Aula 17 – Busca em Árvores e Algoritmo de Dijkstra

Notas de Aula de Teoria dos Grafos

Prof[a]: Patrícia D. L. Machado

UFCG – Unidade Acadêmica de Sistemas e Computação

Sumário

Busca em Grafos

Busca, **Pesquisa** ou **Travessia** (*search*/*traversal*) em grafos refere-se ao processo de visitar, para verificar ou atualizar, cada vértice de um grafo como parte da resolução de um problema. Essas travessias são realizadas de acordo com uma ordem em que os vértices são visitados. A ordem é definida por um algoritmo.

Busca em árvores é um tipo especial de busca em um grafo qualquer conectado no qual o resultado é uma árvore geradora, ou seja, o objetivo é visitar todos os vértices do grafo exatamente uma vez e a árvore resultante denota todos os caminhos entre um vértice raiz e os demais vértices do grafo.

Existem diferentes tipos de algoritmos ou variantes para realizar busca em árvore, sendo a **Busca em Largura** e a **Busca em Profundidade** os mais conhecidos. Estudaremos estes algoritmos nas próximas aulas.

A JGraphT, através do pacote org.jgrapht.traverse, oferece diferentes algoritmos de busca implementados com *iterators*: a diferença entre eles é a ordem em que os vértices são visitados. Estas implementações não retornam uma árvore explicitamente, mas visitam os vértices de acordo com uma árvore estabelecida pelo algoritmo.

Busca em Árvores

Conceito

Em aulas anteriores, vimos a definição formal de grafo conectado. No entanto, de forma automática, como podemos determinar se um grafo é conectado?

Com base no conceito de busca em árvores, podemos aplicar um algoritmo que determina, por exemplo, se o grafo possui uma árvore geradora. Como vimos em aulas anteriores, um grafo é se e somente se possui uma árvore geradora.

A fim de ilustrar a estratégia geral dos algoritmos de busca para determinar se um grafo é conectado, vamos considerar a Figura 1 abaixo que ilustra o grafo *G*. Considere que as arestas que estão destacadas representam as arestas de um dos seus subgrafos *F*. O grafo *F* é uma árvore? Se sim, *F* é uma árvore geradora?

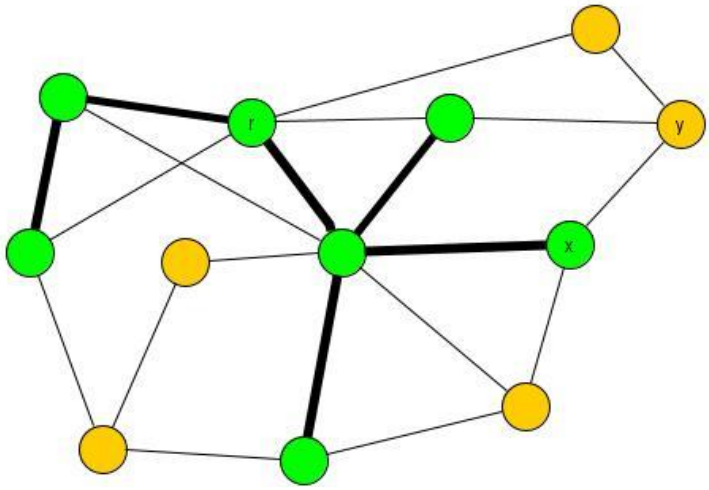


Figura 1

F é uma subárvore de *G*, mas não é uma árvore geradora porque não possui todos os vértices de *G*.

Para todo subgrafo *F* de *G*, definimos como o corte de arestas associado a F, $\partial_G(V(F))$, o conjunto de arestas que relacionam os vértices de *F* com os outros vértices de *G* que não pertencem a *F*. Tais arestas estão representadas de forma tracejada na Figura 2 abaixo.

