M1 Semester 2 Project

Optimization of parcel delivery systems

Year 2020-2021

Thibaut JUZEAU, Leo TILLIEU, Charles FENART,
Chunyang WANG, Zhenyu XIANG, Eloi TEXIER

TABLE OF CONTENTS

1 PROJECT APPROACH

1.1 MOST KNOWN PROBLEMS SOLVED / GENERALIZED

1.1.1 TSP – Travelling Salesman Problem

The Travelling Salesman problem is very simple to understand but also very complex to solve. A traveller plan to visit several connected cities once and he wants to minimize his total distance. The difficulty lies in the number of potential paths. For n vertices, we get:

$$Nb\ solutions = \frac{(n-1)!}{2}$$

Formula explanation: we remove the initial vertex, so we get n-1 vertices. The factorial represents the possibility of going to all the other vertices from one of them This problem is a particular use case for this project. And finally, this number is divided by two because order does not matter.

This problem is a **particular use-case** of our solution.

1.1.2 VRP – Vehicle Routing Problem

The vehicle routing problem is another very easy to understand but hard to solve problem. Here we have a single deposit with several delivery men who each have a specific capacity, that must serve clients with variable size of command.

This problem is a **particular use-case** of our solution.

1.1.3 PDRP – Package Delivery Routing Problem

The real problem we are solving with our solution is this one. As in the one before, we have several clients each with an order of a specific size, and several travellers with each a maximum capacity. But in this case, we have several deposits, and the travellers have their own origin which does not have to be a deposit.

This problem is the **common use-case** of our solution.

1.2 TYPE OF INFORMATION

It has two paradigms in optimization problems: we can have all the information from the start (static) or receive the information as things progress (dynamic). This has a big impact on the method of solving. For this project, we have decided to work on a static resolution.

1.3 THE PROBLEM OF OPTIMIZATION PROBLEMS

The complexity class of simple TSP problem is NP-Hard. Applying its “number of solutions” formula to little number of vertices show why does this means (indicative times):

# of Cities	No. of solutions	Time
5	12	12 microseconds
8	2520	2,5 milliseconds
10	181440	0,18 second
12	19958400	20 seconds
15	43589145600	6 hours
18	177843714048000	5,64 years
20	60822550204416000	1927 years

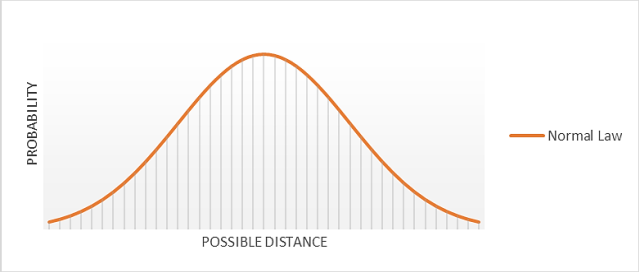
It is not realistic to calculate each case to find the best path, even with the simplest problem. Therefore, we make usage of heuristics: a calculation method that quickly provides a feasible solution, not necessarily optimal.

1.4 OUR RANDOM-BASED HEURISTIC

By analogy with the Monte Carlo method, our heuristic must be based on randomness, so that many draws have a high probability of including a path close to the optimal. So, we did it.

1.4.1 First approach: fully random generation

To test our solution (agile approach), we simply shuffled the array of vertices id before returning it.



Not ideal: need a lot of generations to statistically obtain “good” path.

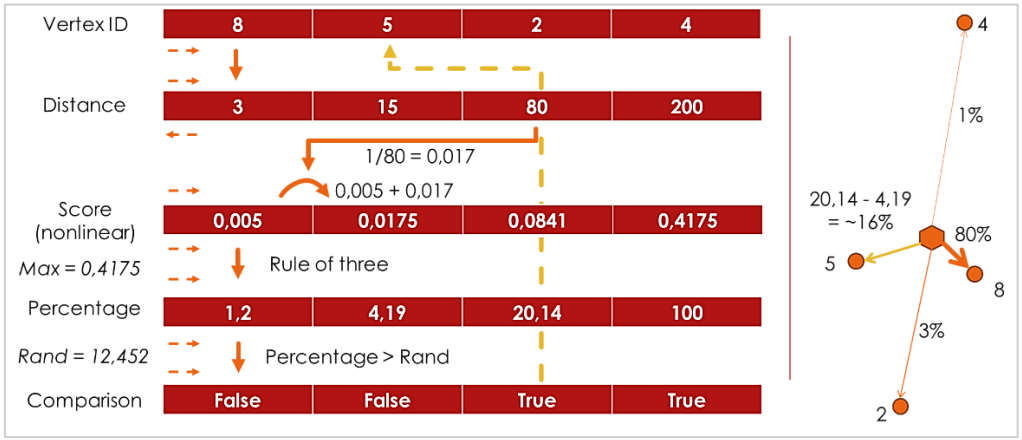
1.4.2 Second approach: n closest vertices

We improved this curve by selecting the n closest vertices respecting the constraints, sorting them by distance, and then selecting one at random position.

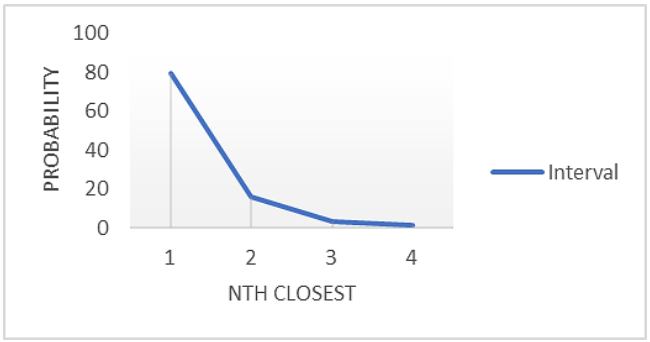
Better but still improvable: the random selection applies a gaussian probability of selection on vertices list, so we have a greater probability of drawing an "average" vertex.

