Mobility in Smart Cities REPORT

Killian MAXEL
Master IoT 2023-2024

Teacher: Mr. Philippe CANALDA

INTRODUCTION

This report aims to explain how three algorithms studied in course work and their importance in the mobility of users. These algorithms are Dynamic Carpooling, CIMO (Ordered Multimodal Route Planner), and Louage.

Then we will talk about the makes change algorithm and its variants implemented as different programs in groups.

Finally we will explain our implementation concerning transitivity.

Dynamic Carpooling Intra-Modal with Transshipment

The Dynamic Carpooling Intra-Modal with Transshipment (DCIT) algorithm is a solution designed to facilitate efficient drive sharing among drivers and passengers traveling along similar routes. This algorithm aims to reduce traffic congestion, to optimize transportation resources, and to provide flexibility to users. Here is how it works, step by step.

- ❖ Normalized Time Windows

The first step is the normalization of the time windows provided by drivers and passengers. It converts time windows into a standardized format to facilitate accurate matching and to enable effective comparison and compatibility assessment between users.

- ❖ Itinerary Division

Then, to simplify the matching process, the algorithm divides the trip into two components: a single itinerary and a return itinerary. This separation ensures that matching is based on the specific needs and preferences of drivers and passengers for both the outbound and return journeys.

- ❖ Verification of Return Itinerary

The return itinerary ensures that drivers can retrieve their vehicles at the end of the journey is essential. Verification of a return itinerary is conducted to secure the convenience and commitment of participating drivers.

- ❖ Matching Process

Once the previous steps are completed, the algorithm proceeds to match drivers and passengers. This matching process takes into account several factors:

- ➢ Location: matching users traveling along on similar routes.
- ➢ Destination: ensuring passengers share the same or near destinations.
- ➢ Time Windows: aligning users with compatible travel schedules.
- ➢ Vehicle Capacity: the capacity of each vehicle is considered to optimize ride sharing.
- ➢ Constraints and Preferences: user-specific constraints and preferences, such as smoking, music, pet, etc.

❖ Flexibility and Adaptability

The algorithm is designed to adapt to various scenarios and user requirements. Its flexibility allows it to address different transportation needs and adapt to diverse user profiles, it contributes to the effectiveness in handling a wide range of carpooling scenarios.

❖ Testing and Evaluation

Testing and evaluation ensures effectiveness in reducing traffic congestion and improving transportation efficiency. Real-world testing and simulations demonstrate the algorithm's capacity to optimize routes, reducing the number of vehicles on the road, and enhancing its efficiency.

