Advanced Algorithms and Data Structures Project

January 2020
Group n°5

Components

Davide Cafaro
Carmine Carratù
Francesco Chiarello
Luca Cuomo
Emanuele D'Arminio

Problem 1

Usage

In order to use the **BTree** implementation you have to install `numpy`. You can install it using the command `pip install numpy` or using the line `pip install -r requirements.txt` that will download and install the same version of the library used in developement phase.
To test the data structure we have provided a script test.py that tests all the functionalities of a BTree by calling the standard methods of the ADT **MutableMapping**.

Solution description

Data structure description and analysis of I/O complexity

We have chosen to optimize the I/O complexity instead of the computational one, since the BTrees are used to achieve low I/O complexity. In order to do that, our choice was to implement the **Node** of the BTree in such a way that every single node can be stored in just one block of memory.
Therefore, we have chosen to use a contiguous sorted array to store the items into the Node. Doing this we are sure that for each operation (Search, Insert and Delete) we have $O(log(n)/log(B))$ (where n is the number of the items in the tree and B is the maximum number of items that fit into a memory block) block transfers because every Node is stored in a single block even into the internal memory.
This second constraint is important for the I/O complexity. In fact if you use another data structure for the items, instead of a contiguous array, you can still have a Node that fits into a block of internal memory, but you could lose this property in external memory, because of **paging**.

In order to have a Node stored in a contiguous part of the memory we have encapsulated the Node into a numpy object of type `numpy.dtype`. Using a self-made dtype you can create a type that is similiar to a **C struct**. In fact, the Node is a structure that contains a numpy.array of a `dtype(key, value)` (**elements**), a numpy.array of pointers that contains the references to the children of the node (**children**), an attribute size that keeps the logic size of the Node (**size**) that is the number of items into the node) and the reference to the parent of the node (**parent**).
After having done this you can be sure that the Node is **C_CONTIGUOUS** checking the flag of the numpy object `npobject.flags("C_CONTIGUOUS")`. If this value is True, it means that the array is stored as a contiguous block of memory.

Compute the order

Now, when you want to use a BTree you can choose the types of the key and the value, for example you can use a BTree(int, int) or a BTree("U16", float64). Accordingly when we initialize the BTree we need to compute the order **d** based on the types that the user passes.
The **BLOCK_DIMENSION** can be changed into the file btree.py.
This computation is made by the formula:

$$Remaining\ dim = Block\ dim - node\ dim - size\ dim$$

$$Order = (Remaining\ dim + pair\ dim)\ //\ (pair\ dim + node\ dim)$$

Where *size_dim* is the dimension in memory of the variable size, *node_dim* is the dimension in memory of a reference (that depends on the python version that you are using) and *pair_dim* is the dimension in memory of a pair **(key,value)** of the type passed by the user.

Computational complexity

Since a sorted array was used to implement a node, the time complexity for the main operations is:

1. Add element : $O((log(d) + d) * (log(n)/log(d-1)))$
   where log(d) is the time needed for the research in a node (f(b)).
   (d) is the time needed to handle an overflow g(b) because whe need to do d insert operations.
   (log(n)/log(d-1)) is the height of the tree.
2. Remove element : $O((log(d) + d) * (log(n)/log(d-1)))$
   the only term that is different is g(b) that is the time to
   handle an underflow that depends if it is handled with a fusion
   $O(d)$ or a transfer $O(1)$.
3. Search an element : $O(log(d) * (log(n)/log(d-1)))$

- n is the number of elements of the tree
- d the maximum number of children of a node
- $log(d)$ the time needed to search an item in a node
- $log(n)/log(d)$ is the height of the BTree

Problem 2

Usage

Run the file simulator.py from either an IDE or the terminal. In order to work out imports correctly using the terminal, you should `cd` in the directory first and run the program using `python simulator.py`.

Run description

The simulator will then prompt you to insert an aging interval: the number of time slices after which a job waiting in the queue will have its priority updated. It's exactly the `x` parameter defined in the problem description.
From now on, the process of updating the priority of a waiting job will be referred to as *aging*.