1.4.3 Third approach: draw probability

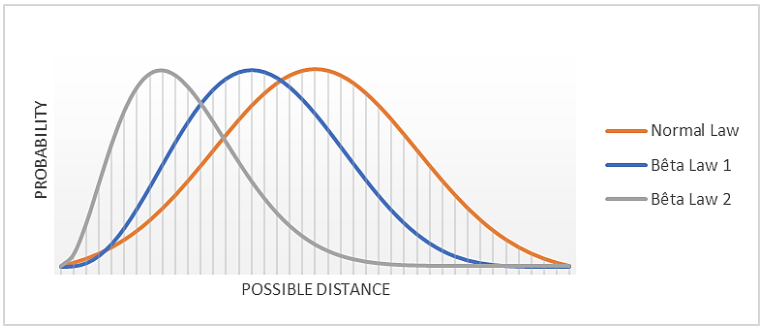
Our current summit selection is more sophisticated. For each vertex respecting the constraints, the series of the inverse of the distances is computed, from the furthest vertex to the closest. thus, the closer a vertex is, the larger its value interval and the higher selection probabilities it gets. We therefore obtain a selection probability for each vertex.



The resulting probability of selecting the closest vertex looks much better:



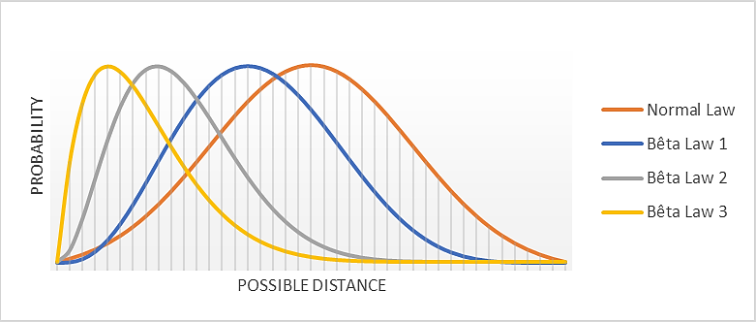
This process can be done recursively to select smaller distance on n next vertices, but without taking account of constraints.



Close to the best: e^{-x} curb style probability of closest vertex selection and no vertex discrimination.

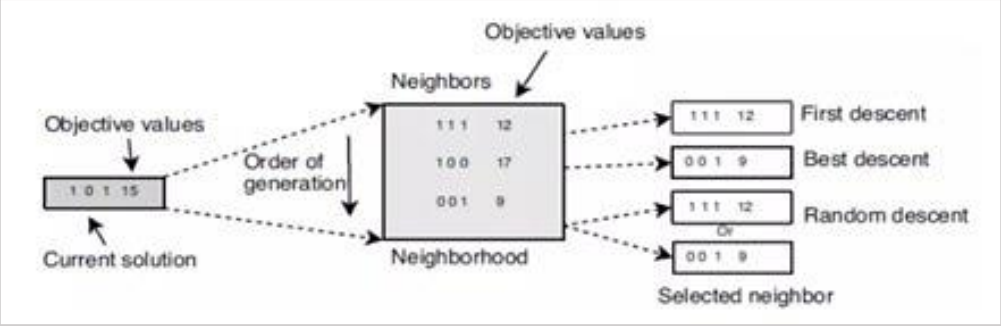
1.4.4 Side approach: optimizations

We worked in parallel to reduce the distances of the generated paths: a left shift of histogram's bars.



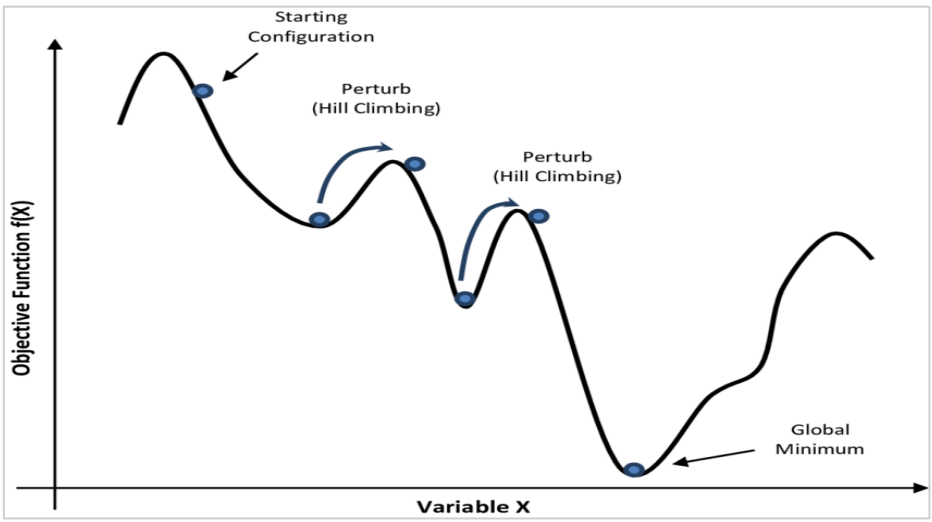
1.4.4.1 Local search

Local search is a greedy algorithm for searching the best solution in the local specified range. The strategy we used in this project is to find the best neighbourhood in the neighbour area and replace it as the next best solution. After several iterations, we can get the best solution in the specified range.



