i++) {
                visitados[i] = 0;
        }
        return (procuraRec(g, o, d, visitados));
}
```

```
int procuraRec (Graph g, int o, int d, int v[]) {
        int found = 0;
        EList it;
        v[o] = 1;
        if (o == d) found = 1;
        else
                for (it = g[o]; it != NULL && !found; it = it->next)
                        if (!v[it->dest])
                                found = procuraRec (g, it->dest, d, v);
        return found;
}
```

Vamos definir agora uma função que, dado um grafo e um nodo desse grafo, determina quantos vértices são alcançáveis a partir desse vértice.
Este problema é, tal como referido acima, conhecido como uma travessia do grafo a partir de um vértice.

A estratégia que vamos usar é conhecida como travessia **depth first**.

```
int travessiaDF (Graph g, int o) {
        int visitados[NV] = {0};
        return (DFRec (g, o, visitados));
}

int DFRec (Graph g, int o, int v[]) {
        int count = 1;
        v[o] = 1;

        for (EList it = g[o]; it != NULL; it = it->next)
                if (!v[it->dest])
                        count += DFRec (g, it->dest, v);

        return count;
}
```

A principal diferença consiste no teste de paragem ($found$ não tem de ser igual a 0), ou seja, esta última não termina mal encontre o destino, pelo contrário, são esgotados todos os destinos possíveis e, por isso, são visitados todos os nodos alcançáveis a partir do vértice inicial.

A função $DFRec$ acima pode ser usada para determinar o tamanho (número de vértices) da maior componente ligadad de um grafo não orientado:

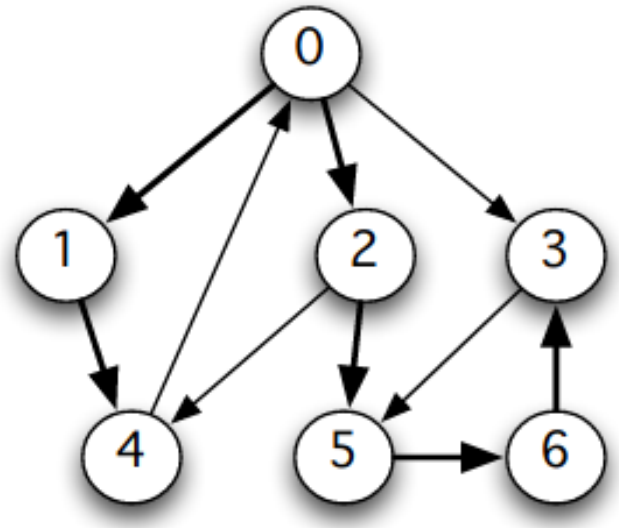
```
int maiorCL (Graph g) {
        int visitados[NV] = {0};
        int max = 0, c;

        for (int i = 0; i < NV; i++)
                if (visitados[i] == 0) {
                        c = DFRec (g, i, visitados);
```

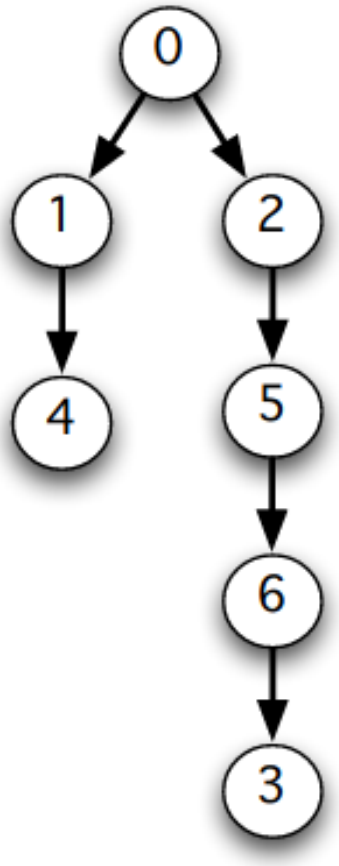
```
                    if (c > max) max = c;
                }

        return max;
}
```

> **Note**
>
> Mais uma vez estamos a tirar partido do mecanismo de passagem de arrays como argumentos do C, em que as alterações feitas a tais arrays persistem após a invocação da função.

(Verificar se um grafo é bi-partido; more here - grafo não orientado)

Uma forma usual de representar uma árvore correspondente a uma travessia baseia-se numa propriedade simples das árvores: cada nodo da árvore (com exceção para a raiz) tem exatamente um antecessor. Dessa forma, a informação pode ser representada com um array em que na posição i se encontra o antecessor do vértice i.





0	1	2	3	4	5	6
-1	0	0	6	1	2	3

-1 assinala a raiz da árvore.
Pai do 1 é o 0, do 3 é o 6, etc.

```
int travessiaDF2 (Graph g, int o, int p[]) {
        int visitados[NV];
```

```
        for (int i = 0; i < NV; i++) {
                visitados[i] = 0;
                p[i] = -2;
        }
        p[o] = -1;
        return DFRec2(g, o, visitados, p);
}

int DFRec2 (Graph g, int o, int v[], int p[]) {
        int count = 1;
        EList it;
        v[o] = 1;
        for (it = g[o]; it != NULL; it = it->next)
                if (!v[it->next]) {
                        p[it->dest] = o;
                        count += DFRec2(g, it->dest, v, p);
                }
        return count;
}
```

O uso do array visitados não é estritamente necessário. Pode verificar-se que $v[i] == 0$ sse $p[i] == -2$. Desta forma a definição acima pode ser reescrita.

```
int travessiaDF2 (Graph g, int o, int p[]) {
        for (int i = 0; i < NV; i++)
                p[i] = -2;
        p[o] = -1;
        return DFRec2(g, o, p);
}

int DFRec2 (Graph g, int o, int p[]) {
        int count = 1;
        for (EList it = g[o]; it != NULL; it = it->next)
                if (p[it->dest] == -2) {
                        p[it->dest] = o;
                        count += DFRec2(g, it->dest, p);
                }
        return count;
}
```

Breadth First

A estratégia de travessia apresentada acima usa a recursividade como mecanismo de navegação sobre o grafo. Uma outra forma de fazer uma travessia passa pela manutenção explícita da informação sobre quais os vértices que devem ainda ser visitados.
Uma forma de armazenar tal informação é usando uma fila (queue) onde os vários vértices candidatos a serem visitados vão sendo adicionados. O processo termina quando já não houver vértices nessa fila.

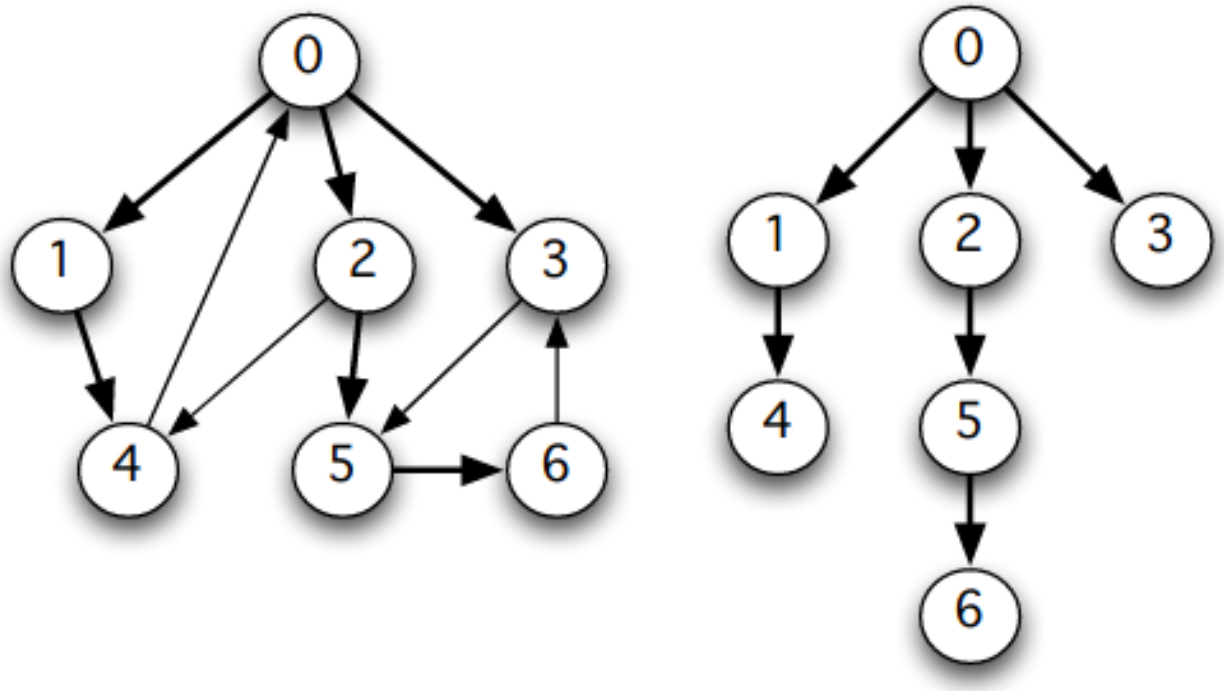
```
int travessiaBF (Graph g, int o) {
        int visitados[NV];
        int count = 0;

        Queue q;
        q = emptyQueue();
        enqueue(q, o);
        visitados[o] = 1;

        while (!empty(q)) {
                o = dequeue(q);
                count++;
                for (EList it = g[o]; it != NULL; it = it->next)
```

```
                        if (!visitados[it->dest])
                                enqueue(q, it->dest);
        }
        return count; // quantos vértices são alcançáveis a partir de 'o'
}
```

> A queue nunca tem mais do que NV elementos e nunca um vértice pode entrar na queue mais do que uma vez.

Tal como fizemos com a travessia em profundidade, podemos construir a árvore associada à travessia usand um array de antecessores que é passado como argumento (e que servirá ainda para desempenhar o papel do array $visitados$).