Then, the simulator runs in an infinite loop, printing the current time slice as an incrementing integer which starts from 0.
In the first time slice it has no jobs to execute, so it simply asks if you want to add a new job to the queue. If the answer is affirmative, it then prompts you for the job name, priority and length.
If at a certain iteration the simulator has a job to execute, it does it before asking for a new job and prints the job name and the remaining time slices for this job.

The simulator will try to validate user input.

Solution description

The simulator uses the `LazyJobScheduler` class that you can find in the `schedulers` package.
It implements the `BaseJobScheduler` Abstract Base Class in the same package, which defines a simple Job class and provides the following behaviors typical of a priority queue:

- `get_max_priority_job`: returns the job with the highest priority and removes it from the waiting queue;
- `add_job`: adds a new job to the waiting queue;
- `increment_time`: it is used to let the scheduler know when a time slice has passed, which may then decide to do something, such as aging jobs.

Our LazyJobScheduler does not check if it needs to age jobs at each time slice: it only does it when it needs to yield the max priority job.
It does so by keeping track of the arrival time of the jobs, not of their waiting time, and of the current time. When it is time to provide a job, the scheduler evaluates all the jobs it is handling and checks whether they age and by how much - in fact, more than one aging interval may have passed for the waiting jobs since the last time they were updated. In the meanwhile, since it is already visiting all the jobs, it also keeps track of the minimum (so the highest) updated priority of the jobs, doing everything in one pass of the jobs.

Data structure description and complexity analysis

The LazyJobScheduler does not need any particular data structure to implement its behavior: a positional unsorted linked list was used.

Every time slice

In this way, the scheduler doesn't execute any actions but incrementing its time counter, so it only does a very simple $O(1)$ operation when no new job is requested.
If a job needs to be added, it is added in $O(1)$ too thanks to the unsorted linked list implementation.

Once a job has finished being processed

The scheduler visits all the jobs and does constant time operations for each one: it checks if it has passed an integer number of aging intervals in the queue and updates its priority and arrival time accordingly, just as if the jobs were updated every time slice. It then keeps track of the minimum during the visit.
At the end, it removes the selected job in constant time, thanks to the choice of the linked list.
Therefore, every time a job finishes processing, and only then, a $O(n)$ operation is executed.

Problem3

Solution

In order to implement the DFS algorithm iteratively, we have changed the structure of the Vertex class. We have added two variables:

1. **dfs_predecessor**, that represents the vertex from which we reached the current one, set by default to None.
2. **edges_to_visit**, that memorizes the iterator of the incident edges of the current vertex, set by default to None.

The implementation is composed by two functions, **dfs_complete(g)** and **dfs(g,s)** that are in the **iterative_dfs.py** module.

The **dfs_complete(g)** function takes as input a graph, instance of the Graph class, present in the graph.py module.
It looks for each not yet visited connected component of the graph received as an argument, and calls the *dfs(g,s)* function on each of these connected components. If the graph is connected, this results in only one call to the *dfs(g,s)* function.

The **dfs(g,s)** function takes as input the graph and a vertex representing the starting point of the DFS visit (the *source*).
Starting from the source, the visit consists in saving the iterator for the incident edges in *edges_to_visit*, setting the *dfs_predecessor* to None, and considering the vertices adjacent to it.
For each adjacent vertex, the algorithm checks if it has already been visited, by inspecting the value of the *edges_to_visit* attribute (finding it set to None means the vertex has not been visited yet), and, if it's not the case, it sets the source as its *dfs_predecessor*, and makes this vertex the current one. The algorithm then proceeds as for the source.
When all the vertices adjacent to a certain one have been visited, the algorithm "goes back" to the *dfs_predecessor* of the current node, by setting it as the new current one.
The algorithm terminates when the predecessor of the current node is None, because this indicates that this node is the source of the visit.