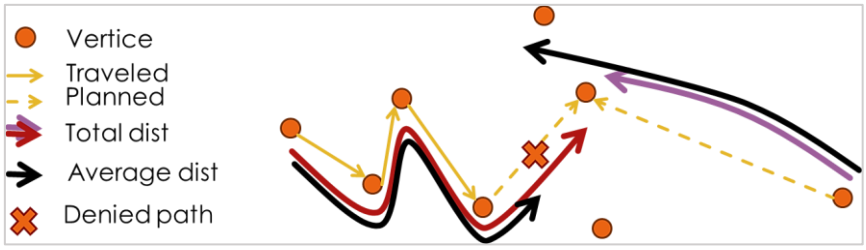
1.4.4.2 Simulated annealing

Simulated annealing algorithm is a single-solution algorithm. This probabilistic technique approximate the global optimum of a given function. Specifically, it is a metaheuristic to approximate global optimization in a large search space for an optimization problem.



1.5 DISTRIBUTION AMONG SEVERAL TRAVELLERS

It is necessary to ensure the distribution of the vertices among the travellers to avoid imbalances between the paths. To do this, each turn and for each traveller, we start by adding the distance travelled and the distance planned. Then, we calculate the average of these total distances. Finally, the movement of travellers whose total distance is less than this threshold value is authorized, and the others are cancelled.

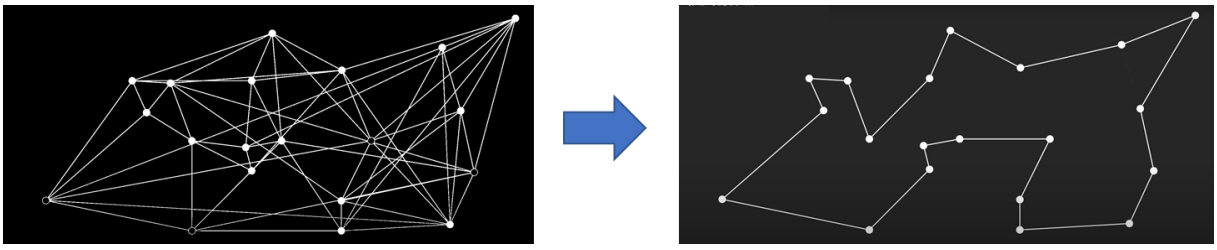


1.6 VALUABLE CRITERIA'S: KEY PERFORMANCES INDICATORS

Heuristics are based on distance only. However, the criteria for choosing the best path are not limited to distance. We have therefore determined 7 key performance indicators, which are the basis of a score calculation: total distance, minimum distance travelled, maximum distance travelled, interval of distance travelled, average distance, median distance, and percentage of travellers used. The lower the values are, the lower the score is and the better the path generated is.

1.7 GO FURTHER: PATH FUSION – NOT ENDED

We generate a lot of paths, which we seek to improve and which we analyse individually. Still, it might be interesting to study the result of their fusion: each path being analogous to an ant, one wonders if the colony would find a "smooth path" naturally.



For that, we thought of making a matrix of the number of selections of each arc, then taking the most popular arcs and finally adjusting the generated path to meet the constraints. This part was not done.

However, before carrying out this operation, it was necessary to bring together the paths of all travellers. For this, we reused our heuristic with a new two-dimensional distance matrix. Indeed, although a path has two distinct ends (start and end), the constraints impose an order on the path, so the distances of the matrix correspond to the distance between the end vertex of the first path and the start vertex of the second path.

1.8 TEAMWORK: ONLINE SHARING

We have created a google drive dedicated to this project.

Our first idea about this drive was to store there our results to be able to display them in a nice way on a dashboard. We found a way upload our results in a google sheet, with the Google Sheets API v4. We first needed a credential file that we got on the API website to create the first connection between the google service and our application. Once we had this file, we were able to get sheets from the drive and edit them or to create a new one. We also needed a way to connect to the google drive API, to upload the images we got. To do so we got another credential file, for the Google Drive API v3 this time, which allows our application to upload all kind of data to the drive.

Later, we had idea was to download the input data from the drive, doing so we were able to all share the same whereas we were working on the same feature or not. To do so, and as we needed the same API as for the upload of images, we used the same credential file. Once we had it, we could easily download a file with simply passing its name as a parameter or method.

1.9 DASHBOARD

We have developed a reporting dashboard to be able to analyse our results more easily. We then developed a more in-depth analysis by integrating our input data and crossing our generated data.

Most of the graphs presented are clickable: selecting an element will apply a filter on all the others. Likewise, filters and sorts can be applied at the top right of the graphs.

Finally, our tool, based on Data Studio, is totally public:

datastudio.google.com/u/2/reporting/719e36c2-23e3-4697-8a94-30e2cb147167/

2 CORE IMPLEMENTATION

We chose C++ for this part because of its efficiency for computation and its memory management.

2.1 LIBRARIES

2.1.1 Common

The lightest formatting language is json, which is why we use it to transfer our entries. To handle json, we used a GitHub library: github.com/nlohmann/json.

The calculation and display of our KPIs being common to all algorithms, we have developed a library with seven independent functions with an impersonal name. Thus, their content can be changed if the KPIs evolve and in all cases, their display is standardized.