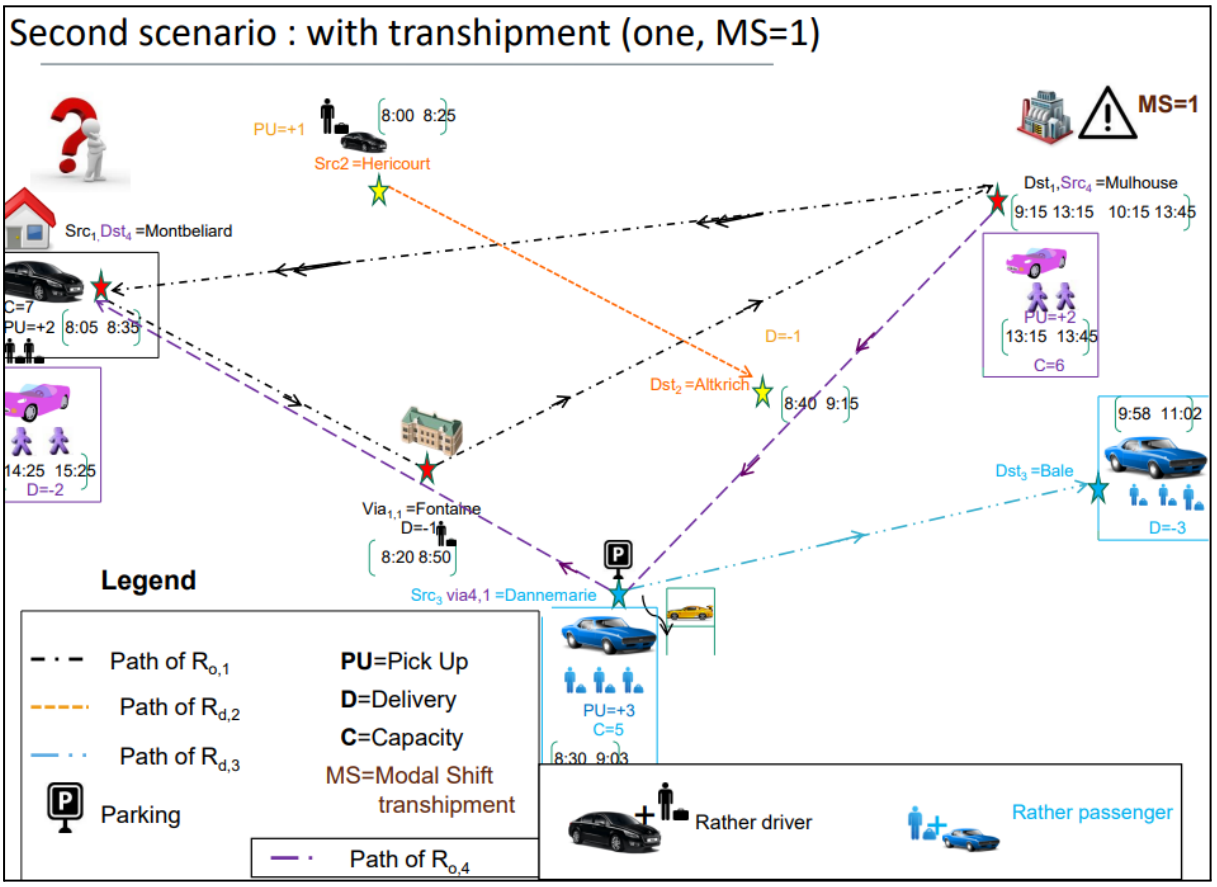


Image from mr Canalda's slides showing how the solution works

Louage

The Louage transportation system in Tunisia is a challenge of matching transportation offers and demands using an innovative incremental greedy pairing algorithm. So it is a multi-constraint and multi-objective optimization challenge, solved by employing the

incremental greedy pairing algorithm that incrementally builds a solution by making locally optimal choices at each step. Here are the different aspects of Louage in details:

Incremental Greedy Pairing Algorithm:

Through an iterative process, the algorithm creates pairs of offers and demands, based on their compatibility and predefined optimization criteria, it adapts to new offers and demands without the need for a full re-computation, making it efficient and responsive to real-time changes. This robust management system that collects, processes, and executes offers and demands is what provides optimized transportation matches.

So this algorithm operates by incrementally matching offer and demand in a greedy way, aiming to maximize the number of validated requests based on their order of registration. To achieve this, each iteration follows these steps:

- Initial normalization of all time windows for request offers (driver) and demands (passenger) to validate each of them.
- Addition of a driver to a journey and then of one passenger or more, using FIFO (First In First Out) which allows that both offer and demand's time windows stay valid, and that there are still available slots.
- Updates on the passenger's time window to align with the driver's on the journey.
- Steps repeated until there are no more drivers or passengers available.

