**Documentation Project**

**Advanced Algorithm and Data Structures**

Group XX (non so che numero siamo)

Exercise 1:

Exercise 2:

*Import:*

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

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\_\_(self,slice\_to\_increment):**

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(self,k,v):**

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(self):**

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 3:

*Import:*

* **from TdP\_collections.graphs.my\_graph import My\_graph**

The exercise requires the implementation of an iterative DFS algorithm without using an auxiliary data structure. It’s has been developed modifying the Graph’s Vertex class. It has been added a parameter: **\_pre**. The **Vertex** class has the following public methods (there are other methods which are used to solve exercise 5 so they are not illustrated here):

* **\_\_init\_\_(self,x)**: the constructor of the class. It requires the parameter x that is the element that must be instanced.
* **element(self)**: this method allows to return the element of the **Vertex**.
* **pre(self):** this method allows to return the previous element from which the current **Vertex** is discovered.
* **set\_pre(self,x=None)**: this method allows to modify the attribute pre of the **Vertex**.

The script takes in input the graph on which the DFS must be performed and the source vertex. The discovered vertices are stored in a list, so the first element of this list is the source vertex. The algorithm, iterating on all the incident edges of the source vertex, checks if the other vertex (not the source one) has been already visited, with the **\_pre** attribute. If the **Vertex** has not been visited it is stored in a variable, the \_**pre** attribute is and it is added to the nodes list, otherwise continue. ####Now for the successive vertex is checked if any of the incident edges there is a not visited vertex, if there are the algorithm continue to go deeper, if all near vertex are visited the algorithm go back and search again until all the vertex of the vertex are visited and the next incident edge of the source node is checked. The algorithm stops when all the incident edge of the source node are checked and returns a list containing all the connected nodes of the graph.

Exercise 4:

*Import:*

* **from tree.concrete\_tree import ConcreteTree**

This exercise requires us to implement a Dynamic Programming algorithm that’s able to install a software on the minimum number of users of a social network so that for every pair of friends, at least one of them has the software. The network is modelled as a tree. From now on, the sentence “color a node” becomes equivalent to “install a software for a given user”. In order to solve this problem, we have provided our implementation of the **Tree** class, the **ConcreteTree** class. It contains the **\_Node** class with the following attributes:

* **\_element:** the element stored in the node;
* **\_parent:**  a reference to the node’s parent;
* **\_children:** a list that contains all the children of the node;
* **\_colored:** an attribute to specify if the node must be colored or not;
* **\_pos:** the position of the **node** in the array.

The **\_Node** class also contains the method:

* **\_\_init\_\_(self,element,parent=None,children=None):** it’s used to instantiate the class.

The class **ConcreteTree** also contains the class **Position** which inherits from the **Tree.Position** class. This class has the following methods:

* **\_\_init\_\_(self,container,node):** this method is the constructor for the class;
* **element(self):** this method returns the element stored in the node;
* **get\_parent\_position(self):** this method returns the position, in the list, of the node’s parent;
* **node(self):** returns the node stored in the position;
* **color(self):** color the node;
* **set\_array\_position(self,index):** this method is used to store the position of the node in the list;

**\_\_str\_\_(self):** this method is implemented in order to print, when the tree is visited, the element stored in the position and if that position is colored or not;

* **\_\_eq\_\_(self,other):** this method checks if this position and the other position are the same position.

The class **ConcreteTree** contains the following methods:

* **\_\_init\_\_(self):** this is the constructor of the class;
* **\_\_len\_\_(self):** returns the number of the elements present in the tree;
* **root(self):** returns a **Position** that represents the root of the tree, or None;
* **parent(self,p):** returns the **Position** of the node that is the parent of the node stored in p;
* **children(self,p):** returns a list of **Position** in which are stored the children of the node in the **Position** p;
* **\_add\_root(self,e):** this method allow us to insert as root the element e;
* **\_add(self,p,e):** this method adds, as child of p, a new **Node** that stores the element e;
* **num\_children(self,p):** this method returns the number of children of the **Position** p;
* **\_validate(self,p):** returns the node stored in the **Position** p, if it’s valid;
* **\_make\_position(self,node):** returns the **Position** in which the node is encapsulated.

The idea behind the algorithm is traverse the tree in a preorder way. For each element, we set the position in the list as attribute of the *i-th* **Position**, and we store, in the list, the *i-th* **Position** and the number of its children those are not colored (we later explain why). Then we iterate from the last element of the list to the first element and for each element we check the number of its children:

* number of children less or equal to zero: continue the execution;
* number of children greater than zero: we color it, we get the position of the parent of the element because we decrease the number of the non-colored children (if a node has all children colored we don’t need to color it) and go to the next iteration.

This algorithm is obtained from a similar algorithm that uses the recursion to solve the problem. The base cases for the recursive algorithm are:

* empty tree;
* tree with one node;
* tree with height equal to 1.

In the first two cases we don’t need to color any node, in the last case we need to color only the root. Starting from a node the algorithm is executed on its children; the starting point is the tree’s root.

To transform this algorithm in a DP algorithm we have introduced an auxiliary data structure and we have chosen a list in order to respect the specific about time complexity. The algorithm is explained above.

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, we have implemented the class **My\_graph** which inherits from the **Graph** class in the Tdp.collection package. This class adds, as attribute 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\_\_(self,x,colored=False):**

this method has, as input variables, the element that should be stored and the colour for the vertex. 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(self):**

this method returns the color associated with the specific vertex.

* **Color(self):**

this method colors the specific vertex.

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

* **not\_colored\_vertex(self,v):**

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

* **get\_vertices(self):**

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

* **not\_colored\_degree(self,v):**

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(self,value):**

this method has, as input parameter, a value. It looks for the vertex which stores that specific 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. The optimum solution for this problem is quiet difficult to express because it depends also on the topology of the graph (in a “star” graph if we color just one node we have achieved the optimum) but we can make some theoretical consideration: this solution, because is a greedy solution, is 2-approximated (according to the Graham’s Theorem) as upper bound and, because our problem is analogous to a weighted vertex cover problem we know that, according to the Dinur-Safra’s Theorem, there is, as upper bound, a 1.3-approximation solution, so we can certainly say that the performance of our algorithm is between these two bounds.