Authors

**Dornetshumer Simon**

**Pammer Thomas**

Programming for Business tasks

**Nicolas Forget**

07 2024



Programming for Business tasks – Final Project - Report

Table of Contents

[1. Read\_instance.jl 3](#_Toc171942549)

[1.1. Read\_instance 3](#_Toc171942550)

[1.2. Create\_pre\_succesors 3](#_Toc171942551)

[2. Task1.jl 3](#_Toc171942552)

[2.1. Solve\_ILP 3](#_Toc171942553)

[2.2. Find\_connections 3](#_Toc171942554)

[2.3. Find\_all\_tours 3](#_Toc171942555)

[2.4. connect\_solution 4](#_Toc171942556)

# Read\_instance.jl

In this Julia file we read in the given instances and generate the pre- and successors of each node of each instance. This code is used in all other tasks as we need the data in every subtask.

The read instance file includes two functions:

## Read\_instance

This function uses the DelimitedFiles package in order to read in all lines of each instance. The first 3 lines of each instance represent n, the number of total nodes, m, the number of delivery points and Q, the actual delivery points. From line four to the end of the file every line represents an arc with the costs or length of this connection. This function returns n, m, Q as well as the arcs and costs of this instance.

## Create\_pre\_succesors

This function uses n and the arcs as input and returns a vector of vectors for the predecessors and for the successors each.

# Task1.jl

In task 1 we use the packages CPLEX and JuMP to solve the ILP. We include the read\_instance file as we need the data to solve this Integer Linear Programming problem. In order to run the implementation, you just need to source it.

## Solve\_ILP

The solve\_ILP function creates the ILP Model without constraint four. This means it creates a model where it is ensured that every node is left the same number of time as it is entered, that every delivery point gets served and that all decision variables are integer but not that there is one tour which servers all this delivery points as there is no restriction containing subtours. It returns an ILP model which will be used in the following functions of this code.

## Find\_connections

The find\_connections function looks for all the connections of the visited nodes. It creates the subtours which we need to make constraint four. It takes as input the current node c as well as the current subtour vector subtour and the visited BitVector which describes if a node is already visited and the decisions variables x of the current solution. It then pushes the current node c to the current subtour and looks for all connections of this node. It is worth mentioning that a node is connected with another if the value of the decision variable x is above 0.5 (could be any value greater than zero and smaller than 1). It then iterates through as long as there is still an unvisited node in the current tour. This function is used in the the find\_all\_tours function in order to receive the subtours and is also programmed in it as it adapts the used vectors.

## Find\_all\_tours

The find\_all\_tours function uses the values of the decision variable x as well as the number of nodes and the delivery points as input and creates all subtours of the current solution which are then returned as a vector of vectors. Therefore, an empty vector of vectors is created for all subtours and a BitVector in order to check if a node is already visited. Then the function iterates through all the delivery points as every subtour contains at least one delivery point as otherwise this subtour would have no profit but additional costs. Before using the find\_connections function in order to look for all nodes In this subtour it is checked if this node has not been visited yet. Then the subtour is generated and pushed as a vector to the vector of vectors “all components” which at the end has all subtours in it. The Vector of subtours is then needed in order to create constraint 4 which is done in the connect\_solution function.

## connect\_solution

The connect\_solution function uses the model, the delivery points as well as the number of nodes and the vector of vectors of successors as input. Then the Boolean variable subtour\_detected is generated and set to true. These variables checks if there are still subtours in the solution. If this variable is true, the model gets optimized. Then all the subtours are generated with two functions described above. Then for every subtour constraint four is added. The if condition !all checks if there are all delivery points in the current subtour. If this is the case, then there is one tour which includes all the subtours and so this is the optimal solution for this problem. To check the result, we then print out the objective value.

With the use of these four functions task one is completed and can be used in task 3 in order to compare it to the implementation of task 2.

# Task 3

In task 3 we compared the processing times of task 1 and task 2 for the different instance sets.

Therefore we use the functions calc\_comp\_time\_task1 and calc\_comp\_time\_task2. As an input we take the vector with the names of the instances. Then we first read in the data of each instance using the commands programmed in the read\_instance.jl file and then use the commands of each task. To calculate the average processing time, we use the @elapsed command and then first add all computation times together and then divided the sum through the number of instances we looked at.

In the following table we can see the comparison between the computation times of the two implementations:

|  |  |  |
| --- | --- | --- |
| Computation Times in seconds | Task 1 | Task 2 |
| n-20\_m-5 | 0,145 | 0,027 |
| n-35\_m-7 | 0,326 | 0,034 |
| n-50\_m-10 | 4,228 | 0,087 |

Table 1: comparison of computation time

So we can conclude that the implementation is by far the more efficient way of solving this problem. The difference between the two implementation gets higher the bigger the given instances.