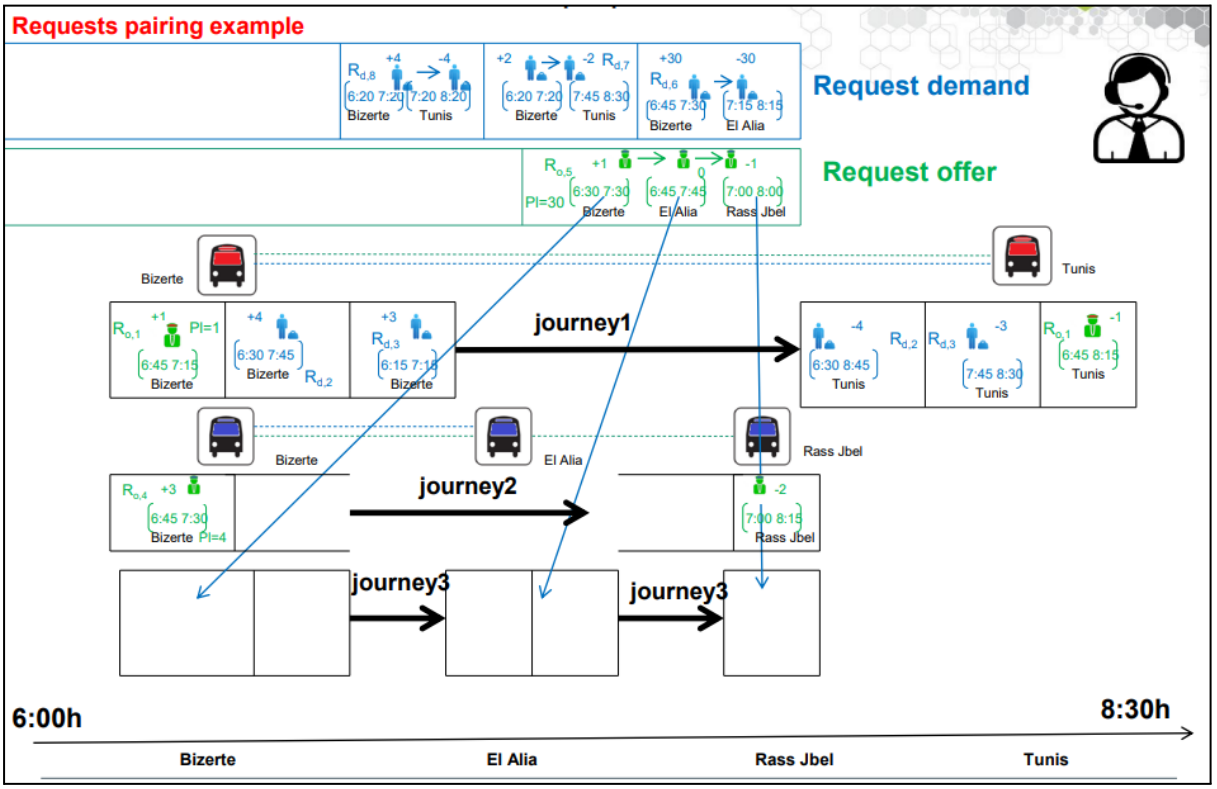


Image from mr Canalda's slides showing how the solution works

Evaluation:

The Louage performance is evaluated thanks to the number of requests processed, the quality of matches generated by the system, and the computational efficiency of the incremental greedy pairing algorithm.

By leveraging advanced algorithms and digital technologies, Louage allows an optimization which contributes in the improvement of the quality of life for tunisians by reducing their travel times, maximizing vehicle occupancy, and enhancing the revenue for transportation providers.

CIMO

CIMO is an algorithm designed to compute multimodal itineraries, considering real-time transportation demands, it is constituted of two crucial phases: pre-processing and query, each one with its objectives.

- ❖ Pre-processing Phase:

During this phase a tree is built using several transportation modes, representing buses, trains, metros, and more as a network of nodes and edges, with nodes denoting stops or stations and edges depicting the connections between them. Furthermore, CIMO allows a dynamic nature of transportation demand by creating a "time-dependent" graph, factoring in variables like time of day and day of the week.

- ❖ Query Phase:

The query phase is what allows CIMO to do real-time itinerary calculation. It accomplishes this thanks to:

- ➢ Cost Function: a dynamic programming function that evaluates the optimal itinerary based on two factors: minimizing the number of correspondences and minimizing the global transport time.

- ➢ Heuristic Search: The algorithm explores the transportation network by initiating at the origin of the query (the root) and advances using breadth-first search (BFS), examining all potential paths through the network.

- ➢ Termination Conditions: The algorithm terminates when it reaches the destination specified in the query or when it has examined all feasible paths through the network.

❖ Efficiency and Real-Time Processing:

CIMO is particularly efficient in its real-time itinerary calculations, by taking into account the time-dependency of transportation demand and by optimizing using a cost function that simultaneously considers correspondences and transport time, in short it minimizes travel challenges and optimizes multimodal travel for users.

