Projet d'Algorithmie Appliquée

Amira ELOUAZZANI
Max SENRENS-MAUNY
Théo MORIN
Côme PERIN

21 janvier 2025

1 Introduction

This project aims to implement one or more solutions to improve the metropolitan/-railway network in Île-de-France, particularly in the context of the Paris 2024 Olympic Games.

The project will be carried out in several stages :
— Problem modeling
— Implementation of solution algorithms
— Addition of extensions

The subject is available here.

2 Problem Modeling

To model the problem, two types of points of interest will be considered :
— Metro/train stations (often denoted as S)
— Olympic sites (often denoted as O)

First of all, we focus on the following problem :

Main problem

INPUT : **Metro stations** and **Olympic Sites** of *Paris 2024* and $x \in \mathbb{R}^+$
OUTPUT : Find the smallest set of metro stations to be developed such that all Olympic venues are at most x minutes walking distance from an accessible metro station.

To do so, edges are drawn between **metro stations** and **olympic sites**, if and only if the walking time between them is lower or equal to x. This construction process could be time consuming if not carried out properly, thus, refer to 4 to see further details.

3 Problem Formulation

The problem is to find a certificate for the `BIPARTITE ANNOTATED DOMINATING SET` problem, for a minimal integer k.

Bipartite Annotated Dominating Set (BADS)

Instance : A bipartite graph G with marked vertices and an integer k.
Question : Does G have a set of vertices S of size k such that every vertex not colored with c is either in S or has a neighbor in S ?

This problem is a special case of the `ANNOTATED DOMINATING SET` problem.

Annotated Dominating Set

Instance : A graph G with marked vertices and an integer k.
Question : Does G have a set of vertices S of size k such that every unmarked vertex is either in S or has a neighbor in S ?

The graph modelisation given by 2 ensures that the resulting graph is bipartite. Each **metro stations** is then marked. `BADS`'s certificate provides a dominating set of the unmarked vertices (*i.e.* **olympic sites**). A valid certificate (with a minimal k) easily leads to a solution of problem 2. The only case that needs to be handle is if an **olympic station** ends up in the dominating set of vertices. Knowing that the set of vertices has size k, the olympic station can be removed and replaced by one of its adjacent **metro station**.

4 Graph construction

4.1 Construction of vertices :

Vertices are simply parsed from a `json` file containing informations about **metro stations** and **olympic sites**, according to the modelisation given by 2.

4.2 Edge Construction :

In this section two methods of edge calculation will be presented. The first one is simple to realize and gives acceptable computation time, wereas the second one is more complex but gives better computation time.

4.2.1 Edge Construction by Pair Computation

This construction simply consider each pairs of **(metro station, olympic site)** and computes the walking time between them (using `shortest_path_length` from **osmnx** library). However, simply considering every pair of this kind in the whole graph would be too time consuming, because of time complexity of `shortest_path_length`. Instead, we only consider the pairs of the form **(S, O)** where **O** is an olympic site and **S** is a metro station *enclosed in a ball centered in* ***O*** *with radius* $x * walking\_speed$. Since the geodesic distance is always lower than the real distance, every **S** that should be adjacent to an **O** is indeed enclosed in this ball.

4.2.2 Edge Construction by *Dijkstra*

First solution :

The main issue with the previous method is that even if we restrict the set of pairs, executing `shortest_path_length` successively remains unefficient. The next idea is to execute the *Dijkstra* algorithm on the graph representation of the world of **osmnx**. This algorithm computes every distances from a point **O** within a ball that grows over the execution. The computation is stopped once this ball has radius $x * walking\_time$, ensuring that the set of stations that need to be adjacent to **O** is equal to the set of stations that has been visited.

Second solution :

We wanted to test another implementation to compare its performance with an alternative solution. In this new approach, Dijkstra's algorithm is used via `single_source_dijkstra` to calculate the distances between a given Olympic site and all the nodes in the graph in a single operation. This method allows all necessary distances from an Olympic site to be pre-calculated, avoiding the need to recalculate distances for each station individually. It is particularly advantageous when several stations are located within the area of influence of a site, as it considerably reduces the number of redundant calculations, making full use of the graph structure and the advantages of Dijkstra's algorithm.

Using multithreading to generate osmnx graphs :

For our last solution, we wanted to parallelize the generation of the graph `osmx`. Multithreading allows us to take advantage of the parallelization of calculations for each Olympic site. As each site requires the creation of a graph and the calculation of distances via Dijkstra, these operations are independent of each other. By using a pool of threads or processes, multiple sites can be processed simultaneously, making efficient use of multicore processor resources. This is particularly advantageous in scenarios where the number of sites is large and osmnx graphs require considerable computing time. Multithreading thus enables a drastic reduction in total processing time, while maintaining a balanced workload on available resources.

5 Solutions

Some of our realizations are explained in this section. The existence of a solution can be checked linearly before any useless computations : we ensure that each **olympic site** has at least **one neighbor**.

5.1 Brute Force

The first approach down-to-earth is to enumerate every possible solutions (*i.e.* set of stations vertices) by increasing the size of the solution. When a solution is found, we are then ensured that it has the smallest possible size. However, altought straightforward, this method is **exponential** ($O(2^{\#metro\_stations})$ and inconceivable on real applications.

