MOBILITY IN SMART CITIES PRACTICAL WORKS

Antonin WINTERSTEIN

Teacher: Mr. Philippe CANALDA

Master 1 IoT, Belfort-Montbéliard
2023-2024

Introduction

The Makes Change problem is an optimization problem in computer science and mathematics which involves finding the minimum number of coins needed to make change for a specific amount of money using a given set of coins. The goal is to determine the most efficient way to provide change, minimizing the total number of coins used while ensuring that the sum of the coins equals the desired amount.

For that, several approaches with various algorithms can be implemented such as iterative greedy, iterative or recursive solutions.

Iterative greedy approach

The greedy approach in the case of the Makes Change problem involves making the most immediately advantageous choice by selecting the largest coin which is less than or equal to the remaining amount and using it as much as possible before moving to the next coin. It is a linear complexity approach.

It tries to minimize the total number of coins used by always selecting the largest coin that doesn't exceed the remaining amount.

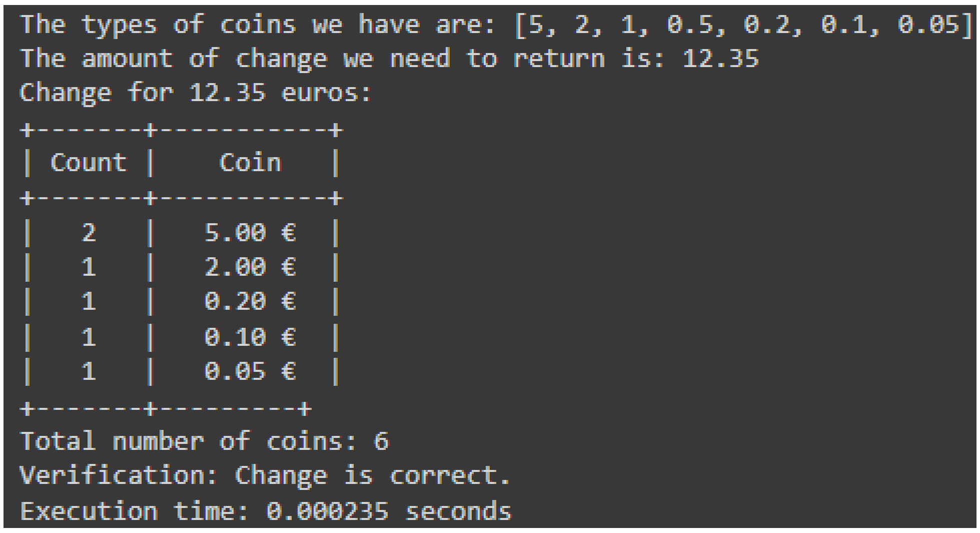
Here are the steps involved in the algorithm:

1. Start with the largest coin available.
2. Calculate the possible number of coins that can be used for the current denomination and the remaining amount, subtract it.
3. Move on to the next smaller coin value and repeat step 2.
4. Continue this process until the remaining amount is reduced to zero.

Iterative greedy approach tests

I implemented this solution in Python and tried several tests:

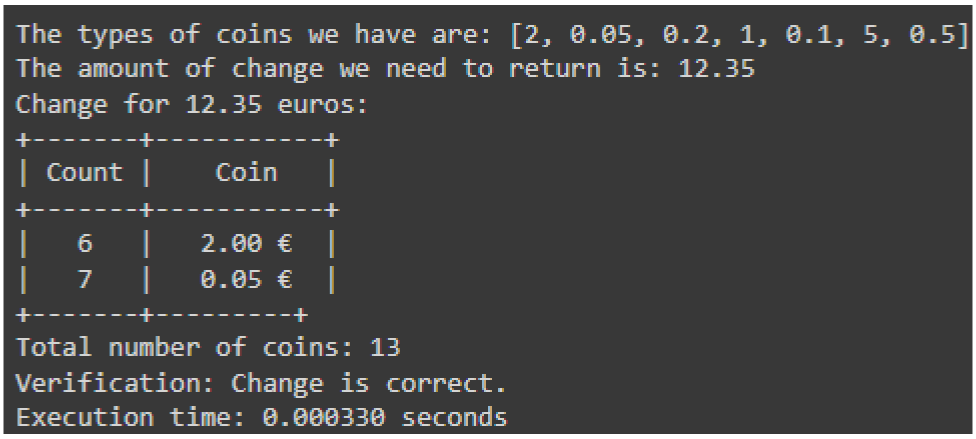
First test: First I tried on a list of coins in descending order of value L = {5, 2, 1, 0.5, 0.2, 0.1, 0.05} and an amount of change m = 12.35. In that case, the result I got is the most optimal in terms of the number of coins used (6), which is great since the execution time is really fast.



```
The types of coins we have are: [5, 2, 1, 0.5, 0.2, 0.1, 0.05]
The amount of change we need to return is: 12.35
Change for 12.35 euros:
+-------+----------+
| Count |   Coin   |
+-------+----------+
|   2   |  5.00 €  |
|   1   |  2.00 €  |
|   1   |  0.20 €  |
|   1   |  0.10 €  |
|   1   |  0.05 €  |
+-------+---------+
Total number of coins: 6
Verification: Change is correct.
Execution time: 0.000235 seconds
```

First test result

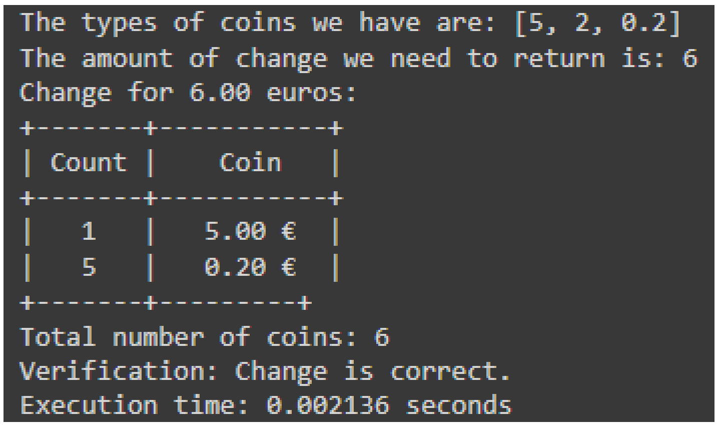
Second test: Then I tried on the same list of coins but without the good ordering as before. I have L = {2, 0.05, 0.2, 1, 0.1, 5, 0.5} and an amount of change m = 12.35. In that case, we are still getting the good amount of change but since it takes the first value which is "2" and tries to use it as much as possible, we are far from having the optimal solution like we had before (here we use 13 coins). Thus, it is really important to order the coins values.



```
The types of coins we have are: [2, 0.05, 0.2, 1, 0.1, 5, 0.5]
The amount of change we need to return is: 12.35
Change for 12.35 euros:
+-------+----------+
| Count |   Coin   |
+-------+----------+
|   6   |  2.00 €  |
|   7   |  0.05 €  |
+-------+---------+
Total number of coins: 13
Verification: Change is correct.
Execution time: 0.000330 seconds
```

Second test result

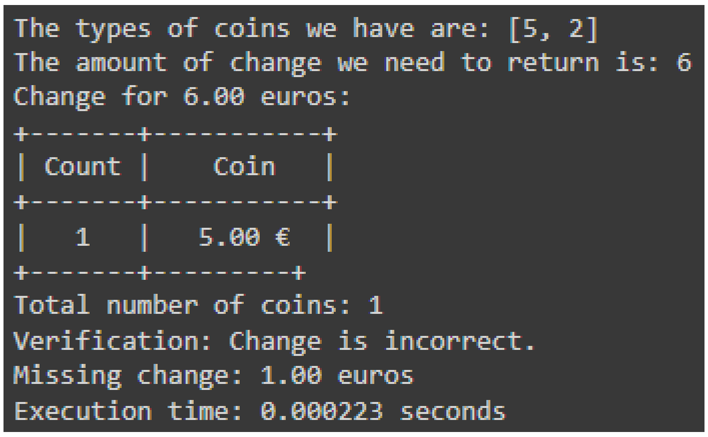
Third test: Then I tried on a different list of coins L = {5, 2, 0.2} and an amount of change m = 6. Here, if we think a little bit, we can see that the optimal number of coins is 3 by taking 3 times the 2 euros coin. However, since it is a greedy approach, it will first use the 5 euros coin leading to a non optimal result with 6 coins used.



```
The types of coins we have are: [5, 2, 0.2]
The amount of change we need to return is: 6
Change for 6.00 euros:
+-------+-----------+
| Count |    Coin   |
+-------+-----------+
|   1   |   5.00 €  |
|   5   |   0.20 €  |
+-------+---------+
Total number of coins: 6
Verification: Change is correct.
Execution time: 0.002136 seconds
```

Third test result

Fourth test: Then I tried on the same list of coins as before but removed the last coin value. Here I have L = {5, 2} and an amount of change m = 6. As before, the optimal solution is by using 3 times the 2 euros coin. However, it won't find the solution and won't be able to return the good amount of change because it starts with 5 euros and doesn't have a next coin value which is available to fill the rest while it could have been possible using the 2 euros coin.



```
The types of coins we have are: [5, 2]
The amount of change we need to return is: 6
Change for 6.00 euros:
+-------+-----------+
| Count |    Coin   |
+-------+-----------+
|   1   |   5.00 €  |
+-------+---------+
Total number of coins: 1
Verification: Change is incorrect.
Missing change: 1.00 euros
Execution time: 0.000223 seconds
```

Fourth test result

Overall, the greedy approach is a simple and intuitive way to solve this problem but it has limitations which we need to be aware of.