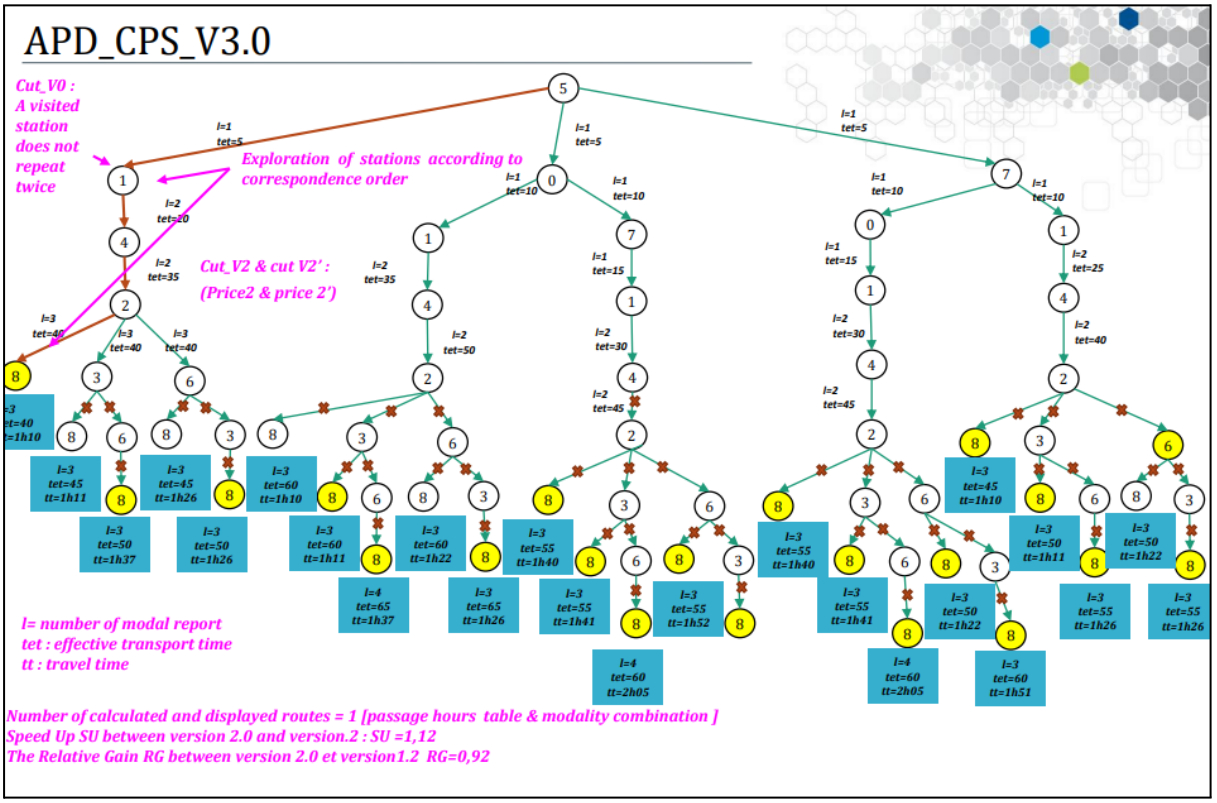


Image from mr Canalda's slides showing how the solution works

Conclusion concerning the 3 algorithms

In conclusion, the Dynamic Carpooling Intra-Modal with Transshipment algorithm, the Louage transportation system in Tunisia with the incremental greedy pairing algorithm, and CIMO are all transportation optimization solutions, each of them bring unique benefits and features to the transportation sector.

The Dynamic Carpooling algorithm is an adaptable approach reducing traffic congestion and improving transportation efficiency, confirmed by its real-world testing.

The Louage transportation system in Tunisia, employing the incremental greedy pairing algorithm, enhances transportation service matching with its multi-constraint, multi-objective optimization. Its implementation was successful according to the statistics presented, digital technology can transform transportation services in Tunisia.

CIMO is a versatile algorithm that enhances the calculation of multimodal itineraries. Its robust pre-processing phase and efficient query phase, along with its adaptability to real-time factors, position is helpful in multimodal transportation.

In summary, while DCIT, CIMO, and Louage have all transportation-related objectives, they differ in terms of functionalities, indeed DCIT targets carpooling, CIMO specializes in real-time multimodal itinerary calculations, and Louage concentrates on optimizing the allocation of transportation services. Each algorithm is implemented in order to address specific transportation challenges and goals.

So these solutions represent advancements in transportation optimization, with the potential to answer to various challenges and improve the overall transportation experience of users.

