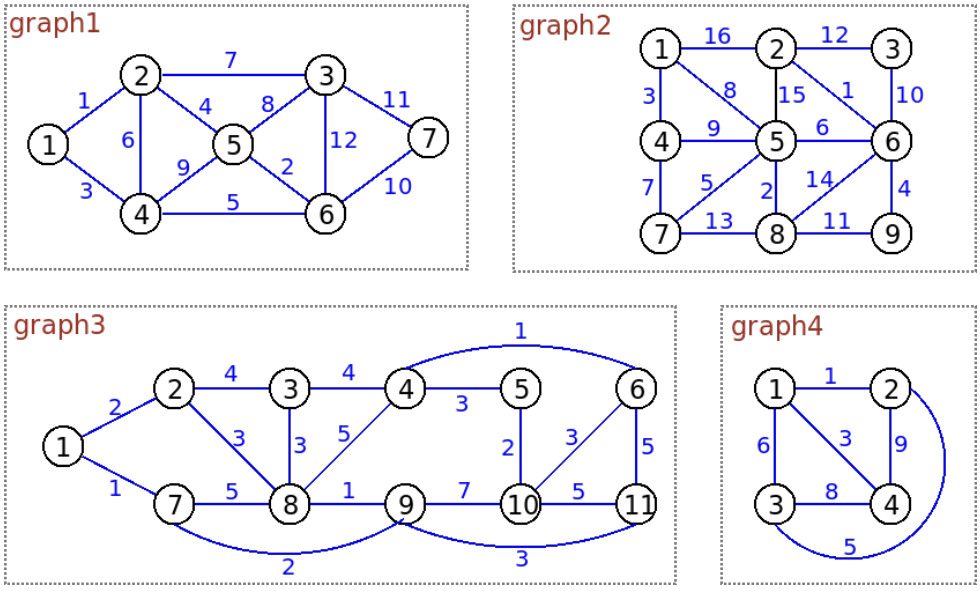
10ª aula prática – *Minimum Spanning Trees* e estruturas de dados associadas

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, disjointSets.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 na aula prática anterior. 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 tua tarefa nesta aula podes ver aqui as suas ilustrações:

Alguns grafos não dirigidos e pesados



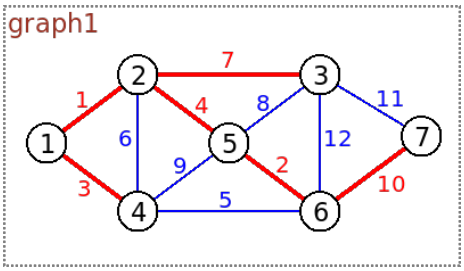
Uma ***minimum spanning tree (MST)***, conhecida em português como árvore de suporte de custo mínima, árvore geradora mínima ou árvore de extensão mínima, é um subconjunto de |V|-1 arestas com o menor custo possível que forma uma árvore ligando todos os nós (ver slides 2 a 5 da teórica nº 19)

1. Calculando manualmente uma *minimum spanning tree*.

Para garantir que percebeu o conceito de MST e os algoritmos de **Prim** e **Kruskal** *(clicar para ver visualizações)* que estão na base desta aula, comece por calcular a MST de *graph2* (deixamos ficar *graph3* e *graph4* como exercício opcional). Deve calcular usando os dois algoritmos dados, explicando os passos dados, as arestas adicionadas e o custo total. Como exemplo pode ver uma solução para *graph1*.

Algoritmo de **Prim**, começando no nó 1 (ver slides 7 a 9 da teórica nº 19)

. 2º nó adicionado: 2, através da aresta (1,2) de custo 1
. 3º nó adicionado: 4, através da aresta (1,4) de custo 3
. 4º nó adicionado: 5, através da aresta (2,5) de custo 4
. 5º nó adicionado: 6, através da aresta (5,6) de custo 2
. 6º nó adicionado: 3, através da aresta (2,3) de custo 7
. 7º nó adicionado: 7, através da aresta (6,7) de custo 10



A MST deste grafo é indicada pelas arestas a vermelho e tem custo 1+3+7+4+2+10 = 27

Algoritmo de **Kruskal** (ver slides 3 a 6 da teórica nº 20)

. 1ª aresta adicionada: (1,2) de custo 1
. 2ª aresta adicionada: (5,6) de custo 2
. 3ª aresta adicionada: (1,4) de custo 3
. 4ª aresta adicionada: (2,5) de custo 4
. 5ª aresta adicionada: (2,4) de custo 7 [arestas (4,6) ou (2,4) têm menor peso, mas introduziriam um ciclo]
. 6ª aresta adicionada: (6,7) de custo 10 [arestas (3,5) ou (4,5) têm menor peso, mas introduziriam um ciclo]

2. ***Heaps Binários.***

Uma **fila de prioridade** (*priority queue*) é uma estrutura de dados para armazenar um conjunto de elementos e fornecer operações de inserir um elemento ou retirar o elemento mais prioritário.