All solutions iterative with best solution approach

The iterative approach refers to a method of finding all possible combinations of coins that can be used to make a specific amount of change without using recursive function calls. This approach involves using loops and a data structure to manage the state of the problem and keep track of the different combinations. It can be efficient if there is a small amount of possible combinations but can take a very long time if lots of combinations are available.

Here are the steps involved in the algorithm:

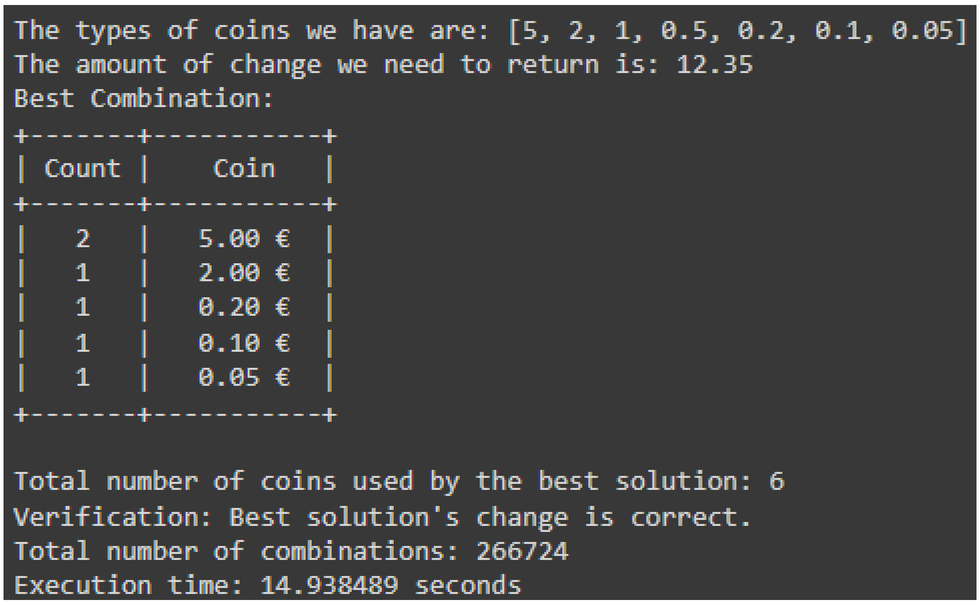
1. Start with an empty stack or queue to keep track of states (current amount, coin combination, and coin index).

2. Use a loop to systematically explore different combinations of coins.
3. In each iteration, consider adding different counts of each coin denomination to the current combination.
4. If the current combination equals the target amount, add it to the list of valid combinations and keep the solution with the least amount of coins.
5. Continue exploring possibilities until all combinations have been considered.
6. Once the loop completes, it's possible to show the optimal solution among all valid combinations.

All solutions iterative with best solution approach tests

I implemented this solution in Python and tried several tests:

First test: First I tried on a list of coins in descending order of value L = {5, 2, 1, 0.5, 0.2, 0.1, 0.05} and an amount of change m = 12.35. Note that the ordering of coins doesn't matter in this solution since in any case, all solutions are computed. I of course got the most optimal solution but it took a long time to compute (around 15 seconds) because there are a lot of combinations (266 724 here). In comparison to the greedy approach, it is 63 568 times slower!

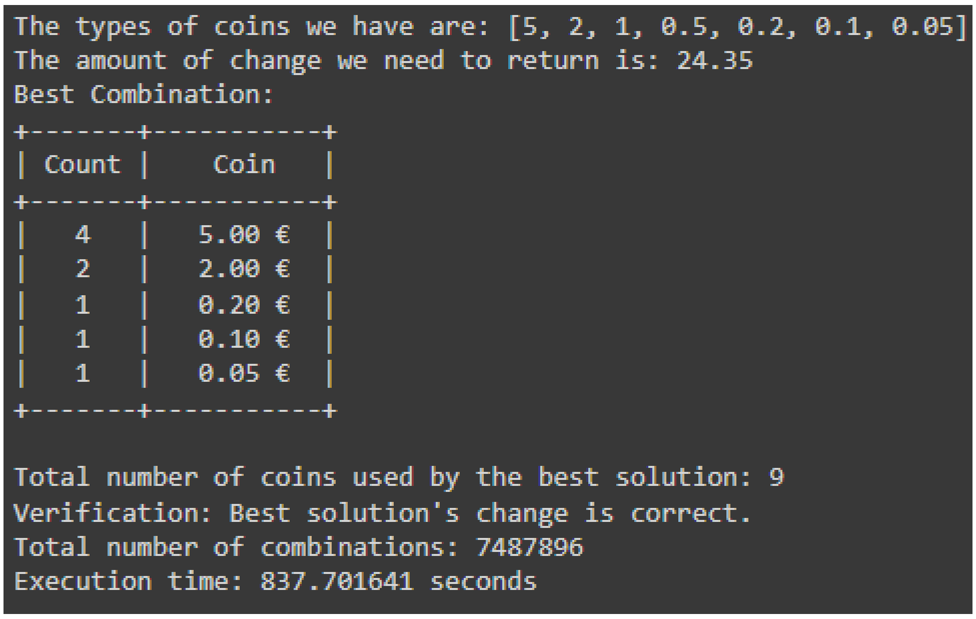


```
The types of coins we have are: [5, 2, 1, 0.5, 0.2, 0.1, 0.05]
The amount of change we need to return is: 12.35
Best Combination:
+-------+-----------+
| Count |    Coin   |
+-------+-----------+
|   2   |   5.00 €  |
|   1   |   2.00 €  |
|   1   |   0.20 €  |
|   1   |   0.10 €  |
|   1   |   0.05 €  |
+-------+-----------+

Total number of coins used by the best solution: 6
Verification: Best solution's change is correct.
Total number of combinations: 266724
Execution time: 14.938489 seconds
```

First test result

Second test: I then tried using the same list of coins L = {5, 2, 1, 0.5, 0.2, 0.1, 0.05} but with a bigger amount of change m = 24.35. Again, I got the most optimal solution but at the cost of performance. Indeed, it took 837 seconds (nearly 14 minutes) to compute all solutions (7 487 896) and print the best one which is not applicable in real life.

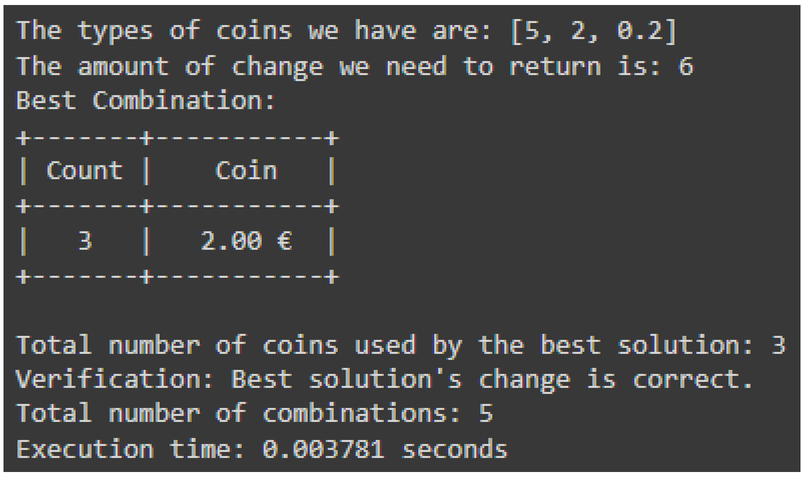


```
The types of coins we have are: [5, 2, 1, 0.5, 0.2, 0.1, 0.05]
The amount of change we need to return is: 24.35
Best Combination:
+-------+-----------+
| Count |    Coin   |
+-------+-----------+
|   4   |   5.00 €  |
|   2   |   2.00 €  |
|   1   |   0.20 €  |
|   1   |   0.10 €  |
|   1   |   0.05 €  |
+-------+-----------+

Total number of coins used by the best solution: 9
Verification: Best solution's change is correct.
Total number of combinations: 7487896
Execution time: 837.701641 seconds
```

Second test result

Third test: I then tried using a smaller list of coins L = {5, 2, 0.2} and an amount of change m = 6. Here, since the number of combinations is really small, the execution time is fast and the best combination can be found while the greedy approach couldn't.



```
The types of coins we have are: [5, 2, 0.2]
The amount of change we need to return is: 6
Best Combination:
+-------+-----------+
| Count |    Coin   |
+-------+-----------+
|   3   |   2.00 €  |
+-------+-----------+

Total number of coins used by the best solution: 3
Verification: Best solution's change is correct.
Total number of combinations: 5
Execution time: 0.003781 seconds
```

Third test result

While this solution can be efficient when there are only a few possible combinations, it may become impractical for larger input values due to the exponential growth in the number of combinations to consider. These limitations must be taken into account.