MAKES CHANGE

The Makes Change problem is a problem in computer science for optimization, it involves finding the most efficient way to give change for a given amount using a set of coins. In this study, we explore two distinct approaches to solving this problem: an iterative greedy solution and a recursive one. We also need to return the best solution, a solution and all possible combinations. To do so we will use Java language using IntelliJ IDE.

Background and specifications

Before starting any implementation, here are keywords to keep in mind:

Iterative Greedy Solution:

- Greedy approach, always choosing the largest coin denomination.
- Limitation: May not always yield the optimal solution.

Modified Iterative Approach:

- Considers decreasing L (remaining amount) for optimized coin selection.
- Orders combinations for efficient processing.

Stopping Criteria:

- Controls the search process in the iterative approach.
- Balances computational efficiency with solution quality.

Recursive Approach:

- Provides a complementary, memory-efficient alternative.
- May have higher runtime complexity.

The iterative greedy solution is a straightforward approach, where the algorithm selects the largest available coin denomination at each step until the target amount is reached. However, this method may not always yield the optimal solution.

Using what is expected, we have to implement

- a greedy algorithm using iterative approach, generating a solution (may not be the right one)
- an algorithm using iterative approach generating all possible combinations
- a greedy algorithm using iterative approach generating the BEST solution
- a recursive algorithm generating all possible solutions

Implementation and Results

Iterative Greedy Solution (**makeChangeBest**):

- This algorithm uses a greedy approach by always selecting the largest coin denomination available to reduce the remaining amount.
- It starts by converting the given amount to cents and initializes an array to store the minimum number of coins needed for each amount.
- The algorithm iterates through each coin denomination and calculates the minimum number of coins needed to make change for each amount from the smallest to the target amount.
- Finally, it keeps the best combination of coins found to achieve the target amount with the minimum number of coins.
- We need to be sure that it is the best solution (not a random solution)

Modified Iterative Approach (**makeChangeAllIterative**):

- This algorithm explores all possible combinations of coins iteratively.
- It starts with the highest coin denomination and loops through all possibilities to find combinations that sum up to the target amount.
- The result is a list of all valid combinations.

Recursive Approach (**makeChangeAllRecursive**):

- This algorithm uses recursion to find all possible combinations of coins that sum up to the target amount.
- It explores different combinations by recursively adding coins to the current combination until the target amount is reached.
- The result is a list of all valid combinations.
-

Now, let's have a look on the results:

1) Iterative Greedy Solution (**makeChangeBest**):

We conducted an investigation to assess whether the ordering of the coin set affected the selection of the best solution. For an amount of 12.35, we tested two different coin sets: {5, 2, 1, 0.5, 0.2, 0.1, 0.05} and {2, 1, 0.5, 0.2, 0.1, 0.05, 5}. The experiment confirmed that the algorithm identifies the same best solution, even when the coin sets are not ordered identically. The following two combinations represent the best solution for the given amount:

[5.0, 5.0, 2.0, 0.2, 0.1, 0.05]
[2.0, 0.2, 0.1, 0.05, 5.0, 5.0]

This demonstrates the robustness of the algorithm in consistently providing the optimal change, regardless of the ordering of the coin denominations.

2) Modified Iterative Approach (**makeChangeAllIterative**):

Because of memory errors with the previous set (long runtime), we decided to work on a smaller inputs, with an amount of 10 and a set of coins equals to {5, 2, 0.5}, the algorithm provides several valid combinations of coins that sum up to 10, such as [2.0, 2.0, 2.0, 2.0, 0.5, 0.5, 0.5, 0.5]. The total number of combinations is 10.

3) Recursive Approach (**makeChangeAllRecursive**):

Because of the limitations of the previous algorithm, and to compare results with this recursive method, we decided to use the same small set as before. The algorithm provides the same combinations as the modified iterative approach, so to compare with the previous algorithm, here are the different runtimes using System.nanoTime():

- **makeChangeAllIterative: 68 000 * 10^-9s**
- **makeChangeAllRecursive: 110 800* 10^-9 s**

Obviously the recursive approach is time consuming, but when coming to larger input, contrary to the iterative approach, we have a lower memory consuming approach.
Indeed, even if there were memory constraints for our iterative algorithm, we decided to test the recursive one with the initial inputs of 12.35 and coins {5, 2, 1, 0.5, 0.2, 0.1, 0.05}, the recursive algorithm finds a total of 266,724 different combinations and is working well (without memory error)

So when generating the best solution, the iterative greedy solution provides the right amount with the minimum number of coins, Whereas when we generate all solutions, the iterative solution has memory constraints while recursive approaches find all possible combinations for a given amount without problem.