Time complexity

The time complexity of the algorithm is the same of the recursive one, $O(n+m)$.
We can prove this by analizing separately the two functions in the **iterative_dfs.py** module.
The **dfs_complete(g)** function performs a constant-time if statement on every node of the graph `g`, and a call to **dfs(g, s)**, on a certain source node, for each of its connected component.
In the **dfs(g, s)** function, the inner while performs constant-time operations on each of the edges adjacent to a current one (named *s*), and it's exectued, toghether with a constant-time if statement, for each vertex in the connected component of `g` that contains `s` (outer while). So this functions'

complexity is $O(\sum_{v \in G(s)} deg(v))$, where *G(s)* is the connected component of `g` that contains `s`. Since this function is called by **dfs_complete(g)** on each of the connected components of `g`, and the connected components are disjoint sets of vertices, this results in a total complexity of $O(\sum_{G(s) \in g}(\sum_{v \in G(s)} deg(v)))$, that is $O(\sum_{v \in g} deg(v))$, that is $O(n + m)$.

Alternative solutions

We have considered two different solutions in addition to the proposed one.
These are illustrated below.
For each of them, we report only the implementation of the **dfs(g,s)** function, because it's the core function of the algorithm.

The first alternative solution doesn't save the iterator as an attribute of the Node instance, but still adds two instance memeber to the class: *dfs_predecessor*, with the same meaning of the chosen implementation, and *visited*, that stores a boolean value, indicating wheter the node has been already visited or not.
The Python implementation of the **dfs(g,s)** function for this alternative solution is reported below:

```
def dfs(g, s):
    # visit the root of the DFS tree
    s.visited = True
    # the root has no predecessor
    s.dfs_predecessor = None

    while True:
        # Get an iterator over the incident edges in the node
        edges_to_visit = g.incident_edges(s)
        while True:
            try:
                e = next(edges_to_visit)
            except StopIteration:
                break
            v = e.opposite(s)
            if not v.visited:
                # Visit the node on the opposite side of the edge
                v.visited = True
                v.dfs_predecessor = s
                # Move on it
                s = v
                edges_to_visit = g.incident_edges(s)

        if s.dfs_predecessor is not None:
            # go back to the previous node in the DFS tree, to continue
            # the inspection of its incident edges
```

```
            s = s.dfs_predecessor
        else:
            # all the incident edges for the root have been "relaxed",
            # so all the nodes which are reachable from the root have
            # have been visited, hence the algorithm can stop
            break
```

We didn't choose this as our final solution, because the fact that we don't save the iterator implies that, every time we go back to a previously visited node, to inspect its adjacent vertices, we consider again all the edges incident to it, and this results in quadratic complexity on the single vertex, and this brings to poorer perfomances, in terms of time complexity, with respect to the chosen solution. In fact, the first edge in the iteration is considered $deg(v)$ times, the second $deg(v) - 1$, the third $deg(v) - 2$ and so on. So, for a single vertex, we have a complexity that is $O(deg(v)^2)$, and this results in a total complexity on the connected component of $O(\sum_{v \in G(s)} deg(v)^2)$, where *G(s)* is the connected component of `g` that contains `s`. By calling this function on each of the connected components of graph `g`, we obtain a time complexity on the whole graph that is $\Omega(n + m)$.

The second alternative solution instead modifies the implementation of the **incident_edges(self, v, outgoing=True)** method in the Graph class, in order to make it return a list of the edges incident to the vertex `v`, in the way shown below:

```
    def incident_edges(self, v, outgoing=True):
        """Return all (outgoing) edges incident to vertex v in the graph.

        If graph is directed, optional parameter used to request incoming
edges.
        """
        self._validate_vertex(v)
        adj = self._outgoing if outgoing else self._incoming
        # Need to explicitly create a list, because in Python3
dictionary.values() returns a dict_values object,
        # that is not indexable.
        return list(adj[v].values())
```

Moreover, to implement this solution, we have added to the Graph.Vertex class two members, *dfs_predecessor*, that we have already specified, and *edges_to_visit*, which, in this implementation, represents the index, in the list returned by *graph.incident_edges(v)*, of the next edge that we have to "relaxe", for that node. This solves the problem of considering several times edges that we had already considered, but it's still inefficient, because the creation of the incident edges list, in the way shown above, has complexity $O(deg(v))$, where v is the current vertex, and this operation is repeated each time we come back to this vertex from an adjacent one, that, in the worst case, is $O(deg(v))$ times, so this algorithm has complexity similar to the first alternative, and that's why we haven't chosen this as our definitive solution.

