Notable Applications of Greedy Algorithms

Practical Exercises

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

Spring 2024

Important Note
Graph.h has been updated to facilitate the implementation of the following exercises. Please carefully review the *Graph.h* file. The main updates include:

- The vertex now holds information about *incoming* edges in addition to outgoing/*adj* edges.
- The edge now holds information about the *orig* vertex in addition to the *dest* vertex.
- Introduction of a new method *addBidirectionalEdge(orig, dest, w)* where two edges are created, one in each direction (orig→dest and dest→orig). In this case, both edges will store a pointer to the corresponding "twin" inverted edge in their *reverse* attribute.
- Introduction of new attributes (e.g., *path*, *flow*) and methods (e.g., *getPath()*, *getFlow()*) that will support the implementation of upcoming algorithms.

Exercise 1
In the **ex1.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)
```

Suggestion: 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 2
In the **ex2.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> 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 3

In the **ex3.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* 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)
```