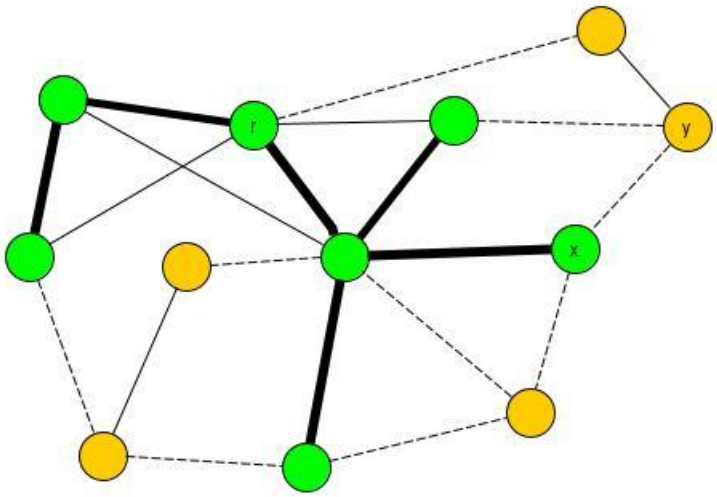


Figura 2

Note que, podemos adicionar qualquer aresta de $\partial_G(V(F))$ em *F* e obteremos como resultado uma árvore, já que estas são exatamente arestas que relacionam vértices de *F* aos demais vértices de *G*. Isto não acontece com as demais arestas do grafo: arestas entre vértices de *F* adicionariam um ciclo em *F* e arestas entre vértices de *G* deixariam *F* desconectado.

Ao adicionar uma aresta de $\partial_G(V(F))$ em *F*, passamos a ter mais um vértice de *G* em *F*. Portanto, se *F* é uma árvore que não é geradora, o corte representa um conjunto de arestas candidatas que podem ser adicionadas a *F* a fim de obtermos uma árvore geradora de *G*.

Desta forma, seja *T* uma subárvore de G. Se V(T) = V(G), então T é uma árvore geradora de G e G é conectado. Neste caso, $\partial_G(V(T)) = \emptyset$. A Figura 3 abaixo ilustra este exemplo.

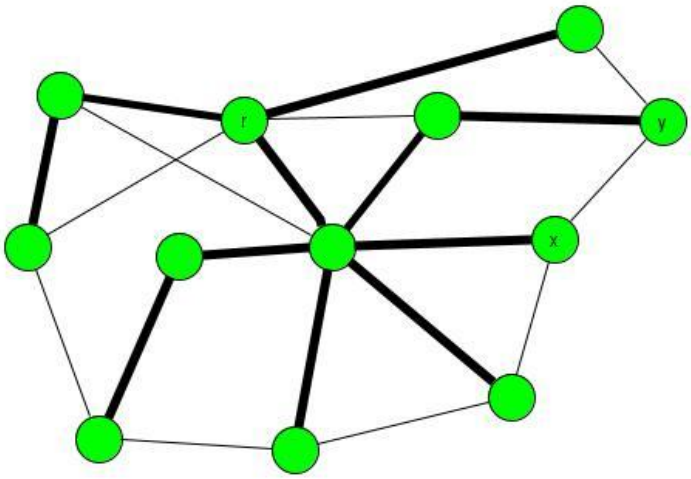


Figura 3

Mas, se $V(T) \subset V(G)$, então duas situações podem ocorrer:

- Se $\partial_G(V(T)) = \emptyset$, então *G* é desconectado. A Figura 4 abaixo ilustra este caso.
- Se $\partial_G(V(T)) \neq \emptyset$, então para todo $xy \in \partial_G(V(F))$, onde *x ∈ V(T)* e *y ∈ V(G)\V(T),* o subgrafo obtido pela adição de *xy* a *T* é uma subárvore em *G*. Este caso está ilustrado na Figura 2.

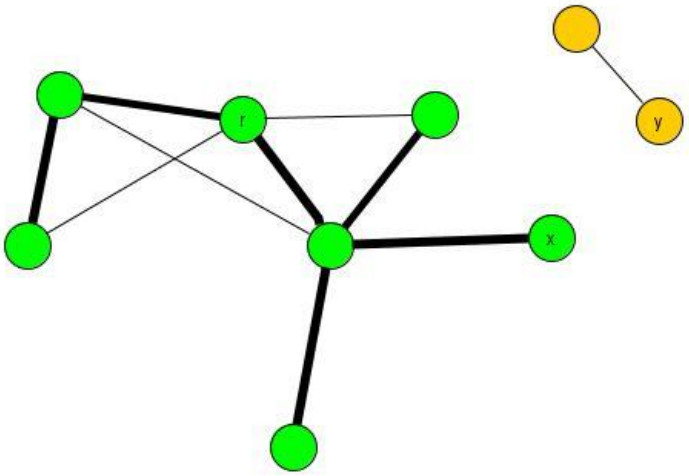


Figura 4

Em resumo, esta é a estratégia básica utilizada na busca em árvores. Partindo da árvore trivial contendo apenas um vértice *r*, constrói-se uma árvore com raiz r através da adição sucessiva de arestas no corte de arestas aplicado aos vértices da árvore até obtermos uma árvore geradora (grafo é conectado) ou não geradora quando um corte vazio for obtido e a árvore não contém todos os vértice do grafo.

