CSE 6140 / CX 4140 Assignment 2
due Oct 2, 2020 at 11:59pm on Canvas

Ting Liao GTID: 903278773
(collaborator: Chin Wang GTID: 903506612 / Shen-Yi Cheng GTID: 903514405)

10/2/2020

Please upload two files:

1. a single PDF named `assignment.pdf` containing your solutions for Problems 1 and 2; and a report for Problem 3.
2. a single zip file named `code.zip` containing your code, README, and results for Problem 3

1 Dynamic Programming: Atlanta MARTA [13 pts]

The Metropolitan Atlanta Rapid Transit Authority (MARTA) is the principal public transportoperator in the Atlanta metropolitan area. It was Formed in 1971 as strictly a bus system, and today, it is transporting almost 450,000 passengers a day (bus and train). Currently, MARTA Passes are the cheapest option for those who regularly use MARTA for transportation. Assume the MARTA Passes are sold in three following forms:

- Daily: A 1-day pass sold for $tickets[0]$ dollars;
- Weekly: A 7-day pass sold for $tickets[1]$ dollars;
- Monthly: A 30-day pass sold for $tickets[2]$ dollars.

For instance, $tickets = \{2, 7, 20\}$ means we need to pay \$2, \$7, and \$20 for each daily, weekly,and monthly pass, respectively. A ticket cost is always a positive value. Therefore, $tickets[i] > 0, i \in \{1, 2, 3\}$. The passes allow consecutive days of travel. For example, if we get a weekly pass on day 5, then we can travel for 7 consecutive days which are: day 5, 6, 7, 8, 9,10, and 11. George P. Burdell is a student at Georgia Tech and he has already organized his commuting plan for the upcoming year. In his plan, each day of year is specified by an integer identification number from 1 to 365. Therefore, he can represent his commuting plan as an array of integers. For instance, $days = \{8, 9, 10, 11, 14, 17, 18, ...\}$ means George needs to commute to the school on the 8th, 9th, 10th, 11th, ... days of year. He asked you to help him find the minimum amount of money that he should spend to purchase MARTA Passes for commuting to school in the next year.

a) (3 pts) Prove the optimal substructure.
For Optimal Substructure:

Subproblem (S) is that which of the following is cheapest:
1. Buy 1-pass : cost 2 dollars
2. Buy 7-pass : cost 7 dollars
3. Buy 30-pass : cost 20 dollars

By comparing these three costs, we could get the minimum cost. So now we can solve for
S x n problems.

So, we can divide the subproblem into three cases:
Case 1: OPT select ticket 1-pass
Case 2: OPT select ticket 7-pass
Case 3: OPT select ticket 30-pass

Then, the optimal substructure:

Case 1: OPT(i-1) + 1-pass.price

Case 2: OPT(max(i-7, 0)) + 7-pass.price

Case 3: OPT(max(i-30, 0)) + 30-pass.price

Proof by induction:

True that when i = 0, OPT(i) = 0
Assume all true for j ¡ I
By induction we know that OPT(i) is correct.

For OPT(i+1) = (min(OPT(i) + 1-pass.price,
OPT(max((i+1)-7, 0)) + 7-pass.price,
OPT(max((i+1)-30, 0)) + 30-pass.price)