In summary, while the iterative greedy solution offers simplicity, it may suffer from memory inefficiency for larger inputs, in our case, when we generate all solutions. Modifying our iterative approach may be a solution to optimize memory usage, but further research is needed. We could also try the algorithm's implementation using C language, which may be more effective, especially for its runtime and its memory garbage management.

MAKES CHANGE PROGRAMS

This part is a continuation of the makes change algorithm work, It is asked to implement seven different variants of the previously presented makes change, joined with outputs in text format. This work was done thanks to the help of Antonin Winterstein and Guillaume Martin, using Python language this time.

Program 1: Using a greedy algorithm to find a solution to the makes change problem, the program sets the available coins at disposition (coins) and uses the largest available coins first to make change, the found combination is the output stored in *sol_1.txt* with its execution time.

Program 2: This program computes all the possible solutions without using any recursivity. The output containing all combinations is stored in *sol_2*.txt with the execution time. However, because of the size of the text file, you have to generate it yourself using your favorite IDE and Python.

Program 3: A recursive approach is used (dynamic programming), to generate all valid combinations. Using an in-depth walk, valid combinations are printed when found, the goal is to avoid large memory consumption. All the combinations are stored in *AllTheSolutionsTest1.txt* with the execution time, which may be slower due to the frequent I/O operations.

Program 3 bis: This program is the same as the previous one except that the solutions generation is modified to prioritize the maximization of the quantity of the processed value's units.

Program 4: Program 4 returns all combinations without displaying them in real-time, they are stored in a 2D array, to avoid a long runtime. The code also checks if the result is the same as the one found earlier to avoid duplicates. All combinations are stored in *sol_4.txt* with the execution time. However, because of the large size of the file and as explained as before in program 2, you have to generate it.

Program 5: It is an extension of the third program, but it evaluates the cost of each solution according to the number of units of the coin values. It returns the best solution incrementally by replacing previous values with an improved solution. The output is stored once again in a text file, which is called *CORRECT_SolutionsWhichImproveIncrementally.txt*, you will find the execution time of program3 and this one.

Program 6: This program is an extension of Program 4, it stores the best solution in a table among others. The combinations are stored in a 2D array and are ordered to ensure the maintenance of valid combinations. The combination is stored in sol_4-6.txt with the execution time.

Program 7: This program implements a cut to optimize the search of the best solution, the cost is computed as the program continues, and if the partial cost exceeds the best-known solution cost, the program stops processing, we do it to reduce computation. The output is in the text file *sol_4-7.txt* with the execution time taken.

TRANSITIVITY

Transitivity is a property of relationships between elements or objects that says, in mobility's case, that if we can go from a place A to a place B, and go from a place B to a place C, then we can join C from A.

Thanks to graphs represented with an array of states and an array of edges (represented as a pair between the starting point and the ending point), or matrices like the Origin-Destination one, it is possible to check, remove or add transitivity (transitivity closure) between locations.

In summary, transitivity means that if A is related to B, and B is related to C, then there is a link from A to C.

The ability to include or eliminate transitive links has advantages, indeed eliminating unnecessary transitive links can reduce complexity, and result in more efficient and optimized algorithms, it can also enhance clarity, and so improve performances. Moreover adding missing transitive links can ensure a logical representation, which is essential for accurate analysis and decision-making across applications.

To test our implementations, we decided to use 2 types of data:

- one which is in the form of a graph where we define the states id and the edges as an array of pairs (the first element is the starting point, the second element the destination point)

- the other as a matrix, a 2d array that says that if a value at index (i,j) is 1, then there is an edge between state i and state j.