```
int travessiaBFTree (Graph g, int o, int ant[]) {
        int q[NV]; int inicio, fim;
        int count = 0;
        EList it;

        for (int i = 0; i < NV; i++) {
                ant[i] = -2;
        }
        inicio = fim = 0;
        q[fim++] = o;
        ant[o] = -1;

        while (inicio < fim) {
                o = q[inicio++];
                count++;
                for (it = g[o]; it != NULL; it = it->next)
                        if (ant[it->dest] == -2) {
                                ant[it->dest] = o;
                                q[fim++] = it->dest;
                        }
        }

        return count;
}
```



Ordenação topológica

Algoritmo de Tarjan

Este algoritmo é baseado na travessia depth-first. Há no entanto alguns problemas que têm de ser tratados para que tal estratégia possa produzir o efeito desejado.

Em primeiro lugar deve ser possível determinar se a ordenação é possível, i.e., se o grafo tem ou não ciclos. Em segundo lugar devemos ter em atenção que à partida não sabemos se um dado vértice pode ou não aparecer no início da sequência.

O segundo problema é relativamente fácil de resolver se começarmos por calcular a sequência por ordem inversa: a inclusão de um vértice em tal sequência (invertida) far-se-á *depois* de incluídos todos os vértices alcançáveis a partir dele.

Além disso, uma travessia a partir de um único vértice pode não ser suficiente para percorrer todo o grafo, pelo que devem ser iniciadas travessias suficientes para garantir que todo o grafo é visitado.

A forma de resolver o primeiro problema consiste em incluir no processo a informação sobre os vértices que estão a ser visitados.
O algoritmo de travessia depth-first apresentado atrás usa um array visitado em que cada posição determina se já foi ou não iniciada uma travessia a partir desse vértice.
Em vez disso, vamos usar um array *estado* em que cada posição i pode estar um de três valores possíveis:

- 0 se **nunca foi iniciada** uma travessia a partir do vértice i. Dizemos que o vértice i não está visitado.
- 1 se **já foi iniciada** uma travessia a partir do vértice i mas **ainda não terminou**. Dizemos que o vértice i está em visita.
- 2 se **já terminou** uma travessia iniciada no vértice i. Dizemos que o vértice i está **visitado**.

Esta alteração pode ser feita facilmente: a atualização do estado de cada vértice deve ser feita (1) no início de cada travessia (a primeira instrução) o vértice deve ser marcado como em visita e (2) no final (última instrução) o vértice deve ser marcado como **visitado** e adicionado ao final da sequência.

Por uma questão visual é ainda costume caraterizar estes estados dos vértices usando o seguinte *código de cores*:

- **Branco** para os vértices não visitados,
- **Preto** para os vértices visitados,
- **Cinzento** para os vértices em visita.

Antes de prosseguirmos na codificação desta solução convém atentar numa propriedade (invariante) importante sobre esta transformação:

> Em cada estado do processo existe um caminho que liga todos os vértices marcados como **em visita**.

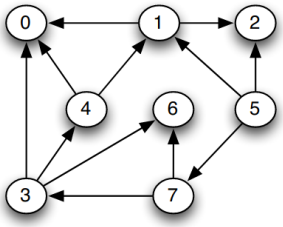
Esta observação é crucial para a determinação da existência de ciclos (e consequente insucesso da ordenação topológica): se nos depararmos com um sucessor do vértice em análise para um outro cujo estado seja **em visita** foi detetado um ciclo.

```
int tarjanTS (Graph g, int sep[]) {
        int color[NV];
        int v, j, t, r, temp;

        // inicializa o estado de cada vértice como não visitado
        for (v = 0; v < NV; v++) {
                color[v] = 0; // White
        }

        // incoca-se a função de travessia até que todos os vértices passem
        // a ser visitados
        r = 1; t = 0;
        for (v = 0; v < NV; v++)
                if (color[v] == 0)
                        // o valor r é modificado caso seja encontrado um ciclo
                        // retorna o número de vértices que foram colocados em seq
```

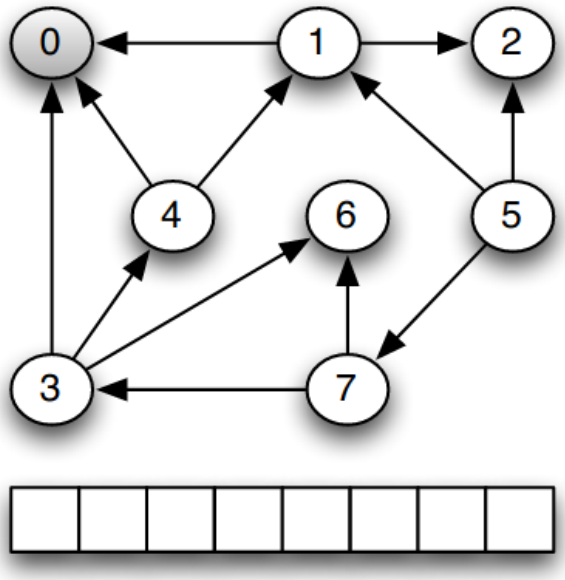
```
                        t += dfirstTopSort(g, v, color, &r, seq + t);

        // inverter sequência
        for (v = 0, j = t; v < t; v++, j--) {
                temp = seq[v];
                seq[v] = seq[j];
                seq[j] = temp;
        }

        return r;
}
```

```
int dfirstTopSort (Graph g, int o, int color[], int *success, int seq[]) {
        int t;
        EList it;
        color[o] = 1; // GREY
        t = 0;

        for (it = g[o]; it != NULL; it = it->next)
                if (color[it->dest] == 0) // WHITE
                        t += dfirstTopSort(g, it->dest, color, success, seq + t);
                else if (color[it->dest] == 1) // GREY
                        *success = 0;