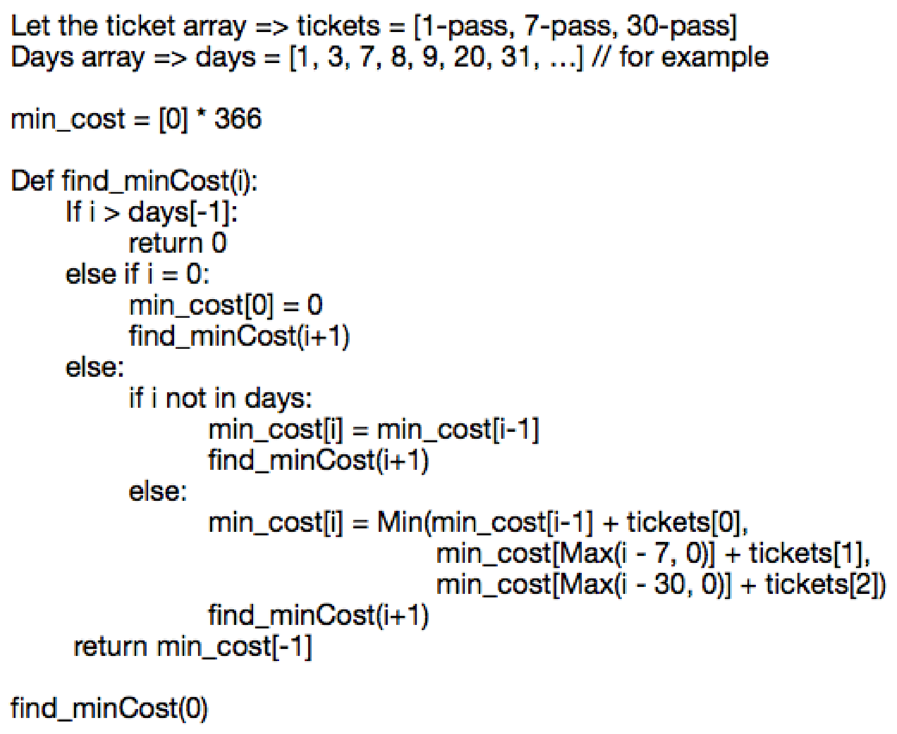
Therefore, OPT(i+1) is still optimal. Our Algorithm is proved to be optimal.

Basically, while checking for every day commuting, we could always execute this subproblem to find the minimum cost of that day. For example, if we have to commute on days [1, 4, 6, 7], we could solve for the subproblem on each day. First, on the first day, buying 1-pass will cost 2 dollars, 7-pass will cost 7 dollars and 30-pass will cost 20 dollars. It seems that buying 1-pass will be the cheapest option. However, after we repeat the process for every commuting day, we will eventually get the result that on the last day, buying 1-pass will cost 8 dollars, 7-pass will cost 7 dollars and 30-pass will cost 20 dollars. Therefore, we can get the optimal solution by using this optimal substructure and DP algorithm.

b) (4 pts) Write a recursive expression for calculating min Cost including the base case.
Recurrence relation:

OPT(i) =
(0 if i = 0
(min(OPT(i-1) + 1-pass.price,
OPT(max(i-7, 0)) + 7-pass.price,
OPT(max(i-30, 0)) + 30-pass.price) otherwise)

The following is the pseudo code by using the recurrence relation idea:

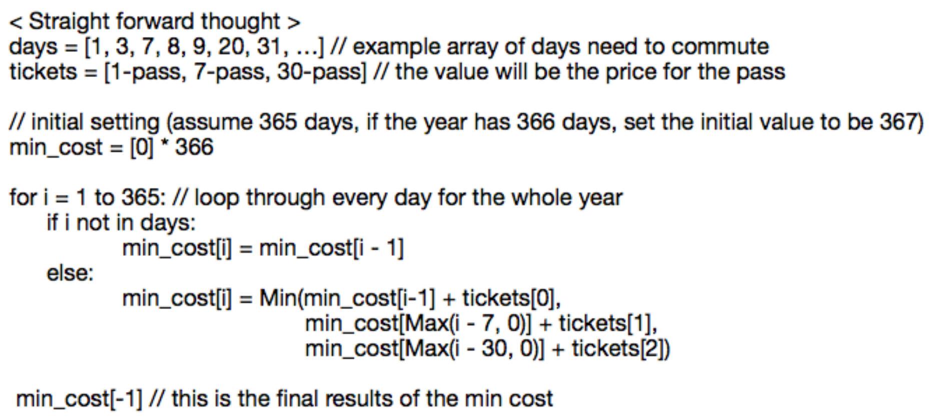


```
Let the ticket array => tickets = [1-pass, 7-pass, 30-pass]
Days array => days = [1, 3, 7, 8, 9, 20, 31, ...] // for example

min_cost = [0] * 366

Def find_minCost(i):
    If i > days[-1]:
        return 0
    else if i = 0:
        min_cost[0] = 0
        find_minCost(i+1)
    else:
        if i not in days:
            min_cost[i] = min_cost[i-1]
            find_minCost(i+1)
        else:
            min_cost[i] = Min(min_cost[i-1] + tickets[0],
                              min_cost[Max(i - 7, 0)] + tickets[1],
                              min_cost[Max(i - 30, 0)] + tickets[2])
            find_minCost(i+1)
    return min_cost[-1]

find_minCost(0)
```