Uma maneira eficiente de implementar é usar uma **heap** (*clicar para visualizar*), como explicado nos slides 20 a 25 da teórica nº 19. O C++ disponibiliza a classe **priority_queue** que é semelhante a uma (max)heap mas não permite atualizar (melhorar) a prioridade de um elemento (operação necessária para o algoritmo de Prim e que também será necessário para o algoritmo de Dijkstra.).

O intuito deste exercício é disponibilizar código completo para uma classe genérica **MinHeap<K,V>** que está preparada para todas as necessidades do algoritmo de Prim. Para isso permite armazenar pares (*chave*, *valor*) com chaves do tipo K a terem como prioridade valores do tipo V. Os métodos disponíveis publicamente para interagir com a classe são os seguintes:

```
MinHeap(int n, const K& notFound); // Create a min-heap for a max of n pairs (K,V)
                                   // with notFound returned when empty
int getSize();                // Return number of elements in the heap
bool hasKey(const K& key);    // Heap has key?
void insert(const K& key, const V& value);      // Insert (key, value) on the heap
void decreaseKey(const K& key, const V& value); // Decrease value of key
K removeMin(); // remove and return key with smaller value
```

insert, ***decreaseKey*** e ***removeMin*** garantem tempo logarítmico; ***getSize***, ***hasKey*** têm tempo constante

Um exemplo de uso é dado a seguir:

```
MinHeap<string, int> q(10, "---");
q.insert("A", 42); q.insert("B", 20); cout << q.removeMin() << endl;
q.insert("C", 10); q.decreaseKey("A", 5);
cout << q.removeMin() << endl << q.removeMin() << endl << q.removeMin() << endl;
```

```
B
A
C
---
```

Explicação: No primeiro *removeMin* a chave com valor mínimo (20) é "B", depois é "A" (o seu valor foi reduzido para 5) e finalmente "C". O último *removeMin* devolve "---" porque nessa altura a heap está vazia e é devolvido o elemento *notFound*.

a) Um pequeno jogo. Implementa a seguinte função no ficheiro ***FunWithGraphs.cpp***:

```
static int FunWithGraphs::game(const vector<int>& v)
```

Complexidade temporal esperada: $\mathcal{O}(n \log n)$
(onde *n* é o tamanho do vector *v*)

Este procedimento deve simular o seguinte jogo:

- Considerar no início um conjunto *S* com todos os números do vector *v*
- Repetir o seguinte até *S* ficar só com único número: retirar os dois números *n1* e *n2* com menor soma dos dígitos e inserir novamente no conjunto a diferença entre *n1* e *n2* (ou seja, |*n1-n2*|)
- Devolver o número final que ficou no conjunto

Exemplo de chamada e output esperado:

```
cout << FunWithGraphs::game({100, 2, 30, 400}) << endl;
```

```
272
```

Explicação: 1ª iteração: S={100, 2, 30, 400} com somas de dígitos 1, 2, 3 e 4 | retirar 100 e 2, inserir |100 – 2| = 98
2ª iteração: S={98, 30, 400} com somas de dígitos 17, 3 e 4 | retirar 30 e 400, inserir |30 – 400| = 370
3ª iteração: S={370, 98} com somas de dígitos 10 e 17 | retirar 370 e 98, inserir |370-98| = 272
4ª iteração: S={272} | é o último número no conjunto, devolver 272

Sugestão: usar uma *minHeap<int, int>* (números como chaves e soma dos dígitos como valores); no início inserir todos os números e depois é só repetir o passo de retirar dois e inserir a diferença até ficar só com um elemento.

3. Algoritmo de Prim. Função a implementar:

```
int Graph::prim(int r)
```

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

Deve devolver o custo de uma *minimum spanning tree* (*MST*) criada pelo algoritmo de Prim quando este é iniciado a partir do nó *r*.

Para implementar podes seguir como guião o pseudo-código do slide 12 da teórica nº 19 (o custo é dado pela soma dos *v.distance*). Podes modificar a classe *Graph* (ex: qual seria o sítio apropriado para criar variáveis *distance* e *parent*, que são atributos de um nó?)

Para perceber o que o programa faz é aconselhado que vejam as várias iterações do algoritmo a serem feitas (ex: imprimindo). O exercício 1 um desta aula fornece um exemplo para *graph1* da ordem em que os nós são adicionados e os custos associados.