        seq[t++] = o;
        color[0] = 2; // BLACK
        return t;
}
```

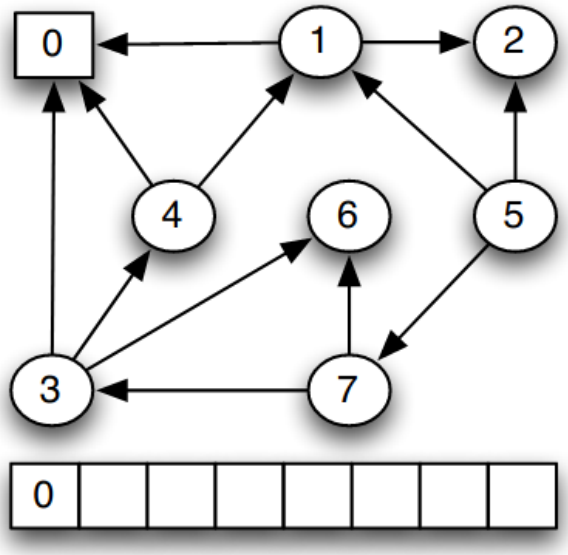
Evolução do algoritmo



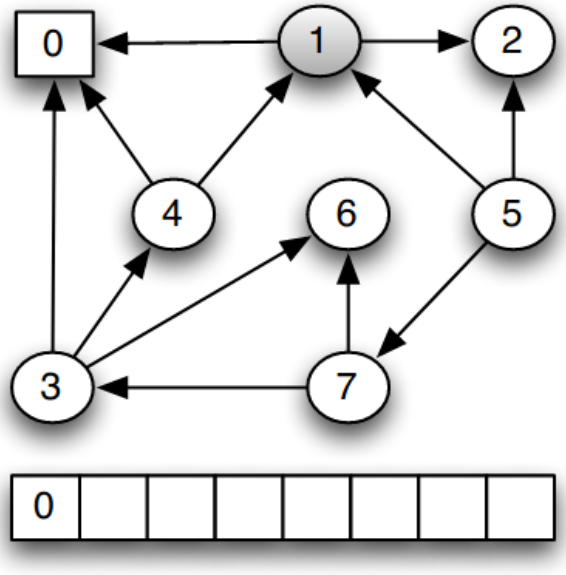
1: Inicia-se uma travessia a partir do vértice 0. Este é marcado como em visita.



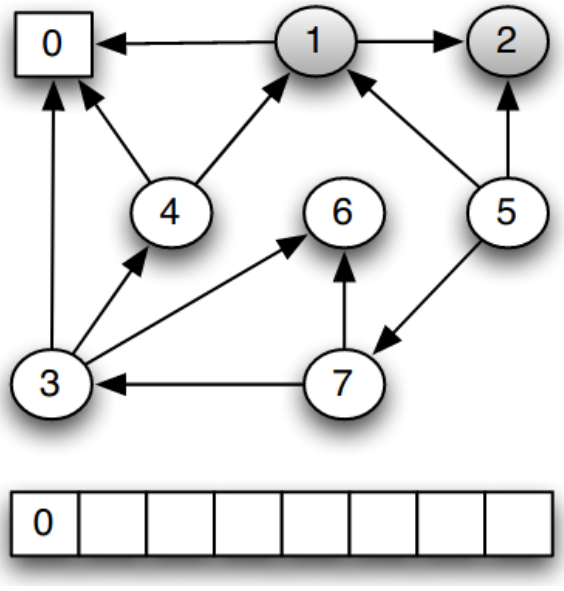
2: Como o vértice 0 não tem sucessores termina a visita: o vértice é marcado como visitado e adicionado à sequência.



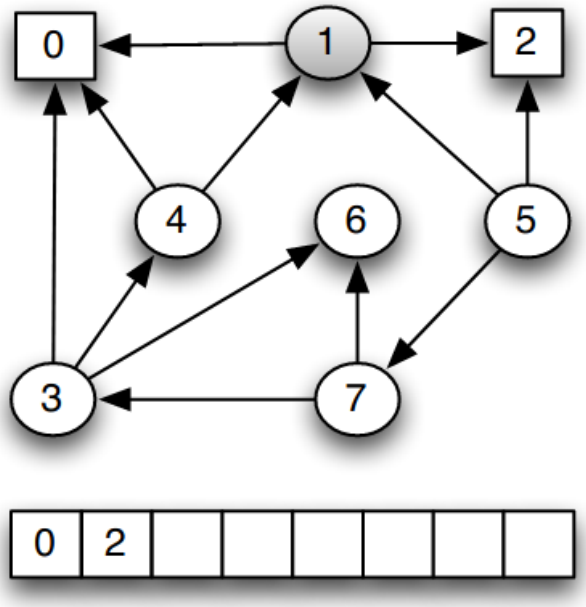
3: Inicia-se uma travessia a partir do vértice 1. Este é marcado como em visita.



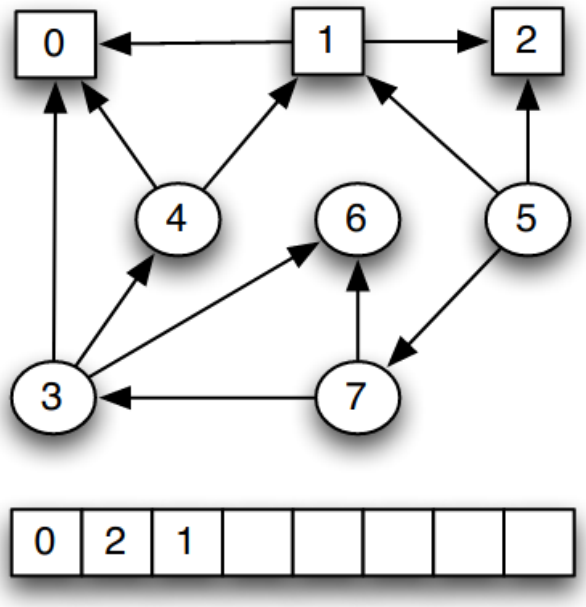
4: Inicia-se uma travessia a partir do vértice 2. Este é marcado como em visita.



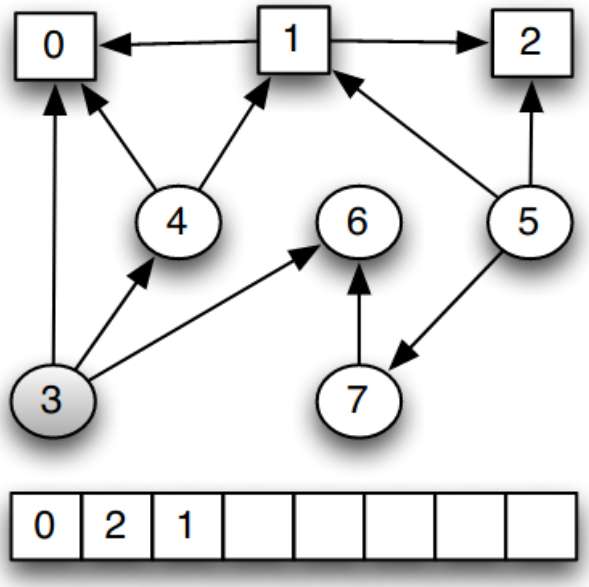
5: Como o vértice 2 não tem sucessores termina a visita: o vértice é marcado como visitado e adicionado à sequência.



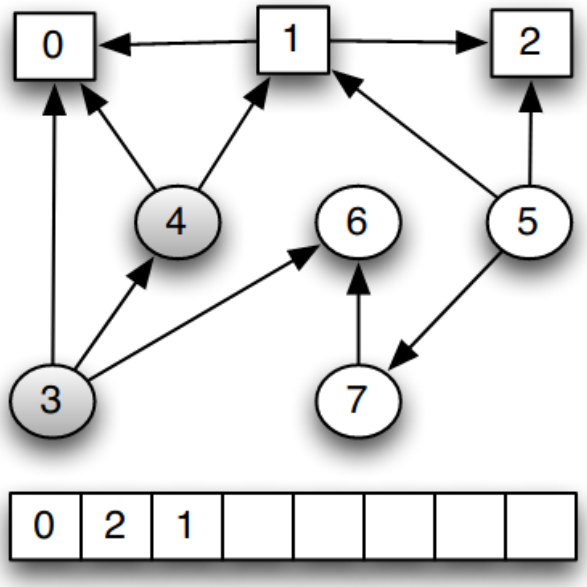
6: Como o vértice 1 não tem mais sucessores termina a visita: o vértice é marcado como visitado e adicionado à sequência.



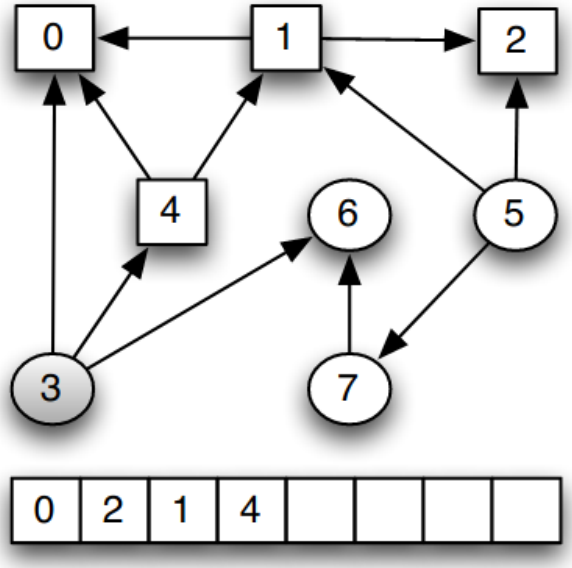
7: Inicia-se uma travessia a partir do vértice 3. Este é marcado como em visita.



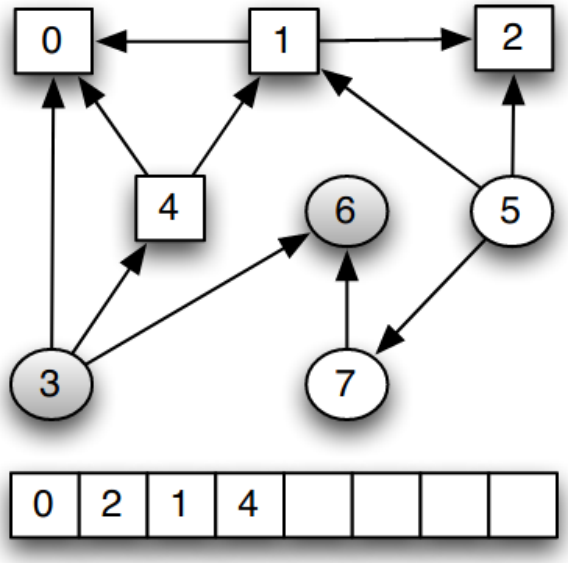
8: Inicia-se uma travessia a partir do vértice 4. Este é marcado como em visita.



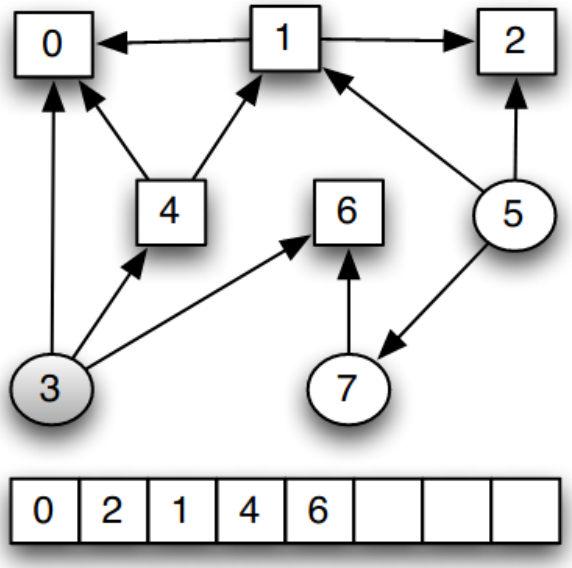
9: Como o vértice 4 não tem sucessores não visitados termina a visita: o vértice é marcado como visitado e adicionado à sequência.