Se o objetivo é determinar **se um grafo é conectado**, então qualquer algoritmo de busca em árvore pode ser usado. No entanto, quando a **ordem** é **considerada**, informações diferentes podem ser produzidas:

- Busca em **Largura** identifica **distâncias** (caminhos de menor tamanho) entre o vértice *r* e os demais vértices do grafo.
- Busca em **Profundidade** identifica **cortes de vértices** do grafo.

Definições Básicas

Vamos agora introduzir alguns conceitos básicos em árvores enraizadas.

Seja *T* uma r-árvore. O **nível** (*level*) de um vértice *v* em *T* é o tamanho do caminho (*r,...,v)*. Na árvore apresentada na Figura 5 abaixo, o nível do vértice r, a raiz da árvore é 0 (por *default*), enquanto que o nível do vértice *f* é 2, tamanho do caminho (r,c,f).

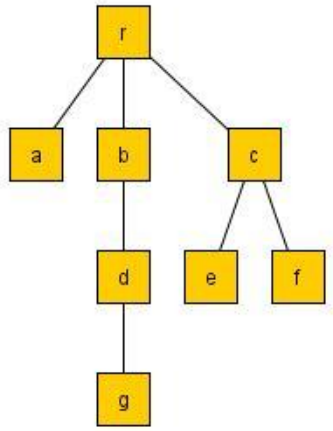


Figura 5

Todo vértice no caminho (*r,...,v)* é chamado de ***ancestral*** *(ancestor)* de *v* e todo vértice do qual *v* é ancestral é chamado de **descendente** de *v*. Na Figura 5, os vértices *r* e *b* são ancestrais de *d* e o vértice *g* é descendente de *d*.

O ancestral imediato é chamado de **predecessor ou pai** (*parent*), denotado por *p(v)*. E os vértices para os quais *v* é pai são chamados de **sucessores ou filhos**. Na Figura 5, o pai de *d* é *b* e o filho de *d* é *g*.

Como discutimos em aulas anteriores, árvores são grafos não-direcionados. No entanto, quando definimos uma raiz, estabelecemos uma orientação no grafo que define os caminhos da raiz para todos os outros vértices. Desta forma, podemos definir uma árvore com raiz, através de um conjunto de pares ordenados usando a função predecessor de um vértice. Em cada par, o primeiro elemento p(v) é o predecessor imediato do segundo elemento *v* na árvore.

$$E(T) = \{(p(v), v) : v \in V(T) \setminus \{r\}\}$$

A árvore da Figura 5 pode ser definida como:

$$E(T) = \{(r,a),(r,b),(r,c),(b,d),(d,g),(c,e),(c,f)\}$$

onde p(*a*) = *r*, p(*b*) = *r*, p(*c*) = *r*, p(*d*) = *b*, p(*g*) = *d*, p(*e*) = *c*, p(*f*) = c.

Algoritmo de Dijkstra

O algoritmo de Dijkstra, proposto em 1956, tem como objetivo encontrar o menor caminho entre dois vértices de um grafo ponderado (*shortest path*). Porém, uma de suas variações calcula uma árvore de caminhos menores de um vértice *v* para os demais vértices do grafo. Estudaremos esta versão.

A estratégia básica é a seguinte. Dado um certo vértice *v* como origem, o objetivo é encontrar o menor caminho entre *v* e todos os seus vértices adjacentes $(u_0, \cdots, u_k)$. Em seguida, para o vértice u_i com menor distância para v que ainda não foi visitado, o objetivo é encontrar o menor caminho entre este vértice e cada um de seus vértices adjacentes. E assim sucessivamente até visitar todos os vértices. Note que este caminho mais curto não é necessariamente o que segue a aresta onde os vértices são terminais, mas pode ter sido encontrado em iterações anteriores do algoritmo.