All solutions recursive with best solution approach

The recursive approach involves breaking down the problem into smaller subproblems and solving them using recursive function calls to explore different combinations of coins until a valid solution is found or all possibilities are exhausted. Just like the iterative approach, it won't be really efficient if there is a large number of recursive calls.

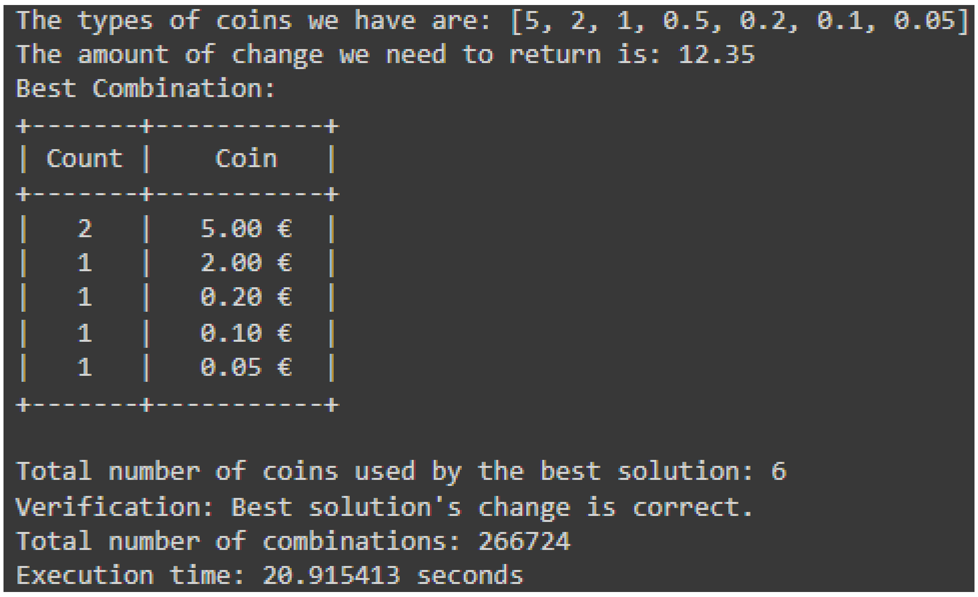
Here are the steps involved in the algorithm:

1. For each coin, it explores all possible combinations by calling recursively itself with the updated values.
2. After exploring all possibilities with a coin, remove it and go to the next coin to try other combinations.
3. The recursion continues until all possible combinations have been explored.

All solutions recursive with best solution approach tests

I implemented this solution in Python and tried several tests:

First test: First I tried on a list of coins in descending order of value L = {5, 2, 1, 0.5, 0.2, 0.1, 0.05} and an amount of change m = 12.35. Note that like for the iterative, the ordering of coins doesn't matter in this solution since in any case, all solutions are computed. In the same way, it also took a long time (around 21 seconds), even more than the iterative one.

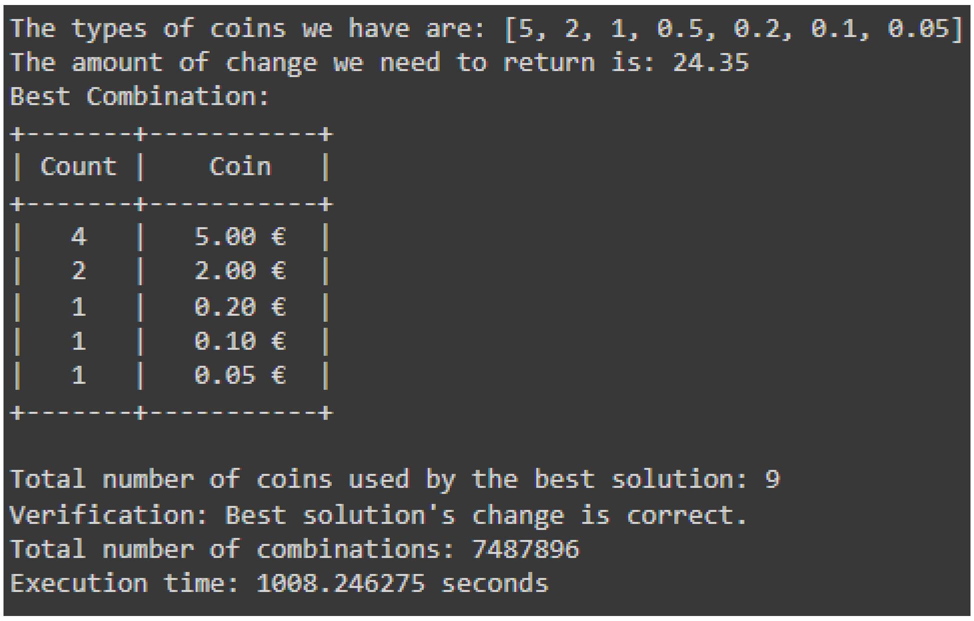


```
The types of coins we have are: [5, 2, 1, 0.5, 0.2, 0.1, 0.05]
The amount of change we need to return is: 12.35
Best Combination:
+-------+-----------+
| Count |    Coin   |
+-------+-----------+
|   2   |   5.00 €  |
|   1   |   2.00 €  |
|   1   |   0.20 €  |
|   1   |   0.10 €  |
|   1   |   0.05 €  |
+-------+-----------+

Total number of coins used by the best solution: 6
Verification: Best solution's change is correct.
Total number of combinations: 266724
Execution time: 20.915413 seconds
```

First test result

Second test: I then tried using the same list of coins L = {5, 2, 1, 0.5, 0.2, 0.1, 0.05} but with a bigger amount of change m = 24.35. Again, it gives the same results as the iterative version but with even more time to wait with 1008 seconds (nearly 17 minutes).

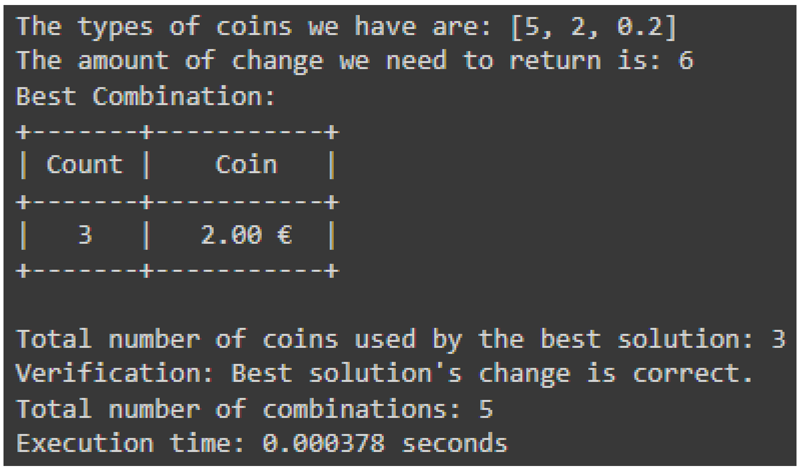


```
The types of coins we have are: [5, 2, 1, 0.5, 0.2, 0.1, 0.05]
The amount of change we need to return is: 24.35
Best Combination:
+-------+-----------+
| Count |    Coin   |
+-------+-----------+
|   4   |   5.00 €  |
|   2   |   2.00 €  |
|   1   |   0.20 €  |
|   1   |   0.10 €  |
|   1   |   0.05 €  |
+-------+-----------+

Total number of coins used by the best solution: 9
Verification: Best solution's change is correct.
Total number of combinations: 7487896
Execution time: 1008.246275 seconds
```

Second test result

Third test: I then tried using a smaller list of coins L = {5, 2, 0.2} and an amount of change m = 6. Here, the results are the same as the iterative one with the recursive one being a little bit faster in that case.



```
The types of coins we have are: [5, 2, 0.2]
The amount of change we need to return is: 6
Best Combination:
+-------+-----------+
| Count |    Coin   |
+-------+-----------+
|   3   |   2.00 €  |
+-------+-----------+

Total number of coins used by the best solution: 3
Verification: Best solution's change is correct.
Total number of combinations: 5
Execution time: 0.000378 seconds
```

Third test result

Just like the iterative version, recursive can be a handful to find the best combination when the possible combinations are small. However, when there are a lot, it is not optimal and even worse than the iterative. Indeed, it may introduce overhead that makes it less efficient.

