Greedy Algorithms

Practical Exercises

Departamento de Engenharia Informática (DEI)
Faculdade de Engenharia da Universidade do Porto (FEUP)

Spring 2025

Exercise 1

In the **ex1.cpp** file implement the *prim* algorithm. This algorithm finds the minimum spanning tree from the first vertex v in the graph, to all other vertices. The function returns the graph's set of vertices.

```
std::vector<Vertex<T> *> prim(Graph <T> * g)
```

Suggestion: Since the STL does not support mutable priority queues, you can use the provided **MutablePriorityQueue** class, which contains the following methods:

- To create a queue: `MutablePriorityQueue<Vertex<T>> q;`
- To insert vertex pointer *v*: `q.insert(v);`
- To extract the element with minimum value (*dist*): `v = q.extractMin();`
- To notify that the key (*dist*) of ***v*** was decreased: `q.decreaseKey(v);`

Suggestion: Use the *visited*, *dist* and *path* attributes (and associated methods) from the **Vertex** class.

Exercise 2

In the **ex2.cpp** file implement *kruskal*, which uses Kruskal's algorithm to find the minimum spanning tree.

```
std::vector<Vertex<T> *> kruskal(Graph <T> * g)
```

Suggestions:

- Since the STL does not support union-find disjoint sets, you can use the provided **UFDS** class, which contains the following methods:
  - To create an UFDS with N nodes: `UFDS ufds(N);`
  - To determine the set of a node *v*: `ufds.findSet(v);`
  - To determine if two nodes *u* and *v* belong to the same set: `ufds.isSameSet(u,v);`
  - To connect two sets, identified by one of their nodes each: `ufds.linkSets(u,v);`
- Use the *path* attribute (and associated getter and setter) from the **Vertex** class and the *selected*, *orig, dest* and *reverse* attributes (and associated getters and setters) from the **Edge** class.

- Implement the auxiliary method called *dfsKruskalPath*, which uses a Depth-First Search (DFS) to update the vertices' *path* attribute so that it has their ancestor in the MST. This attribute is used by the unit test to check if the output is a correct MST.

```
void dfsKruskalPath(Vertex<T> *v)
```

Exercise 3

In the **ex3.cpp** file implement the following public method:

```
void unweightedShortestPath(Graph<T>* g, const int &origin)
```

This method implements an algorithm to find the shortest paths from a vertex (vertex which contains element `source`) to all other vertices, ignoring edge weights.

In the **ex3.cpp** file implement also the following public method:

```
vector<T> getPath(Graph<T>* g, const int &origin, const int &dest)
```

This function returns a vector with the sequence of the vertices of the path, from the `origin` to `dest`, inclusively. It is assumed that a path calculation function, such as `unweightedShortestPath`, was previously called with the `origin` argument, which is the `source` vertex.

Exercise 4

In the **ex4.cpp** file, implement the following public function:

```
void dijkstra(Graph<T>* g, const int &origin)
```

This method implements the Dijkstra algorithm to find the shortest paths from ***s*** (vertex which contains element ***origin***) to all other vertices, in a given weighted graph (see theoretical class slides). Update the **Vertex** class with member variables `int dist` and `Vertex<T>* path`, representing the distance to the start vertex and the previous vertex in the shortest path, respectively. Since the STL doesn't support mutable priority queues, you can use the class provided ***MutablePriorityQueue***.

Exercise 5

In the **ex5.cpp** file implement *edmondsKarp*, which uses the Edmonds-Karp algorithm to find the maximum flow from the source vertex source to the sink vertex target in the graph.

```
void edmondsKarp(Graph<T>* g, int source, int target)
```

<u>Suggestion</u>: Use the *visited* and *path* attributes (and associated getters and setters) from the **Vertex** class and the *orig* and *flow* attributes (and associated getters and setters) from the **Edge** class. Use the *weight* attribute from the **Edge** class as the edge's capacity (i.e. maximum allowed flow).

Exercise 6

In the **ex6.cpp** file implement *fordFulkerson*, which uses the Ford-Fulkerson algorithm to find the maximum flow from the source vertex source to the sink vertex target in the graph.

```
void fordFulkson(Graph<T>* g, int source, int target)
```

<u>Suggestion</u>: *fordFulkerson* does not impose a strategy to find an augmenting path from the source to the target Vertex in the residual network. As a title of example, use Depth-First Search (DFS) algorithm to find such augmenting path.