O algoritmo está ilustrado abaixo. O algoritmo recebe como entrada um grafo, *Graph*, e um vértice origem, *source*. Das linhas 3-10, as variáveis a serem utilizadas para armazenar os resultados parciais e finais são inicializadas. São elas:

- **Q** – conjunto de vértices ainda não visitados. Inicialmente, contém todos os vértices;
- **dist** – função que mapeia para cada vértice do grafo a distância entre este e o vértice *source*. Inicialmente esta distância é definida como infinita (*INFINITY*), exceto para o vértice source (linha 10);
- **pred** – função que determina o predecessor de um vértice v na árvore resultante. Inicialmente, o predecessor de cada vértice é indefinido (*UNDEFINED*).

```
1  function Dijkstra(Graph, source):
3      create vertex set Q
5      for each vertex v in Graph:           // Inicialização
6          dist[v] ← INFINITY                // Distância não conhecida entre v e source
7          pred[v] ← UNDEFINED               // Predecessor de v em um caminho ótimo a partir de source
8          add v to Q    // Todos os vértices são inicialmente adicionados a Q (vértices não-visitados)
10     dist[source] ← 0                      // Distância de source para source
12     while Q is not empty:
13         u ← vertex in Q with min dist[u]
                                 // Vértice com menor distância para source é selecionado primeiro
15         remove u from Q
17         for each neighbor v of u:         // Todos os vértices ainda em Q.
18             alt ← dist[u] + weight(u, v)
19             if alt < dist[v]:             // Um caminho menor para v foi encontrado
20                 dist[v] ← alt
21                 pred[v] ← u
23     return dist[], pred[]
```

Da linha 12 a linha 21, é realizada a travessia no grafo enquanto o conjunto de vértices não visitados, **Q**, não for vazio. Na linha 13, é escolhido um vértice *u* para ser visitado. Este é o vértice com menor valor de distância em **dist** que ainda não foi visitado (pertence a **Q**). O vértice *u* é removido de **Q** porque será visitado (linha 15). Para cada vizinho *v* de *u*, *alt* é calculado como a

distância de *source* para *u* (**dist**[*u*]) somada ao peso da aresta entre *u* e *v* (linha 18). Note que o valor de *alt* é a distância entre *source* e *v* passando por *u*.

Na linha 19, se *alt* < **dist**[*v*], onde **dist**[*v*] é a distância calculada até o momento entre *source* e *v*, então um menor caminho para *v* foi encontrado. Neste caso, **dist**[*v*] passa a ser *alt* e **pred**[*v*] passa a ser *u* (o caminho anterior é abandonado porque um menor foi encontrado; linhas 20,21).

O algoritmo retorna as funções **dist**, com as distâncias entre *source* e os demais vértices, e **pred** onde a árvore está representada.

Vejamos um exemplo sobre o grafo abaixo na Figura 6 considere o vértice Frankfurt (Fra abreviado) como origem.

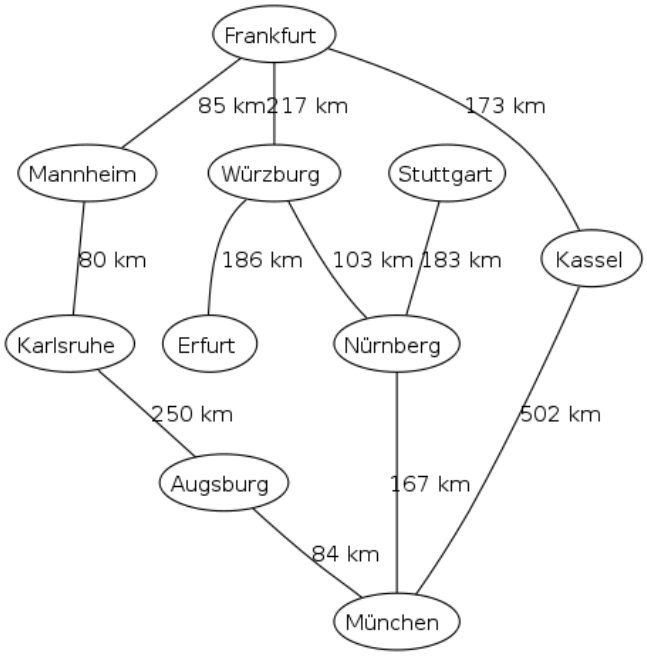


Figura 6[1]

Inicialmente, as variáveis do algoritmo são inicializadas tal como na tabela abaixo.