TD2 - Makes change

Introduction

This TD is the continuation of the first one which I did with Guillaume Martin and Killian Maxel. Seven programs were asked which we provided with the according results in .txt format.

Program 1

This program uses a greedy algorithm to find a solution to the making change problem. It initializes the available coin denominations (coins) and attempts to make change using the largest available coins first. The combination is stored in sol_1.txt with the execution time.

Program 2

This program exhaustively calculates all possible solutions to the making change problem without using recursion. All the combinations are stored in sol_2.txt with the execution time. However, since the size of the file is too large for GitHub, you will have to generate it yourself.

Program 3

This program uses a recursive approach, often referred to as dynamic programming, to generate all valid solutions to the making change problem. Solutions are generated through an in-depth walk, and valid combinations are displayed as they are found. This approach aims to avoid excessive memory consumption but may be slower due to I/O operations. All the combinations are stored in AllTheSolutionsTest1.txt with the execution time.

Program 3 bis

This program is the same as Program 3 except that the generation of solutions is returned to favor the maximum number of units of the value being processed.

Program 4

This program calculates all solutions without displaying them in real-time. Solutions are stored in a 2D array, and the order of valid combinations is maintained. It also checks if the result is the same as the one done earlier. All the combinations are stored in sol_4.txt with the execution time. However, since the size of the file is too large for GitHub, you will have to generate it yourself.

Program 5

This program extends Prg3 by evaluating the cost of each solution based on the number of units of the coin values. It calculates the best solution incrementally by displaying values that improve the solution. The combination is stored in CORRECT_SolutionsWhichImproveIncrementally.txt with the execution time of the previous program and the modified one.

Program 6

This program improves on Prg4 by storing the best-calculated solution in a solution table.Solutions are stored in a 2D array, and the order of valid combinations is maintained. The combination is stored in sol_4-6.txt with the execution time.

Program 7

This program implements a cut mechanism to optimize the search for the best solution. The solution's cost is calculated as the program progresses, and if the partial cost exceeds the best-known solution cost, the program stops processing. It reduces computation. The combination is stored in sol_4-7.txt with the execution time.

TD3 - Transitivity

Introduction

Transitivity describes a property of relationships between elements or objects. Thanks to matrices like the Origin-Destination one, it is possible to check, remove or add transitivity between points. For example, if there is a relationship from element A to element B and from element B to element C, then there is also a relationship from element A to element C. In other words, if A is related to B, and B is related to C, then A is related to C through a transitive link.

Being able to add or remove transitive links can be really useful. Indeed, removing unnecessary transitive links can simplify data and reduce complexity, leading to more efficient algorithms, improved clarity, and better performance. On the other hand, adding missing transitive links can help establish logical connections and ensure completeness in representations, which can be crucial for accurate analysis and decision-making in various applications.

Check if a matrix is transitive

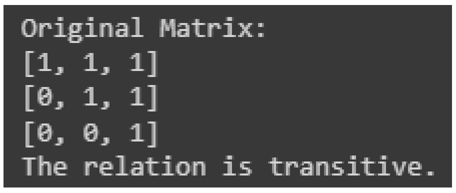
To check if a matrix is transitive or not, we can follow this process:

1. Iterate through the rows (i) and columns (j) of the matrix.
2. For each cell (i, j) that contains a value of 1 (indicating a relation from element i to element j), check for transitive links.
3. To check for transitive links, iterate through the columns (k) of the matrix for the same row (j) where the relation from (i, j) was found.
4. If there is a relation from (j, k), check if there is already a relation from (i, k).
5. If there is no relation from (i, k) when there is a relation from (j, k), then the transitive property is violated, and the matrix is not transitive.

Check if a matrix is transitive tests

I implemented this solution in Python and tried several tests:

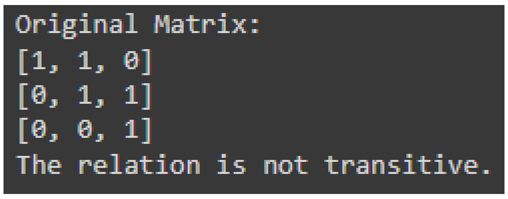
First test: First I tried on a 3*3 matrix which is transitive. Indeed, for all points with a relation, there exists its transitive equivalence.



```
Original Matrix:
[1, 1, 1]
[0, 1, 1]
[0, 0, 1]
The relation is transitive.
```

First test result

Second test: Then I tried on a 3*3 matrix which is not transitive. Indeed, we can see that the relation between A-C is missing.



```
Original Matrix:
[1, 1, 0]
[0, 1, 1]
[0, 0, 1]
The relation is not transitive.
```

Second test result

Add transitive closure to a matrix

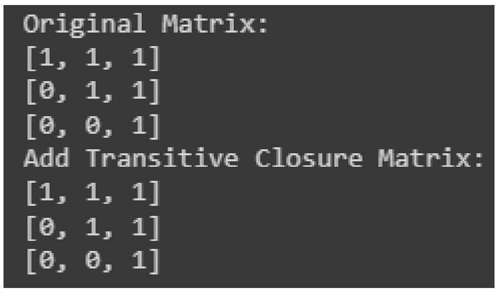
To add the transitive closure to a matrix, we can follow this process:

1. Iterate through the matrix, considering each element as a potential intermediate node in a transitive link.
2. For each pair of nodes (i, j) in the matrix where there is a direct link (i to j), check for other nodes (k) that can create transitive links.
3. To check for transitive links, iterate through the matrix again to find nodes (k) where there is a direct link from (j to k) and from (i to k).
4. If such transitive links are found, update the matrix to indicate the transitive connection from (i to k).
5. Repeat this process for all possible pairs of nodes.

Add transitive closure to a matrix tests

I implemented this solution in Python and tried several tests:

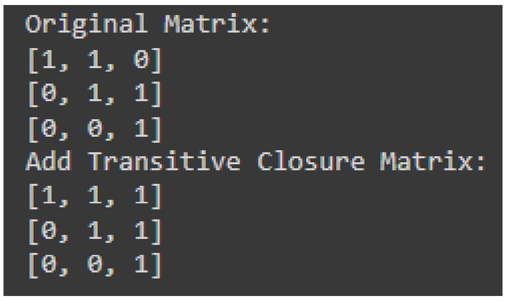
First test: First I tried on a 3*3 matrix which is already transitive. In that case, nothing changes which is the normal behavior.



```
Original Matrix:
[1, 1, 1]
[0, 1, 1]
[0, 0, 1]
Add Transitive Closure Matrix:
[1, 1, 1]
[0, 1, 1]
[0, 0, 1]
```

First test result

Second test: Then I tried on a 3*3 matrix which is not transitive. Indeed, we can see that the relation between A-C is missing. Thus, the program fixes that by adding this transitive link.



```
Original Matrix:
[1, 1, 0]
[0, 1, 1]
[0, 0, 1]
Add Transitive Closure Matrix:
[1, 1, 1]
[0, 1, 1]
[0, 0, 1]
```

Second test result

Remove transitive closure to a matrix

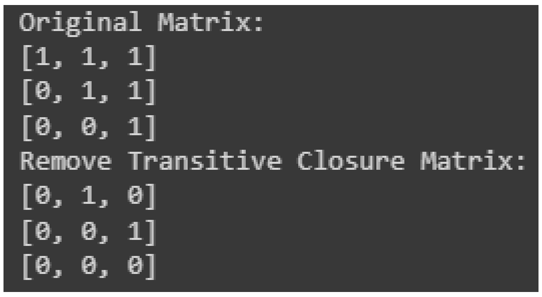
To remove the transitive closure to a matrix, we can follow this process:

1. Perform reflexive reduction by setting diagonal elements to 0. This ensures that no element has a reflexive relation with itself.
2. Iterate through the matrix to find elements (i, j) with a direct link (i to j).
3. For each such element, check for other nodes (k) that have transitive links with (i) and (j).
4. If a transitive link is found from (i to j) and a direct link exists from (j to k), set the transitive link from (i to k) to 0, effectively removing the transitive connection.
5. Repeat this process for all elements in the matrix.