The most basic functions of our heuristic are grouped together in another library:

- Our first useful function is this *getNthClosest* which takes as parameters a vector of distances, and a number n, and that returns a vector of n values which are the indexes of the smallest distances. In the last version the parameter n was not useful anymore because we chose to use all the vertices, but we kept it in the function for retro compatibility. This function is mainly used in *getPossibleNextPeak*.
- Another function we use a lot is *getPossibleNextPeak*, that has as inputs a vector of distances, a vector of vertices and an integer nbClosest. This function is used to return the next vertices sorted by distances. The result of this function is mainly used as input data for *getIdPoint*.
- And the main function of our heuristic is *getIdPoint*. This function takes in arguments a vector of vertices sorted by distances, and another vector containing the distances. This function returns the id of a point, with a probability based on its distance (see 1.4.3).

2.1.2 For generation

We have developed common functions for input and output, to standardize communication between engines and python:

- The input function takes the input arguments of the program plus non-constant addresses: the input data json, the recursion deepness integer and the choice of return to the origin. It displays the id (in case of error), initializes the random seed by modulate current timestamp with given id (same seed give same path), convert the contents of the data file into json and cast other inputs to initialize addresses.
- The output function takes the input json, the generated path and the choice of return to the origin. It displays the KPIs (see 2.1.1), then for each traveller, the distance followed by the vertex's ids of its specific path. if the traveller does not have to travel, it displays a default value.

2.1.3 For optimization

We have developed common functions for input and output, for the same reason as for the generation:

- The input function is close to the one of generation. It takes the same arguments plus the non-constant address of the generated path json. The main difference is the reuse of the seed without going through a timestamp (false random).
- The output function takes the input json, the initial path, the final path, and the choice of return to origin. It compares the scores of the two paths and displays the results as for generation only if the score of the final path is better.

Also, we have developed a check-neighbour method for judging the effectiveness of each solution. The main operation is checking every element in the generated solution. When the element represents restaurant, depending on the storage number, if there is no storage left, it means this solution cannot work. If there's still storage left, then plus one on the same location in the array with the same length of the solution for choosing the client number. When the element represents client, checking the array we create before, if the value on this location of the restaurant related to this client bigger than zero which means it is effective otherwise it cannot work. If the method works, it will also decrease the number of orders asked by this client and increase the storage number at the same time.

2.2 PATH GENERATION

All generation algorithms take advantage of commons input and output functions. All these functions are grouped together in a common library (see 2.1.2).

Our final path generation is based on four steps, which allows our program to pick the next vertex for each traveler in a random but optimized way:

1. The first step is to pick for every traveller the next vertex randomly using the *getIdPoint* with the remaining deposits and the clients this traveller is ready to deliver as parameter. After this first step, each traveller is assigned a point where he or she would like to go next.

2. As several travellers might want to go to the same point, the goal of the second step is to determine among travellers which one is the most fitted to go a certain point. To do so we are picking randomly, once again with *getIdPoint*, which traveller would be the best for each point. Once the best traveller is picked, we check if he or she has enough free storage, and if it is the case, we assign this traveller to this vertex.

3. Now, we have assigned travellers to vertices, but we still need to check if it is worth it to send this traveller to this point. Because if the traveller is very far away, it might be faster to wait for a closer traveller to be free than to send this one. To check this, we are keeping in memory the average distance travelled by all the travellers and comparing it with the distance that a traveller must travel from its current point to the potential next point. If this distance is lower than the average, then the traveller can go to this next point, else nothing happened.

4. Finally, the fourth step is to check if at least one traveller has moved. If not, it means that all the distances that travellers must travel are greater than the average. In this case, to avoid being stuck in an infinite loop, we send the traveller who is the closest to the vertex where he or she wants to go to this vertex.

Once all these steps are accomplished, we start again until there are no more vertices to go.

2.3 PATH OPTIMIZATION

All optimization algorithms take advantage of common input and output functions. Likewise, they call upon the same function for checking the constraints. All these functions are grouped together in a common library (see 2.1.3).

2.3.1 Local Search

We used a generated solution as the input for applying LS to make improvement. Applying the switch method to get new neighbourhoods, we use the check-neighbourhood method for judging the effectiveness of each new neighbourhood, finding the best one of them and replace it as the current solution. When there are not enough vertices for swap or when the number of trials is reached, the function end.

2.3.2 Simulated Annealing

We set the parameters: T-start, T-end, length, α.

Same as LS, we use a generated solution as input and applying switch and check-neighbourhood methods to get the new effective neighbourhood. We choose the element used in the switch method randomly. Then we calculate the value of the fitness function and compare it with the current solution. If it is better, then replace it as the new current solution otherwise we will also accept it as the current solution with the possibility of $e^{-\frac{|\Delta f|}{t}}$. In each temperature, this kind of operation will repeat 'length' times with the decrease of the temperature: $T_{new} = \alpha T_{old}$. When the temperature is lower than the T-end, the function comes to end and output the current solution.

2.4 PATH FUSION – NOT ENDED

As said in 1.7, only the path linker part has been developed. In it, we reuse the seed of the path, we select the first path at random (invisible in the final matrix) then, as long as there are paths to connect, we draw the next one with our heuristic (see *getIdPoint* at 2.1.1). Finally, we display the paths id.

3 LAUNCHER IMPLEMENTATION

We chose python for this part because of its efficiency in developing features and the lightness of its syntax. We display the steps and the associated timestamp to keep the user informed of the progress and to assess the effectiveness of each step. The first being precisely dedicated to the compilation of functions decorated by *njit* (*numba* library), and the creation of the results folder if it does not exist yet.