(a)

Figure 1: Pseudo code for using recurrence relation

c) (6 pts) Give the pseudocode of a linear Dynamic Programming algorithm to return the minimum cost of commuting every day in the array "days", if the cost of MARTA passes is given in a three-element array "tickets". Analyze the space and time complexity of your algorithm.



```
< Straight forward thought >
days = [1, 3, 7, 8, 9, 20, 31, ...] // example array of days need to commute
tickets = [1-pass, 7-pass, 30-pass] // the value will be the price for the pass

// initial setting (assume 365 days, if the year has 366 days, set the initial value to be 367)
min_cost = [0] * 366

for i = 1 to 365: // loop through every day for the whole year
    if i not in days:
        min_cost[i] = min_cost[i - 1]
    else:
        min_cost[i] = Min(min_cost[i-1] + tickets[0],
                          min_cost[Max(i - 7, 0)] + tickets[1],
                          min_cost[Max(i - 30, 0)] + tickets[2])

min_cost[-1] // this is the final results of the min cost
```

(a) Min cost of commuting

Figure 2: Pseudo code for a linear Dynamic Programming algorithm

Time complexity is O(N)
Space complexity is O(N)

2 Dynamic Programming: Buy More [12 pts]

Chuck is the new manager of the local Buy More, a consumer-electronics store whose main revenue source comes from selling computers. Chuck has accurate predictions of the quantity of computer sales in the next n months, where d_i denotes the number of sales in month i. Assume that sales occur at the beginning of each month, with unsold computers stocked in inventory until the beginning of the next month. It costs C to keep a single computer in stock for a month. The Buy More's inventory can keep up to I computers in stock. Each month Chuck can choose to order any number of computers (which conveniently arrive before that month's sales), but can only keep up to I computers in stock at the end of the month. Each time that Chuck submits an order, there is a fixed shipment cost of R. Chuck currently has no computers in inventory. Help Chuck design an algorithm that is polynomial in n and I to meet the n monthly demands (d_i), whilst minimizing the cost.

Note that the cost consists of both the fixed cost R for submitting an order and the marginal cost C for each computer kept in inventory per month. For every month i, the demand d_i must be met. No more than I computers may be left in inventory at the end of the month.