Remove transitive closure to a matrix tests

I implemented this solution in Python and tried several tests:

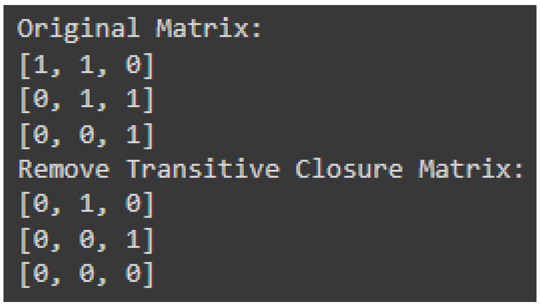
First test: First I tried on a 3*3 matrix which is already transitive. Thus, the program removes all the transitive links only letting necessary links.



```
Original Matrix:
[1, 1, 1]
[0, 1, 1]
[0, 0, 1]
Remove Transitive Closure Matrix:
[0, 1, 0]
[0, 0, 1]
[0, 0, 0]
```

First test result

Second test: Then I tried on a 3*3 matrix which is not transitive. Indeed, we can see that the relation between A-C is missing. In that case, only the reflexive links are getting removed.



```
Original Matrix:
[1, 1, 0]
[0, 1, 1]
[0, 0, 1]
Remove Transitive Closure Matrix:
[0, 1, 0]
[0, 0, 1]
[0, 0, 0]
```

Second test result

TD4 - Shortest path

Introduction

The objective is to develop a program that, given a subset of nodes and edges from a graph, can calculate the distances between these nodes and a designated reference node. Based on these distances, it will allow the ordering of the list of nodes.

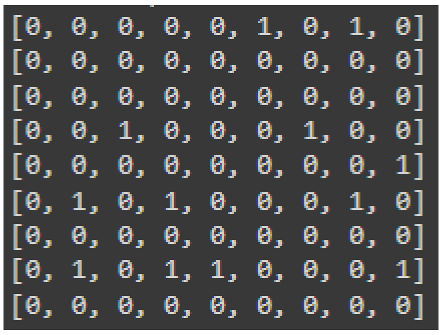
For that, the primary objectives are to:

1. Calculate the shortest path from a node to all the other ones.
2. Order the nodes based on their increasing distances from the reference node (or in alphanumeric order in case of same distances).

For that, we consider a graph G = (N = {0,1,2,3,4,5,6,7,8}, E = {(0,5), (0,7), (5,7), (5,3) , (3,6),(3,2), (7,3), (7,8), (7,1), (7,4), (4,8), (5,1)}) where N is equal to the nodes and E the edges of the nodes.

Matrix representation

I represented an adjacency matrix where each row and column represent nodes, and the values indicate the presence (1) or absence (0) of edges between nodes.



```
[0, 0, 0, 0, 0, 1, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 1]
[0, 1, 0, 1, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 1, 1, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
```

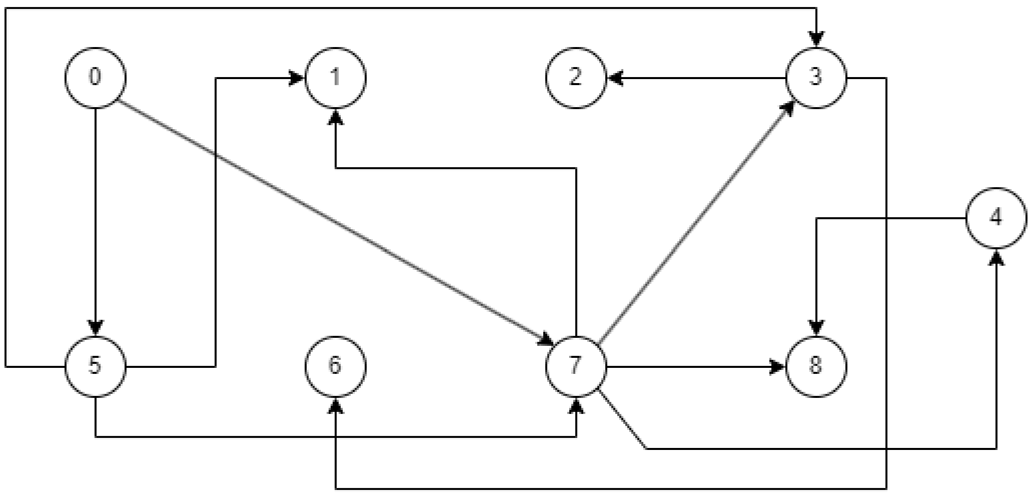
Matrix representation of the graph

Characterization of relationships

The graph is directed because it is not specified that the edges are linked between each other. Some nodes have no outgoing edges (e.g., 1, 2, 4, 6, 8), while others have multiple outgoing edges. There are cycles in the graph (e.g., nodes 0, 5, 7, and 3 form a cycle).

Sagittal representation

Here is the sagittal representation I made for the provided graph:



Sagittal representation of the graph

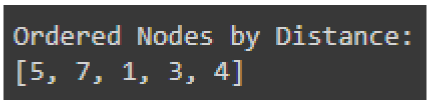
Algorithm

To calculate the shortest paths from any point to 0, we can use Dijkstra's algorithm. Then, it is possible to sort the nodes by distance and then by alphanumeric order in case of ties within the subset of nodes.

To calculate a sublist corresponding to a subgraph, we can specify the subset of nodes and edges from the original graph.

If we try applying the algorithm on the subset of nodes N0 = {1,3,4,5,7} and the associated edges E0 = {(0,5), (0,7), (5,7), (7, 3), (7,1), (7,4), (5,1)}, the result should be L0 = {0,5,7,1,3,4}.

The result I'm getting is really similar to the expected result. Indeed, I only don't have the "0" value because it is not within the subset of nodes N0.


```
Ordered Nodes by Distance:
[5, 7, 1, 3, 4]
```
Result of the algorithm on the subset N0 and E0 of the graph G