3.1 INPUT FILES FORMAT

3.1.1 Json config file

For this project, many parameters (21 + 4 per optimization algorithms) are defined by the user to give him maximum freedom. Theses parameters are divided into five sections (example annex1):

- The root section specifies two global elements: the name of the data file to be processed and the choice to have travellers return to their origin.
- The KPI section is used to specify the importance of each indicator. These are the weights of the weighted average which fixes the score of each path (see 1.6).
- The generation section is used to specify the algorithm to use (the highest version by default), the number of paths to generate and the level of recursion to apply in the search for the best next vertex.
- The optimization section is used to list the algorithms. For each, we can specify the threshold value and its use.
- The result section is used to specify the data retrieval format and to adjust the graphs creation.

The use of configuration files makes it possible to reinforce the reproducibility of our system. In addition, the configuration to be used by the system is specified by the name of the file, so it is possible to have a multitude of configuration files.

3.1.2 Txt data file

Our input data is divided into three different parts (example annex2):

1. In the first part, we are storing different vehicles references with their characteristics. The first column is an alias which is used later to link a traveller to a vehicle, the second column is the speed of the vehicle, which is finally not used in our version, and finally the last column is the storage of the vehicle which represent the amount of shipment this kind of vehicle can carry. If this last value is set to zero, we put it to one hundred thousand (virtual infinity).

2. The second part is about travellers. Each traveller is defined by a name, a longitude, a latitude, and a vehicle. The name is used to assign the path to the good traveller, the latitude and longitude represents the traveller's starting position and finally the vehicle must match one alias of the first part.

3. And finally, the third part describe the link between deposits and customers. Each deposit has a name, a longitude and latitude, and one or several clients. Each of these clients are separated by a dash and have a name, a longitude and latitude, but also an amount of shipment that must be delivered to this place. By default, this last value is set to one.

In a typical input file, we can find all the data required to generate a path, and the user is free to add has many vehicles, travellers, deposits, and clients as he wants.

3.2 GATHERING AND PARSING OF INPUTS

We start by parsing the user arguments. The first, mandatory, specifies the configuration file to use. The second, optional and true by default, specifies whether to work online or not (see 3.9).

The content of the configuration file is then loaded, locally or online (see 3.9.2) before processing some of its fields. Indeed, we check the existence of the algorithms to use, and we add their paths to the configuration json. In case of mismatch, we stop with an error message. We then retrieve the content of the data file in the same way.

The next step is to isolate the data blocks and check their number. It is in fact necessary to remove the header lines and the variable number of "empty" lines (spaces, tabs, line break) between two blocks. If we get less than three blocks, we display an error. Then, we map the names of the vehicles and their characteristics to replace the aliases of the travellers (vehicles names).

Finally, we divide those data into two sets, what is necessary for the computation and what is not, and to generate a matrix of the distances in a minimal time:

- We start by pre-allocating all memory locations in the fastest way, without sharing addresses, so that we only need to fill our structured data sets. However, this pre-allocation requires memorizing the memory cursor of each table (no push-back).
- Then, for each traveller row, we apply a parser which splits the cells and checks their number and type (conversions). The returned values are then applied to the traveller index of the sets.

- The same goes for vertices after a split on the separator: the first vertex is treated as a deposit and all subsequent ones are treated as clients. This association is also integrated into the set: the origin has an id list of the client vertices and each client knows the id of its origin vertex.

- Finally, we calculate the distance between each vertex in a spherical basis of terrestrial diameter based on four matrices (longitude and latitude of two points). This makes it possible to have real distances "as the crow flies". For this very intensive part in calculations, we use functions compiled Just-In-Time with the help of *Numba*. Zero distances outside the diagonal are prohibited: they are replaced by the smallest value with two decimal places possible (0.01). These computations are done once for the distance between all the vertices (n square matrix), but also for the distance between each traveller and all the vertices (matrix 2, n).

We then save the content of our data set in a temporary file intended for the executables: indeed, the number of characters is quickly too large for a console call. Converting a python object to a string results in a json, which meets our needs. However, displaying the *numpy* arrays used to speed up compiled functions requires a few cleaning steps: we set the printing options upstream, to display the maximum number of elements as floats to two decimal places. Then, we removed from the string five different patterns linked to the *numpy* arrays. Finally, we need to replace single quotes with double quotes to ensure the correct reading of the json on the C++ side.

3.3 SUMMARY AND USER AGREEMENT

With config settings and data arrays, we can easily display a summary of the forthcoming job. Since those data can be online and a heavy configuration may not be supported by all computers, we ask the user for confirmation before starting the job (example annex3).

3.4 PATH GENERATION

We use *Popen* to launch the executable in parallel processes from command line without blocking our process, as many times as we want. The only argument that changes is the id, which is an increment to have different random seeds, based on the python's PID (process id) to reinforce the randomness. Also, we pipe *stdout* and *stderr* in text format to get whatever will be printed during execution as a set of normalized, utf-8 encoded lines, once the process is finished.

Once all the processes have been started, we watch for the end of executions asynchronously. If none has finished, the system is put on hold for a few moments before rechecking. When a process ends, we remove it from the list and analyse its outputs. In case of error, we display its id (after subtracting the PID) and all other lines received. Otherwise, we pre-process the results before waiting for another end of execution. Pre-processing begins with a verification of the seed: if it is already in the results,

then we already have the path. Otherwise, we must calculate the weighted average of the KPI, convert the strings and add a row to the results. Then, when we have all the results, we sort them by score.