Additional considerations

The provided algorithm performs the same conceptual operation of the recursive one, that is, identify, for each vertex of a given connected component of the given graph, its predecessor in the DFS tree, and mark the predecessor of the source vertex of the DFS visit as None. To build a graph representing the DFS tree of one of the connected components, meant as and additional Graph class instance, some additional computation is necessary. This computation consists in creating a new graph, and fullfil it with the vertices of the connected component, linked according to the information contained in the dfs_predecessor field of the Node instances.
The function implementing this operation is not provided, since it wasn't part of the recursive DFS implementation neither. However, this does not causes our time complexity analysis to be incorrect, since this additional computation would require time $O(n)$, that would be additional with respect to the time required by the DFS algorithm, so the overall complexity would still be $O(n + m)$.

Usage instructions

To use the algorithm, it's only necessary to call the dfs_complete function, in the **iterative_dfs.py** giving as an argument the desired graph, with the only constraint that this graph is an instance of the class given in the **graph.py** file.
The **main.py** module builds a trivial graph and calls the algorithm on it.

Problem 4

Introduction

The problem is an example of a vertex cover problem on a tree. So ,for simplicity, in this and in all the documents and files related to the problem number 4, we will refer to the vertex cover on a tree problem and not to the bacefook problem.

Description of the algorithm

The algorithm that we used is dynamic and takes time O(n) where n is the number of nodes of the tree. In the following paragraphs we will describe how the algorithm works and what are the steps followed in order to build a dynamic algorithm.

How the problem is divided in subproblems:

The optimal solution for a tree, is obtained considering the optimal solutions of the subtrees rooted in the children and grandchildren of the root.

How the optimal solutions of the subproblems are combined in order to solve a problem of bigger size

Define VC(r) as the minimum vertex cover for the tree rooted in r.
For each node i of the tree we consider two cases:

1. The node i is inserted in the vertex cover, in this case the optimal solution of his children must be considered.
   $VCin(i) = 1 + \sum_{k=1}^{children(i)} VC(k)$
2. The node i is not inserted in the vertex cover.
   In this case, all the children of i have to be inserted in the vertex cover, and the optimal solutions of his grandchildren must considered.
   $VCout(i) = children(i) + \sum_{k=1}^{grandchildren(i)} VC(k)$

At the end the minimum of this two quantites is considered:
$VC(i) = min(VCin(i), VCout(i))$

Borderline cases:

1. If the tree is empty, the solutions is 0
2. If the node i has no children, the solution for VC(i) is 0

In which order the subproblems are solved:

Recursion is used in order to visit the tree, so the problems are solved in the same order of an inorder visit of a tree.

Notes

VC(i) could be evaluated multiple times, because there is overlapping between subproblems.
When the algorithm computes VC(i) for a node i it memorizes it in the node in order to avoid this problem, so when a node needs the VC() of other nodes he look for the value in the node itself without recalculeting it.

How to test the script

The function that evaluates the vertex cover for the tree is placed in the **vertex_cover_tree.py** file.
This function need as parameter a tree, that have to be an istance of the Tree class, that we implemented in the **tree.py** file.
We have not implemented all the ADT Tree interface because it was not necessary for the purposes of the excercise.

The test.py file is organized in this way:

First we create a Tree istance of the Tree class in the **tree.py** file, this is tree is build randomly using the **build_random_tree** function that takes 2 parameters:

1. The number of nodes of the Tree
2. Maximum number of children for each Node of the Tree

Then the Tree is dumped, even if the dump is very raw because we used it just for testing purpose.
After the Tree is passed to the **vertex_cover_tree** present in the **vertex_cover_tree.py** module, that evaluates the minimum vertex cover for the Tree and label the nodes as **True** or **False**, if a node is True then he is in the vertex cover otherwise no.

Problem 5

Introduction

The problem is an example of the **vertex cover problem** on a graph. So ,for simplicity, in this and in all the documents and files related to the problem number 5, we will refer to the vertex cover problem and not to the bacefook problem.

