Bilkent University

Department of Computer Engineering

IE 400

Principles of Engineering Management

Final Report

Muhammad Umair Ahmed - 21600539

Elif Ozcelik - 21703185

Serdar Demirkol - 21001489

Instructor: Arnab Basu

Teaching Assistant: Mücahit Aygün

May 11, 2020

Table of Contents

1. Introduction

The objective of this report is to find the minimum time to collect the homeworks from a number of students and deliver them back to the teacher by choosing the shortest path from a given set point. So our problem is a travelling salesman problem which is very common in Operations Research. It is a binary Integer Programming problem. The basic problem is that a salesman has to travel to certain cities by going through the shortest path possible and visiting every city once. The decision variable Y_{ij} is a binary variable. Value 1 for Y_{ij} means the path from "*i*" to "*j*" is selected and 0 means it is not selected.

2. Modelling Integer Programming

Basis of the problem is travelling salesman with some changes. Solving times for students is additional to the original travelling salesman problem. Apart from that, the general structure of the problem is the same: Finding the optimal path to visit each node/city/person only once and returning back to the source. Since we have N students to visit and we start and end at the professor, we assigned number 0 to the professor and each student will have a number ranging from 1 to N denoted with i. X_i is given in the question, that is solving time for student i. T_{ij} is the travelling time of the assistant from student i to student j. Y_{ij} is a boolean value, representing if the assistant goes from student i to student j or not. X_i's are greater or equal to zero and T_{ij}'s are greater than zero.

Objective function is: **min** $\left(\sum_{i=0}^{N} \sum_{j=0}^{N} T_{ij} Y_{ij} + \sum_{i=1}^{N} X_i \right)$

Y_{ij} = 1, if assistant goes from i to j

0, otherwise

$\forall X_i$, i = 1,...,N

$\forall T_{ij}$, i = 0,...,N , j = 0,...,N

Constraints are:

$$X_i \geq 0$$

$$T_{ij} > 0$$

$$\sum_{j=1}^{n} x_{ij} = 1, \qquad i = 1,\ldots,n$$

$$\sum_{i=1}^{n} x_{ij} = 1, \qquad j = 1,\ldots,n$$

$$x_{ii} = 0, \qquad i = 1,\ldots,n$$

$$x_{ij} \in \{0,1\}, \qquad \forall\, i,j = 1,\ldots,n$$

This formula is the basis for our program and we made changes to it to fit in our problem.

3. Modelling Dynamic Programming

Our path network between the assistant and the students represents the possible connections between decisions and expresses time of going from one to another. The model encompasses a value-iterative algorithm which successively converges to the solution. The principle of optimality, which permits finding the minimum (shortest) path as a function of the maximum number of arcs allowed, is: regardless of how a particular state was arrived at, the remaining decisions as to what path to take to the terminal must constitute an optimal solution. The time of going on the paths may be input. Dynamic model flexibility allows for testing the sensitivity of a decision process to changes in the terminal point.

4. Data Generation

This part is done by generating X_i values for each student as $X_i \in [300, 500]$ once and then, generating T_{ij} values in range of $[100, 300]$ and generating $Nln(N)$ times and then taking average. But, for integer programming, if the values we get are decimal numbers after taking the average; we round up or down based on the value of the decimal part. To be specific: we rounded up .5 and bigger and rounded down lesser numbers. These generated values will be used for both Integer Programming and Dynamic Programming parts.

5. Solving via Integer Programming

In this section, how's and why's of the code will be explained. Code itself has self-explanatory comments in it but this report can be used as an additional referencing environment. We begin by generating data that will be used for both IP and DP parts. That data will be presented in a separate file. For solving the problem, we used "Miller - Tucker - Zemlin Formulation" as a base and adapted it to our problem. For the most part, we did not need an outside source but we used its subtour elimination concept.

```
"mdl.add_constraints((mdl.u_vars[student_i] - mdl.u_vars[student_j] +
num_of_nodes * mdl.order_vars[student_i, student_j] <=
num_of_nodes-1)

                        for student_i in student_nums_list for
student_j in student_nums_list if student_i != student_j and
student_i >= 1 and student_j >= 1)

    mdl.add_constraints((mdl.u_vars[student_i] >= 0)

                        for student_i in student_nums_list if
student_i >= 1)

    mdl.add_constraints((mdl.u_vars[student_i] <= (num_of_nodes-1))
```

```
                    for student_i in student_nums_list if
student_i >= 1) ”
```

These code lines are equivalent of :

$$u_i - u_j + nx_{ij} \leq n - 1 \quad 2 \leq i \neq j \leq n$$

$$0 \leq u_i \leq n - 1 \quad 2 \leq i \leq n$$

1. This formula is taken from Miller-Tucker-Zemlin formulation

6. Solving via Dynamic Programming

For dynamic programming, the basic concept is relatively similar. We take any arbitrary node(student) to be the next destination and deduct the selected node from the set. By continuing forward this way until no node is left in the set we get a solution for the problem but we have no assurance that this is the optimal solution. To avoid this nondeterminism, we make recursive calls each time we select a node with another student. That is, we make separate k calls to choose k different nodes to be the next destination and compare their results with each other and only return the smallest(optimal) solution to the previous call. Actual formulation is:

$$g(i,s) = \min(w(i,j) + g(j, \{s-j\})) \quad j \in s$$

Where *"i"* is the starting node and *"s"* is the set of all non-visited nodes. Function *'g'* is a DP function that is called recursively. Function *'w'* is the time to go from node *"i"* to node *"j"*. What this algorithm does is: After selecting a destination node it keeps on selecting from unvisited nodes set in each iteration until no node is left to select and adds $w(i,0)$ where *"i"* is the last visited node and 0 is the number of the professor. After each call completes the set and generates a solution. Then we return the value from each call and compare them with each other. For example

