Project two report

Design and Analysis of Algorithms

Alessio Pepe - 0622701463
Paolo Mansi - 0622701542
Teresa Tortorella - 0622701507

31/12/2020

0 Introduction

0.1 Work division

First of all, we would like to point out that the composition of the group for the second part of the project and therefore for its entire duration, from 15/12 to 31/12, the member Dario Zito did not participate in the work as he did communicated to the responsible professors.

The first three exercises were carried out one by each member of the group, the first by Teresa Tortorella, the second by Alessio Pepe and the third by Paolo Mansi. For all 3 exercises a phase of continuous review by the other team members followed, who contributed to improvements before reaching the final version.

For the fourth exercise the work was carried out in the following way: in a first phase all the members of the group documented themselves on the problem, having identified this as a TSP. Following an interface phase, some functions were implemented individually and often worked together, in pair programming (but in three), to put together the functions and evaluate the functioning of the algorithm.

0.2 Code Organization

The developed code is in the `project2` package. The function required in exercise 1 is in the `kc1c2_coverage.py` file. An algorithm to estimate the experimental approximation of the algorithm is provided in the file `kc1c2_experimental_valutation.py`. The functions required in exercise 2 is in `all_change.py` file. In `negative_cycle_detector.py` file there is the function required by exercise 3. What was requested in exercise 4 is provided in the `project2.exchange_tour` package.

Each module has a test code inside to verify the functionally of what is required.

0.3 Content organization

Each exercise is provided with chapter explanations on design, algorithms, and everything else necessary to quickly understand the code and the choices.

1 Exercise One

The exercise defines a **(k, c_1, c_2)-cover of T** as: *given a (2,8)-Tree T saving Currency objects, an integer k, and two currency codes c_1 and c_2, a (k, c_1, c_2)-cover of T is a set of nodes of T that jointly contain at least k currencies whose code is between Cl and (extremes included).*

The first exercise asks to implement a greedy algorithm that tries to compute the (k, C1, C2)-cover of (2-8)-Tree T saving Currency objects, with the minimum number of nodes. The algorithm must return None if there are less than k currencies in the tree whose code is between c1 and c2.

1.1 Algorithms explanation

The basic idea of the algorithm is as follows:

```
1. INPUT: T:abTree, k:int, c1:string, c2:string.
2. OUTPUT: list of currency; Null if coverage do not exists.
3.
4. FUNCTION search (t:ABTREE, k:INT, c1:STR, c2:STR) : LIST
5. DO
6.     solution: LIST := []
7.     N := 0
8.     p: POSITION, i: INT := t.search(c1)
9.
10.    // Adjust position
11.    IF p(i) not in [c1, c2] THEN
12.        p, i := t.after(p, i)
13.    END IF
14.    REPEAT
15.        IF p not in solution THEN
16.            solution.add(p)
17.        increment k bt the number of elements in [c1, c2] in p
18.        p, i := t.after(p, i)
19.        IF reached end of tree THEN
```

```
            BREAK
        END IF
    UNTIL n < k AND c1 <= p(i) <= c2

    RETURN solution or Null IF n < k
END FUNCTION
```

The first step of the algorithm involves the search for c1, or a neighbor if that does not exist. If the position and the index found through the search are not None, they are saved, otherwise None is returned. If the neighbor is before c1, the algorithm moves forward.

The search continues in following currencies until you reach coverage of at least k.

If the position found, whose key is included in the requested range, is not present in the solution list, this is added.

Checking in all elements of the node, if the key of a given position is between c1 and c2 the counter is incremented.

The algorithm returns None if there are less than k currencies in the tree whose code is between c1 and c2.

1.1 Design choices

It was decided to use a solution list in which the found positions are saved and a counter, initialized to zero, to check coverage.

The greedy algorithm developed runs in a time that is at most

$$O((n^2)/(log^2 a)(b-1))$$

The optimal solution, if the tree has node in which all the currencies (or all the necessary ones) belong to them, is given by ($\frac{k}{b}$ - 1 rounded up) nodes. The solution given, in the worst case, first sees k nodes where there is only one element of the `(k, c1, c2)-cover`. So, it returns k nodes in the worst case.