The algorithm

The algorithm that we have designed is greedy and consists of the following steps:

- for each vertex of the graph that has not been already added to the vertex cover, call it v, consider all its neighbour vertices that are not in the vertex cover.
- for v and for each neighbour, call it u, compute the degree. The degree is the score assigned to each vertex and it is the number of adjacent vertices not in the vertex cover.
- for each couple (v, u), add to the vertex cover the vertex with the highest degree. If v is added to the vertex cover, stop the iteration for the remaining couples (all the couples (v, u) are satisfied).

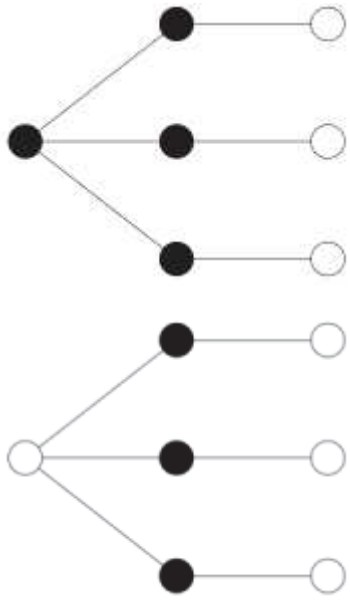
Complexity

The complexity of this algorithm is O(n + m) where n is the number of vertices and m is the number of edges.

Does the algorithm return an optimal solution?

The algorithm does not return always an optimal solution.

For example the optimal solution for this graph is the one in the first image.
Our algorithm could provide the solution of the second image, that is not optimal.



Approximation ratio

We are not able to find a precise approximation ratio for our algorithm. We notice that it's similar to an another algorithm, that we implemented and tested too. This algorithm, at each iteration, insert in the vertex cover the vertex with the higest degree (number of adjacent vertices not in the vertex cover). If a vertex has degree zero (is isolated), don't insert it. It is demonstrated that this algorithm, in the worst case, is log(n) worse respects the optimal one. So our algorithm is probably log(n) approximate.
We don't use this algorithm because is very slow compared to our algorithm (that is linear) and the performance is more or less the same.

This is not the algorithm with the best approximation ratio.
The following simple algorithm (that we implemented and tested too) achieve a 2 approximation ratio:
The algorithm simply takes ad edge and add each vertex of this edge to the vertex cover (removing or setting as removed these vertices and all the incident edges of these vertices). So, in this way, in the worst case, the solution will contain twice the number of vertices respects the optimal solution.
We decide to not use this algorithm because in practice, using the graphs randomly generated, it performs worse than our algorithm.

How to test the script

Our algorithm is a function called vertex_cover in the **vertex_cover.py** module, it takes in input a graph istance of the class Graph present in the module **graph.py** module. It evaluates the vertex cover on the graph and labels the vertices as **True** or **False**, if a vertex is True it is in the vertex cover otherwise not.
In order to test the function our implementation of the graph has to be used. Our implementation has a method that dumps a graphcal representation of the graph. If you want to use this method, you have to import networkx and matplotlib.pyplot, otherwise you don't need to import any modules for testing the function. When building a new graph, you have not to pass a value, for the element of the vertex and for the element of the edge, different from the default one. In fact, the algorithm uses these values during the computation and assumes that at the begging they are the default ones, assigned as default in the graph.

The test.py

In this module, we just call a function that we made in order to evaluate the performances of different solutions the **evaluate_performances** present in the **test_functions.py** module.
If you want to use it,you need Python >= 3.7.

This function takes as input:

1. An array of functions, that contains the functions for which the performances must be evaluated.
2. The number of graphs on which the performances must be evaluated (They are made of 100 vertices for default).
3. The probability that a vertex has and edge linking it with any other vertices.

This function creates as random graphs as specified in the second parameter using the function **build_random_graph** in the **test_functions.py** that takes in input the number of vertices and the probability specified in the third parameter.
Then each function is runned on each graph, calculating the time it takes and the value of the vertex

cover that it returns.
For each graph the correcteness of the result is tested using the **is_vertex_cover_correct**.
At the end a mean of this values is calculated and returned.

If you dont want to use this function, all you have to do is creating a graph instance of the Graph class, with the default values of elements for vertices and edges, and pass it to the vertex_cover.