Finally, we display the number of distinct paths obtained as well as the algorithm used. For each row of results, we display an increment (necessary to compare the generation with any optimizations), the seed that was used to generate this result, the score obtained, the details of the KPIs, and the distance as well as the ordered list of vertices visited by traveller for each traveller (example annex4).

3.5 PATH OPTIMIZATION

The operation of this part is very close to the previous one: for each algorithm enabled, we still use *Popen* for each distinct generated path and retrieve the outputs asynchronously, with just a reuse of the seed. However, this time only certain paths will be optimized (lower score) and should be kept. In this case, we duplicate the result line of the generations to keep the id, and we update certain fields.

We display the results in the same way as before, adding the percentage of improved paths at the beginning, to evaluate the algorithm's ability to find better paths, and the percentage of reduction of the score for each line, to evaluate the algorithm's ability to improve paths.

3.6 PATH FUSION – NOT FULLY TESTED

This part works in two steps:

First, it is necessary to have one-piece paths per generation, and therefore to assemble the paths of travellers if necessary. To do this, we generate a matrix of the distances between the end of the first paths and the start of the second (importance of the order of travel), then we use run, a synchronous command line execution mechanism to call our first executable with same seed again.

Then, we save all the one-piece paths in a temporary file, which we pass to our second executable by using run again.

3.7 GROUPING AND MERGING OF DATA – CSV FORMATTING

This part is dedicated to the crossing of the initial and generated data to prepare the analysis report.

We start by generating "atomic" datasets, made up of distinct lines. Some are simple (list of deposits, clients, travellers, ...) and others are complex, and require preliminary calculations (path from an origin to a destination by generation and algorithm, ...).

Then, we merge these datasets two by two on keys (one or more) common to the two datasets, which can drastically multiply the number of rows and justifies the use of a dashboard (see 4).

Finally, we get three datasets that can be saved locally or sent online (see 3.9.3): vertices, orders, and executions.

- **Vertices** groups the name, position, and type (deposit/client) of the vertex,
- **Executions** groups the path of the traveller, his own distance, the total distance, the score, and the random seed of each type (generated / optimization algorithm), generation and traveller.

- **Orders** groups together the traveller information, the traveller's path, and the path distance between the deposit to the client, for each type (generated / optimization algorithm), generation and pairs depot – customer quadruplet.

3.8 GRAPH GENERATION

We take the n (limit of graph number) first paths among generated and optimized, sorted by best generated path. To do this, we generate an id translator that allows us to replace the paths generated by the optimized ones if the score is better (= lower). The title specifies the version displayed.

We generate the common elements once (mandatory vertices position, optional names, map, and links), then for each result, we create a thread and transmit it a copy of the graph (by pickling the figure's data into a memory buffer) to add the plotting of the paths.

To make the GIF, we save the intermediate images in PNG format in a buffer memory array, with indexing reset to the beginning. Then, we can transmit these images one by one to a frame converter, and the resulting frames to an assembler. Finally, we use a library which compresses the gifs by removing the common information from one frame to another.

3.9 ONLINE COLLABORATION

To download and upload files from the cloud, we have decided to use Google services which are very well documented in python and easy to use. So, we created a class thanks to which we can connect to several APIs and use them to retrieve the data we need or to upload files.

3.9.1 Initialisation

The instance constructor uses the tokens files to connect to the drive. If they expire, it updates them using the credentials files. It then builds its API interaction services and stores the id of key elements (csv file and image folder).

3.9.2 Data download

If the execution is done with online data, the configuration and the input data are retrieved from the drive. To do this, the synchronization instance launches a first request to know the id of the desired files and a second to retrieve their content, which must be decoded in utf-8.

3.9.3 Data upload

If the execution is done with online data, we delete all the images from the result folder whose name corresponds to that of the input data file, so as to replace them with the new data without cluttering the drive. This is done through a scan request and multiple conditionals delete requests.

We can then upload our media (PNG or GIF) one by one, keeping the id applied to each media. These ids, combined with an URL basis, are applied to all the lines of the execution's dataset whose execution number corresponds.

Once this whole operation has been carried out, necessary to display the images on the dashboard, the cells of the online csv must be cleaned and then refilled. For more convenience, we automate the calculation of the number of columns based on the dataset ones.

4 REPORTING DASHBOARD FOR DATA ANALYSIS

We used Google Data Studio, a free online reporting tool, to offer a more in-depth analysis of our results and to easily share them with a public link: datastudio.google.com/u/2/reporting/719e36c2-23e3-4697-8a94-30e2cb147167/.

Each graph can be sorted or filtered at the top right, and the elements are generally clickable: interacting with will filter all the elements on the page. Likewise, tables can be sorted by clicking on the column headers. To remove an applied sort or filter, an arrow appears at the top left.

4.1 OVERVIEW

The first page presents the positioning of all the vertices on a world map and the number of clients per deposit. Those graphs, as well as the numbers displayed (active travellers, number of deposits and number of clients) are linked to the selected generation (once at the time) and to the selected deposit (all or some). Finally, we get the distribution of vehicles among travellers.

4.2 ALGORITHM COMPARISON

The second page presents a table with the distance and the detailed path of each traveller, for each execution (with its seed) and algorithm used. The bottom graphs are used to display the distances and total scores by generation and algorithms: selecting a row filters the table to know which path to assign to each traveller. We can also select a row of the table or select only certain algorithms and generations with the drop-down lists at the top. Finally, on the left, we have a summary in three values: average, minimum and maximum traveller distance, currently displayed in the table.