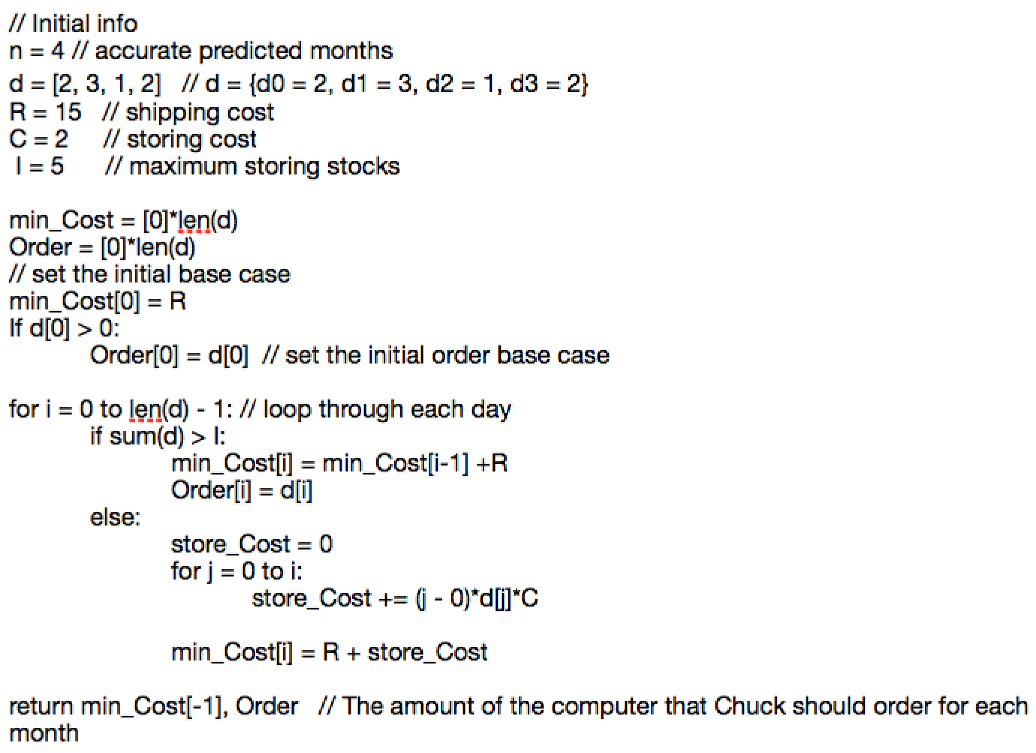
a) (4 pts) Give the recurrence relations including base cases.

Base case: OPT(0) should only have the shipping cost R

Case 1: OPT(i) =¿ including day i
Case 2: OPT(i-1) + R =¿ not including day i, day i will need to pay for the shipping fee R

So, OPT(i) =
(R if i = 0
min(OPT(i), OPT(i-1) + R) otherwise)

b) (8 pts) Give the pseudocode of a bottom-up approach to implement your dynamic programming algorithm. Also include the pseudocode to find the optimal solution which tells Chuck how many computers he should order each month. Analyze the time and space complexity of your complete algorithm.



```
// Initial info
n = 4 // accurate predicted months
d = [2, 3, 1, 2]   // d = {d0 = 2, d1 = 3, d2 = 1, d3 = 2}
R = 15   // shipping cost
C = 2     // storing cost
 I = 5     // maximum storing stocks

min_Cost = [0]*len(d)
Order = [0]*len(d)
// set the initial base case
min_Cost[0] = R
If d[0] > 0:
        Order[0] = d[0]  // set the initial order base case

for i = 0 to len(d) - 1: // loop through each day
        if sum(d) > I:
                min_Cost[i] = min_Cost[i-1] +R
                Order[i] = d[i]
        else:
                store_Cost = 0
                for j = 0 to i:
                        store_Cost += (j - 0)*d[j]*C

                min_Cost[i] = R + store_Cost

return min_Cost[-1], Order   // The amount of the computer that Chuck should order for each
month
```

(a)

Figure 3: Pseudo code for Min Cost of Buy More

2.1 examples

$$n = 4, \quad \{d_0 = 2, d_1 = 3, d_2 = 1, d_3 = 2\}, \quad R = 15, \quad C = 2, \quad I = 5$$

- An order for 8 computers cannot be placed at the beginning of month 0 because this would result in $8 - d_0 = 6$ computers leftover at the end of month 0, which exceeds the inventory limit I.
- If an order size of 2 is placed for month 0, 3 for month 1, 1 for month 2 and 2 for month 3, the total cost will be: $R + R + R + R = 15 + 15 + 15 + 15 = 60$
- If orders are placed of size 3 for month 0, 4 for month 1, 0 for month 2 and 1 for month 3, the total cost will be: $R + 1C + R + 2C + 1C + R = 15 + 2 + 15 + 4 + 2 + 15 = 53$

3 Programming Assignment [25 pts]

You are to implement *either* Prim's or Kruskal's algorithm for finding a Minimum Spanning Tree (MST) of an undirected multi-graph, and evaluate its running time performance on a set of graph instances. The 13 input graphs are RMAT graphs [1], which are synthetic graphs with power-law degree distributions and small-world characteristics. Please note that, there might be multiple edges between vertices, you just treat them like they are different edges (i.e., do not sum up the weights, or randomly pick one edge).