5.2 Progress

For this method, we aimed to be more strategic by creating both a solution and a witness that demonstrates the absence of a solution, thus significantly reducing the

number of configurations we need to test.

The first implementation focused primarily on reducing the number of configurations without including any progressive aspect. It was a straightforward application of the algorithm described in the proof of the complexity of step 2.a of the progress algorithm from the course material. The key idea revolves around "profiles" : each vertex in the graph that is not an Olympic site (i.e., a metro station) has a profile. This profile is represented as a binary word where the i-th bit is 1 if the vertex is a neighbor of the i-th Olympic site and 0 otherwise.

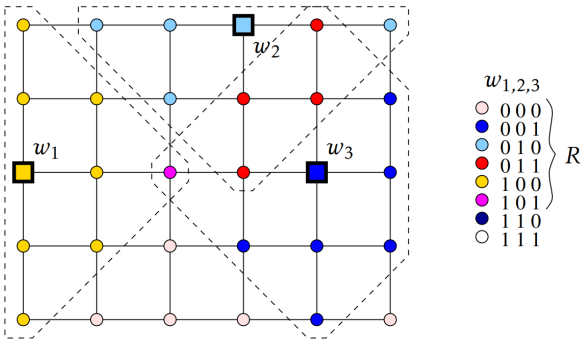

FIGURE 1 – In this example, profiles = colour, only 6 different profiles but can be reduced to only two by inclusion, source : *https ://dept-info.labri.fr/ gavoille/UE-AA/cours.pdf*

Once the profiles are created, we sort them by their decimal value using Python's built-in `sorted()` function. Initially, we used a custom implementation of counting sort, but it was not suitable for our data since the binary words were too long. After sorting, we eliminate all unnecessary profiles, such as null profiles and profiles that are included in another one.

For instance, the profile `[00110]` is included in `[10110]`, so we discard `[00110]`.

After this reduction step, we are left with significantly fewer profiles. The next task is to find the smallest union of these profiles that forms the "full" binary word `[111...1]`. Unfortunately, we do not have a better option than to try every combination (i.e., a brute-force approach), but this is efficient in practice because many useless profiles have already been removed.

This version was quite efficient, except for the profile creation step due to a poor initial implementation, which we plan to optimize in the next versions.

The latest version introduces the progressive idea by building a witness W smaller than the full set of Olympic sites and iterating on this. We start with W containing only one vertex and search for a dominating set. If we cannot find one, we return `false`. If we find one, we then check whether this set is also a dominating set for the entire graph. If it is, we return `true` ; otherwise, we add a non-dominated vertex (an Olympic site) to W and continue searching for a dominating set for this new W.

The main issue with this version is that it needs to recompute the profiles multiple times, as each iteration uses a different set of Olympic sites. However, the profile computation is costly, making this version slower than the initial, non-progressive implementation, on some examples. Once the profile creation step is optimized, this version should outperform the previous one.

This was the most crucial step because profiles are the corner stone of the algorithm and many bugs we faced along the course of the project originated from them. There were

issues of non-consistent profile when using the programs multiple times on the same graph, wrong profiles, bit that should be 0 and were 1, profiles being modified unwillingly and so on. But in the end we found a proper and simple implementation based on Adjacency lists that cleared previous bug and was efficient for the algorithm.

5.3 Branch and Bound

Branch-and-Bound Approach :

Branch-and-Bound is a recursive method that explores the solution space to find the minimum dominating set of stations. The algorithm systematically eliminates branches that cannot yield a better solution than the best one found so far. This elimination is based on a bounding condition :

$$\text{if } |S| \geq |S_{\text{best}}|, \text{ prune the branch.}$$

Here, S represents the current partial solution, and S_{best} is the best solution found up to that point. If adding more stations to S cannot improve the solution, the algorithm backtracks.

Heuristic Selection :

To enhance efficiency, the algorithm uses a heuristic to prioritize the most challenging sites to cover. Specifically, the heuristic selects the undominated Olympic site u with the smallest number of neighboring stations :

$$u = \arg\min_{o \in \text{U}} \text{Deg}(o),$$

where U is the set of undominated Olympic sites, and $\text{Deg}(o)$ is the degree of site o (i.e., the number of neighboring stations within the walking distance threshold). By focusing on these bottleneck sites, the algorithm reduces the search space and accelerates convergence.

Pruning and Recursive Search :

The algorithm explores potential solutions recursively, adding stations to the partial solution S one by one. At each step :

- **Edge Tracking :** It maintains a set of dominated edges (connections between stations and Olympic sites) to avoid redundant domination.
- **Unique Coverage Check :** A station is only added to S if it provides unique coverage for one or more undominated edges.
- **Backtracking :** Once a branch is fully explored, the algorithm backtracks and removes processed sites from consideration.

Solution Refinement :

After a solution is found, a post-processing step removes redundant stations to ensure the solution is minimal. Stations that do not contribute unique coverage to any edge are excluded :

$$S_{\text{refined}} = \{s \in S \mid \text{DominatedEdges}(s) \not\subset \text{AllDominatedEdges}\}.$$

This step ensures that the final solution represents the smallest set of stations necessary to dominate all Olympic sites.