Tunisian Louage's Transportation Solution

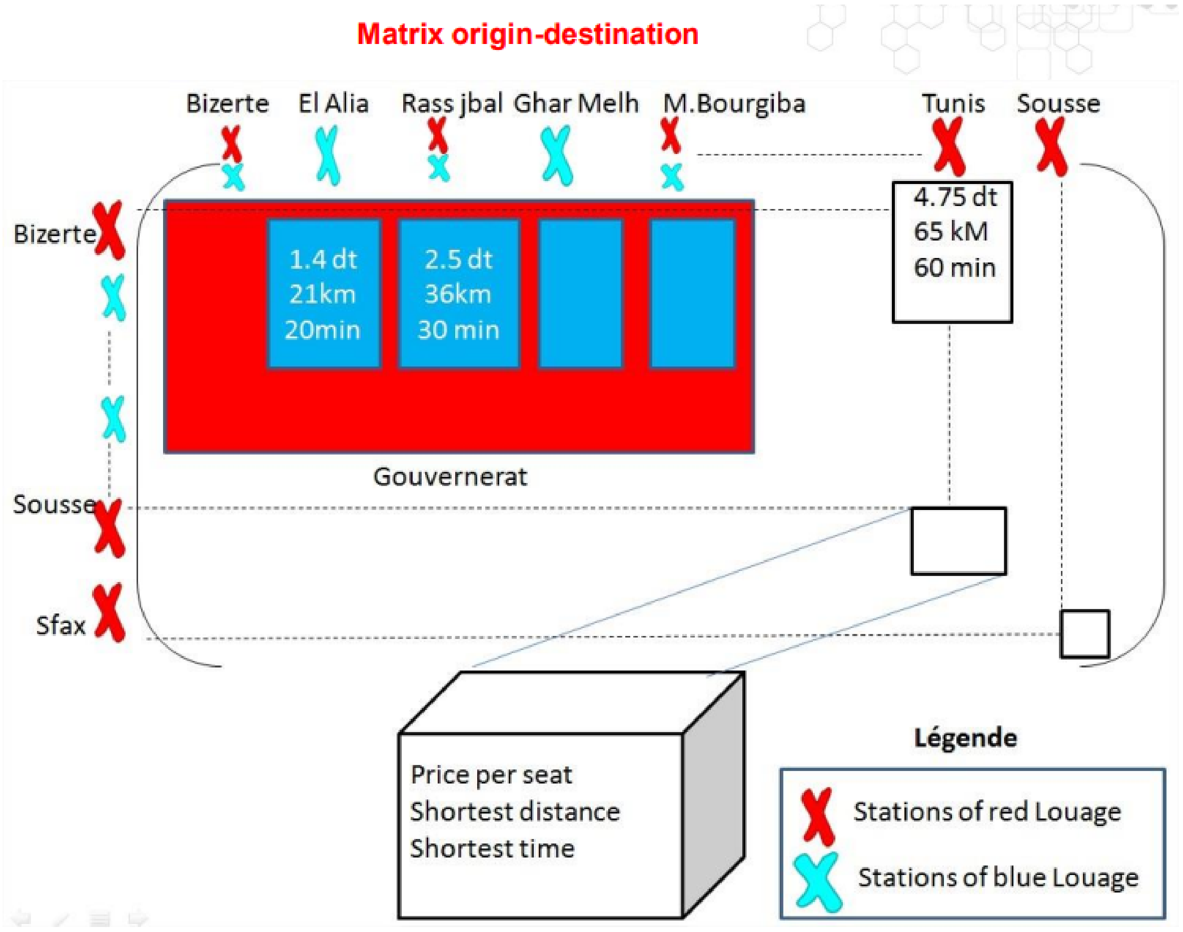
Introduction

The Louage system is a public transportation solution that covers the entire territory of Tunisia. It is designed to complement other transportation offers in the country. Louage is a response to the challenges of providing efficient and affordable transportation in Tunisia, particularly in rural areas where traditional public transportation is often lacking. The Louage system faces several challenges, including the need to balance supply (the drivers) and demand (the passengers) but also the issue of optimizing routes. Indeed, each driver has a different itinerary, type of louage, time-window, and number of available seats. In the same way, the passengers also have their personal itinerary with their possible time-windows. It makes it really difficult to efficiently satisfy everyone.

It is defined as a multi-constraints and multi-objectives problem which we try to solve. Examples of problems are:

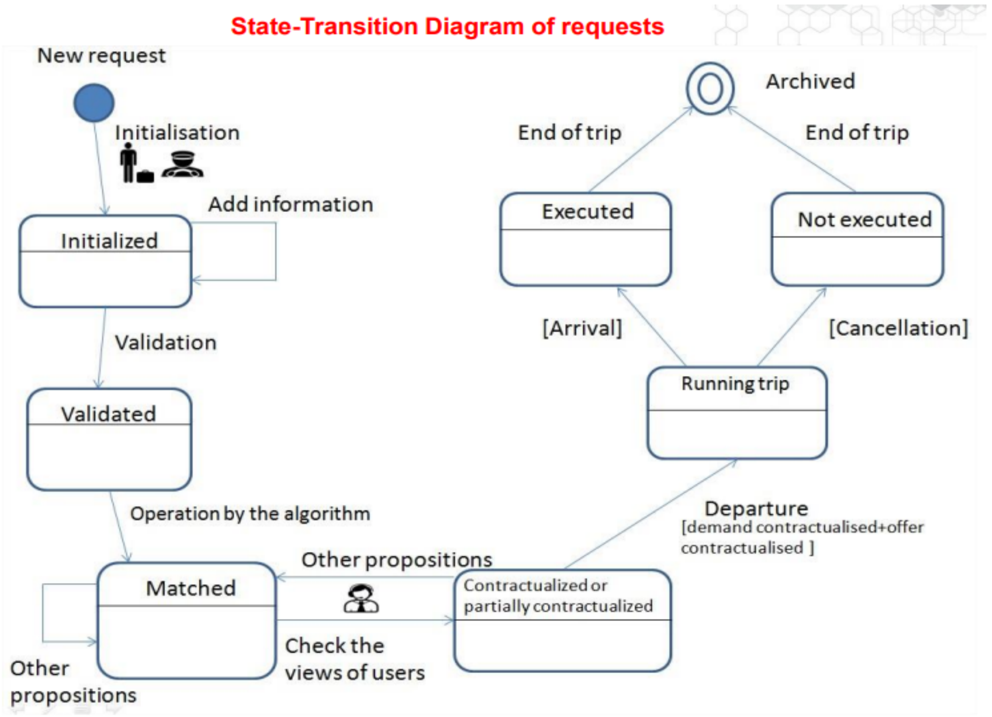
- The Traveling Salesman Problem (TSP) which involves finding the shortest possible route that visits a set of cities exactly once and returns to the starting city, with the goal of minimizing total travel distance or cost.
- The Vehicle Routing Problem (VRP) which deals with determining the most efficient routes for a fleet of vehicles to deliver goods or services to a set of customers, considering factors like distance, time, or cost.
- The Dial-a-Ride Problem which is about efficiently picking up and dropping off passengers at specific locations while considering factors like travel time, waiting time, and vehicle constraints.

To represent this system, it is possible to use a Matrix origin-destination where each element can take several values between two locations like for example the shortest distance, shortest time to travel, the price and so on.



Example of origin-destination matrix from Mr.Canalda's slides

We can also present the general process using a State-Transition Diagram of offers and demands.



State-Transition Diagram of offers and demands from Mr.Canalda's slides

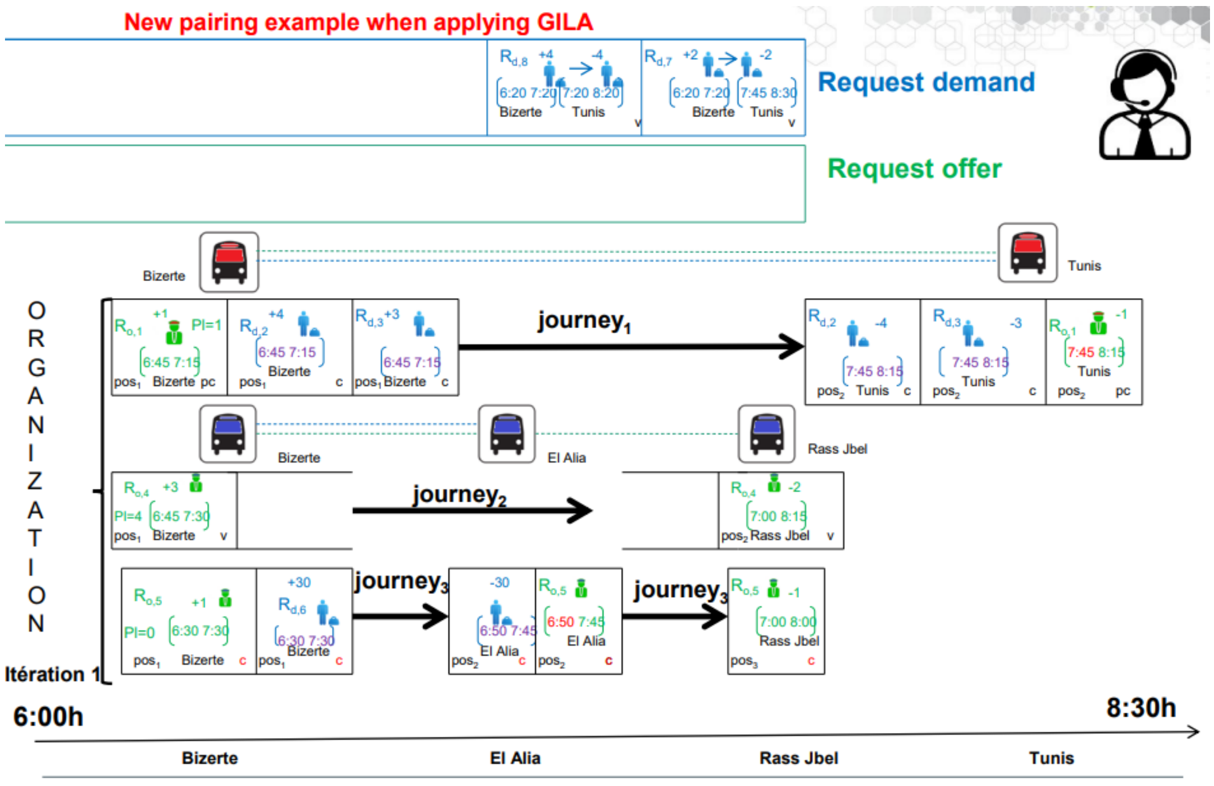
Greedy Incremental Louage's Algorithm (GILA)

The GILA algorithm aims for pairing offers and demands in the Louage system in Tunisia by implementing a mono objective which is:

"Maximizing the number of contractualized requests respectful to registration order."