3.1 Static Computation

The first part of the assignment entails coding either Prim's or Kruskal's algorithm to find the cost of an MST given a graph file. You may use the programming language of your choice (C/C++, Java or Python).

We provide a wrapper function in all three languages to help you get started with the assignment. You may call your own functions inside the wrapper. We also have implemented a timer in the wrapper that records the running time of your algorithms. To implement these algorithms, you may make use of data structure implementations in the programming language of your choice; e.g. in python, the heapq library may be used for implementing priority queues and set operations may be used for implementing the union-find data structure. In Java, java.util.PriorityQueue may be used for implementing priority queues and java.util.Set may be used for set operations.

The 'graph file' format is as follows:

> Line 1: N E (N = number of vertices, E = number of edges)
>
> Every subsequent line contains three integers: u v weight (u &v are end points of edge, weight = weight of edge between u and v. Please note each undirected edge is only listed once in the file.)

The MST calculation is to be implemented in the `computeMST` function, as indicated in the wrapper code.

3.2 Dynamic Recomputation

The next part of the assignment requires you to update the cost of the MST given new edges to be added to the graph. You are provided with a 'changes file' and the format is as follows:

> Line 1: N (N = number of changes/edges to be added)
>
> Every subsequent line contains three integers: u v weight (u & v are end points of the new edge to be added, weight = weight of edge between u and v)

You are to implement the function `recomputeMST` as indicated in the wrapper code that computes the new MST given the new edge to be added into the graph. You are responsible for maintaining the old MST before recomputing.

Note: it is very easy to complete this part of the assignment by simply adding the new edge and calling your old `computeMST` function from part 1 (Subsection 3.1). The objective is to minimize computation and efficiently recompute the cost of the MST.

3.3 Execution

The wrapper code is set up to require three command line arguments: <executable> <graph_file.gr> <change_file.extra> <output_file>. The <graph_file> is the one described in Subsection 3.1 and the <change_file> is the one described in Subsection 3.2.

3.4 Experiments

You are required to run your code for all 13 input graphs provided. The wrapper functions we provide in C++, Java, and Python describe the following procedure. For each graph:

- parse the edges (`parseEdges`): **to be implemented**
- compute the MST using either Prim's or Kruskal's algorithm (`computeMST`): **to be implemented**
- write the cost of the initial MST and time taken to compute it to the output file: **provided in wrapper code**
- For each line in the <change_file>,
  - Parse the new edge to be added: **provided in wrapper code**
  - Call the function `recomputeMST`: **to be implemented**

- Write to the output file the cost of the new MST and the time taken to compute it: **provided in wrapper code**
- Note: Each new edge should be added cumulatively, such that in the output file, the cost of MST in line j should never be more than the cost of MST in line i for $j > i$.

Name the output files as such: <graph_file>_output.txt and place them in the folder *results*.

3.5 Report

Write a brief report wherein you:

a) List which data structures you have used for your choice of algorithm (Prim's/Kruskal's). Explain the reasoning behind your choice and how that has influenced the running time of your algorithms, and the theoretical complexity (i.e., specify the big-Oh for your implementation of each of the two algorithms - computeMST and recomputeMST).

b) Plots: Plot the running time as the number of edges in the graph increases (across the 13 graphs you were given) for both the static and dynamic calculations, i.e. one plot showing on the x-axis the number of edges the graph has and the y-axis the time for the static MST calculation, and another plot where the y-xis the time needed to insert 1000 edges (as given the changes files) using your recomputeMST code. Discuss the results you observe. How does the empirical scaling you observe match your big-Oh analysis? How does the behavior vary with the dynamic recomputation?

3.6 Deliverables

- code for initial, static MST implementation
- code for dynamic MST recomputation
- README, explaining how to run your code
- Report (Subsection 3.5)
- output files (13) within the *results* folder- one for each graph

References

[1] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *Proc. 4th SIAM Intl. Conf. on Data Mining,* Florida, USA, April 2004.