Q	Fra, Man, Wur, Stu, Kar, Erf, Nur, Kas, Aug, Mun
dist	**Fra→0**, Man→∞, Wur→∞, Stu→∞, Kar→∞, Erf→∞, Nur→∞, Kas→∞, Aug→∞, Mun→∞
prev	Fra→_, Man→_, Wur→_, Stu→_, Kar→_, Erf→_, Nur→_, Kas→_, Aug→_, Mun→_

Na primeira iteração, o vértice *Fra* é escolhido por ter o menor valor em **dist**. O valor de **dist** é calculado para seus vizinhos, *Man*, *Wur* e *Kas* e cada um deles passa a ter *Fra* como predecessor

[1] Imagem de http://en.wikipedia.org/wiki/Breadth-first_search

já que o valor que possuíam antes em **dist** era infinito. Assim, as variáveis passam a conter os seguintes valores. Note que *Fra* é removido de **Q**.

Q	Man, Wur, Stu, Kar, Erf, Nur, Kas, Aug, Mun
dist	Fra→0, **Man→85**, **Wur→217**, Stu→∞, Kar→∞, Erf→∞, Nur→∞, **Kas→173**, Aug→∞, Mun→∞
prev	Fra→_, **Man→Fra**, **Wur→Fra**, Stu→_, Kar→_, Erf→_, Nur→_, **Kas→Fra**, Aug→_, Mun→_

Na próxima iteração, o vértice em **Q** com menor valor em **dist** é *Man.* O valor de *alt* é calculado para seus vizinhos, *Kar* (85+80) e *Fra* (85+85). Apenas o valor de *alt* para *Kar* é menor que o valor em **dist**. Assim, este passa a ter Man como predecessor e *alt* será o seu novo valor em **dist**. Note que *Man* é removido de **Q**.

Q	Wur, Stu, Kar, Erf, Nur, Kas, Aug, Mun
dist	Fra→0, Man→85, Wur→217, Stu→∞, **Kar→165**, Erf→∞, Nur→∞, Kas→173, Aug→∞, Mun→∞
prev	Fra→_, Man→Fra, Wur→Fra, Stu→_, **Kar→Man**, Erf→_, Nur→_, Kas→Fra, Aug→_, Mun→_

Na próxima iteração, o vértice em **Q** com menor valor em **dist** é *Kar.* O valor de *alt* é calculado para seus vizinhos, *Man* (165+80) e *Aug* (165+250). Apenas o valor de *alt* para *Aug* é menor que o valor em **dist**. Assim, este passa a ter *Kar* como predecessor e *alt* será o seu novo valor em **dist**. Note que *Kar* é removido de **Q**.

Q	Wur, Stu, Erf, Nur, Kas, Aug, Mun
dist	Fra→0, Man→85, Wur→217, Stu→∞, Kar→165, Erf→∞, Nur→∞, Kas→173, **Aug→415**, Mun→∞
prev	Fra→_, Man→Fra, Wur→Fra, Stu→_, Kar→Man, Erf→_, Nur→_, Kas→Fra, **Aug→Kar**, Mun→_

Na próxima iteração, o vértice em **Q** com menor valor em **dist** é *Kas.* O valor de *alt* é calculado para seus vizinhos, *Fra* (173+173) e *Mun* (173+502). Apenas o valor de *alt* para *Mun* é menor que o valor em **dist**. Assim, este passa a ter *Kas* como predecessor e *alt* será o seu novo valor em **dist**. Note que *Kas* é removido de **Q**.

Q	Wur, Stu, Erf, Nur, Aug, Mun
dist	Fra→0,Man→85, Wur→217, Stu→∞, Kar→165, Erf→∞, Nur→∞, Kas→173, Aug→415, **Mun→675**
prev	Fra→_, Man→Fra, Wur→Fra, Stu→_, Kar→Man, Erf→_, Nur→_, Kas→Fra, Aug→Kar, **Mun→Kas**

De forma similar segue a visita aos vértices Wur. Resultando no status abaixo para as variáveis.