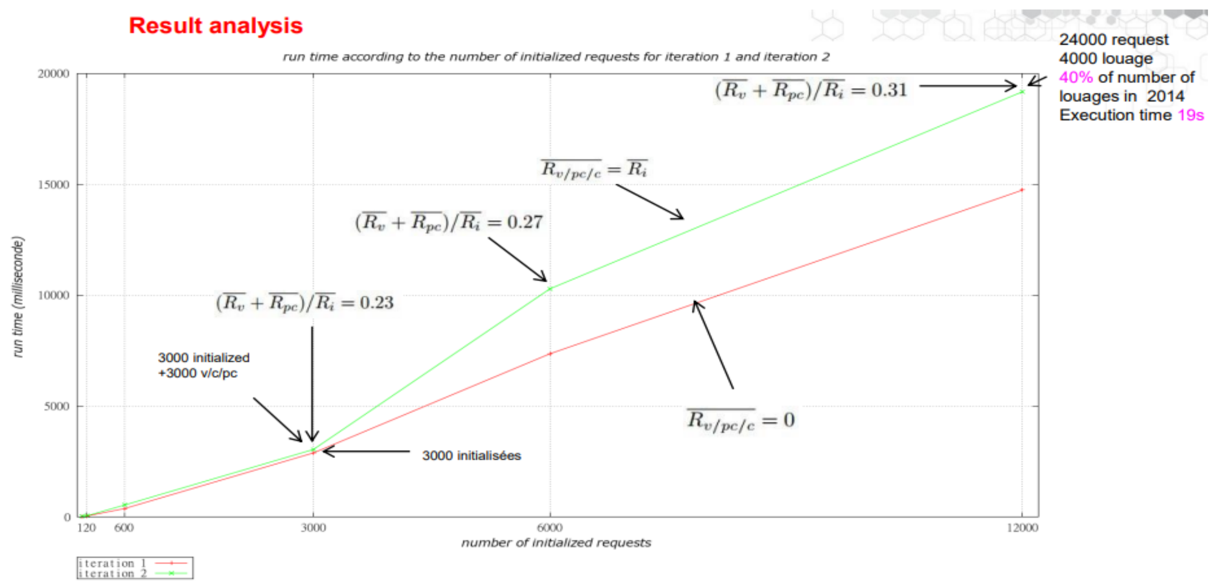
The algorithm works by incrementally pairing offers and demands in a greedy manner, with the objective of maximizing the number of contractualized requests in registration order. For that, for each iteration, it follows this principle:

1. It starts by normalizing all time windows of request offers (drivers) and demands (passengers) in order to validate them.
2. Then, it adds the driver to a journey and then passengers to this journey in a First In First Out (FIFO) manner if the time windows of both the offer and the demand are valid and if there are still places.
3. The time window of the passenger is then updated to correspond to the one of the driver.
4. The previous process is done until no more drivers or passengers are available.



Result of the application of the GILA algorithm on an example from Mr.Canalda's slides

The GILA algorithm is useful because it provides a way to efficiently and effectively pair offers and demands in the Louage system, which can help to optimize routes, reduce waiting times, and improve overall passenger satisfaction. Moreover, the complexity of this algorithm is linear (O(n)) since it is done in a greedy manner which is a great point to reduce the execution time of the program.



Run time of the GILA algorithm from Mr.Canalda's slides

CIMO

Introduction

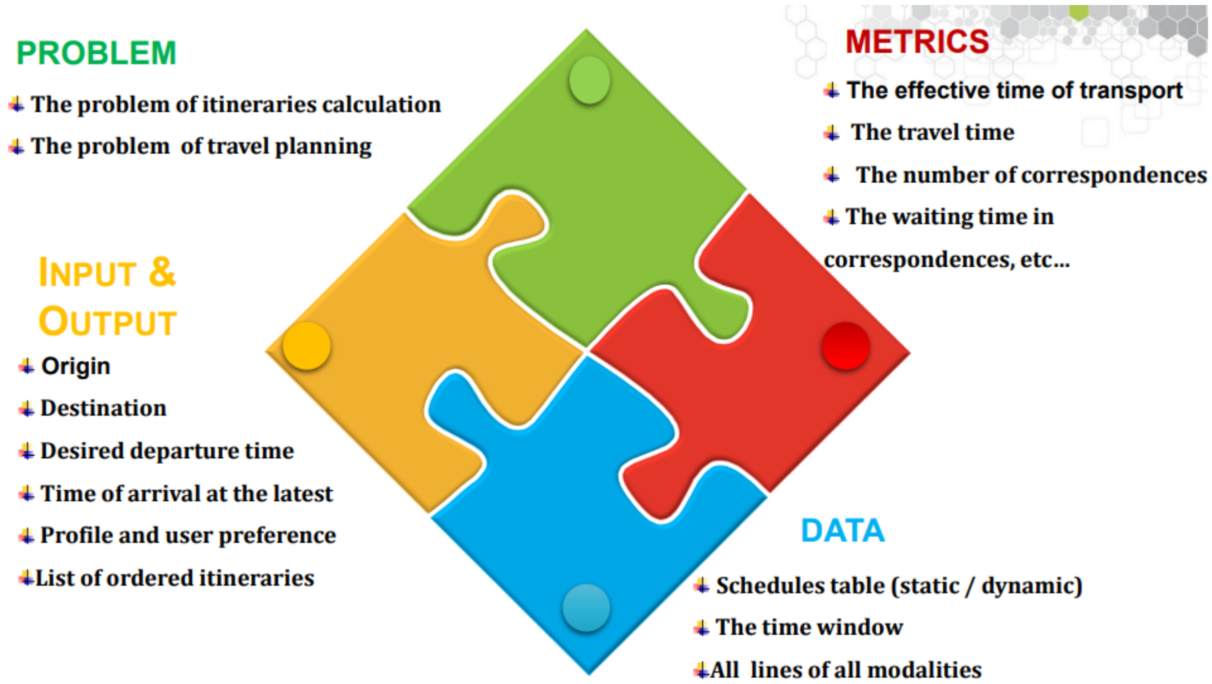
Traveling is an integral part of everyday life, playing a crucial role in our personal, professional, and social experiences. It enables us to go to work, visit friends and family, access education and healthcare, and explore new destinations.

For that, several modes of transport can be used (e.g cars, buses, trains, trams, bicycles or our own two feets). These modes of transport determine how efficiently we can move from one point to another, impacting our quality of life.

However, the challenge for an individual is to know when he should leave his origin and by what modes of transport he should take to reach his destination on time. Indeed, in most cases, we have to combine several modes of transport since they don't deliver people right on the destination (train stations, bus stops...)

We then need to address the problem of managing all these modes of transport in order to provide efficient itineraries for travelers and allow them to know when to leave, where and by what means. This is a common problem in urban areas where multiple transportation options are available, but finding the best itinerary can be time-consuming and confusing.

Several solutions have already been proposed. However, in most cases, they are not implementing several modes of transport (i.e they are single information systems) and are not dynamic (advances, delays etc. are not taken into account).


PROBLEM
The problem of itineraries calculation
The problem of travel planning
METRICS
The effective time of transport
The travel time
The number of correspondences
The waiting time in correspondences, etc...
INPUT & OUTPUT
Origin
Destination
Desired departure time
Time of arrival at the latest
Profile and user preference
List of ordered itineraries
DATA
Schedules table (static / dynamic)
The time window
All lines of all modalities

The core elements to solve the problematic from Mr.Canalda's slides

Ordered Calculator of Multimodal Itineraries (CIMO)

CIMO is an efficient 2-phases calculator of multimodal itineraries for real trans-territories based on a dynamic programming algorithm. It allows us to find the optimal solution given the earliest departure time, location and the latest arrival time, location minimizing the travel time and if possible, the number of modal transfers.

An itinerary is presented as a list of stations and should respect some constraints like the fact that the same line can't be taken two times

The algorithm implements the cut + price + share principle in order to select the itineraries and also reduce the computing time. It has been tested on a prototype with 9 stations (3 bus stations in Belfort, 3 bus stations in Montbéliard and 3 train stations between Belfort and Montbéliard) and works as follow:

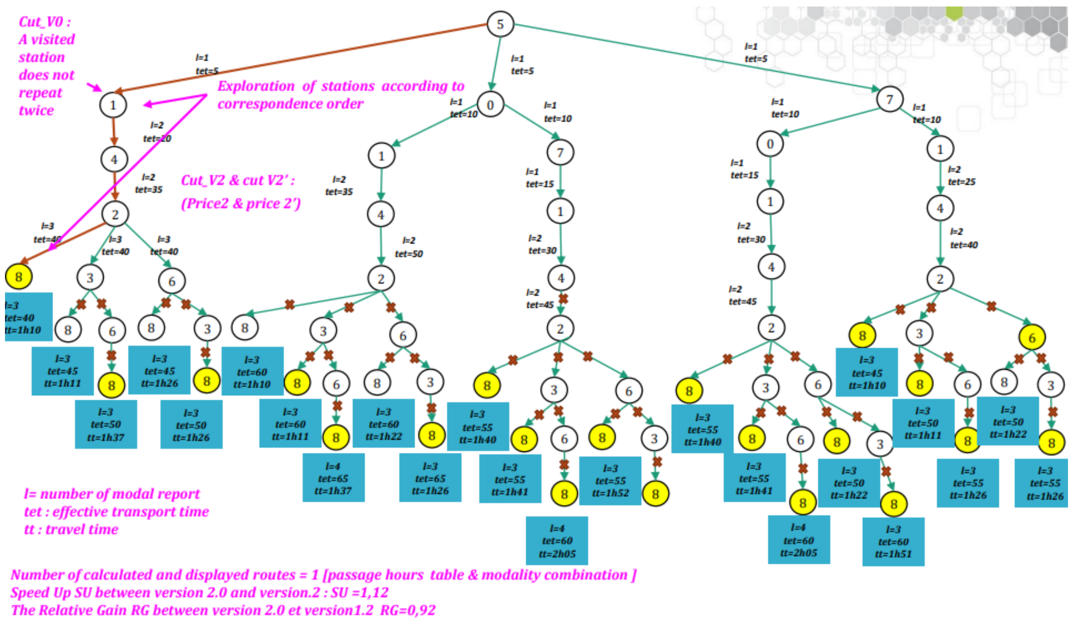
1. From an origin station, find all the other possible stations and note for each the travel time and number of modal reports.
2. Repeat step 1 until we find the arrival station.