Exemplo de chamada e output esperado:

```
Graph g1 = FunWithGraphs::graph1();
cout << g1.prim(1) << endl;
cout << g1.prim(2) << endl;
```

```
27
27
```

Explicação: seja qual for o nó inicial, o custo da MST é sempre 27 (ver figura da página 1)

Sugestão: Use a *MinHeap* dada (nós como chave e distância como valor) para implementar a fila de prioridade *Q* do pseudo-código (não se esqueça de fazer *decreaseKey* ao atualizar uma distância)

Nota: o slide 30 da teórica nº 20 indica outras alternativas para implementar o algoritmo de Prim usando apenas estruturas de dados da STL (ex: usar *set* e implementar atualização como remoção + reinserção)

4. Conjuntos Disjuntos (*disjoint-sets*).

Uma estrutura de dados **disjoint-set** (também conhecida como *union-find*) é uma estrutura de dados especializada em armazenar conjuntos disjuntos (sem sobreposição) e fornecer operações de unir dois conjuntos (*union*) e descobrir o representante do conjunto de um elemento (*find*). Esta última operação pode ser usada para verificar se dois quaisquer elementos pertencem ou não ao mesmo conjunto.

Uma maneira eficiente de implementar é usar **florestas de conjuntos disjuntos** (*clicar para visualizar*), como explicado nos slides 11 a 18 da teórica nº 20 (esta estrutura de dados não existe na STL ou em qualquer outra parte de C++ padrão).

O intuito deste exercício é disponibilizar implementação inicial de uma classe genérica **DisjointSets<K>** que implementa conjuntos disjuntos de elementos do tipo T, fornecendo todas as operações necessárias para depois serem usadas no algoritmo de Kruskal. Os métodos disponíveis publicamente para interagir com a classe são os seguintes (nota: foi usado *unite* porque *union* é uma palavra reservada em C++):

```
void makeSet(const T& x);      // Create a set with a single element x
T find(const T& x);            // Find the representative of the set of x
void unite(const T& x, const T& y); // Merge the sets of elements x and y
```

Um exemplo de uso é dado a seguir:

```
DisjointSets<int> s;
for (int i=1; i<=3; i++) s.makeSet(i);
cout << s.find(1) << " " << s.find(2) << " " << s.find(3) << endl;
s.unite(2,1);
cout << s.find(1) << " " << s.find(2) << " " << s.find(3) << endl;
s.unite(1,3);
cout << s.find(1) << " " << s.find(2) << " " << s.find(3) << endl;
```

```
1 2 3
2 2 3
2 2 2
```

Explicação: No início temos 3 conjuntos: {1}, {2} e {3} e cada número é o seu próprio representante.
Depois do merge dos dois primeiros conjuntos temos conjuntos {1,2} (com representante 2) e {3}
Depois do merge do conjunto de 1 com o conjunto de 3 ficamos com um só conjunto {1,2,3} (com representante 2)

a) Melhorando a eficiência com "union by rank" e "path compression".

Neste exercício deves modificar a classe e os 3 métodos atrás descritos (*makeSet*, *find* e *unite*) para melhorar a sua eficência (a versão disponilizada é "naive" e pode ter custo linear no *find*).

Complexidade temporal esperada por operação: $\mathcal{O}(1)$ em média, para todos os efeitos práticos
(na realidade a complexidade está relacionada com uma função que cresce mesmo muito lentamente – ver slide 18 para mais pormenores e um link para um artigo científico que prova a complexidade)

A implementação dada é "naive" e pode ter custo linear para operação find no pior caso, se a ordem de uniões induzir uma árvore desequilibrada (ver slides 12 e 13). O que se quer aqui é que implementes duas metodologias heurísticas para melhorar a eficiência (ver slides 14 a 17 para pseudo-código):

- **union by rank:** guardar *rank* de cada nó (profundidade) e unir sempre a árvore mais pequena com a árvore maior, evitando um incremento da profundidade máxima
- **path compression:** ao fazer um find, colocar todos os nós do caminho a apontar diretamente para a raiz, "comprimindo" assim o caminho de todos esses nós até ao representante

Os teste unitários fornecidos para esta função incluem um caso grande com mais de 100 mil elementos e muitas operações que já deve demorar alguns segundos. Pode verificar o tempo de execução na janela do CLion com os resultados dos testes (implemente as melhorias e vejas diferenças no tempo de execução – nota que este não é sequer o pior caso possível e qualquer uma das heuristicas melhora logo o tempo)

5. **Algoritmo de Kruskal.** Função a implementar:

```
int Graph::kruskal()
```

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

Deve devolver o custo de uma *minimum spanning tree* (*MST*) criada pelo algoritmo de Kruskal.

Para implementar podes seguir como guião o pseudo-código do slide 8 da teórica nº 20 (o custo é dado pela soma dos pesos das arestas adicionadas a *T*).