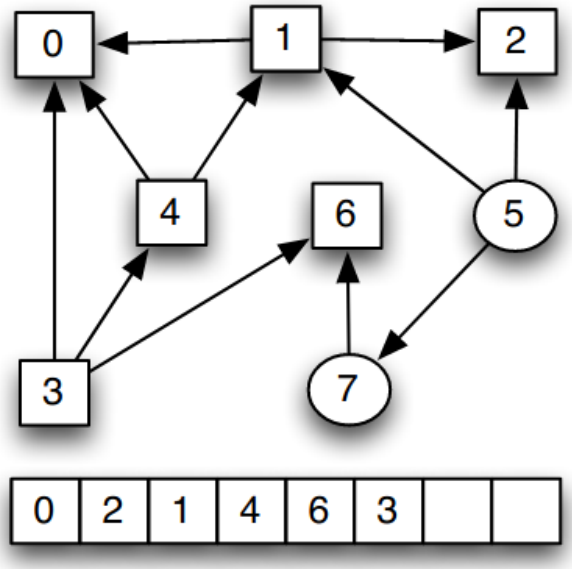
10: Inicia-se uma travessia a partir do vértice 6. Este é marcado como em visita.



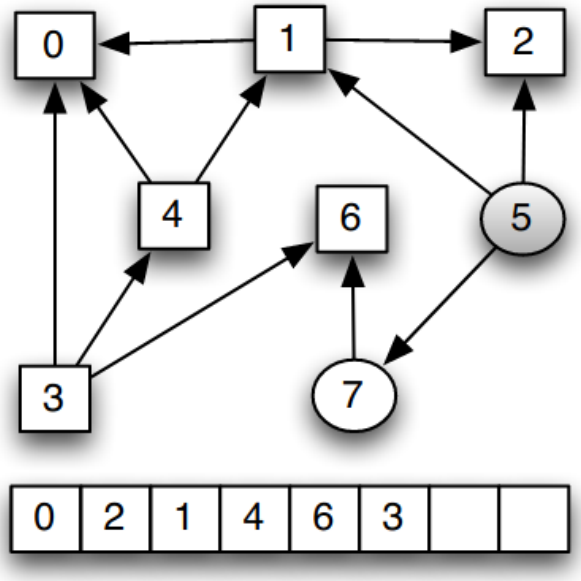
11: Como o vértice 6 não tem sucessores termina a visita: o vértice é marcado como visitado e adicionado à sequência.



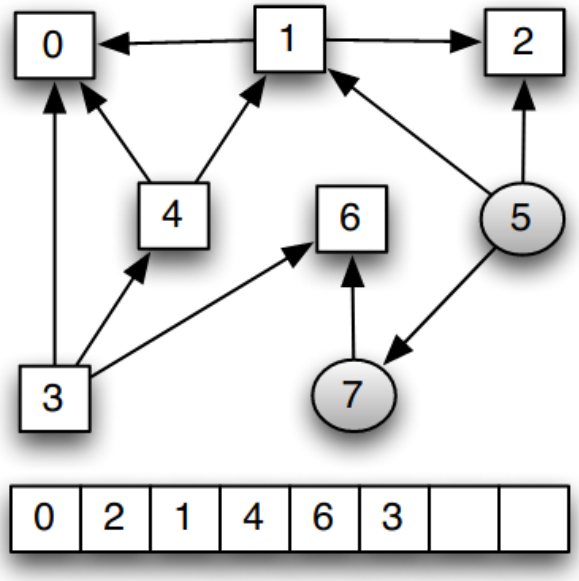
12: Como o vértice 3 não tem mais sucessores termina a visita: o vértice é marcado como visitado e adicionado à sequência.



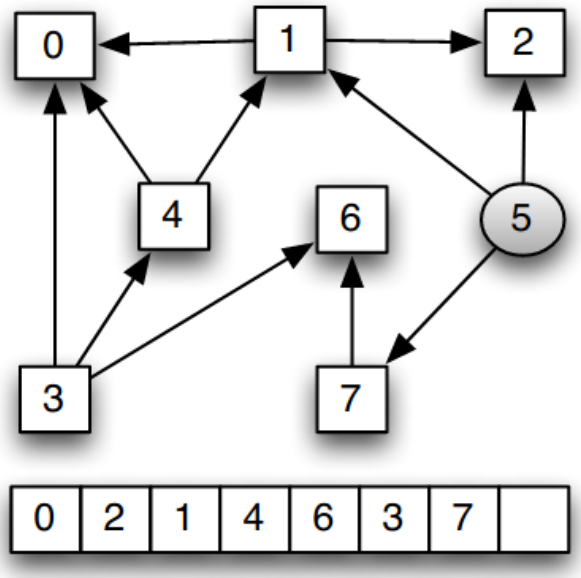
13: Inicia-se uma travessia a partir do vértice 5. Este é marcado como em visita.



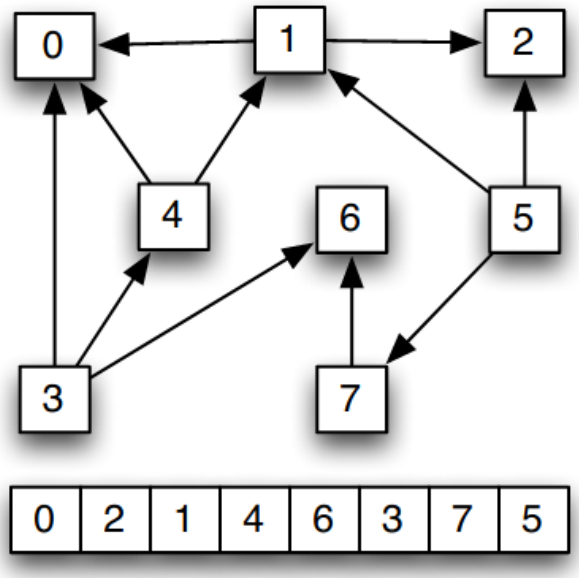
14: Inicia-se uma travessia a partir do vértice 7. Este é marcado como em visita.

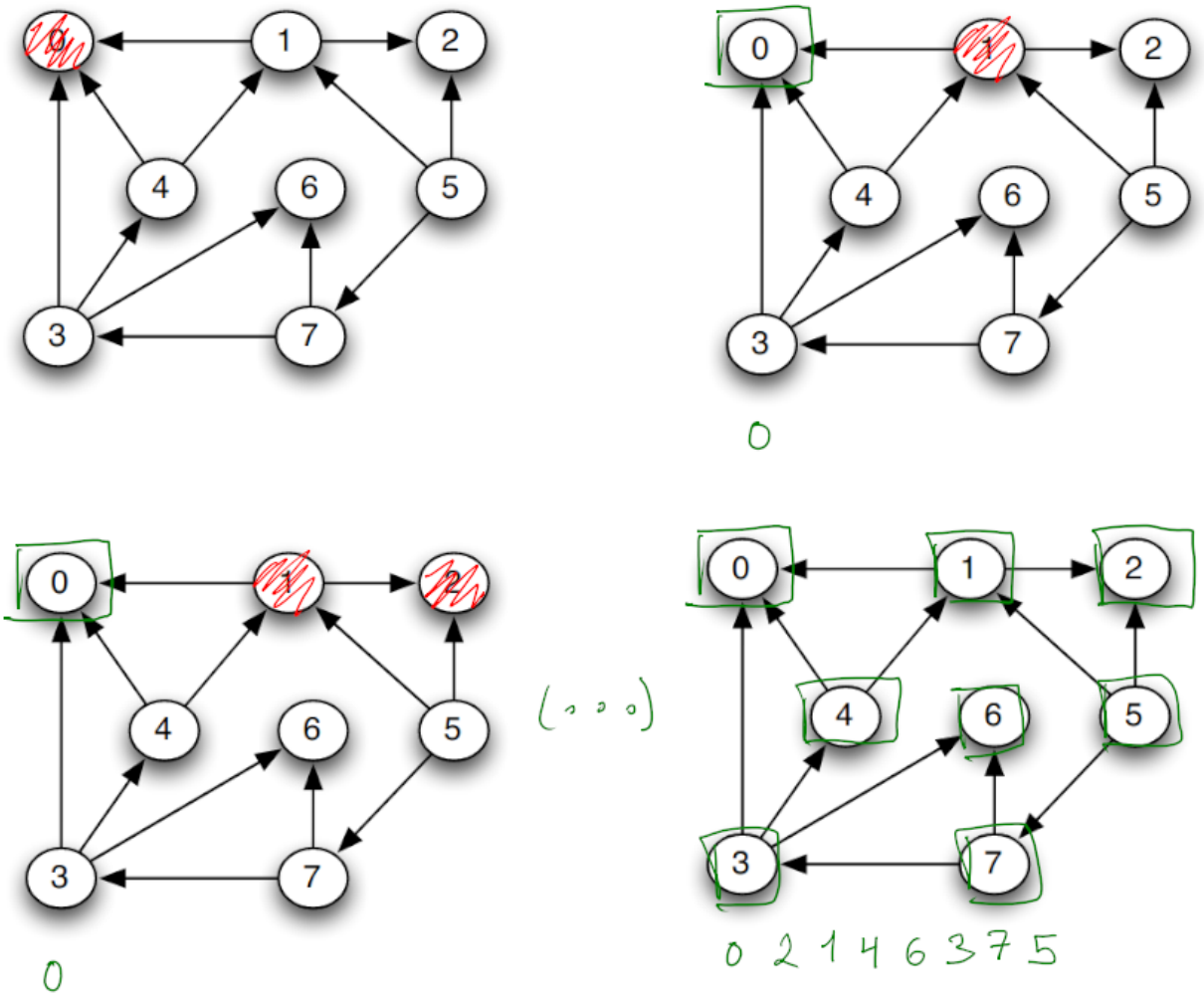


15: Como o vértice 7 não tem sucessores não visitados termina a visita: o vértice é marcado como visitado e adicionado à sequência.



16: Como o vértice 5 não tem sucessores não visitados termina a visita: o vértice é marcado como visitado e adicionado à sequência.





Caminhos mais curtos

Um problema comum em grafos pesados (em que a cada aresta está associado uma distância) consiste em determinar, para um dado par de vértices qual o caminho mais curto (menor distância) que liga esses vértices.

Um problema relacionado, e normalmente com o mesmo custo consiste em determinar, para um dado vértice, os caminhos mais curtos entre esse vértice e todos os seus alcançáveis.