At each step, several conditions are tested in order to reduce the complexity by reducing the number of recursive calls.

Condition 1: A visited station can't be repeated twice.

Condition 2: The number of modal reports (change of modality) can't be greater than 3.

At the end, the best itinerary selected will be the one with stations according to correspondence order. This approach allows CIMO to minimize both correspondences and global transport times, making it a useful tool for travelers who want to find the most efficient itinerary.



Cut_V0 : A visited station does not repeat twice
Exploration of stations according to correspondence order
Cut_V2 & cut V2' : (Price2 & price 2')
l= number of modal report
tet : effective transport time
tt : travel time
Number of calculated and displayed routes = 1 [passage hours table & modality combination]
Speed Up SU between version 2.0 and version.2 : SU =1,12
The Relative Gain RG between version 2.0 et version1.2 RG=0,92

Result of the application of CIMO on an example from Mr.Canalda's slides

Introduction

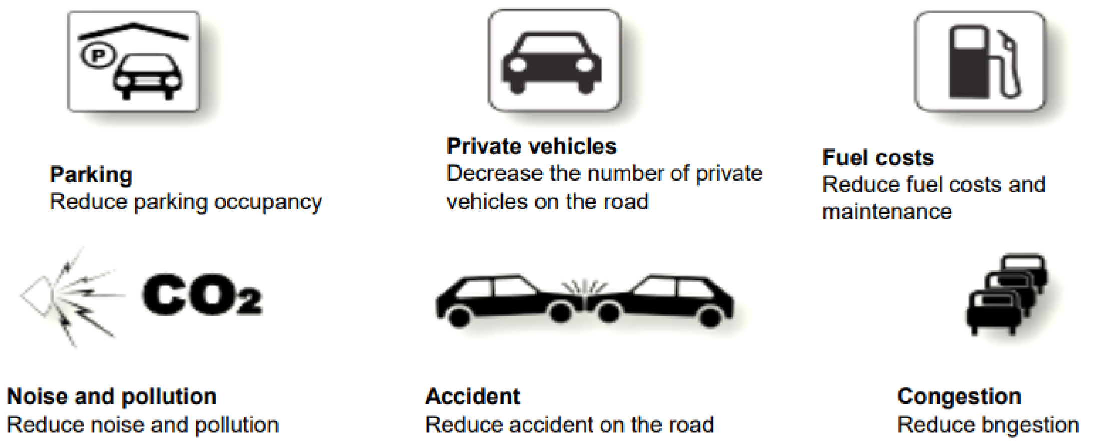
As I said before with CIMO, transportation is really important. However, a lot of people don't use public transports like buses or trains. They rather use their own cars for convenience which leads to other types of problems.

Carpooling is the practice of sharing a car journey with one or more people. The benefits of carpooling is that it reduces the fuel costs, the traffic congestion, the pollution, parking occupancy and so on. But it also presents challenges such as finding compatible carpool partners and coordinating schedules.

We can distinguish two types of carpooling:

- Traditional carpooling: Involves a fixed group of people who share a car journey.
- Dynamic carpooling: Allows for more flexibility and spontaneity by enabling people to find carpool partners and arrange rides on an as-needed basis. It takes into account factors such as transhipment, which allows for changing vehicles along a trip.

Carpooling is both multi-constraints and multi-objectives. Indeed, for constraints, we should take into account time windows of people, capacity of vehicles, roles of people (driver or passenger), the itinerary, the location of everyone... In the same way for objectives, we should maximize the occupancy rate of vehicles in order to minimize their number.


Parking
Reduce parking occupancy
Private vehicles
Decrease the number of private vehicles on the road
Fuel costs
Reduce fuel costs and maintenance
CO2
Noise and pollution
Reduce noise and pollution
Accident
Reduce accident on the road
Congestion
Reduce bngestion

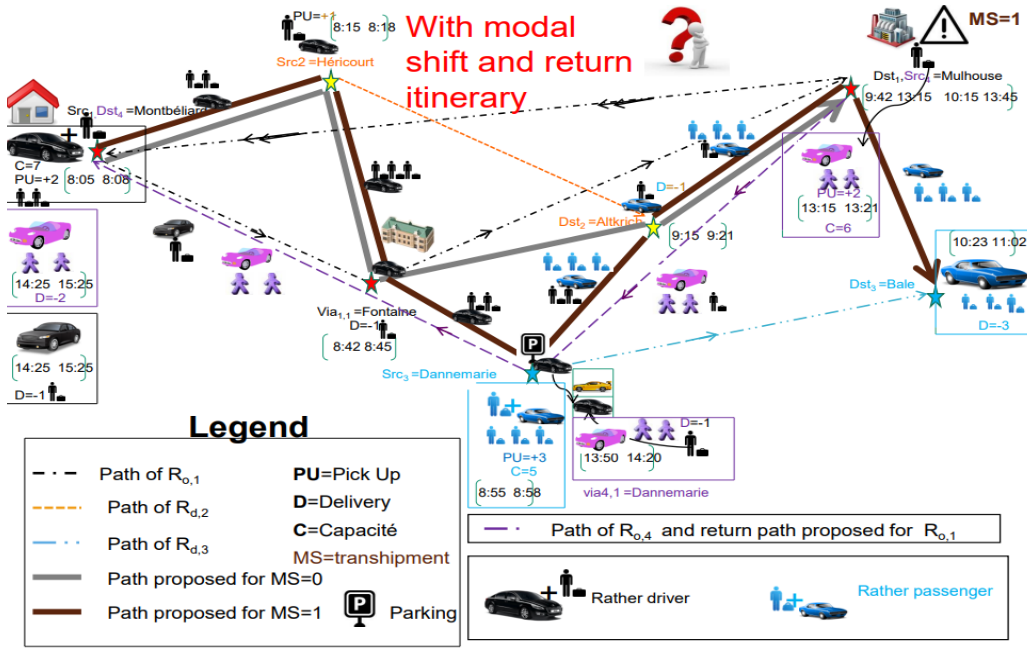
Advantages of carpooling from Mr.Canalda's slides

Algorithm with transhipment

The algorithm with transhipment proposed for dynamic carpooling works by matching drivers and riders based on their starting points, destinations, and preferred modes of transportation. The algorithm takes into account multiple constraints and objectives, such as time windows, capacity constraints, and modal shift authorization, in order to create efficient and effective carpool arrangements.

For that it follows this principle:

1. It begins by processing a request from a driver who has agreed to make a modal shift (i.e. switch from driving to riding or vice versa). The request includes information about the driver's starting point, destination, and preferred mode of transportation.
2. It then searches for a compatible rider who is traveling along a similar route and is willing to share a ride with the driver. This involves taking into account multiple constraints and objectives, such as time windows, capacity constraints, and modal shift authorization.
3. If a compatible rider is found, the algorithm creates a carpool arrangement that meets the needs of both the driver and the rider. This may involve transhipment, which allows for changing vehicles along a trip, in order to optimize the carpool arrangement.
4. It then searches for additional requests from other drivers and riders and repeats the process of matching them based on the constraints and objectives identified in step 2.
5. Once all requests have been processed and carpool arrangements have been made, the algorithm generates a list of carpool trips that includes information about the starting point, destination, mode of transportation, and any necessary transhipment.



Result of the application of dynamic carpooling with transhipment on an example from Mr.Canalda's slides

ANNEXES

GitHub with code of the 4 TDs:
https://github.com/Antonin-Winterstein/mobility_in_smart_cities_practical_works