Practice 2

Abraham Álvarez Cruz

Department of Computer Science
School of Engineering
University of Cádiz

abraham.alvarezcruz@alum.uca.es

16th November 2023

Summary

In this report, we will address the problem of *Exam Scheduling*, providing a solution implemented in C++. I aim to present all aspects related to the problem and its characteristics, along with an analysis of the results obtained from the proposed solution.

Contents

Tables

Figures

1 Introduction

Resource allocation problems are highly valuable from a computational perspective as they can be extrapolated to various domains and tasks closely associated with it. In our specific case, we are addressing the issue of exam scheduling. This problem is characterized by the absence of an analytical solution that would enable us to directly obtain the optimal solution (or at least one of them if multiple optimal solutions exist). The absence of an analytical solution compels us to explore alternative approaches and methods to address the problem. Various methods and approaches exist for finding solutions, but they all share a common challenge—they are computationally expensive, exhibiting exponential complexity in their algorithms.

Our approach has been to reduce the problem to one for which known solutions exist, albeit also of exponential order—the graph coloring problem. To solve the graph coloring problem, we employ a backtracking search scheme. In this scheme, at each step, we process a specific node of the graph under consideration, constructing a partial solution until a set of conditions is met. At this point, the generated partial solution is returned, and we can assert that we have a final or complete solution[1].

The primary challenge with these backtracking search algorithms is their exhaustive search nature, making them inherently slow and time-consuming in finding a solution or potentially not finding one at all. This is because the fact that this family of algorithms systematically checks all possible combinations generated at each step, creating a search tree that explores all potential situations in the problem's search space.
There are methods to optimize these algorithms by seeking ways to "prune" the search tree, constraining the search and reducing the number of operations.

[1]Depending on the algorithm's implementation, the term "final solution" may not necessarily be a valid solution to the problem but rather a partial solution in the process of construction.

2 Methods

To solve the problem, we begin with a pseudocode that provides a solution using the aforementioned backtracking technique. In this algorithm, a small enhancement has been incorporated to prune the search tree when a certain condition is met. Essentially, this condition entails not proceeding along branches where "more colors are used than those required by the best solution found so far"
The pseudocode for the implemented algorithm is displayed in Figure 1. To work correctly, the algorithm would require:

- **G**: Graph to be processed by the algorithm, containing a set of vertices (**V(G)**) and edges (**E(G)**).
- **k**: Largest color used up to the current point in a partial solution.
- **B**: Smallest color used in a complete solution[2].
- **C**: Vector with sets of colors assignable to each vertex of the graph; thus, **C[v]** would represent "the set of colors assignable to vertex v."

The behavior of the algorithm would be approximately as follows:

1. The algorithm checks if there are remaining vertices in the graph to process. If there are none, it returns the largest color used up to that point in the processed branch (k).
2. When there are vertices left to process, it ensures that each of them can be assigned colors (at least one) smaller than an upper bound (B).
3. Since there are vertices left to process with colors smaller than the bound, it implies that we can find a more optimal assignment for the graph vertices. Thus, a vertex (v) is selected, and for each possible color that can be assigned to v and is smaller than the upper bound (B), the following steps are taken:
   (a) Remove the chosen color (c) from the set of colors assignable to the neighbors of the processing vertex (v).
   (b) Make a recursive call to the function, passing the *larger value between the current k and c* (chosen color) as the parameter k.
   (c) The algorithm returns a value, let's call it B'. Update the value of B with B'. This is done unilaterally because, in the worst case, B' will be the same as B, and in any other case, it will be smaller, indicating a better solution than the current one.
   (d) After the recursive call, return the chosen color to the set of colors assignable to the neighbors of the processing vertex.

[2]A complete solution is one that belongs to the set of solutions to the problem, in other words, it is a valid solution to the problem.

```
backGCP(G, k, B, C) → B
if V(G) = ∅
  return k
for all v ∈ V(G)
  if no-colour-under-bound(C[v], B)
    return B
v ← select(G)
for all c ∈ C[v]
  if c < B
    for all u ∈ adjacents(v)
      remove(c, C[u])
    B ← backGCP(G - {v}, max(k, c), B, C)
    for all u ∈ adjacents(v)
      insert(c, C[u])
return B
```

Figure 1: Pseudocode of the Graph Coloring Algorithm.

4. Finally, after processing all possible assignable colors to vertex v, return B, which will be updated to the best value found when processing the current branch.