Um terceiro problema consiste em determinar os caminhos mais curtos entre cada par de vértices.
Este terceiro problema pode ser solucionado por sucessivas invocações de uma função que resolva o segundo. Há no entanto formas alternativas e por vezes mais eficientes de o fazer.

Algoritmo de Bellman-Ford

O algoritmo de Bellman-Ford para o cálculo do caminho mais curto entre um vértice v e todos os seus alcançáveis é um exemplo de programação dinâmica.

Baseia-se no facto de que se o caminho

$$[v_0, v_1, \ldots, v_{n-1}, v_n]$$

é o caminho mais curto que liga v_0 a v_n então

$$[v_0, v_1, \ldots, v_{n-1}]$$

é o caminho mais curto que liga v_0 a v_{n-1}.

Para além disso, e a menos que existam ciclos com custo negativo, o caminho mais curto entre dois vértices nunca tem mais do que $NV - 1$ arestas.
Se existirem ciclos com peso negativo, o caminho mais curtoatntre dois vértices pode ser um caminho infinito, pelo que o resultado deste algoritmo pode não ser significativo.

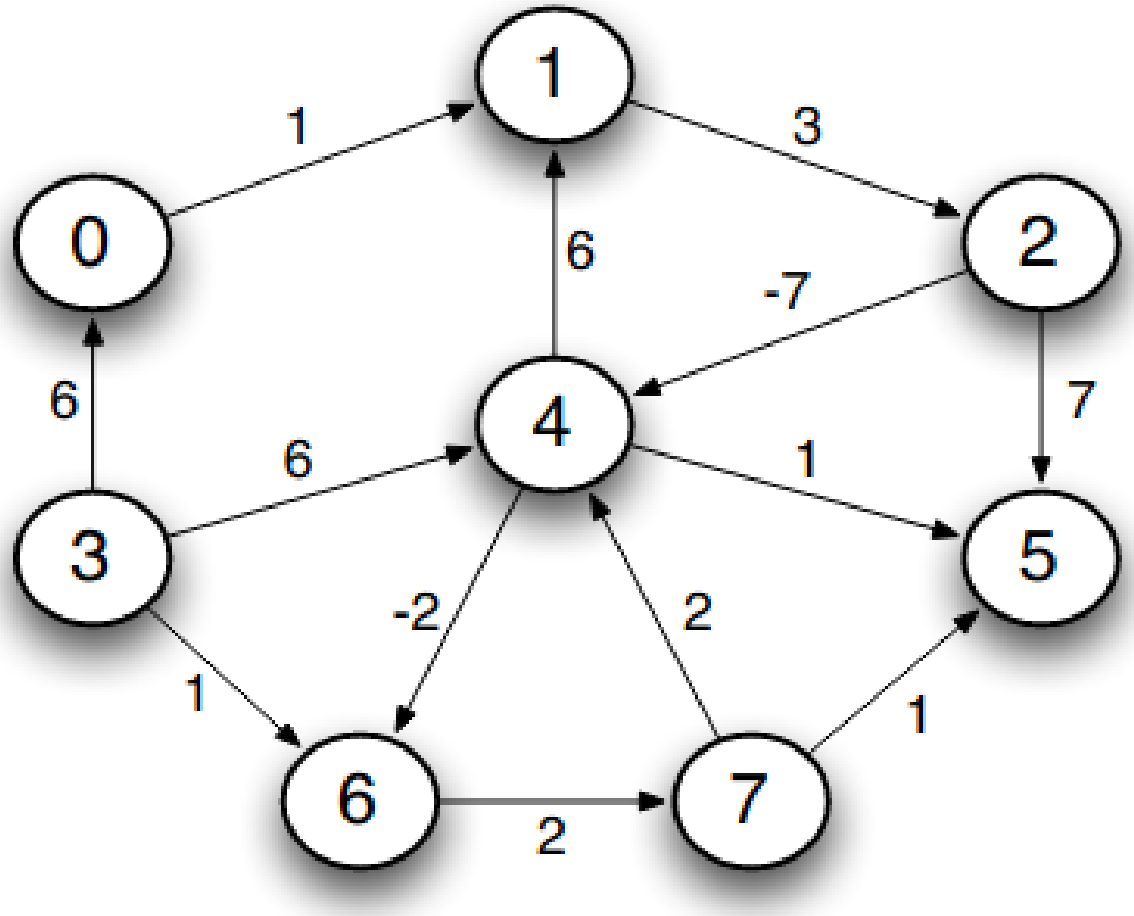
O algortimo itera sobre o número de arestas que compõe os catinhos mais curtos ($i \in [0 \dots N]$), mantendo atualizado um array w de forma a preservar a seguinte propriedade (invariante):

- $w[v] = x$ sse o caminho mais curto que liga o vértice inatial a v usando no máximo i arestas tem peso x.

Para manter válida essa propriedade, cada iteração deve peratrrer todas as arestas e testar se elas podem ser usadas para terminar um caminho associado a v.

