1. **Preâmbulo**

   Na prática de hoje vamos nos interessar no problema de cálculo do caminho mais curto em um grafo. Este problema aparece por exemplo em softwares de GPS que devem calcular itinerários nas estradas. Quando os grafos são grandes, é necessário utilizar algoritmos com boa performance para calcular os caminhos mais curtos em tempos aceitáveis para o usuário: este é o caso das malhas viárias, que correspondem a grafos com várias centenas (até mesmo milhões) de nós.

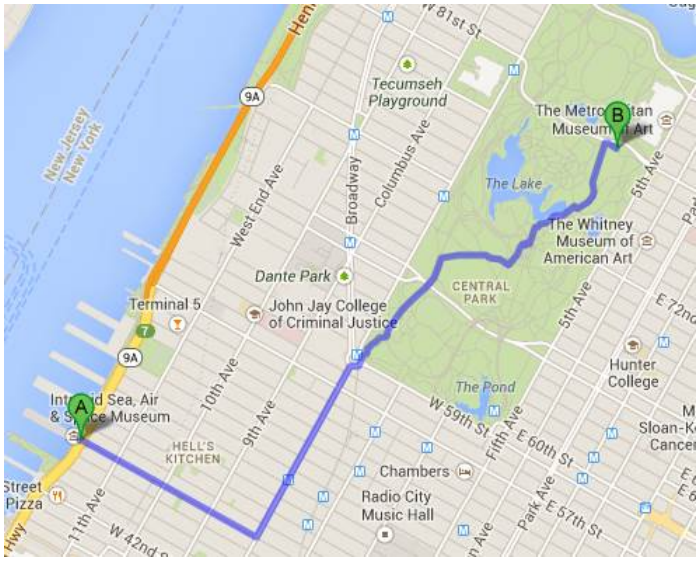


Figure 1: Itinerário calculado pelo Google Maps entre o *Metropolitan Museum of Art* e o *Intrepid Sea, Air & Space Museum* em Nova Iorque

Nós iremos implementar o algoritmo de Dijsktra. Este algoritmo se aplica a grafos conexos cujos pesos nas arestas são positivos ou nulos.

2. **A classe `Graph`**

   Nós vamos trabalhar sobre grafos orientados onde o peso dos arcos são inteiros positivos. Por convenção, os nós serão os inteiros $0, 1, \ldots, n-1$. A classe `Graph` é fornecida com os atributos e métodos públicos a seguir:

   - um atributo inteiro `n` que indica o número de nós.
   - um atributo inteiro `m` que indica o número de arcos.
   - um construtor `Graph(String file)` que constrói um grafo a partir do arquivo `file`.
   - um método `ArrayList successors(int i)` que retorna a lista de sucessores do nó `i`.

- um método `ArrayList predecessors(int i)` que retorna a lista de predecessores do nó `i`.
- um método `int value (int i, int j)` que retorna o peso do arco (i, j) se ele existe, e 0 c.c.

Observação: Você não precisará entender a codificação da classe `Graph`, a descrição acima deve ser suficiente para que você faça a sua prática computacional.

Faça agora no SIGAA o download dos arquivos:

- `sources.zip` contendo:
  - o esqueleto da classe `Dijkstra` a ser completada na prática
  - as classes `Graph`, `Node` e `Arc` (para a modelização dos grafos) e as classes `Fenetre`, `ColoredPoint2D` e `ColoredSegment2D` (para visualização), que não serão modificadas.
- `data.zip` contendo os dados combinatórios e geográficos de algumas malhas viárias.