4.3 ALGORITHM ANALYSIS

The third page presents the result of an execution with one of the algorithms applied. We therefore have the graph generated if it has been, the total and average distances travelled, and the number of clients delivered by travellers. Below, we have the distribution of deliveries among travellers: clicking on one of the travellers filters all the other elements and allows us to know the deposits he visited. Finally, we have the average distance travelled by travellers to deliver clients of each deposit: this gives an idea of the distance between customers and the deposit. By filtering on the deposit, we can see the number of associated customers and therefore the most important deposits.

4.4 PATH FUSION – HIDDEN

This last page simply presents the merged path and the associated distance and score, as well as the number of paths used to obtain this plot.

5 PROJECT AUTO-SETUP

The prerequisites for this project are:

- A Python interpreter between 3.7 and 3.9,
- A C ++ 11 or higher compiler,
- A Windows or Linux based OS which use apt dependency manager.

Also, the user needs to follow listed steps to obtain Google Drive credentials: developers.google.com/workspace/guides/create-credentials.

Credentials needed:

- Google Sheets API v4
- Google Drive API v3

Then, user can start a script automating about 90% of the project specific setup work in four steps:

1. We start by installing the python dependencies listed in the requirements.txt file, via the pip package manager. Then, we download python libraries out of pip before installing them too. There are then two scenarios:
    a. For Linux users, we install a C ++ library via the apt package manager.
    b. For Windows users, we download a port of this library, which we decompress. It will remain to add folder's path to the *Path* environment variables list & restart the system.

2. We then check the number of credentials files in the directory for access to the drive.

3. After that, we compare the list of drawing countries files present on a site with that of the local directory and we download the zip archives of the missing countries. Once everything has been downloaded, we extract the required files and delete the archive.

4. Finally, the script compiles the C++ files to ensure the user has working executables.

To avoid redoing a step, we create a witness file at the end of each step, except compilation. To allow dependency evolution, the dependency witness file is numbered. If you want re execute a step, delete the flag file in the setup folder.

The command to execute from the root folder is: python setup/init.py

6 PROJECT EXECUTION

The user needs to prepare the execution in two steps, either in local (root folder) or in drive (with credentials access):

1. Fill the .data file with input data's,
2. Configure the incoming execution by copying and filling the .json file.

Then, the command to execute from the root folder is: python launcher/main.py <configName> <optionnal: true(default) | false>. The optional parameter specifies if the execution is online or not.

7 DIFFICULTIES ENCOUNTERED AND IDEAS FOR IMPROVEMENT

7.1 DIFFICULTIES

7.1.1 Global to project

At the beginning, we thought about adding a speed to the traveller, but later we figured out that this was not compatible with a static configuration (see 1.2), so we removed it and we did not process it in our dashboard, but we kept it so it could be used to compute each traveller working time.

7.1.2 C++

7.1.2.1 Path generation

We had a few difficulties while developing the path generation algorithm. One of our main difficulty on this part was the implementation of multi travellers and more specifically the balance of vertices among travellers. Our first try was based on multi-threading: our idea was to base the balance of the vertices on the parallelism of the threads. We spent a lot of time developing this version to finally discover that the vertices were not well shared between travellers. After this failure, we had to re spend a lot of time developing our current idea based on average (see 1.5).

We also had a lot of difficulties to find a way to create a weighted random in a vector based on the values of this vector and to make it working well.

7.1.2.2 Path optimization

The development of the optimization's functions caused us some trouble to, because of we have a lot of constraints to consider: the storage, the sequence of clients and deposits… At first, we had developed the optimization as a part of the generation, so afterwards we had to split the generation and the optimization into two separate files which causes some trouble.

7.1.3 Python

On the python side, we did not have big issues, but more a lot of side problems:

- ✓ The major problem was the optimization of the computation of the matrix of distances. The first step has been to compute all the distances manually. Then we used a matrix of objects to group the four values, with *lru_cache* decorator to reduce the amount of calculation: simpler syntax but still heavy. And finally, in the last version, we have compiled functions which handle four matrices in parallel.

- ✓ The toughest problem to fix was maybe optimizing the generation of images, which used to take a lot of time when we had many vertices to print. the solution found are squarely outside the library.

- ✓ One last big problem in python has been the creation of CSV files to send data to our drive and dashboard. It implies to have common keys to merge on, which was pretty hard to conceptualize, especially for optimisations for which we have a different dimension (list of lists of results) and a various number of elements (need of id translation).

7.2 IMPROVEMENTS

7.2.1 Global to project

Our easier idea to implement would be to store the KPI as json in another temporary file. Doing this we could use them in the heuristic and base the creation of a path on them instead of just the distance, as we are currently doing.

One highly impacting idea we had to improve the project is to implement a time constraint to solve the MDRP (Meal Delivery Routing Problem). We thought about defining a max travel time a traveller has to deliver a shipment: for example, we thought about multiplying the distance between a client and a restaurant by some value. In this way, the traveller might have time to grab other meals on the way. We also thought about a way to store if a meal was delivered late or not: our idea for this was to store the opposite of the index of the client vertex. This way, we could still know the index of the client (by getting the absolute value of the index returned by the C++) but we could also have known if this client was delivered in time or not and use the amount of client delivered in time as a new KPI. And no problem with “0” vertex id because it is obligatorily a deposit.