```
void bellmanfordSP (Graph g, int o, int w[], int ant[]) {
        int newcost;
        EList it;

        for (int i = 0; i < NV; i++) {
                w[i] = NE; // max
                ant[i] = -1;
        }

        w[o] = 0;
        for (int i = 0; i < NV-1; i++)
                for (int v = 0; v < NV; v++)
                        if (w[v] != NE)
                                for (it = g[v]; it != NULL; it = it->next) {
                                        newcost = w[v] + it->cost;
                                        if (w[it->dest] == NE || newcost < w[it->dest]) {
                                                w[it->dest] = newcost;
                                                ant[it->dest] = v;
                                        }
                                }
}
```

Evolução do algoritmo


1
0
2
4
3
5
6
7
1
3
6
-7
6
7
6
1
-2
2
1
1
2

	0	1	2	3	4	5	6	7
w =	NE	NE	NE	0	NE	NE	NE	NE

	0	1	2	3	4	5	6	7
ant =	-1	-1	-1	-1	-1	-1	-1	-1

1. As únicas arestas consideradas são as que têm origem no vértice 3: $(3,0)$ com custo 6, $(3,4)$ com custo 6 e $(3,6)$ com custo 1. Isto vai alterar os arrays nas posições 0, 4 e 6.

	0	1	2	3	4	5	6	7
w =	6	NE	NE	0	6	NE	1	NE

	0	1	2	3	4	5	6	7
ant =	3	-1	-1	-1	3	-1	3	-1

2. As arestas consideradas são as que têm origem nos vértices 0 ($(0,1)$), 3 ($(1,2)$), 4 ($(4,1)$ e $(4,5)$) e 6 ($(6,7)$). Isto vai alterar os arrays nas posições 1, 2, 5 e 7.

	0	1	2	3	4	5	6	7
w =	6	7	10	0	6	7	1	3

	0	1	2	3	4	5	6	7
ant =	3	0	1	-1	3	4	3	6

3. Todas as arestas são consideradas uma vez que já foi encontrado pelo menos um caminho para todos os vértices.

	0	1	2	3	4	5	6	7
w =	6	7	10	0	3	4	1	3

	0	1	2	3	4	5	6	7
ant =	3	0	1	-1	2	4	3	6

4. Mais uma vez todas as arestas são consideradas.

	0	1	2	3	4	5	6	7
w =	6	7	10	0	3	4	1	3

	0	1	2	3	4	5	6	7
ant =	3	0	1	-1	2	4	3	6

   Apesar de todas as arestas terem sido consideradas não houve qualquer alteração nos arrays `w` e `ant`. Pelo que será este o resultado final

Este exemplo evidencia uma otimização que é costume ser feita no algoritmo de Bellman-Ford. Se uma iteração do ciclo mais exterior não produz quaisquer alterações, a função pode terminar.
Em termos de complexidade, podemos ver que para um grafo com V vértices o ciclo mais exterior faz no máximo $V-1$ iterações. Cada iteração consiste em percorrer todo o grafo. Pelo que, a complexidade desta função é

$$T_{bellmanfordSP}(V,E) = O((V-1)(V+E)) = O(V^2 + V*E)$$

Algoritmo de Dijkstra

Trata-se de um algoritmo *greedy* que só funciona para grafos com pesos não negativos. Uma consequência de não existirem pesos negativos é que ao adicionarmos novas arestas a um caminho o seu custo total nunca diminui.

Tal como o algoritmo de Prim, o algoritmo de Dijkstra particiona o conjunto dos vértices em três zonas:

- Os vértices já processados (BLACK) e para os quais já se conhece o peso do caminho mais curto que os une ao vértice de partida;
- Os vértices ainda não analisados (WHITE);