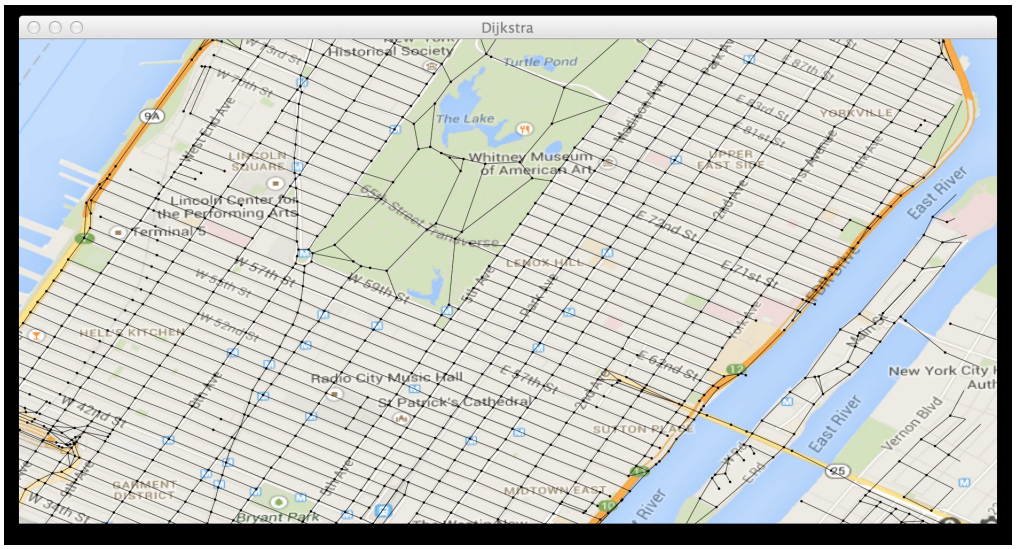
Teste as classes fornecidas com o arquivo `Test1.java`. Você deverá ver o resultado abaixo:

```
Teste 1 : teste da classe Graph
Carregando rede viaria do arquivo mini.gr ... done
n=9
m=16
Node 0 :
   0 - 1 (4)
   0 - 2 (7)
   0 - 4 (10)
Node 1 :
   1 - 3 (5)
   1 - 4 (7)
Node 2 :
   2 - 4 (2)
   2 - 5 (6)
Node 3 :
   3 - 7 (7)
   3 - 6 (5)
Node 4 :
   4 - 6 (8)
   4 - 3 (3)
   4 - 5 (4)
Node 5 :
   5 - 6 (4)
   5 - 8 (2)
```

```
Node 6 :
   6 - 7 (1)
   6 - 8 (5)
Node 7 :
Node 8 :

Carregando rede viaria do arquivo USA-road-d-NY.gr ... done
Carregando coordenadas geometricas do arquivo USA-road-d-NY.co ... done
Carregando imagem de background do arquivo NY_Metropolitan.png ...  done
```

e o mapa seguir:



das classe `Graph` do seu SIGAA assim como dos dois grafos a seguir: `mini.gr` e `rome99.gr`.

3. **Ao trabalho! Façamos Dijkstra!**

   Nosso problema consiste em calcular o comprimento do caminho mais curto entre um nó s e um nó t em um grafo orientado. Para isto, nós vamos implementar o algoritmo de Dijsktra.

   A classe `Dijkstra` fornecida contem os atributos a seguir:

   - um atributo `final Graph g`, contendo o grafo;
   - um atributo `final int n`, contendo o número de nós do grafo `g`;
   - um atributo `final int s`, contendo o nó de partida do caminho mais curto procurado;
   - um atributo `final int t`, contendo nó de destino do caminho mais curto procurado.

- um atributo `Fenetre f` necessário para a apresentação gráfica. Você não deverá se preocupar com a inicialização deste campo.

O esqueleto da classe `Dijsktra` contem um construtor `Dijkstra(Graph g, int source, int dest)`.

Lembre-se de como funciona o algoritmo de Dijsktra através do pseudocódigo abaixo:

```
Dijkstra(G,s,t) :

  Para cada nó i em G
     dist[i] = INFINITO

  dist[s] = 0
  S = {s}

  Enquanto S não vazio
    x = RetireMenorDist(S)
    Se x == t
       Retorne dist[x]
    Para cada sucessor v de x
       Atualize(v,x,G,S)

  Retorne -1
```

Se o algoritmo retorna -1, significa que não existe caminho entre os nós s e t, o que não deve ocorrer na prática de hoje.

A função intermediária `Atualize` funciona da seguinte maneira:

```
Atualize(v,x,G,S) :

  Se dist[v] > dist[x] + G.value(x,v)
    dist[v] = dist[x] + G.value(x,v)
    S = S + {v}
```

Este algoritmo tem portanto necessidade de uma estrutura de dados para memorizar os comprimentos dos caminhos mais curtos encontrados a partir do nó s. É o vetor `dist` da descrição acima. Adicione um campo `int[] dist` à classe `Dijkstra`. O valor `dist[i]` representará o comprimento do caminho mais curto encontrado até então entre o nó s e o nó `i`. No começo do algoritmo, o nó s está a uma distância 0 do nó s, todos os outros nós devem estar a uma distância infinita, uma vez que ainda não exploramos os caminhos. Na prática, você poderá utilizar um número suficientemente grande para representar a distância infinita (por exemplo: `Integer.MAX_VALUE`). Atualize o construtor neste sentido.

4. **Informações complementares a serem guardadas ao longo do algoritmo**

   Por enquanto, nós guardamos as seguintes informações:

   - O conjunto S que contem o conjunto de nós atingidos mais ainda não acomodados. Um nó é considerado atingido se pelo menos um caminho foi encontrado entre s e o nó.
   - O vetor de distâncias `dist` que indica a distância mínima encontrada entre s e cada nó do grafo.

   Nós vamos adicionar as informações a seguir, que serão mantidas durante a execução do algoritmo:

   - um vetor de inteiros `pred` de tamanho n, onde `pred[i]` indicará o predecessor do nó `i` no caminho mais curto obtido entre s e `i`, ou -1 se nenhum caminho ainda foi encontrado até `i`;
   - um vetor de booleanos `settled` de tamanho n, onde `settled[i]` será `true` se o nó `i` foi acomodado pelo algoritmo, e `false` se o nó ainda for suscetível a mudar sua distância até s.

   Agora para verificar que um nó `i` foi atingido basta verificar o conteúdo de `pred[i]`, certo?

   Para o nó s de partida, nós vamos convencionar que inicialmente ele é o seu próprio predecessor. Convencione também que `settled[s] = false` no começo do algoritmo.

   Adicione estes campos à classe `Dijkstra`, e modifique o construtor de modo que estes campos sejam corretamente inicializados.

   A fim de verificar a sua classe `Dijkstra`, execute o arquivo `Test2.java`. Você deverá obter o seguinte resultado:

```
Teste 2 : inicialização da classe Dijkstra
Carregando rede viaria do arquivo mini.gr ... done
---Grafo---
n=9
m=16
Node 0 :
   0 - 1 (4)
   0 - 2 (7)
   0 - 4 (10)
Node 1 :
   1 - 3 (5)
   1 - 4 (7)
Node 2 :
   2 - 4 (2)
   2 - 5 (6)
```

```
Node 3 :
   3 - 7 (7)
   3 - 6 (5)
Node 4 :
   4 - 6 (8)
   4 - 3 (3)
   4 - 5 (4)
Node 5 :
   5 - 6 (4)
   5 - 8 (2)
Node 6 :
   6 - 7 (1)
   6 - 8 (5)
Node 7 :
Node 8 :

---Dijkstra---
origem : 2
destino : 7
0 dist = 2147483647   pred = -1   settled = false
1 dist = 2147483647   pred = -1   settled = false
2 dist = 0   pred = 2   settled = false
3 dist = 2147483647   pred = -1   settled = false
4 dist = 2147483647   pred = -1   settled = false
5 dist = 2147483647   pred = -1   settled = false
6 dist = 2147483647   pred = -1   settled = false
7 dist = 2147483647   pred = -1   settled = false
8 dist = 2147483647   pred = -1   settled = false
```

5. **O caso do conjunto S**

   Na descrição do algoritmo de Dijkstra aparece um conjunto S que corresponde aos nós atingidos mais ainda não acomodados. Sobre este conjunto, nós queremos fazer as operações a seguir:

   - adicionar um nó;
   - retirar o nó de menor distância para o nó de partida s.

   Para ser mais eficiente, nós podemos utilizar uma **heap** a fim de assegurar que estas duas operações sejam realizadas em tempo logarítmico no número de nós deste conjunto. A prioridade de um elemento na heap é definida pela sua distância até o nó s.

   Temos entretanto uma sutileza a tratar aqui. Note que as prioridades dos nós podem mudar, por exemplo se um nó já atingido tem sua menor distância modificada através da relaxação de um arco incidente a ele. Neste caso, é necessário poder modificar a

prioridade deste elemento na heap, o que é de certa maneira complexo. Nós iremos contornar este problema realizando o procedimento a seguir: logo que a prioridade de um nó `i` contido no conjunto S deve ser modificada, iremos inserir em S uma nova ocorrência de `i` com uma nova prioridade. Nós sabemos que esta nova ocorrência de `i` sairá de S antes da antiga, uma vez que os elementos saem de S em ordem crescente de prioridade. Em contrapartida, poderá ser que um nó que saia de S já tenha sido acomodado pelo algoritmo. Isto precisará ser verificado.

Vamos portanto utilizar uma heap que conterá elementos do tipo `Node`, composto de um identificador `id` (o número do nó) e de um valor `val` (sua prioridade, ou seja sua distância até o nó s). A fim de criar um nó correspondente ao nó `i`, iremos utilizar o construtor da classe `Node` com dois argumentos (identificando o nó e sua prioridade): `Node(int id,int val)`.

Adicione um campo `PriorityQueue<Node> naoacomodados` à classe `Dijkstra` (a classe `PriorityQueue` do Java implementa uma heap). Este campo representará o conjunto S na descrição do algoritmo. Modifique o construtor da classe a fim de inicializar corretamente `naoacomodados`.

Os métodos que precisaremos de `PriorityQueue` são:

- `boolean isEmpty()` que retorna `true` se e somente se a estrutura está vazia;
- `void add(Node n)` que adiciona o nó `n`;
- `Node poll()` que retorna o elemento de prioridade mínima, removendo-o da heap.

6. **Agora vamos em frente!**

   Nós temos a partir de agora todo os elementos necessários para codificar uma versão eficaz do algoritmo de Dijsktra.

   Adicione um método `atualize(int v,int x)` que desempenha o papel da rotina `Atualize(v,x,G,S)` da descrição do algoritmo. Em particular, no caso onde a atualização é efetuada sobre `dist[v]`, nós vamos atualizar o predecessor de `v`, e vamos inserir o nó `v` com sua nova prioridade na heap `naoacomodados`.

   Adicione um método `int nextNode()` que desempenha o papel da rotina `RetireMenorDist()` da descrição do algoritmo, que retira de `naoacomodados` os nós de acordo com sua prioridade, até encontrar um nó ainda não acomodado e o retornar (lembre-se dos elementos repetidos que são adicionados a S e que não poderão ser retornados caso já acomodados anteriormente pelo algoritmo). Retorne -1 se a heap estiver vazia ou se todos os nós restantes da heap já tenham sido acomodados.

   Adicione um método `int oneStep()` que realiza uma iteração do laço principal do algoritmo de Dijsktra: ou seja, extrai de S o próximo nó `i` a ser acomodado, marca `i` como acomodado, atualiza todos os sucessores de `i`, e retorna o nó `i` que acaba de ser acomodado. Este método deve retornar -1 caso não reste mais nós a serem acomodados.

   Enfim, adicione um método `int compute()` que calcula e retorna o comprimento de um caminho mais curto. O método funciona da seguinte maneira: enquanto o nó t de

destino ainda não estiver acomodado e que restar pelo menos um nó não acomodado, realizamos uma iteração. Em seguida, é retornado o valor -1 ou o valor do caminho mais curto dependendo se o nó t pode ser alcançado ou não a partir do nó de partida s.

A fim de testar os seus métodos, execute o arquivo `Test3.java`. Você deverá obter como saída:

```
Teste 3 : teste do algoritmo de Dijkstra

grafo pequeno
Carregando rede viaria do arquivo mini.gr ... done
caminho mais curto entre 2 e 7 = 11
caminho mais curto entre 7 e 2 = -1

grafo grande
Carregando rede viaria do arquivo USA-road-d-NY.gr ... done
caminho mais curto entre 190637 e 187333 = 39113
```

E agora para visualizar o resultado, será necessário:

- adicionar uma chamada à função `slow()` no começo da função `oneStep()` (para desacelerar a visualização),
- adicionar a instrução `g.drawUnsettledPoint(f,y)`: assim que o ponto y é adicionado no conjunto `naoacomodados`,
- adicionar a instrução `g.drawSettledPoint(f,x)`: assim que você fixar o valor de `settled[x]` para `true`,
- executar o arquivo `Test4.java`. Você deverá ver avançar o seu conjunto de pontos acomodados (em vermelho) e os pontos de S (em verde).

No fim, você deverá obter o caminho mais curto a seguir:

e a seguinte resposta no terminal:

```
Teste 4 : teste do algoritmo de Dijkstra
Carregando rede viaria do arquivo USA-road-d-NY.gr ... done
Carregando coordenadas geometricas do arquivo USA-road-d-NY.co ... done
Carregando imagem de background do arquivo NY_Metropolitan.png ...  done
caminho mais curto entre 190637 et 187333 = 39113
```

*Este trabalho prático é de autoria de Jean-Christophe Filliâtre (Poly, France)