As we can observe, the algorithm doesn't become overly complex, although it is true that it is not very intuitive due to the incorporation of recursive calls, which complicates the understanding process.

In our case, the implementation of this algorithm requires a series of small additional changes to allow for the storage of the color assignment made. I achieve this by adding an additional parameter, bS, which is a vector where I store, for each vertex in the graph, the color that has ultimately been assigned to it.

Building upon the aforementioned description, the changes I have incorporated into the algorithm are as follows:

1. Remains unchanged. It returns k along with the vector bS.
2. Remains unchanged. It returns B along with the vector bS.
3. Remains unchanged. Additionally, a copy of bS is created (let's call it $bSCopy$). This copy will contain the best assignment found during the recursive calls.
   (a) Remains unchanged.
   (b) Remains unchanged.
   (c) The recursive call would now return two values: B and bS. The update of B is not unilateral since if B is smaller than B, it implies that we have found a better solution, and therefore, we need to update $bSCopy$.
   (d) Remains unchanged.
4. Remains unchanged. It returns the value of B and $bSCopy$.

With the small changes made to the original algorithm, we can now determine not only the minimum number of colors needed to color the graph but also the specific color to assign to each of the vertices.
In practice, the implementation of the algorithm in $C++$ is a bit more complex as it requires considering certain details that may not be apparent at a higher level. Some of these considerations include:

- Use or non-use of pointers.
- Structures used to represent entities (graph, colors, etc.).
- Etc.

n my case, for the representation of the graph, I utilized a class seen in the EDNL subject where an adjacency matrix is employed. I removed some superfluous methods and functions and introduced others to assist in solving the problem or reduce the complexity

of certain operations.
I generally opted for pass-by-value instead of references (even though it involves additional cost) since these algorithms are complex to debug. I preferred to provide a solution (albeit more costly) in a shorter time rather than presenting a more optimal solution at the expense of spending too much time on the subject, paying no attention the rest.

Nevertheless, the presented implementation could be modified with seemingly little complexity, helping to reduce the cost of copying certain expensive structures (vectors and sets).
Finally, the practice guide proposed a small optimization, which involved "selecting the vertex to process based on its degree in the graph." This improvement entails choosing first "those vertices that are more conflictive," being adjacent to a greater number of vertices. By selecting these vertices first, we can reduce the size of the search tree because we can "detect more quickly (in earlier stages) those branches that will not lead to a solution to the problem, thus avoiding postponing this discard task to deeper levels of the search tree."

I implemented this modification in parallel with the non-optimized version. In the following section, I present the results obtained for both versions.

3 Results and discussion

In general, we are already familiar with the asymptotic behavior of the algorithm we are working with. When conducting an exhaustive search, it is nearly impossible for an algorithm to have a time complexity lower than exponential, meaning k^N, where k is a constant value.
Belonging to this category, we can assert that the algorithm is *computationally expensive* or *computationally inefficient*.

3.1 No-Optimized Version

The previous hypothesis aligns well with Figures 2 and 3. In these figures, it can be clearly observed that the asymptotic complexity of our algorithm lies somewhere between 2^N and 10^N. This aligns with our initial idea that this algorithm does not belong to the set of *computationally efficient* algorithms.

It's important to note that, to obtain the results presented here, I had to use a relatively small input size (specifically $N = 12$). For the practice guide proposal, with $N = 15$, the computer took approximately 20-25 minutes to calculate the solution and did not complete the process, emphasizing the computational cost of the algorithm.

3.2 Optimized Version

The optimized version, while yielding slight improvements in the time taken by the algorithm to return a solution, doesn't manage to significantly reduce the algorithm's complexity, still belonging to the same order as in the previous version.
In Figures 4 and 5, I present the results obtained by the optimized version of the algorithm. Again, it can be observed that the asymptotic complexity remains unchanged. This optimization stands as a minor improvement with almost imperceptible residual effects on its asymptotic impact.

Finally, to visualize the improvements in absolute terms of the optimized version compared to the non-optimized one, I have compiled the times obtained for each input size in Table 1.

Input size	Time Taken (optimized version)	Time Taken (classic version)
2	8	6
3	11	18
4	24	48
5	89	216
6	446	1281
7	3327	8206
8	28175	31748
9	260039	269140
10	2866567	2901361
11	33604228	3410909
12	435000891	435730094

Table 1: Times for the execution of the algorithm. The measure is in microseconds.