- A orla (GREY), composta pelos vértices não processados que têm pelo menos uma aresta a ligá-los aos já processados.

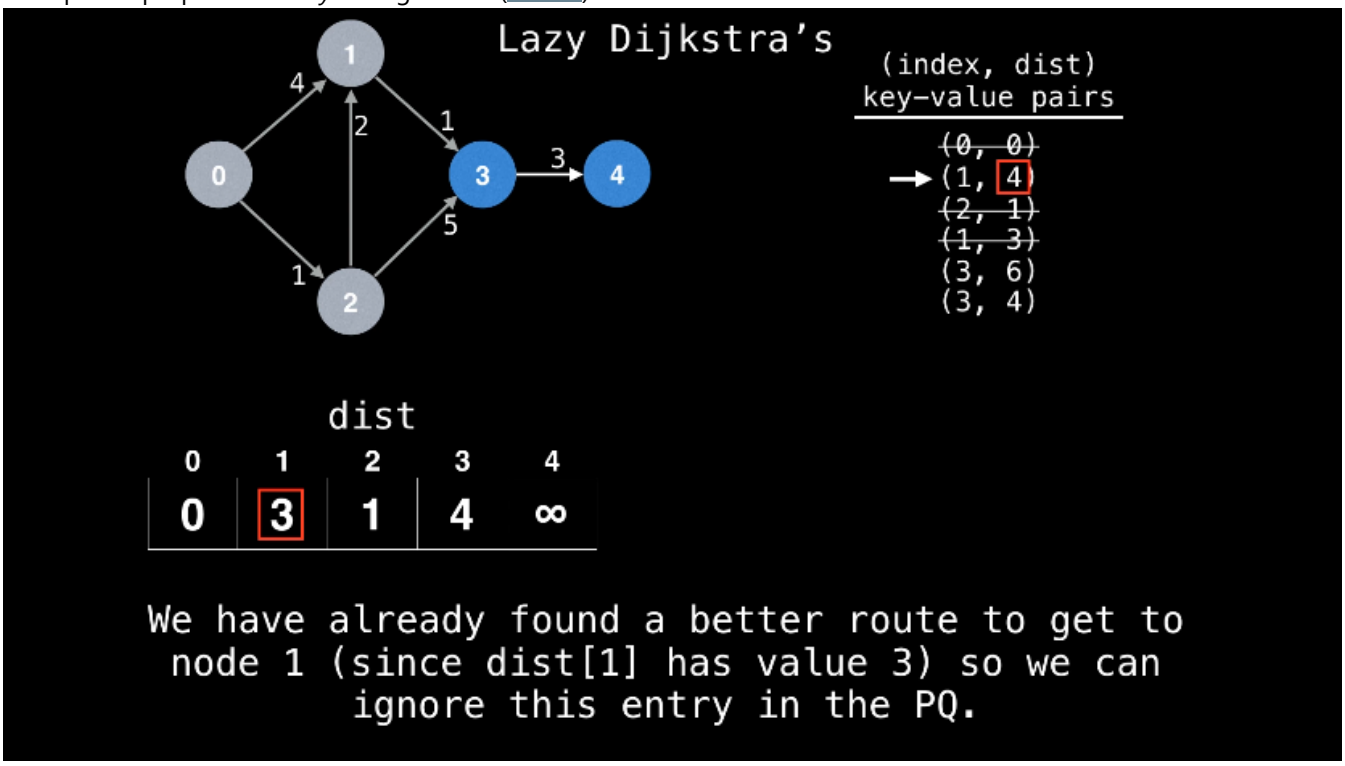
Para cada elemento da orla vamos guardar o custo do melhor caminho conhecido até à altura. Para calcular esse custo, e como cada elemento da orla está ligado a pelo menos um dos vértices já processados, teremos que ter o

mínimo entre a soma das referidas arestas e dos custos das origens dessas arestas.

Tal como acontece com o algoritmo de Prim, o vértice da orla que será selecionado para ser processado será aquele que tem o nenor custo.
O algoritmo termina quando a orla não tiver quaisquer vértices ou quando o vértice for selecionado para sair da orla.

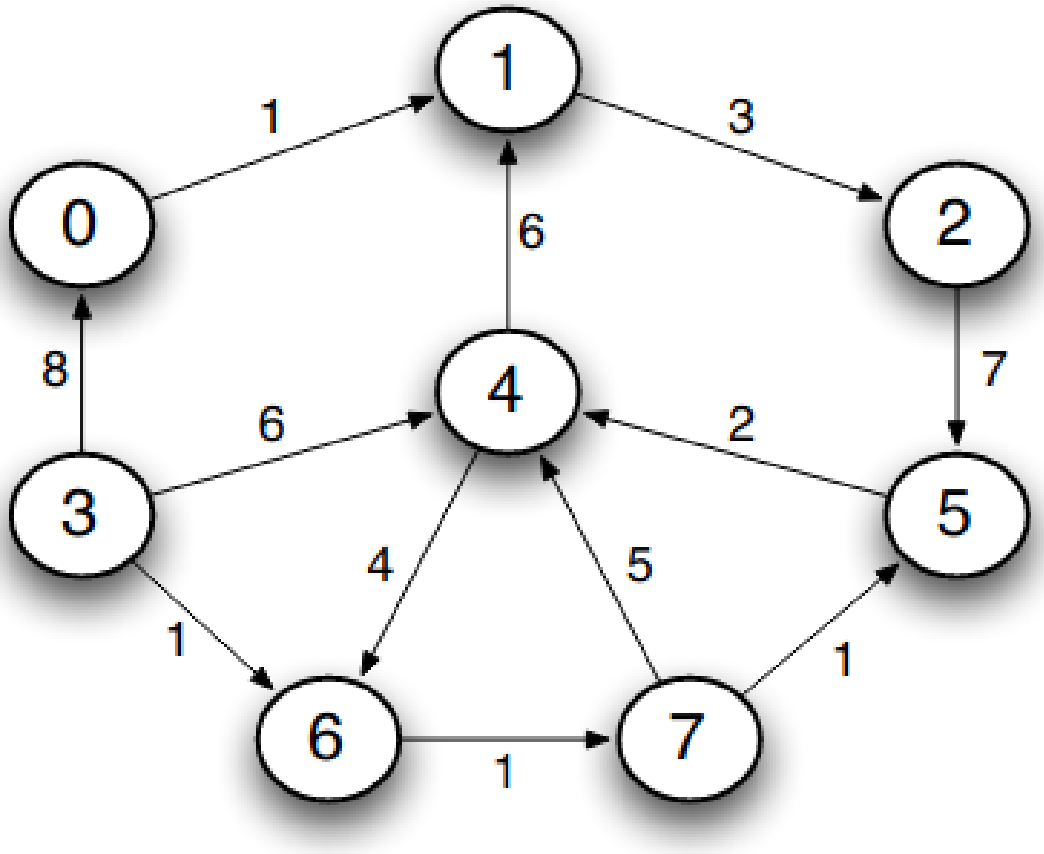
Uma variante deste problema consiste em determinar o caminho mais curto desde um vértice até todos os alcançáveis. Neste caso o algoritmo termina apenas quanso se esgotam os elementos das orla.

Exemplo da propriedade *lazy* do algoritmo: (source)

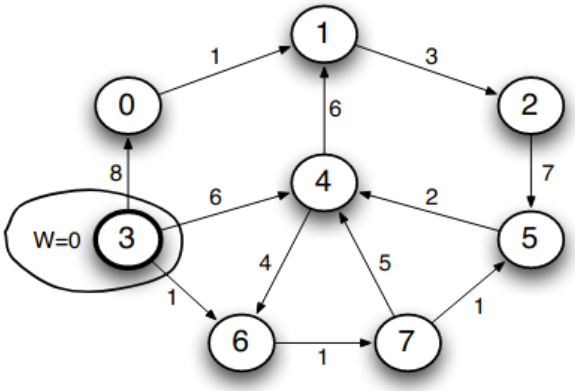


Evolução do algoritmo

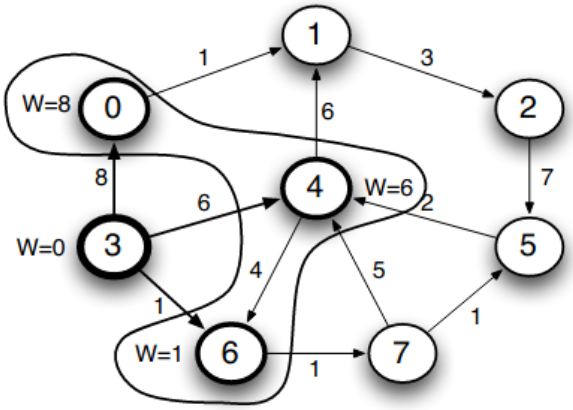
Invocado a partir do vértice 3:



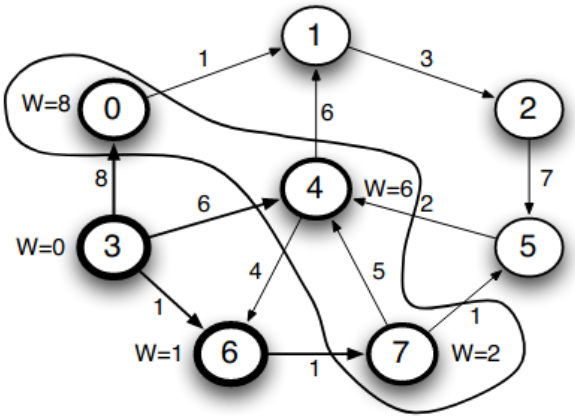
1: A orla é inicializada com o vértice origem. O seu peso é 0 pois é o peso do caminho conhecido. O único vértice da orla é o que será escolhido para ser processado.



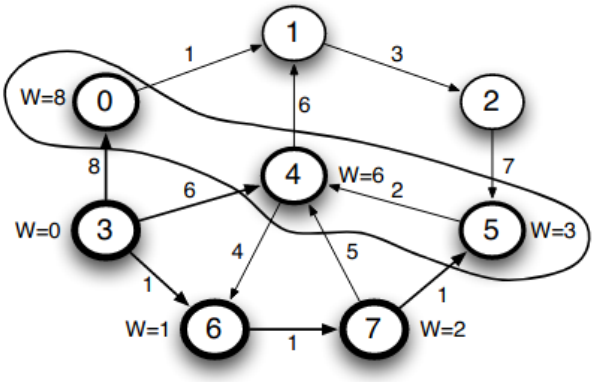
2: Os vértices adjacentes a 3 são adicinados à orla. O peso de cada um obtem-se somando ao peso de 3 (0) ao peso das arestas que os ligam a 3. O próximo vértice a saír da orla será o vértice 6.



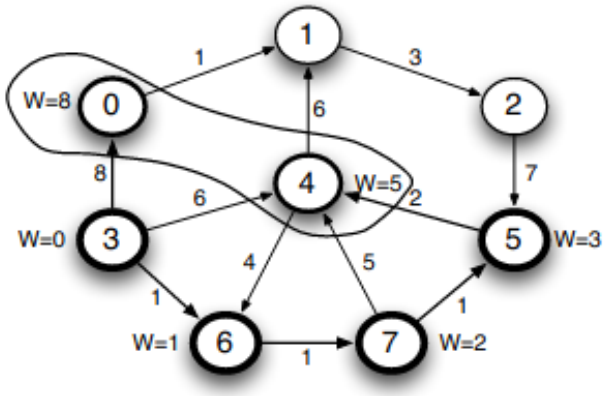
3: O vértice 7 é o único adjacente a 6 eé adicionado à orla. O seu peso calcula-se como o peso de 6 mais o da aresta que liga 6 a 7. O próximo vértice a saír da orla será o vértice 7.



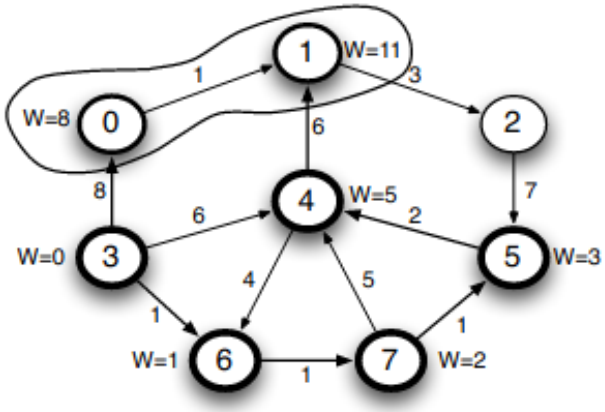
4: Dos vértices adjacentes a 7, o vértice 5 ainda não pertence à orla e por isso é adicionado com peso 3; o vértice 4 já pertence à orla mas o seu peso tem que ser actualizado uma vez que o caminho por que inclui o vértice 7 é de menor custo. O próximo vértice a saír da orla será o vértice 5.



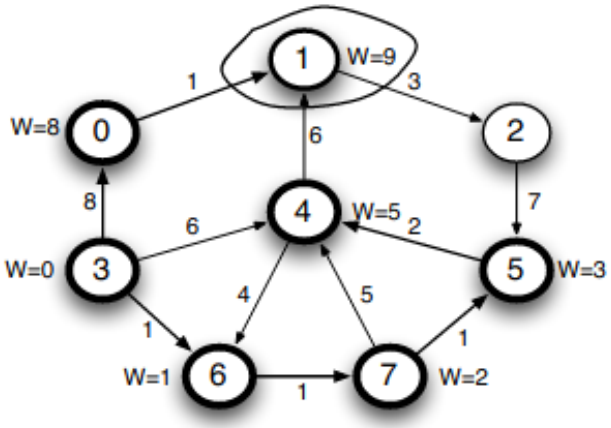
5: O único adjacente ao vértice 5 é o vértice 4 que já está na orla. No entanto o seu peso terá de ser actualizado. O próximo vértice a saír da orla será o vértice 4.



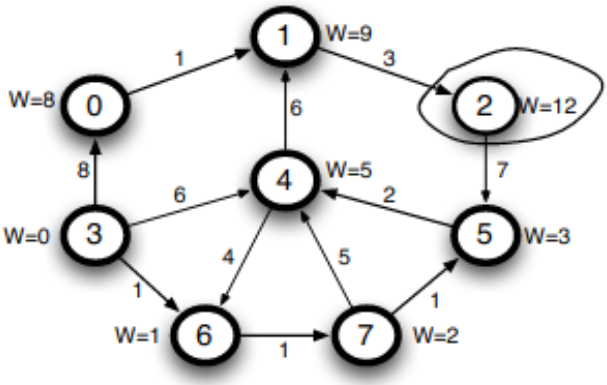
6: O único adjacente ao vértice 4 é o vértice 4 que é acrescentado à orla. O próximo vértice a saír da orla será o vértice 0.



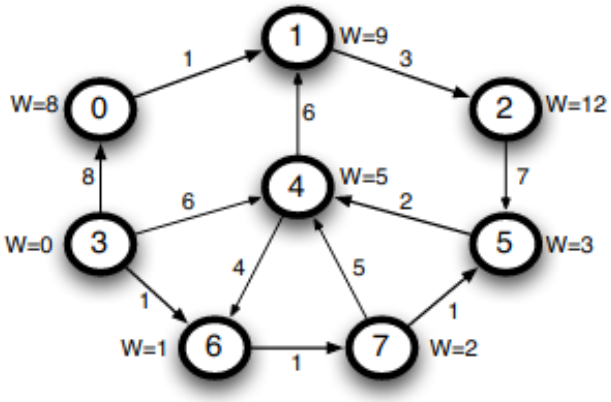
7: O único adjacente ao vértice 0 é o vértice 4 que já está na orla. No entanto o seu peso terá de ser actualizado. O próximo vértice a saír da orla será o único vértice da orla.



8: O único adjacente ao vértice 1 é o vértice 2 que é acrescentado à orla. O próximo vértice a saír da orla será o único vértice da orla.



9: O vértice 2 não tem adjacentes por processar. A orla fica vazia pelo que o algoritmo termina.



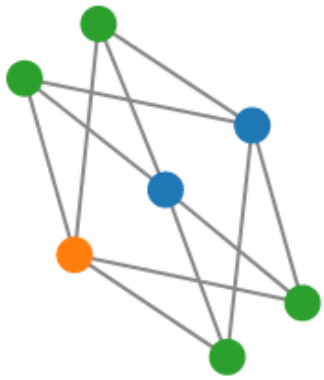
Code