Checking transitivity:

To check if the given matrix or graph is fully transitive or not, we check for each state i, j and k if the value of (i,j) is 1 and (j,k) is 1, then there must 1 at (i,k), if that's not the case, a transitive link is missing, we stop from here and the matrix is not transitive, we return false.

```
Matrix:
[1, 0, 1, 0, 0]
[0, 1, 0, 1, 1]
[1, 0, 1, 0, 0]
[0, 1, 0, 1, 1]
[0, 1, 0, 1, 1]
The matrix is transitive
----
Matrix:
[1, 0, 1, 0, 0]
[0, 1, 0, 1, 1]
[1, 0, 1, 0, 0]
[0, 1, 0, 1, 1]
[0, 1, 1, 1, 1]
0 at coordinates (1, 2) while there are 1 at (1, 4) and (4, 2)
The matrix is not transitive
```

output of the program when checking transitivity on a matrix

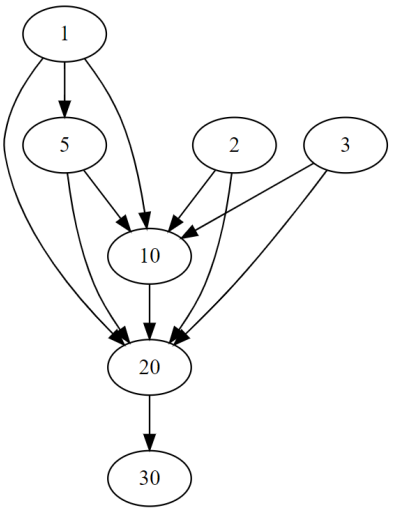
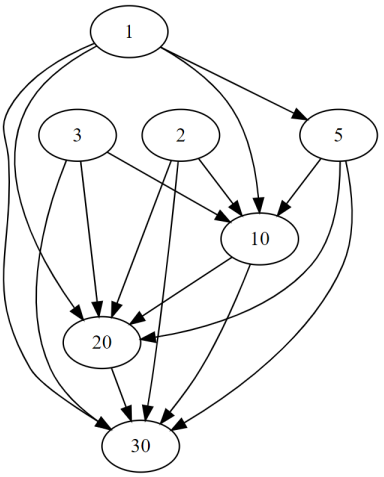
Adding transitivity:

To add missing transitive links to a graph, to make it fully transitive, we need to check if for states i, j and k, if there are edges for (i,j) and (j,k) then (i,k) must exist, if not, we add it.

While there is a modification to be put to the graph or the matrix, the process continues into checking and adding missing transitive links.

```
----
GRAPH
------

states: [1, 2, 3, 5, 10, 20, 30]
edges before:
[(1, 5), (1, 10), (1, 20), (2, 10), (2, 20), (3, 10), (3, 20), (5, 10), (5, 20), (10, 20), (20, 30)]
adding (1, 30) because (1, 20) and (20, 30)
adding (2, 30) because (2, 20) and (20, 30)
adding (3, 30) because (3, 20) and (20, 30)
adding (5, 30) because (5, 20) and (20, 30)
adding (10, 30) because (10, 20) and (20, 30)
edges after:
[(1, 5), (1, 10), (1, 20), (2, 10), (2, 20), (3, 10), (3, 20), (5, 10), (5, 20), (10, 20), (20, 30), (1, 30), (2, 30), (3, 30), (5, 30), (10, 30)]
```

output of the program when transitivity applied on a graph

	
graph before adding transitivity	graph after adding transitivity

```
------
MATRIX
------
initial_matrix:
[0, 0, 0, 1, 1, 1, 0]
[0, 0, 0, 0, 1, 1, 0]
[0, 0, 0, 0, 1, 1, 0]
[0, 0, 0, 0, 1, 1, 0]
[0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0]
add (1,30) because (1,20) and (20,30)
add (2,30) because (2,20) and (20,30)
add (3,30) because (3,20) and (20,30)
add (5,30) because (5,20) and (20,30)
add (10,30) because (10,20) and (20,30)
final matrix:
[0, 0, 0, 1, 1, 1, 1]
[0, 0, 0, 0, 1, 1, 1]
[0, 0, 0, 0, 1, 1, 1]
[0, 0, 0, 0, 1, 1, 1]
[0, 0, 0, 0, 0, 1, 1]
[0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0]
```

Output of the program when transitivity applied on a matrix (different from previous graph)

