Greedy Algorithms

Practical Exercises

DA 2023 Instructors Team

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

Spring 2023

A - Fundamentals

Exercise 1
Consider the description for the **change-making problem** in exercise 3 of the TP2 class sheet.

```
bool changeMakingGR(unsigned int C[], unsigned int Stock[],
 unsigned int n, unsigned int T, unsigned int usedCoins[])
```

a) Implement *changeMakingGR* using a greedy strategy.
b) Does the algorithm always return the optimal solution (i.e. minimum amount of coins) for any set of coin values (*i.e.,* for any coin system)? Give an example of a coin-system where the greedy algorithm does not give the optimal solution.

Exercise 2
The **activity selection problem** is concerned with the selection of non-conflicting activities to perform within a given time frame, given a set **A** of activities (a_i), each marked by a start time (s_i) and finish time (f_i). The problem is to select the maximum number of activities that can be performed by a single person or machine, assuming a given priority and that a person can only work on a single activity at a time. Consider the function *activitySelection* below (which uses the **Activity** class):

```
vector<Activity> activitySelectionGR(vector<Activity> A)
```

Input example: **A** = { $a_1(10, 20)$, $a_2(30, 35)$, $a_3(5, 15)$, $a_4(10, 40)$, $a_5(40, 50)$ }
Expected result: {a_3, a_2, a_5}

a) Formalize this problem.
b) Implement *activitySelectionGR* using a greedy strategy, in which priority is given to activities with the earliest finish time.
c) Indicate and justify the algorithm's time complexity, with respect to the number of activities, n.
d) Prove that the greedy strategy leads to the optimal solution.

Suggestion: Implement the < (less than) operator in the **Activity** class.

Exercise 3

Consider a machine on a factory line that needs to have its tasks scheduled in order to minimize their average completion time. The machine can only process one task at a time and each task has a predefined quantity of time needed for execution. Consider the function *minimumAverageCompletionTime* below, which returns the minimum average task completion time and computes the optimal task ordering on the second argument (*orderedTasks*). The tasks are identified by their execution time, t_i.

```
double minimumAverageCompletionTime(vector<unsigned int> tasks,
                                    vector<unsigned int> &orderedTasks)
```

Input example: **tasks** = [3, 7, 4]
Expected result: [3, 4, 7]

In the input example above, the machine has three tasks to carry out which take exactly 3, 7 and 4 units of time, respectively. The optimal scheduling is [3, 4, 7] (the task with a time of 3, followed by the one with 4 and ending with the one with 7) since the average completion time is ((3) + (3 + 4) + (3 + 4 + 7))/3 = 24/3 = 8. Any other scheduling is sub-optimal. For example, [3, 7, 4] would give an average completion time of ((3) + (3 + 7) + (3 + 7 + 4))/3 = 27/3 = 9.

a) Formalize this problem.
b) Implement *minimumAverageCompletionTime* using a greedy strategy.
c) Indicate and justify the algorithm's time complexity, with respect to the number of tasks, n.
d) Prove that the greedy strategy leads to the optimal solution.

Exercise 4

Consider the **fractional knapsack problem**. This is a variant of the 0-1 knapsack problem (presented in the TP2 sheet) where only a percentage of an item can be placed in the knapsack. For instance, if an item has a value of 4 and a weight of 3 and only 50% of the item is used, then it adds a value of 2 and a weight of 1.5 to the knapsack. Consider the function *fractionalKnapsackGR* below.

```
double fractionalKnapsackGR(unsigned int values[], unsigned int
weights[], unsigned int n, unsigned int maxWeight, double usedItems[])
```

Input example: **values** = [60, 100, 120], **weights** = [10, 20, 30], **n** = 3, **maxWeight** = 50
Expected result: [1, 1, $\frac{2}{3}$] (the total value is 60*1 + 100*1 + 120*$\frac{2}{3}$ = 240)

a) Formalize this problem.
b) Implement *fractionalKnapsackGR* using a greedy strategy.
c) Indicate and justify the algorithm's time complexity, in the worst case, with respect to the number of items, n.
d) Prove that the greedy strategy leads to the optimal solution.
e) Give an example of an integer 0-1 knapsack problem where using an adapted version of the greedy algorithm does not give the optimal solution.

B - Notable Applications

Exercise 5
To solve this and the following exercises, use the **GreedyGraph** class provided in the *exercises.h* file. **GreedyGraph** derives from the **Graph** base class (declared in *data_structures/Graph.h*), which implements a graph's basic functionalities and includes all the necessary attributes for the **Vertex**, **Edge** and **Graph** classes to solve these exercises.

All exercises can be solved without modifying the **Graph**, **Vertex** and **Edge** classes and by only adding auxiliary methods to the **GreedyGraph** class. No class attribute needs to be created on any class.

Implement *prim*, which uses Prim's algorithm to find 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 *> prim()
```

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 6
Using the **GreedyGraph** class provided in the *exercisses.h* file, implement *kruskal*, which uses Kruskal's algorithm to find the minimum spanning tree.