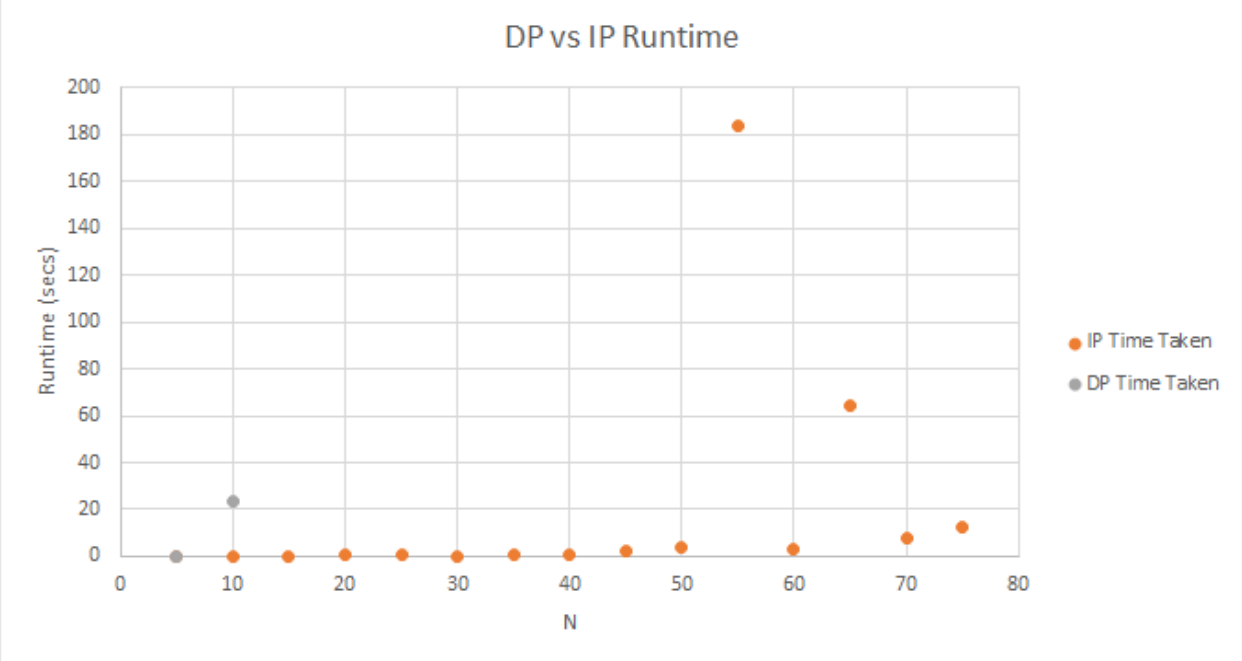
N^{th} step of the function is going from the last node to the professor. We take that $w(i,0)$ value and return it. Since 2 recursive calls were made in the previous step, we compare them and take the minimum to send to one step above. This way

we make sure only the optimal solution is returned. Even if there is more than one optimal solution, our program guarantees to return one. Which again is an optimal solution.

7. Discussion of Runtime vs N

Our code generates random student homework times once for 75 students (+ a 0 for the professor) and travel times for N = 5 to N = 75 by averaging N(ln N) random instances for each N as outlined in the project assignment document.

We then apply our CPLEX-based integer programming function [**data_generator.py::solve_problem()**] for each of the 15 applicable values of N and the dynamic programming-based function [data_generator.py::dp_wrapper()] to solve the problem for the same travel time dataset and student homework time dataset in order to get the optimal path time cost and the path itself from the data. Since a new dataset is generated every time the program is run.



As you can see the integer programming solution takes a few seconds for each instance of N with some abnormal values. This is an indication that the CPLEX library is very efficiently designed and can be used to solve such problems. The

continued recurrence of the abnormality suggests that it may be linked to the implementation of the library somehow. Apart from the abnormalities, the trend remains uniform.

We were unable to go beyond N=10 with dynamic programming without timing out of our 1 hr timeout limit. This makes sense as our dynamic programming solution is in fact a factorial time (O(n!)) solution that employs no memoization or lookup tables in order to eliminate the compute overhead of calculating repeated recursive calls for the same parameters via different branches. If we had used memoization or a lookup table for current best path value, the result would have been better.

8. About our Code

The code for our project is attached with this report and also available at: https://github.com/ahmed-umair/ie400-project

Our python file data_generator.py can be run by first installing the the packages in requirements.tx using pip and then calling data_generator.py for the upper limit of N e.g.

python3 data_generator.py 75

The relevant results are printed out one by one on the screen but are available in organized form in ./generated_data/*.xls files. The files available are as follows:

- ip_time_results.xls: These contain the optimal values, the optimal path (list of Xij's = 1 i.e. chosen arcs), the time taken to calculate that solution for each value of N starting from 5 to the specified command line argument.
- dp_time_results.xls:These contain the optimal values, the optimal path (illustrated, two-way), the time taken to calculate that solution for each value of N starting from 5 to the specified command line argument.

- student_times.xls: These contain the homework times generated for this run of the program for 75 students + 1 professor (=0)
- travel_times_*.xls: These contain the random averaged travel times for arcs generated for each value of N starting from 5 to the given command line argument.