Complexity Analysis :

— **Time Complexity :** The worst-case complexity is exponential, $O(2^n)$, where n is the number of stations. However, the use of heuristics, pruning, and edge tracking significantly reduces the practical runtime.
— **Space Complexity :** The algorithm requires $O(n)$ space for the recursion stack and storing the current partial solution S.

This approach effectively balances exhaustive search with intelligent pruning and heuristic-driven exploration

6 Results presentation

6.1 Graph and user input management

6.1.1 User input

This module interacts with the user to define walking constraints and select a resolution method to calculate optimal connections between stations and Olympic venues. The user can enter a walking time in minutes to limit the scope of calculations, while ensuring that a certain time limit is not exceeded.

— A function prompts the user for a running time in minutes to limit the range of calculated connections.
— Another prompt allows the user to choose one of three resolution methods (Brute Force, Progress, Branch and Bound).

6.1.2 Graph creation and management

This module provides a solution for creating or loading a graph from stored data. The graph is either generated from the Graph or Graph2 class, or loaded from a pickle file to avoid recalculation at each runtime. This mechanism guarantees efficient resource management and time savings, while allowing flexibility in connection analysis.

6.2 Random Graph Creation

A system of benchmark was made in order to compare the diferent methods in terms of rapidity.

As seen, creation of a graphs is time consuming. In order to test our implementations on many and various graphs, a *peeling method* is used. The big graph is computed once and for all, afterwards, many random graphs are obtained by getting a subgraph of the first one. To do so, a certain number of vertices are removed as well as the edges they were connected to. Hence, we have an efficient method to get graphs to execute our benchmark on.

6.3 Benchmark

The Benchmark is quite simple : on each graph, each method is executed a defined number of times, and the average execution times are then compared and displayed. However, the brute force method is only executed when the graph is small enough, to avoid big computation times.

6.4 Results and Comparisons

Figure 2 has been obtained by running the benchmark and exploiting the given results.

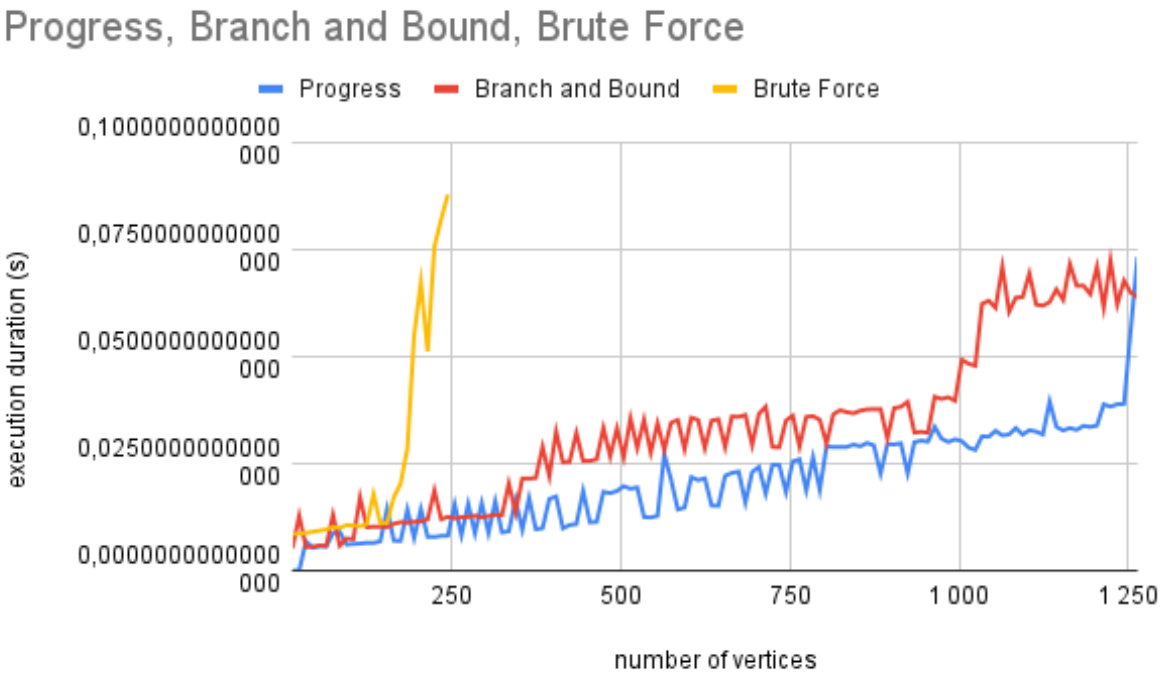

FIGURE 2 – Results of the benchmark

As predicted, the execution time of the *Brute Force* method seems exponential on the diagram. However, every results here are very *graph-dependent*, which means one execution on one graph can not be fully representative of the method, for instance, Brute Force could easily find small solutions on big graphs.

Nevertheless, our implementation of *Progress* seems quicker than the implementation of *Branch and Bound*.