```
std::vector<Vertex *> kruskal()
```

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 an auxiliary method on the **GreedyGraph** class, 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 *v)
```

Exercise 7
Using the **GreedyGraph** class provided in the *exercisses.h* 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.

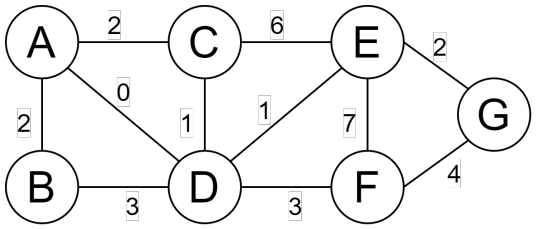
```
void edmondsKarp(int source, int target)
```

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 8
Bernard and Bianca are competing as a team in a computer network competition where each contestant must connect a series of routers between each other, so that there is a path between all pairs of routers. To achieve this, all of the teams in the competition (which are made up of 2 people) receive the same sheet with all of the connections they can create between consecutive routers and a score for each connection. One-by-one, each team member must create their own network, without knowledge about the network of their team member.

The goal is for the teams to **minimize** the sum of the score achieved by each member when creating their connected network. The teams know that there a **single** network that yields the minimum score. They also know that, according to the competition's rules, if both members create the exact same network, they receive 1.000.000 points – a guaranteed loss, taking into account the goal of minimizing the team's total score.

To avoid this scenario, Bernard agreed with his teammate Bianca that he will build the network with the lowest possible score, leaving the other options open for his teammate. On the day of the competition, they receive the following sheet:



a) Indicate which network Bernard created and the score he obtained. Use the algorithm you find the most convenient. Present the steps to reach the solution with the chosen algorithm.
b) Propose an efficient algorithm, in pseudo-code or in C++, so that Bianca can create a network that allows the team to achieve the lowest possible score. The algorithm must work for the general case, not only on the graph above. Indicate the temporal complexity of the algorithm.
c) Indicate the network and score obtained by Bianca on the specific case of the graph above with the algorithm that was proposed in the previous question.

Exercise 9

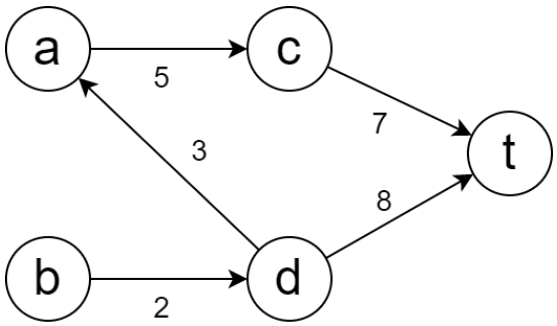
Consider a file with the following text: "ban bad bananas". The goal is to encoding this string using the minimum number of bits necessary.

a) Define a fixed-length encoding for the text. How is the text represented and what is the minimum number of bits needed?

b) Define a variable length-encoding for the text using Huffman encoding. How is the text represented and how many bits does it need?

Exercise 10

An oil extraction station is structured as a set of tanks connected by pipes. Fluids can only traverse the pipes in one direction, due to the pipes' slope. The structure of the station is depicted below, where the numbers denote how much liters can traverse each pipe per second.



All the oil that is extracted from the underground is initially stored in one of two tanks, node a or node b, before moving though the structure to reach the destination tank, located in node t.

a) What is the maximum flow for the oil that can traverse the system, from being mined underground to reaching the destination tank? Represent the problem using a capacity graph. Explain how you computed the maximum flow.

b) In the capacity graph, determine a minimum s-t cut.

c) A new pump is added to the system, allowing oil to be moved not only from tank d to a, but also in the reverse direction. The capacity of the pipe remains at 3 L/s. Does the pump increase the maximum possible amount of flow?

Exercise 11

Give an example of a capacity graph to illustrate why the choice of the algorithm to find augmenting paths in the Ford-Fulkerson method is essential for achieving lower execution times when computing the maximum flow.

Exercise 12

In a dancing event, a set of 7 men and 7 women dance in pairs over several rounds (for simplification purposes, assume all dancing pairs are composed of a man and a woman). These 7 women, however, are rather selective and will only dance with men they are friends with. The tables below show each woman's friends:

Women	**Male friends**
Anna	Anthony, Bruno
Berta	Anthony, Charles
Claire	Anthony
Diane	Bruno, Dennis
Elizabeth	Charles, George
Faith	Dennis, Fred
Grace	Eugene, George

In a given round, the 5 following couples danced:

- Anna and Anthony
- Berta and Charles
- Diane and Bruno
- Faith and Dennis
- Grace and George

a) Using these pairings as a starting point, determine a matching between the women and man that maximizes the number of dancing couples for the next round.
b) Explain why using these initial pairings saves time over finding an optimal matching from scratch.