Therefore, the solution of the algorithm differs from the optimal solution of

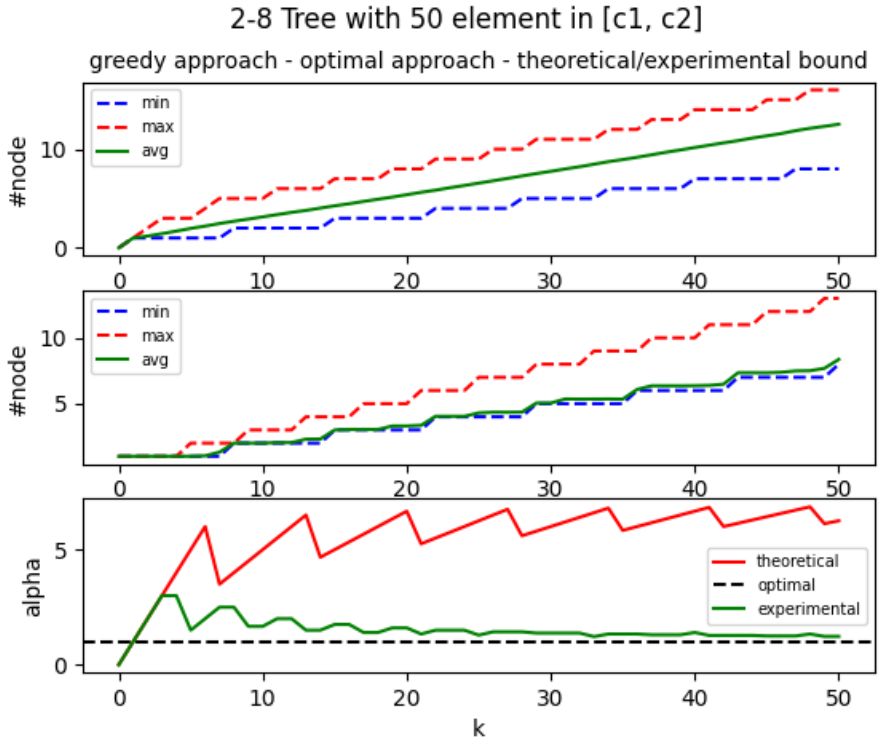
$$ALG(I) \geq (k)/(k/(b-1)\, rounded\ up) OPT(I)$$

So, in the worst case, we have that:

$$ALG(I) \geq (b-1)\, OPT(I)$$

The solution therefore differs a lot when k approaches a multiple of $b-1$ from the left, while it is much more acceptable in the rest of the cases. Furthermore, the worst case described above Is more likely to occur if C1 and C2 are very far from each other, otherwise the probability of obtaining the worst case is much lower.

Using a Monte Carlo simulation, the performance of the algorithm was estimated and compared with an optimal algorithm. Furthermore, the worst case of the greedy algorithm is shown compared with the formal estimate of the approximation.



As hypothesized before the experimentation, in a worst-case situation when k becomes close to the size of the set of currencies to be hedged, a result is reached which tends to become identical.

On average, as can also be seen from the estimate of the alpha parameter, the algorithm never finds a solution that differs much from the optimal one, or in any case in the experimentation never more than double.

1.2 Test module

In the `k-c1-c2_coverage.py` module, following the algorithm's implementations, a test code is provided to verify the functioning of the implementations.

2 Exercise Two

The second exercise requires the implementation of an algorithm that, given in input a Currency and a float number r returns the number and the list of changes in which the float number can be returned using the currency's denomination.

2.1 Design choices

A naive idea could be to generate all possible sequences with an exhaustive algorithm who explore all solution 's space. Unfortunately, such an approach would lead to calculating the same solutions many times, especially for small values. For this reason, it is advisable to use dynamic programming.

Subproblems: generate the possible sequences of cuts from a set $S' \in S$ that form an $n < r$ value.

Base case: to return an empty value there is only one solution, that is no coin.

Characteristic equation:

$$\mathrm{T}[i][j] = \mathrm{T}\big[i - S[j]\big][j] \text{ if } -\mathrm{S}[j] \geq 0 \text{ else } 0 \; + \; \mathrm{T}[i][j-1] \text{ if } -1 \geq 0 \text{ else } 0$$

that is, for the value $i < n$, the sum of the possible solutions including the coin $S[j]$ and those without including the coin $S[j]$.

There is no way to go back to the solutions without saving them, so it must be considered that the copying of the lists has complexity $\#\text{solutions} \cdot \#\text{denomination}$.

To save space in memory, the bottom up technique is used where the problem is the same and the characteristic equation is identical, but it only needs to be increased for each coin considered since the solutions excluding that coin are already present in the cell considered.

The implementations, in both types, are implemented in the functions:

- `all_change(cur, r)`, implemented with dynamic programming, which has time complexity $O(r \cdot \#\text{denomination} \cdot \#\text{solutions} \cdot \#\text{denomination})$ and the same space complexity.
- `all_change_bottom_up(cur, r)`, implemented with dynamic programming with bottom up approach, which has time complexity $O(r \cdot \#\text{denomination} \cdot \#\text{solutions} \cdot \#\text{denomination})$, but $O(r \cdot \#\text{solutions} \cdot \#\text{denomination})$ space complexity.

2.2 Test module

In the `all_change.py` module, following the algorithm's implementations, a test code is provided to verify the functioning of the implementations.

3 Exercise Three

The third exercise requires the implementation of an algorithm for the detection of an arbitrage opportunity for a given currency s, which means a sequence of changes that starts and ends with s such that the total exchange rate of the sequence is negative.

3.1 Design choices

To deal with the different path from one currency to another, it was decided to use the Graph structure, implemented as an Adjacency Map, provided by the `TdP_collections` library. It was used a complete directed graph where each vertex represents a currency and each edge `e(u, v)` represents the rate exchange from the currency u to the currency v.

There were also used a dictionary that associates the currency code to the related vertex, another one to store the distance needed to reach a given vertex and a third one for tracing back and reconstructing the cycle.