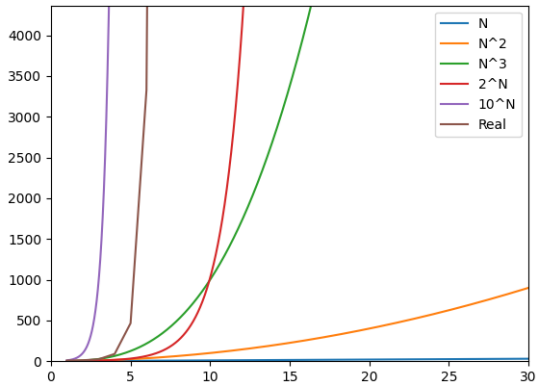

Figure 2: Time vs. Input Size for the non-optimized version of the graph coloring algorithm.

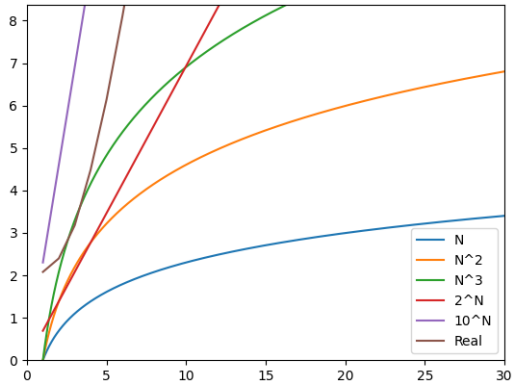

Figure 3: Logarithm of Time vs. Input Size for the non-optimized version of the graph coloring algorithm.

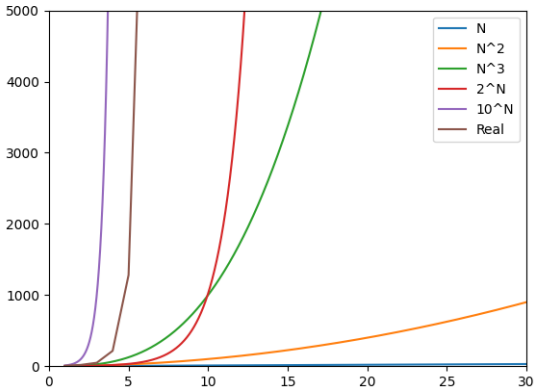

Figure 4: Time vs. Input Size for the optimized version of the graph coloring algorithm.

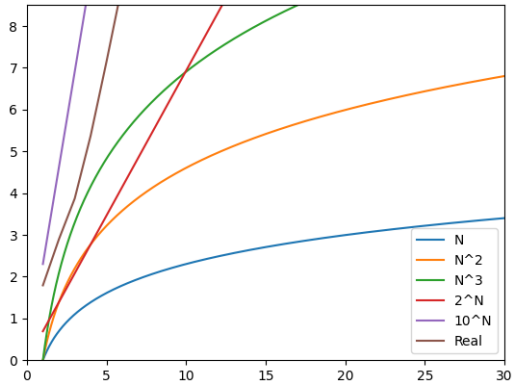

Figure 5: Logarithm of Time vs. Input Size for the optimized version of the graph coloring algorithm.

4 Conclusions

The algorithm in pseudocode is easy to understand once you examine it closely, although it may take a bit of effort to grasp initially. The implementation in *C++* did pose some challenges for me (especially when dealing with pointers), but I was able to address the issues that arose without too much difficulty.
Regarding the problem instance generator, with the numbers presented in class, it is impossible to obtain a non-complete graph. Each and every generated instance has created conflicts with the remaining vertices, causing the algorithm to always be in the worst possible case (having to traverse a greater number of nodes in the search tree).
Also, when calculating the times for each version of the algorithm, I encountered issues due to the amount of time it took for values of $N > 12$. That's why I had to limit the input sizes of the algorithm to a lower range to obtain sufficient times for visualizing the approximate shape of the function on a graph.

Even though all the cases studied have led to an almost exhaustive search of the search tree of the problem, it has helped me better understand the challenges of such algorithms that need to perform complete searches in their search tree. Additionally, it has demonstrated how we can solve a real-world problem (closely related to other types of problems from different areas) by reducing the problem to a known one and for which we have an algorithm that can provide a solution.

Bibliography

[1] Oded Goldreich. *P, NP, and NP-completeness. The basics of computational complexity*. Cambridge University Press, 2010.