A side improvement we would like to make is to use real distances between points, that we would get thanks to google map or OpenStreetMap, to have real distances between points and not "as the crow flies" anymore.

Finally, to make the project completely online and collaborative, we thought about executing the code in the cloud, via a platform like Google Collab who can host and run Jupyter notebooks on Google’s servers. in this way, we could control an execution from our phone, for example, and consult the results on the dashboard, from this same device.

7.2.2 C++ specific

We wondered how to find a way to pass the percentage of client’s command taken at each deposit for a traveller, without success. Currently, we display the repository id as many times as the number of orders loaded, without even knowing the orders taken. We must therefore find an elegant way to retrieve this information.

Also, the optimisations could be made not only on a traveller's path but between several paths, for example by trying to untie several nearby points and reconstruct the paths. Incidentally, the SA is much longer than the other algorithms.

Finally, we could try to merge all our solutions into one using an ant-colony algorithm to get an optimal path based on the results we have got from the generation and the optimization we already have. We have thought about an arc usage matrix as first step, but the exact implementation remains to be done. After that, it is possible to try to cut this path into segments meeting the constraints and optimized for the initial position of the travellers.

7.2.3 Python specific

One problem we still have is that the amount of execution of the C++ we can do simultaneously depends on the computer running the program. To fix this issue, we thought about a new parameter given to the program which would define the number of simultaneous processes. For example, if there are 200 processes needed, and the value of this new parameter is 4, we would run 4 times 50 instances in parallel. Doing this would allow the user to launch as much process as he wants with lower risk of damaging his or her computer.

Another thing that can be improved is the creation of graphs: currently, we create the n best generations, by considering the optimizations of equivalent seed. But we only take the first n generations / optimizations, not the best n paths all result combined: sorting is done on generations, not on each optimization algorithm or on a merged set.

8 CONCLUSION

Our project consists of three relatively independent and completely functional parts:

- ✓ The C ++ path generation and optimization algorithms,
- ✓ The Python wrapper
- ✓ And the Data Studio dashboard

Our generation algorithm is the result of personal reflection during development. Combined with more classic optimization algorithms, we obtain paths that seem good visually, and can serve as a comparative basis for other algorithms in scientific literature, such as genetic algorithms.

Our wrapper is a good base, finely optimized, and able to execute any generation, optimization and soon merging algorithm, using the inputs and outputs defined in the hpp libraries. It also generates CSVs files and summary graphs for subsequent interpretation of the results.

And the last part, our dashboard, provides an in-depth analysis of the data received and the results generated. In addition, it is easy to access through an internet browser and it brings a significant collaborative aspect.

Finally, we would like to thank mister Samuel DELEPLANQUE for all the help, the support and the good tips he gave to us during the six weeks of this project.

ANNEX 1: EXAMPLE OF JSON CONFIG FILE

```
{
    "input_datafile": "light",
    "back_to_origin": false,

    "KPI_weighting": {
        "total dist": 1,
        "% trav used": 1,
        "min dist": 1,
        "max dist": 1,
        "interval dist": 1,
        "mean dist": 1,
        "median dist": 1
    },

    "path_generation": {
        "algorithm": "default",
        "nb_process": 2000,
        "max_recursivity": 0
    },

    "path_optimization": [
        {
            "apply": false,
            "algorithm": "LS",
            "name": "Local search",
            "limit": 50
        },
        {
            "apply": false,
            "algorithm": "SA",
            "name": "Simulated annealing",
            "limit": 500
        }
    ],

    "results": {
        "print_console": false,
        "keep_local": false,
        "graph":
        {
            "make": false,
            "nb_max": 1,
            "show_names": false,
            "link_vertices": true,
            "map_background": false,
            "gif_mode": true,
            "fps": 3
        }
    }
}
```

ANNEX 2: EXAMPLE OF DATA INPUT FILE

```
vehicule : name, speed, capacity
bike,1,2

traveler : name, long, lat, vehicule
martin,-8,2,bike

deposit : name, long, lat  - clients : name, long, lat, qty -  ...
kfc,-3,4 - michel, 7,3 - jean_luc,2,5
pokawa,12,24 - arnaud, 9,-10 - kevin,8,1,5 - erwan,0,-4,2
```

ANNEX 3: EXAMPLE OF JOB SUMMARY

```
19:29:24.943383 - You will run 'light' (3 travelers, 3 deposits and 5 clients) with 'config.json' parameters :
Following KPI values :
  - total dist : 1
  - % trav used : 1
  - min dist : 1
  - max dist : 1
  - interval dist : 1
  - mean dist : 1
  - median dist : 1
10 executions of generator1 algorithm with recursivity of 0, without return to origin
Application of Local search optimisation algorithm on distincts paths with value of 50
Application of Simulated annealing optimisation algorithm on distincts paths with value of 500
A maximum of 5 graphs generation

Continue(y) ?
```

ANNEX 4: EXAMPLE OF RESULTS DISPLAY

```
10 distinc(s) peaks travel(s) order(s) generated with generator1:
- Generation 1, seed 10570 :
    Score of 1999.42
        Key performance indicators :
        - total dist    : 5330.38
        - % trav used   : 1.0
        - min dist      : 1157.95
        - max dist      : 2715.36
        - interval dist : 1557.41
        - mean dist     : 1776.79
        - median dist   : 1457.07
                traveler1 :    1457.07km with resto1 -> resto1 -> client1 -> jean_luc
                traveler2 :    2715.36km with resto2 -> client2 -> resto1 -> michel
                traveler3 :    1157.95km with resto3 -> client3
```