Para perceber o que o programa faz é aconselhado que vejam as várias iterações do algoritmo a serem feitas (ex: imprimindo). O exercício 1 um desta aula fornece um exemplo para *graph1* da ordem em que arestas são adicionadas (e quais não o são, por induzirem um ciclo)

Exemplo de chamada e output esperado:

```
Graph g1 = FunWithGraphs::graph1();
cout << g1.kruskal() << endl;
```

```
27
```

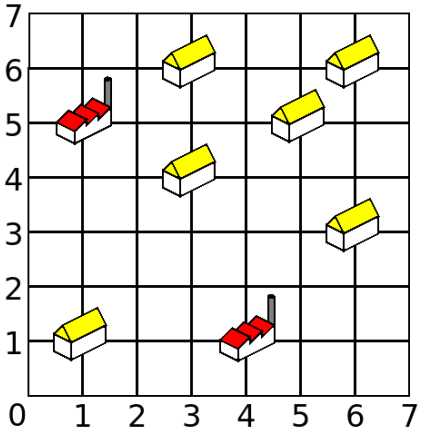
Explicação: o custo da MST é 27 (ver figura da página 1)

Sugestão: Use a classe *DisjointSet* dada para manter as florestas de árvores.
Para ordenar as arestas como pode fazer?

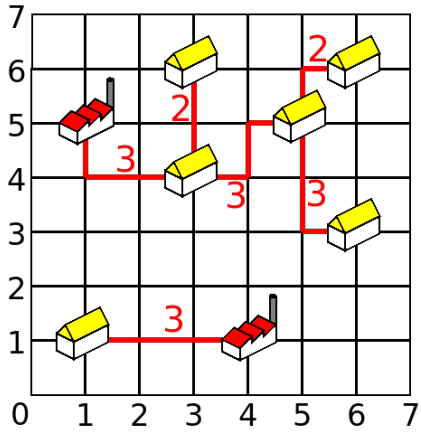
5. *(exercício extra)* **Uma cidade "quadriculada".** Função a implementar:

```
static int FunWithGraphs::gridCity(const vector<pair<int, int>>& plants,
                                   const vector<pair<int, int>>& houses)
```

Complexidade temporal esperada: $\mathcal{O}(n^2 \log n)$
(onde *n* é a soma dos tamanhos dos vetores *plants* e *houses*)



Ficaste responsável pela instalação da rede elétrica da província de Algoritmopólis e queres gastar a menor quantidade de fio possível para estabelecer as ligações a todas as casas. Uma cidade desta província pode ser pensada como um quadriculado. Cada uma das centrais eléctricas e cada uma das casas estão posicionados numa dada coordenada. O mapa da direita ilustra uma cidade com 2 centrais (a vermelho) e 6 casas (a amarelo). Por exemplo, as centrais estão posicionadas nas coordenadas (1,5) e (4,1).



Podes colocar um fio elétrico entre uma casa e uma casa, ou entre uma planta ou uma casa. No entanto, o fio tem de ir pelas linhas do quadriculado, o que significa que para ligar duas casas nas posições (x1,y1) e (x2,y2) precisarias de |x1-x2|+|y1-y2| quantidade de fio. Nota também que um fio não se pode dividir a meio do percurso e não há problema se na mesma linha do quadriculado passarem dois fios diferentes.

A tua tarefa é descobrir qual a menor quantidade de fio necessária para conseguir que todas as casas tenham um caminho via fios elétricos para uma das plantas. Por exemplo, a figura à direita ilustra uma possível maneira de ligar o mapa anterior gastando 16, que é o mínimo possível.

Exemplo de chamada e output esperado:

```
vector<pair<int, int>> plants1 = {{1,5}, {4,1}};
vector<pair<int, int>> houses1 = {{1,1},{3,4}, {3,6}, {5,5}, {6, 3}, {6,6}};
cout << FunWithGraphs::gridCity(plants1, houses1) << endl;
```

```
16
```

Explicação: o exemplo dado é o da figura do enunciado

Sugestão: isto é uma "espécie" de MST mas diferente… É quase como se a nossa MST já começasse com as plantas e sem ter gasto nada…

***(exercícios extra)* Mais ideias para exploração:**
Se te sentires com vontade de continuar a explorar este tema, ficam aqui mais sugestões possíveis:

- Devolver a MST em como um conjunto de arestas e não apenas como o custo
As funções Prim e Kruskal apenas devolvem de momento um inteiro com o custo da MST. Como poderias mudar para devolver a árvore em si? Qual o tipo de retorno? Como gerar a resposta?

- Mais variantes de MSTs
Como calcular uma MST se formos obrigados a usar uma dada aresta? E se uma aresta for proibida?
Se quisermos a segunda melhor MST, como poderiamos fazer?