```
int dijkstraSP (Graph g, int v, int cost[], int ant[]) {
        int res = 0;
        int newcost;
        EList it;
        int color[NV];
        struct fringe ff, *f;
```

```
        int fringesize;
        f = &ff;
        for (v = 0; v < NV; color[v++] = 0);
        initFringe(f); fringesize = 0;
        color[v] = 1; cost[v] = 0;
        addEdgeFringe(f, v, cost); fringesize++;
        while(fringesize > 0) {
                v = getEdge(f, cost); fringesize++;
                res++;
                color[v] = 2; // BLACK
                for (it = g[v]; it != NULL; it = it->next) {
                        newcost = cost[v] + it->cost;
                        if ((color[it->dest] == 0) || (color[it->dest]  == 1 && cost[it->dest] >
newcost)) {
                                ant[it->dest] = v;
                                cost[it->dest] = newcost;
                                if (color[it->dest] == 0) {
                                        addEdgeFringe(f, it->dest, cost);
                                        fringesize++;
                                } else updateFringe(f, it->dest, cost);
                        }
                }
        }
        return res;
}
```

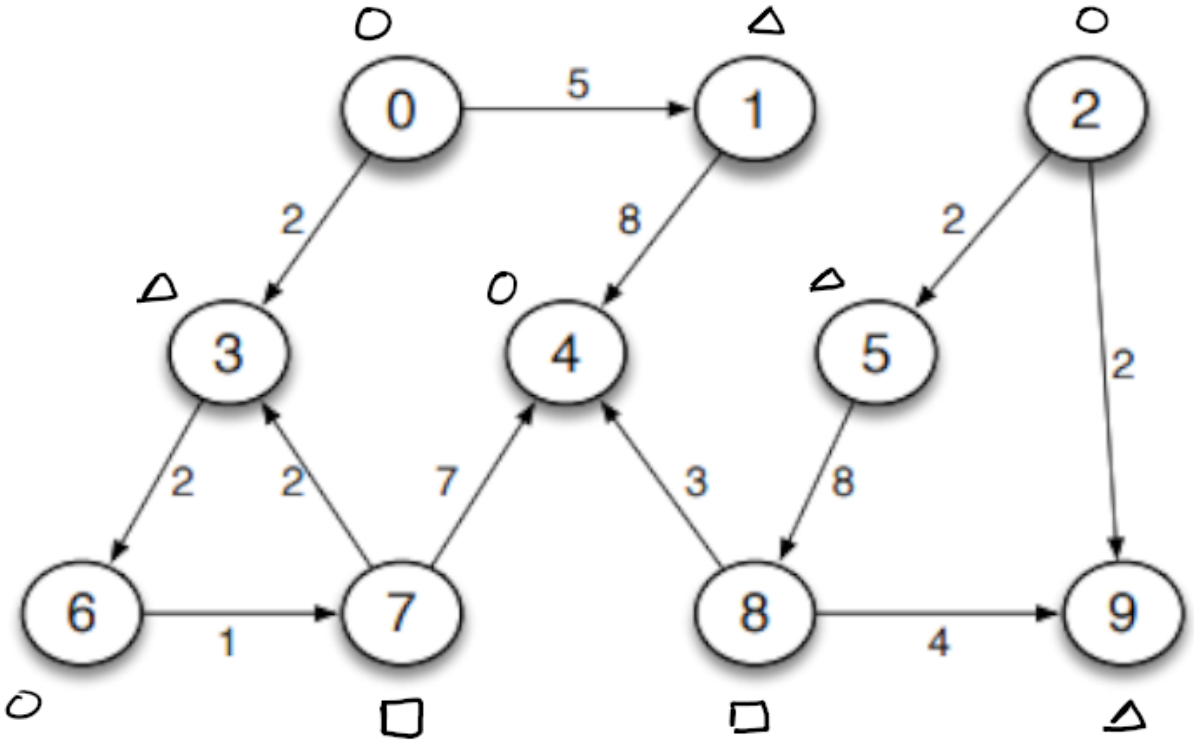
Coloração

Se um grafo é k-colored então o grafo também é $(k+n)$-colored.
(Com $n > 0 \wedge k + n \leq$ nº total de nodos)
Por exemplo:



Este grafo é no mínimo 3-colored, e, por isso, pode ser também no máximo 7-colored (o número total de nodos), ou seja, cada nodo teria uma cor diferente.

Naive approach



(Tomando o grafo como *undirected*)

```
0 -> 1
1 -> 2
2 -> 1
3 -> 2
4 -> 1
5 -> 2
6 -> 1
7 -> 3
8 -> 3
9 -> 2
```

Portanto, a coloração é possível com 3 ou mais cores.

```
// k : number of colors
// i : start
// color : color combination
// Complexidade : O(k^(NV*2))
int Graph_Coloring (GrafoL g, int k, int i, int *color) {
        // Quando o array color está completo
        // Verifica se existem nodos adjacentes com a mesma cor
        // se não existerem a combinação de cores é válida
        if (i == NV)
        if (check_Availability(g, color)) {
                Display_solution(color);
                return 1;
        } else
                return 0;
        // todas a cobinações para o index i
        // e recursivamente para todos os outros
        for (int j = 1; j <= k; j++) {
                color[i] = j;
                if (Graph_Coloring(g, k, i+1, color)) return 1;
                color[i] = 0;
```

```
        }

        return 0;
}
```

Backtracking

Problem set
Introduction
Introduction
https://www.youtube.com/watch?v=DKCbsiDBN6c
https://www.youtube.com/watch?v=gBC_Fd8EE8A