# 5ª Aula Prática – Grafos: Caminho mais curto

## Instruções

- Faça download do ficheiro **cal_fp05.zip** da página da disciplina e descomprima-o (contém os ficheiros **Test.cpp** e **Graph.h,** com código no seguimento das duas aulas práticas anteriores)
- Abra o eclipse e crie um novo projeto C++ do tipo Cute Project (File/New/C++ Project/Cute Project) com o nome **CalFp05** e usando o MinGW GCC.
- Inclua a biblioteca Boost.
- Importe para a pasta *src* do projeto, os ficheiros extraídos: (Import/General/File System)
  - Aparecendo a mensagem a perguntar se quer fazer *Overwrite* ao ficheiro Test.cpp diga que sim.
  - Compile o projeto.
  - Execute o projeto como CUTE Test (Run As/CUTE Test). Se surgir a pergunta de qual compilador usar, escolha MinGW gdb.
- ***Deverá realizar esta ficha respeitando a ordem das alíneas.*** Poderá executar o projeto como CUTE Test quando quiser saber se a implementação que fez é suficiente para passar no teste correspondente.

## Enunciado

Considere a classe **Graph** definida no ficheiro *Graph.h* e já utilizada nas aulas anteriores. Deverá atualizar as classes do ficheiro *Graph.h* adequadamente, a fim de realizar as alíneas que se seguem. Identifique a partir do ficheiro Test.cpp funções auxiliares que sejam necessárias e não sejam pedidas explicitamente nos exercícios.

### 1. Algoritmo de Dijkstra

a) Implemente na classe **Graph** o membro-função público:

```
void dijkstraShortestPath(const T &origin)
```

Esta função implementa o algoritmo de Dijkstra para encontrar os caminhos mais curtos a partir de um vértice de origem (vértice ***s*** cujo conteúdo é ***origin***) para todos os outros vértices (ver algoritmo das aulas teóricas). Precisa de adicionar à classe *Vertex* campos para representar a distância mínima (***dist***) e o vértice anterior no caminho mais curto (***path***) (campos já criados no código fornecido).

Sugestão: Uma vez que a STL não disponibiliza filas de prioridade mutáveis (suportando *decrease_key*), usar uma fila de prioridades mutável de Boost ou usar a classe ***MutablePriorityQueue*** fornecida, que pode ser manipulada da seguinte forma:

- Para criar fila: `MutablePriorityQueue<Vertex<T> > q;`
- Para inserir elemento v (apontador para vértice): `q.insert(v);`
- Para extrair o mínimo (apontador para vértice): `v = q.extractMin();`
- Para avisar que chave (*dist*) de elemento *v* diminui de valor: `q.decreaseKey(v);`

Na classe ***Vextex*** é necessário (passos já realizados no código fornecido):

- Declarar campo `int queueIndex;`
- Declarar `friend class MutablePriorityQueue<Vertex<T> >;`

- Implementar `bool operator<(Vertex<T> & vertex) const` com base na comparação de valores do campo ***dist***.

b) Implemente na classe **Graph** o membro-função:

```
vector<T> getPath(const T &origin, const T &dest)
```

Esta função retorna um vetor com a sequência dos elementos do grafo representando os vértices do caminho de ***origin*** até ***dest***, inclusivé (***origin*** e ***dest*** são os membros-dado ***info*** dos vértices de origem e destino do caminho, respetivamente). Pressupõe-se que a função **`dijkstraShortestPath`** foi chamada previamente com argumento ***origin***.

c) Com base nos dados de desempenho do algoritmo de Dijkstra produzidos pelos testes fornecidos, crie um gráfico para mostrar que o tempo médio de execução é proporcional a $(|V| + |E|) \log_2 |V|$. Os testes de desempenho geram grafos aleatórios em forma de grelha de tamanho N x N, em que nº de vértices é $|V| = N^2$ e o nº de arestas é 4N(N-1).

**PARA PRÓXIMA AULA**

**2. Outros algoritmos de caminho mais curto de um vértice para todos os outros.**

**a)** Implemente na classe **Graph** o membro-função:

```
void unweightedShortestPath(const T &origin)
```

Esta função implementa um algoritmo para encontrar os caminhos mais curtos a partir do elemento ***v*** do grafo (vértice cujo conteúdo é ***origin***) a todos os outros vértices do grafo, ignorando os pesos das arestas.

**b)** Implemente na classe **Graph** o membro-função público:

```
void bellmanFordShortestPath(const T &origin)
```

Esta função implementa o algoritmo de Bellman-Ford para encontrar os caminhos mais curtos a partir do elemento *s* do grafo (vértice cujo conteúdo é ***origin***) a todos os outros vértices, permitindo a existência de arestas com pesos negativos.

**3. Encontrar o caminho mais curto entre todos os pares de vértices.**

**a)** Implemente na classe **Graph** o membro-função público:

```
void floydWarshallShortestPath()
```

Esta função implementa o algoritmo de Floyd-Warshall para encontrar os caminhos mais curtos entre todos os vértices ***v*** do grafo, no caso de grafos pesados (ver algoritmo e estruturas de dados nos slides das aulas teóricas). É necessário adicionar à classe **Graph** as matrizes referidas nos slides.

Adicionalmente, implemente na classe **Graph** o membro-função público:

```
vector<T> getfloydWarshallPath(const T &origin, const T &dest)
```

Esta função retorna um vetor com a sequência dos elementos do grafo representando os vértices do caminho de ***origin*** até ***dest***, inclusivé (onde ***origin*** e ***dest*** são as propriedades ***info*** dos vértices de origem e destino do caminho, respetivamente). Assuma que esta função é chamada depois da anterior.