3.2 Algorithms explanation

As first step of the algorithm, given the set of currency C, the graph is constructed in time complexity $O(n + m)$ where n is the number of currencies and m is the number of rate exchange between currencies.

Then, for the detection of a negative cycle within this graph, an adjustment of the Bellman-Ford algorithm was implemented. For this reason, the greater time complexity that also determines the order O is that of the Bellman-Ford algorithm: $O(n \cdot m)$. Considering that the graph is complete, the number of edges for each node is $n - 1$, so we can write that the complexity of the algorithm is close to $O(n^2)$

According to the Bellman-Ford algorithm, the relaxation of all the edges can produce the alteration of the distance needed to reach each vertex at most V times (where V is the number of vertices) when the graph does not present a negative cycle. Therefore, whether the Bellman-Ford theorem is not satisfied, a negative cycle is present in the graph.

To reconstruct the cycle, at each relaxation of the edges, it is done the update of a dictionary containing for each vertex the one that discovered it, so that it is possible to trace back and find a negative cycle. It is also kept track of the total earn achieved with that cycle.

This algorithm can detect one of the negative cycles present in the graph. If more than one is present, only one of them will be returned, according to the order of the edges' relaxation during the execution of the Bellman-Ford algorithm. Therefore, the cycle returned is simple if contains the given currency s while, if it does not, it contains repeated vertices. In fact, in this last case, the simple cycle is repeated until the total gain is greater than the cost to reach the cycle from s and get back.

The algorithm, so, returns a negative cycle starting and ending with the given currency s, if that cycle exists, `None` if it does not.

3.3 Test module

In the `negative_cycle_detector.py` module, following the algorithm's implementation, a test code is provided to verify the functioning of the implementation. In this test, three different graphs were constructed, and, for a better understanding, a visualization of these graphs is provided in the directory `arbitrage_opportunity_img`.

4 Exercise Four

The third exercise requires the implementation of a local search algorithm for the detection of an exchange tour of minimal rate, which means a sequence of changes that explore each currency only once such that the total exchange rate of the sequence is the minimal possible.

4.1 Design choices

To deal with the different path from one currency to another, it was decided to use the Graph structure, implemented as an Adjacency Map, provided by the `TdP_collections` library. It was used a sparse undirected graph where each vertex represents a currency and each edge `e(u, v) = e(v, u)` represents the rate exchange from the currency u to the currency v.

Considering a graph, this problem be a particular application of the "Travel Salesman Problem", which deals with finding the minimal Hamiltonian cycle on the graph. This problem is considered NP-complete, even on a restrained input. In addition to this, the graph constructed with the given set of currencies is likely to be a sparse graph, which implicate the possible nonexistence of an edge between two currencies. Therefore, the problem is faced in two different steps: first a Hamiltonian cycle needs to be found and then this cycle needs to be optimized until finding the minimal.

The algorithm used for finding the first Hamiltonian cycle is like the `HybridHAM` algorithm but adapted and optimized to this problem. The function used are provided inside the `hybridham.py` file inside the `exchange_tour` package.

The algorithm used for optimizing the Hamiltonian cycle is like a simplification of the LK heuristics, which make use of rotations on two and three edges to reach the optimality. The function used are provided inside the `two_three_opt.py` file inside the `exchange_tour` package.

Inside the `exchange_tour.py` file, within the `exchange_tour` package, the final algorithm is provided. Since it's based on a heuristic, this algorithm doesn't guarantee to always find a Hamiltonian cycle and, since it makes use of a local search algorithm, it doesn't guarantee to

always find the minimum cycle either. Though, the complexity of this algorithm is at most $O(n^3)$, which guarantees a good trade-off between time efficiency and results quality.

4.2 Algorithms explanation

The first step is to use our Hy`bridHAM` implementation, which uses nearest neighbor logic, to search for any Hamiltonian loop. Then call our optimization algorithm.

- `HybridHAM` is divided into 3 phases: find a long enough path, convert it into a Hamiltonian path and finally convert the path into a Hamiltonian cycle. All 3 phases be three LS algorithms:
    - the first takes a path as input and returns as the best neighbor a longer path where the vertex with the lowest possible degree has been added.
    - the second as an algorithm that takes a path and returns a new, longer path as a neighbor, until a Hamiltonian path is found.
    - the third takes a Hamiltonian path as input and chooses among its neighbors to obtain a new Hamiltonian path that is also a Hamiltonian cycle.
- `two_three_opt` is an algorithm inspired by LK heuristics, in a dramatically simplified way. The operation is as follows: given a cycle try to reduce it with 2opt many times until there are variations, when it does not produce any results a 3opt simplification is applied (reduced of the moves already included in the 2opt). This for several times depending on the logarithm to the base two of the number of currencies in the set. When no changes occur in all these operations, the algorithm terminates.

4.3 Test module

In the `exchange_tour/exchange_tour.py` module, following the algorithm's implementation, a test code is provided to verify the functioning of the implementation. In this test, four different graphs were constructed, and, for a better understanding, a visualization of these graphs is provided in the directory `exchange_tour/dataset_img`.