**Documentation Project**

**Advanced Algorithm and Data Structures**

Group XX (non so che numero siamo)

Exercise 1:

Exercise 2:

*Import:*

* **time**
* **random**
* **AdaptableHeapPriorityQueue** from **priority\_queue.adaptable\_heap\_priority\_queue**

The exercise requires the implementation of a non pre-empitive scheduler. It’s has been developed the class Scheduler that inherits from **AdaptableHeapPriorityQueue**. It also stores the number of time slices after which the priority of a task in the scheduler itself must be incremented. The scheduler class has the following public methods:

* \_\_init\_\_:

The constructor of the class. It requires the parameter slice\_to\_increment.

In this method the super() constructor is called, the value slice\_to\_increment is saved and it’s defined another attribute of the class: number\_time\_slice that indicates how many time slices have been spent.

* add\_job:

This method allows the insertion of a new job in the scheduler. It requires as input parameters, the priority of the task *k* and a tuple *v* that contains the name of the job and how many time slices it requires. This method verifies that the priority of the job is a valid number (otherwise an Exception is raised), re-arrange the information present in the tuple and insert the job into the Scheduler via the *add*() method present in **AdaptableHeapPriorityQueue.**

* job\_execution:

This method prints on the output information about the current job in execution, if the scheduler is empty the message “*The scheduler has no tasks.”* is printed. This method is responsible for the update of the priority of the job, the method *update*() in the class **AdaptableHeapPriorityQueue** is used.

In the same script is also present a function, **random\_add(scheduler)**, that randomly adds a job in the scheduler. The job could be chosen among a list: *CPU*, *Memory* and *I/O.* The priority of each task is randomly generated and even the number of time slices for each task is randomly generated. When the script is run the first time, it requires as input the number of time slices after which the priority must be incremented. A check is present to verify that the user has inserted a valid number. After this “setup” phase, five jobs are added to the scheduler. In the infinite loop, it is generated a random value and if it is greater than a threshold, a new task is added to the scheduler.

Exercise 5:

*Import:*

* **random**
* **time**
* **from graphs.my\_graph import My\_graph**

This exercise requires us to implement the solution of the previous exercise considering the BaceFook network as a graph and to implement the solution with a greedy algorithm. First of all we have implemented the class **My\_graph** which inherits from the **Graph** class in the Tdp.collection package. This class adds, as slot of the Vertex inner class (which inherits from the **Graph.Vertex** class), the element ***colored.*** This element is used to mark if a Vertex of the graph should have the software or not. The class **My\_graph.Vertex** also has the methods:

* \_\_init\_\_:

this method has, as input variables, the element that should be stored and the colour for the vertex (this input variable has the default value of “False”). It calls the **Graph.Vertex** constructor via the super().\_\_init\_\_() method, which has, as input, the value that is stored in the graph. The method also assigns the value of the colored parameter to the colored slot.

* colored:

this method returns the color associated with the specific vertex.

* color:

this method colors the specific vertex.

The **class My\_graph** instead has the following methods:

* not\_colored\_vertex:

this method is a generator. It iterates over all the vertices connected to a vertex v, that is the input parameter, those are not colored.

* get\_vertices:

this method returns a list of all the vertices of the graph.

* not\_colored\_degree:

this method computes the difference between the total number of the incident edges of a vertex and the number of incident edges that, at the opposite of v, have already a colored vertex.

* get\_vertex:

this method has, as input parameter, a value. It looks for the vertex which stores that particular value.

The function **color\_vertex** implements the algorithm: its input is a graph. We iterate among all the vertices in the graph, so the specific vertex could be:

* colored, in this case we skip and pass to the next vertex;
* not colored, in this case we iterate among all the vertices connected to this vertex. If the first vertex has a “*not\_colored\_degree”* higher than the second one we color the first one and start with a new vertex, otherwise we color the second vertex and continue among the remaining vertices.

The script “Exercise5.py” also contains the generation of 100 graphs of 100 vertices, vertices are connected randomly. The proposed solution is then applied on all these graphs and, each time an execution ends, we print information about the execution time in nanoseconds and the number of colored graphs. This solution, because is a greedy solution, is 2-approximated as upper bound**….DA CONTINUARE………..**