JOHANNES KEPLER
UNIVERSITY LINZ

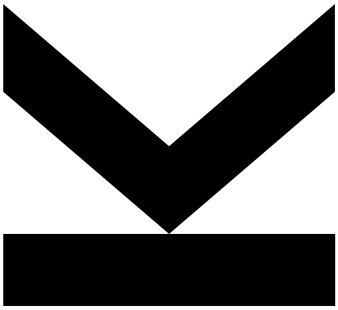
Authors
Dornetshumer Simon
Pammer Thomas

Programming for Business tasks

Nicolas Forget

07 2024

PROGRAMMING FOR BUSINESS TASKS – FINAL PROJECT - REPORT



JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at
DVR 0093696

TABLE OF CONTENTS

1. 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:

1.1. 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.

1.2. 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.

2. 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.

2.1. 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 times 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.

2.2. 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 relates to 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 find_all_tours function to receive the subtours and is also programmed in it as it adapts the used vectors dynamically.

2.3. 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 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 to create constraint 4 which is done in the connect_solution function.

2.4. 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 variable 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 the 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.

3. Task2.jl

As in Task 1, the CPLEX and JuMP packages are also required for Task 2. Additionally, the read_instance file is necessary to run the code without errors. This task is divided into two subtasks. The first step involves creating a new distance matrix that contains the shortest distances between each pair of delivery locations. In the second step, we will perform a cutting plane approach using the formulation with the connectivity cuts. For the first step, we can reuse some functions from the 10th lesson of the Shortest Path Problems, including the three structures: Graph, ShortestPathProblem, and Label.

3.1. findNextNode

The findNextNode function from the shortestPath lecture file can be reused entirely for this task.

3.2. hasPath

The hasPath function from the shortestPath lecture file can be reused entirely for this task, as well.

3.3. DijsktraAlgorithm

The DijkstraAlgorithm function can be reused, but a small change must be made. Instead of using a function to retrieve an optimal solution as in the lecture, it will simply return the minimal distance from traveling between two locations.

3.4. computeShortestPath

The inputs for this function are a Graph object, the indices of the delivery locations, and the number of delivery locations. In the computeShortestPath function, an m x m distance matrix will be created and filled with the shortest distances between each pair of delivery locations. An m x m matrix with zeros is initialized at the beginning of the function. This matrix is then populated with the shortest distances between each pair of delivery locations, except for the locations themselves, using Dijkstra's algorithm. Once the first step of this task is done the function returns the new distance matrix.

3.5. buildTSPmodel

The buildTSPmodel function, which is the first function of the second step, is reused entirely from our second homework.

3.6. addConnectivityCut

The addConnectivityCut function is also reused entirely from our second homework.

3.7. connectivityCutsAlgorithm

This function can be reused almost entirely from the second homework, but with one slight change: only the objective value will be printed, whereas in the homework, the optimal solution of the delivery locations is also printed. After implementing all the necessary functions for this task, we can run them to obtain the objective value for the given instances.

4. Task3.jl

In task 3 we compare 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.

4.1. Calc_comp_time_task1/2

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 divide 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 of task 2 is by far the more efficient way of solving this problem. The difference between the two implementations gets higher the bigger the given instances.

5. Task4.jl

In task 4 the requirement is to adapt the solution of task 2 so that is usable for a company to also obtain the optimal solution itself. That's why the project contains a folder company with the file task4.jl. Many functions and all structures and packages of task 2 can be reused to solve task 4.

5.1. findNextNode

This function can be totally reused from Task 2.

5.2. hasPath

The hasPath function from task 2 can be reused entirely for this task, as well.

5.3. DijsktraAlgorithm

The DijsktraAlgorithm function must be changed a little bit to solve the new task, because here it is necessary to compute the shortest tour between the two delivery locations given as input parameters. First, an undefined integer vector named shortestPath must be initialized to store the tour. Then the variable node must be set to the destinationNode to get the tour backwards, because with the Label structure we can iteratively push the origin node into the tour. If this is done the tour just has to be reverted and returned together with the distance.

5.4. computeShortestPath

In this function, in addition to task 2, an undefined vector of vectors had to be created in which all shortest tours for each combination of delivery locations are stored and returned.

5.5. buildTSPmodel

Nothing must be changed in this function regarding to task 2.

5.6. addConnectivityCut

Nothing must be changed in this function regarding to task 2.

5.7. connectivityCutsAlgorithm

In the connectivityCutsAlgorithm function, only the desired paths now had to be joined together. This was done as follows: first the starting location was inserted into the tour and then it was checked which of the shortest paths started with current last node at the tour and ended with the next delivery location. If this was the case, this shortest path was added to the tour. The next delivery location was known, because the order of the delivery locations was calculated via the code of the second homework. Then only the tour was output together with the objective value and the solution for the company was completed.

5.8. computeTour

Nothing must be changed in this function regarding to task 2. In the end we have again all necessary functions implemented, and the optimal solution and the objective value can be obtained for the company.