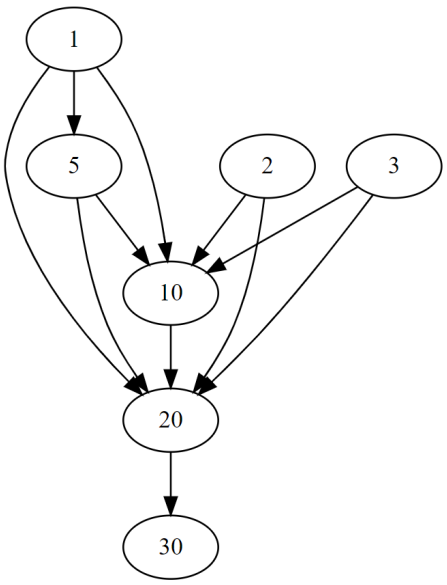
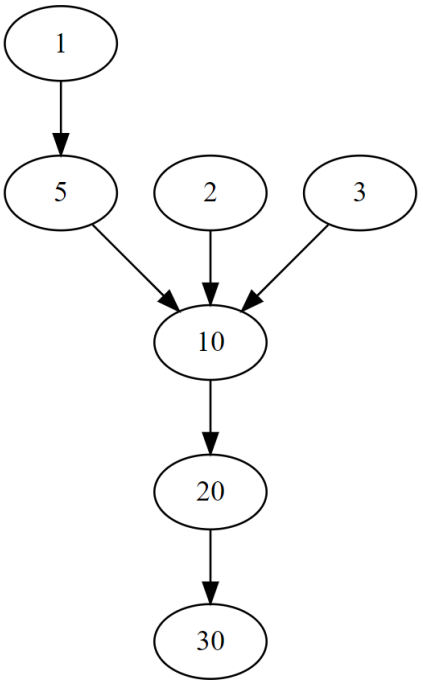
Removing transitivity:

The process here is almost the same as before, except that we need to remove any link that could make the matrix or the graph transitive, to do so we check if at (i,j) and (j,k) we have an existing edge at (i,k), if that's the case we remove it.

Once it is removed, the process is restarted on the updated graph or matrix, until there are no modifications left.

```
first graph example:
states: [1, 2, 3, 4]
original edges: [(1, 1), (1, 2), (2, 3), (3, 4), (4, 1), (4, 2)]
Transitive relation to remove found: (4, 2) because (4, 1) and (1, 2)

final edges: [(1, 1), (1, 2), (2, 3), (3, 4), (4, 1)]

----------

second graph example:
states: [1, 2, 3, 5, 10, 20, 30]
original edges: [(1, 5), (1, 10), (1, 20), (2, 10), (2, 20), (3, 10), (3, 20), (5, 10), (5, 20), (10, 20), (20, 30)]
Transitive relation to remove found: (1, 10) because (1, 5) and (5, 10)
Transitive relation to remove found: (1, 20) because (1, 5) and (5, 20)
Transitive relation to remove found: (2, 20) because (2, 10) and (10, 20)
Transitive relation to remove found: (3, 20) because (3, 10) and (10, 20)
Transitive relation to remove found: (5, 20) because (5, 10) and (10, 20)

final edges: [(1, 5), (2, 10), (3, 10), (5, 10), (10, 20), (20, 30)]

----------

third graph example:
states: [1, 2, 3, 4, 5]
original edges: [(1, 3), (3, 2), (3, 5), (4, 1)]

final edges: [(1, 3), (3, 2), (3, 5), (4, 1)]
```

Output concerning the execution of the program when removing any transitive link from given graphs

	
the graph before the removal of transitive links on second graph	the graph after the removal of transitive links on second graph

```
initial matrix:
[0, 0, 0, 1, 1, 1, 0]
[0, 0, 0, 0, 1, 1, 0]
[0, 0, 0, 0, 1, 1, 0]
[0, 0, 0, 0, 1, 1, 0]
[0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0]
0 at index (0, 4) because 1 at (0, 3) and 1 at (3, 4)
0 at index (0, 5) because 1 at (0, 3) and 1 at (3, 5)
0 at index (1, 5) because 1 at (1, 4) and 1 at (4, 5)
0 at index (2, 5) because 1 at (2, 4) and 1 at (4, 5)
0 at index (3, 5) because 1 at (3, 4) and 1 at (4, 5)
final matrix:
[0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0]
```

Output concerning the execution of the program when removing any transitive link from given matrix

APPENDICES AND REFERENCES

GitHub:
https://github.com/Nexusprime22/Mobility_and_smart_cities_MAXEL