Q	Stu, Erf, Nur, Aug, Mun
dist	Fra→0, Man→85, Wur→217, Stu→∞, Kar→165, **Erf→403**, **Nur→320**, Kas→173, Aug→415, Mun→675

prev	Fra→_, Man→Fra, Wur→Fra, Stu→_, Kar→Man, **Erf→Wur**, **Nur→Wur**, Kas→Fra, Aug→Kar, Mun→Kas

Na próxima iteração, o vértice em **Q** com menor valor em **dist** é *Nur.* O valor de *alt* é calculado para seus vizinhos, Wur (320+103), *Stu* (320+183) e *Mun* (320+167). Note que o valor de *alt* para *Mun* é menor que o valor em **dist** (considerando o caminho anterior vindo por Kas). Assim, este passa a ter *Nur* como predecessor e *alt* será o seu novo valor em **dist**. *Nur* é removido de **Q**.

Q	Stu, Erf, Aug, Mun
dist	Fra→0,Man→85, Wur→217, **Stu→503**, Kar→165, Erf→403, Nur→320, Kas→173, Aug→415, **Mun→~~675~~487**
prev	Fra→_, Man→Fra, Wur→Fra, **Stu→Nur**, Kar→Man, Erf→Wur, Nur→Wur, Kas→Fra, Aug→Kar, **Mun→~~Kas~~Nur**

A visita aos vértices *Erf*, *Aug*, *Mun* e *Stu* segue de forma similar e no final obtemos os seguintes valores para as variáveis que são retornadas pelo algoritmo.

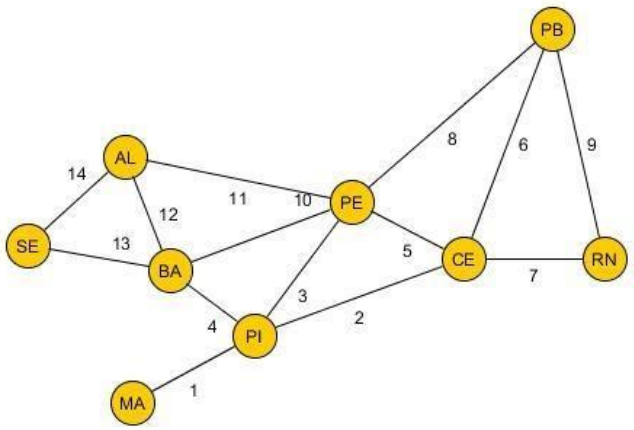
Q	
dist	Fra→0,Man→85, Wur→217, Stu→503, Kar→165, Erf→403, Nur→320, Kas→173, Aug→415, Mun→487
prev	Fra→_,Man→Fra, Wur→Fra, Stu→Nur, Kar→Man, Erf→Wur, Nur→Wur, Kas→Fra, Aug→Kar, Mun→Nur

A JGraphT implementa o algoritmo de Dijkstra através da classe **DijkstraShortestPath**. Nesta classe podemos encontrar o método getPath que retorna o menor caminho entre 2 vértices e o método getPaths que retorna os menores caminhos de um vértice para todos os outros do grafo.

Exercícios Propostos

1. Seja T uma sub-árvore de um grafo G. Mostre que, se $V(T) \subset V(G)$ e $\partial G(V(T)) = \emptyset$, então G é desconectado.

2. Para o grafo abaixo, encontre o menor caminho entre o vértice PE e os demais usando o algoritmo de Djikstra.



3. Considere o algoritmo de Djikstra. Apresente um exemplo onde o algoritmo pode retornar caminhos diferentes em diferentes execuções. Justifique.

4. Considere o algoritmo de Dijkstra. É verdade que o menor caminho entre dois vértices é sempre aquele de menor tamanho, ou seja, com o menor número de arestas possível? Justifique.

5. Considere o problema de encontrar o maior caminho entre dois vértices. Como o algoritmo de Djikstra pode ser adaptado para resolver este problema?

Referências

J. A. Bondy and U. S. R. Murty. Graph Theory. Springer, 2008,2010.

- 4.1, 4.2 (excluindo Cayley's Formula), 4.3 (apenas a definição de co-árvore)
- 4.1 (excluindo Binary Tree), 4.2, 4.3
- 6.3

https://en.wikipedia.org/wiki/Graph_traversal

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm