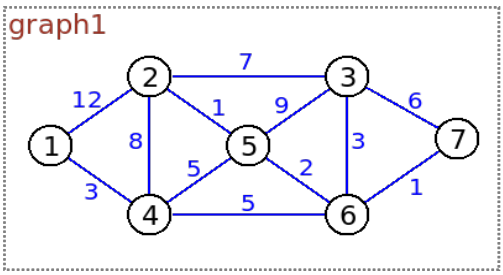
11ª aula prática – Caminhos mais curtos e apoio ao 2º trabalho prático

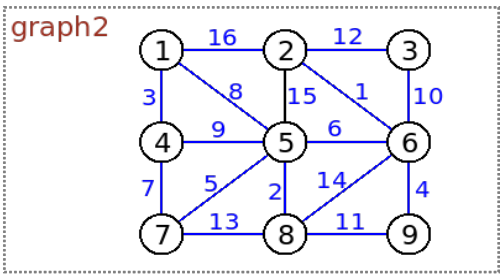
Instruções

- Faça download do ficheiro ***aed2122_p10.zip*** da página da disciplina e descomprima-o (contém a pasta ***lib,*** a pasta ***Tests*** com os ficheiros ***funWithGraphs.cpp, funWithGraphs.h, graph.cpp, graph.h, minHeap.h*** e ***tests.cpp***, e os ficheiros ***CMakeLists*** e ***main.cpp***)
- Se desejar pode criar métodos e/ou classes auxiliares para resolver estes exercícios

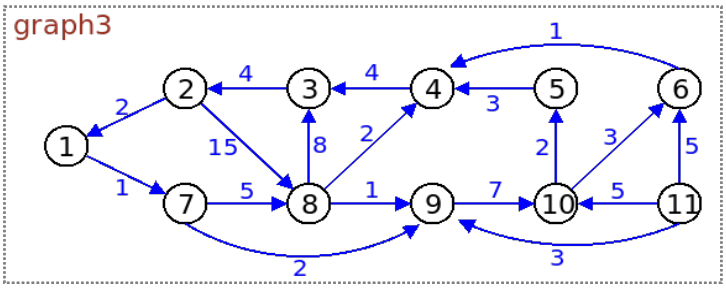
Nesta aula iremos usar como base a classe ***Graph*** introduzida nas aula prática anteriores. A classe ***FunWithGraphs*** contém alguns grafos não dirigidos e pesados "prontos a usar" e que são usados nos testes unitários desta aula. Para facilitar a sua tarefa nesta aula pode ver aqui as suas ilustrações:

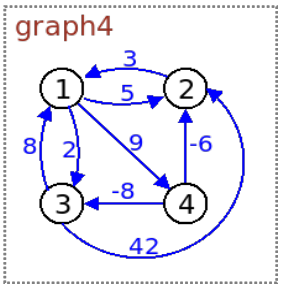
Alguns grafos não dirigidos e pesados





Alguns grafos dirigidos e pesados





1. Algoritmo de Dijkstra e caminhos mais curtos

Uma componente fundamental do 2º trabalho é o cálculo de caminhos mais curtos. Quando o grafo não é pesado pode simplesmente ser usada uma pesquisa em largura (ex: exercício 4 da 9ª aula prática). Se o grafo for pesado (e não existirem pesos negativos) pode usar o algoritmo de Dijkstra. Pode ver uma explicação na aula teórica nº 21 (slides 7 a 14), que inclui pseudo-código.

O objetivo deste exercício é dar-lhe a oportunidade de testar uma implementação do algoritmo de Dijkstra e verificar se ela está a devolver os resultados esperados. Pode fazer uma única implementação do algoritmo e depois cada um dos métodos devolve uma parte diferente dos resultados obtidos (sugestão: acrescentar na `struct Node` de `Graph` dois atributos inteiros - *dist* e *pred* – e um booleano - *visited*).

Pode usar a *MinHeap* dada (nós como chave e distância como valor) para implementar a fila de prioridade *Q* do pseudo-código (não esquecendo de fazer *decreaseKey* ao atualizar uma distância). O slide 14 indica outras alternativas para implementar o algoritmo de Dijkstra usando apenas estruturas de dados da STL (ex: usar *set* e implementar atualização como remoção + reinserção).

a) Distância entre dois nós. Função a implementar:

```
int Graph::dijkstra_distance(int a, int b)
```

Complexidade temporal esperada: $O(|E| \log |V|)$
(onde |V| é o número de nós e |E| o número de ligações)

Esta função deve calcular a distância do nó *a* para o nó *b*, ou seja, qual o menor custo (soma dos pesos) de um caminho entre *a* e *b*. Se não existir nenhum caminho a função deve devolver -1.

Exemplo de chamada e output esperado:

```
Graph g1 = FunWithGraphs::graph1();
cout << g1.dijkstra_distance(1,2) << endl;
cout << g1.dijkstra_distance(4,3) << endl;
Graph g3 = FunWithGraphs::graph3();
cout << g3.dijkstra_distance(8,11) << endl;
cout << g3.dijkstra_distance(11,8) << endl;
```

```
9
8
-1
22
```

Explicação:

- *graph1:* distância entre nós 1 e 2 é 9 (caminho: 1→ 4→ 5→ 2, com custo 9=3+5+1)
- *graph1:* distância entre nós 4 e 3 é 8 (caminho: 4→ 6→ 3, com custo 8=5+3)
- *graph3:* não existe nenhum caminho entre 8 e 11
- *graph3:* distância entre nós 11 e 8 é 22 (caminho: 11→ 6→ 4→ 3→ 2→ 1→ 7→ 8, com custo 22)

b) Caminho mais curto entre dois nós. Função a implementar:

```
list<int> Graph::dijkstra_path(int a, int b)
```

Complexidade temporal esperada: $O(|E| \log |V|)$
(onde $|V|$ é o número de nós e |E| o número de ligações)

Esta função deve devolver o caminho mais curto entre o nó *a* e nó *b*, ou seja, o de menor custo entre *a* e *b*. A lista devolvida deve ter um tamanho igual ao número de nós do caminho e nela devem vir sequencialmente os nós do caminho (o primeiro nó deve ser *a* e o último nó deve ser *b*). Se existir mais do que um caminho, qualquer um é aceite. Se não existir nenhum caminho a função deve devolver uma lista vazia (de tamanho zero).

Exemplo de chamada e output esperado:

```
Graph g1 = FunWithGraphs::graph1();
list<int> v1 = g1.dijkstra_path(1,2);
cout << "size = " << v1.size() << ":";
for (int i : v1) cout << " " << i;
cout << endl;
list<int> v2 = g1.dijkstra_path(4,3);
cout << "size = " << v2.size() << ":";
for (int i : v2) cout << " " << i;
cout << endl;
Graph g3 = FunWithGraphs::graph3();
list<int> v3 = g3.dijkstra_path(8,11);
cout << "size = " << v3.size() << endl;
```

```
size = 4: 1 4 5 2
size = 3: 4 6 3
size = 0
```

Explicação:

- *graph1:* caminho mais curto entre nós 1 e 2 é 1→ 4→ 5→ 2, com custo 9=3+5+1
- *graph1:* caminho mais curto entre nós 4 e 3 é 4→ 6→ 3, com custo 8=5+3
- *graph3:* não existe nenhum caminho entre 8 e 11

2. Tarefas para 2º trabalho prático.

O objetivo deste exercício é deixá-lo começar já a fazer coisas para o 2º trabalho prático. **A primeira coisa que deve fazer é ler atentamente todo o enunciado** (disponível no moodle).

Na sua essência, as suas tarefas podem ser divididas em 4 grandes componentes:

(i) Ler Ficheiros → (ii) Criar Grafo(s) → (iii) Algoritmos sobre o(s) grafo(s) → (iv) Menu

Pode usar a aula para iniciar o trabalho, quer do ponto de vista do seu planeamento, quer do ponto de vista da implementação na prática. Aqui ficam algumas ideias sobre o que pode ir começando nesta aula:

(i) Leitura dos Ficheiros

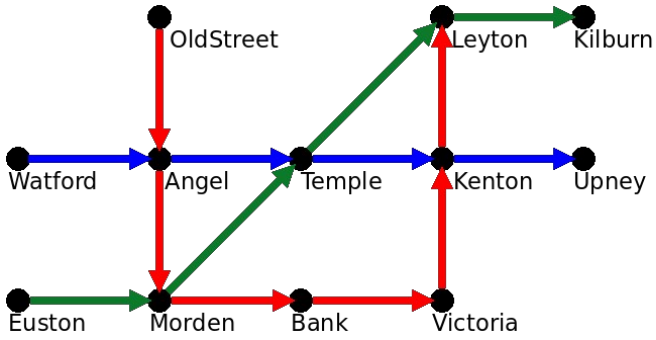
Faça download de dataset.zip e espreite os ficheiros existentes, graantindo que compreende a maneira como estão organizados (com a ajuda do enunciado que os descreve em detalhe).

Implemente código para ler os nós. Os atributos vêm separados por vírgulas (pode, entre outras maneiras, optar por usar a função `getline`, que admite a chamada com um delimitador – espreite a documentação para ver no final um exemplo que separa pelo caracter `';'`). Inicialmente pode apenas imprimir o que leu para garantir que conseguiu fazê-lo corretamente.

Implemente código para ler as linhas de maneira similar. Depois, para cada linha de código `CODE` leia os seus ficheiros `LINE_[CODE]_[DIR].csv` correspondentes, em ambas as direções (`DIR=0` e `DIR=1`)

(ii) Criar um ou mais grafos descrevendo a rede dos STCP

Uma linha é dada como sequência de paragens e no fundo induz ligações entre paragens consecutivas. A imagem seguinte ilustra o conceito para três hipotéticas linhas dadas como sequências de paragens:



Red Line	Green Line	Blue Line
OldStreet	Euston	Watford
Angel	Morden	Angel
Morden	Temple	Temple
Bank	Leyton	Kenton
Victoria	Kilburn	Upney
Kenton		
Leyton		

Não sendo obrigatório (tem liberdade para fazer como preferir), pode usar a classe `Graph` dada nas aulas como a base da sua representação da rede dos STCP. Note que pode acrescentar atributos a `struct Node` (ex: nome da paragem) e `struct Edge` (ex: nome da linha à qual a aresta pertence). Pode também manter a representação interna dos nós como inteiros. Para isso, apenas precisa de ter maneira de mapear cada paragem num inteiro (ex: usar um `map<string, int>`), sendo que ao ler os nomes das paragens pode desde logo associar cada código (que é único) a um inteiro entre 1 e *n*.

Procure implementar estas ideias criando um grafo inicial que reflete as linhas dadas no *dataset*. Na página seguinte pode encontrar um exemplo de paragens reais na rede dos STCP e de uma rede induzida pelas sequências de paragens. Se existir mais do que uma linha a ir de uma paragem para outra, pode optar por criar um multigrafo (com várias possíveis arestas de diferentes linhas entre um mesmo par de nós) ou manter tudo como um grafo simples (associando a cada aresta um conjunto de linhas).

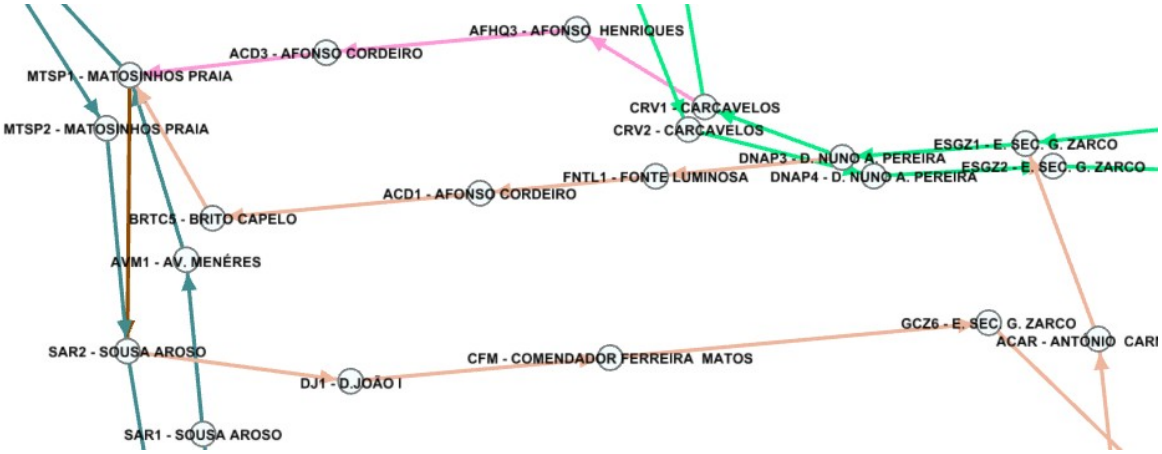


Ilustração de parte da rede dos STCP obtida a partir do dataset

No que toca ao "peso" de cada aresta, a sua opção pode refletir as tarefas que pretende fazer a seguir. Por exemplo, para o cálculo de caminhos mais curtos em termos de distância física, pode colocar como peso na aresta *(a,b)* o valor de *distância(a,b)* calculado com a fórmula de Haversine como explicado no enunciado (note que pode mudar o tipo do atributo `weight` se precisar de mais do que um tipo de peso pode optar por ter mais que um grafo ou por associar a cada aresta mais do que um atributo de peso).

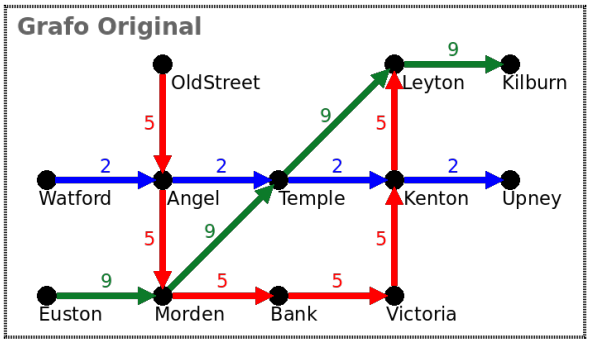
Para implementar a parte de mudanças de autocarro com deslocação a pé, pode receber como argumento uma dada distância *d* e criar ligações entre todas as paragens que estão a menor distância que *d*. Por exemplo, para a figura de cima, uma escolha de *d* adequada pode levar à criação de uma ligação entre as paragens "`AVM1 - Av. Menéres`" e "`BRTC5 - Brito Capelo`", representando que um utente pode sair numa dessas paragens para apanhar um novo autocarro na outra paragem.

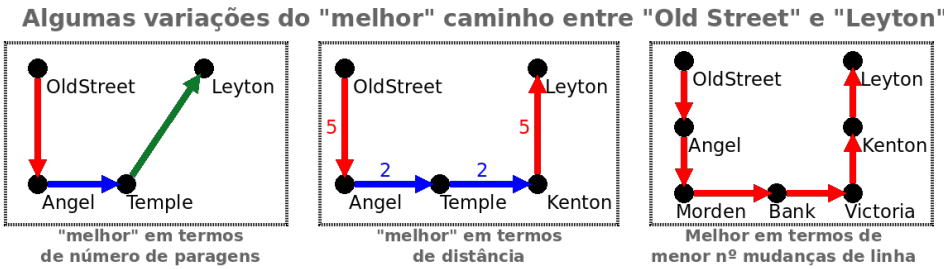
(iii) Algoritmos de grafos na rede dos STCP

Depois de ter o grafo pode implementar algoritmos para encontrar caminhos mais curtos, usando como base os algoritmos que desenvolveu nas aulas práticas.

- **Pesquisa em Largura (BFS)** para encontrar caminhos mais curtos se não quiser ter em conta o peso, mas apenas o número de arestas (ou seja, o número de estações do caminho).
- **Algoritmo de Dijkstra** para encontrar caminhos mais curtos num grafo pesado (onde o significado do que encontra depende dos pesos que usar)

Note como o enunciado lhe oferece diferentes noções de "melhor percurso". A imagem de baixo ilustra um possível grafo e diferentes variações do conceito:





Algumas das variações de "melhor percurso" são diretas em termos de implementação e outras podem obrigar a pensar mais um pouco (também não podemos dar-lhe já a solução para tudo, certo? ☺). Note que pode oferecer a qum utiliza o seu projecto diferentes combinações (ex: melhor percurso segundo um certo critério e em caso de empate segundo um outro critério.

(iv) Menu e interface com o utilizador

Depois do resto feito, esta é a parte mais fácil do trabalho, mas deve expor todas as funcionalidades que tenha implementado e deve ter o cuidado de as tornar o mais *"user friendly"* possível. Por exemplo, qual a melhor maneira de ilustrar um percurso calculado? Será apenas com um número? Com uma sequência de paragens sem indicação de que linhas seguir? Como é mostrado por exemplo no site dos STCP ou em qualquer outra aplicação de navegação em transportes públicos?

Esperamos que este projeto prático com dados reais possa ser motivante e desejamos a todos um bom trabalho, fazendo notar que estaremos sempre disponíveis para tirar dúvidas.

(exercícios extra) **Mais ideias para exploração:**
A prioridade deve ser nesta fase o trabalho, mas se quiser depois explorar esta parte de caminhos mais curtos deixamos ficar uma sugestão:

- Implementar o algoritmo de Bellman-Ford.
E se existissem pesos negativos como em *graph4*? Nesse grafo, qual é o caminho mais curto entre os nós 1 e 2? O que devolveria (erradamente) o algoritmo de Dijkstra? Implemente o algoritmo de Bellman-Ford como explicado nas aulas (ver slides 17 a 19 da daula teórica nº 21) e use *graph4* para poder verificar se o algoritmo funciona.

- Existência de ciclos negativos.
Use a extensão do algoritmo de Bellman-Ford dada nas aulas (slide 21 da aula teórica nº 21) para verificar se existe algum ciclo negativo num dado grafo (que implicaria, por exemplo, que a distância entre qualquer par de nós do ciclo pode ser tão baixo quanto quisermos, se percorrermos várias vezes o ciclo para ir reduzindo a distância). Crie grafos com e sem ciclos negativos para verificar empiricamente a sua implementação.