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

Shasha Liao

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. 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.

Solution:
Goal: calculate the minimum amount of money that George should spend to purchase MARTA Passes according to his commuting plan for $n = 365$ days in the next year. Denote that value by $OPT(n)$.
Define subproblem: calculate the minimum amount of money he should spend by optimal solution for the first j days in the next year $OPT(j)$.

Consider an optimal solution O for the days $\{1, 2, ..., j\}$. On day j, there are two possible cases for George's commuting plan.

- **Case 1**: George dose not plan to commute on day j. Then $OPT(j) = OPT(j-1)$
- **Case 2**: George plans to commute on day j. Then he has three options:
  - **Option 1**: George uses a daily pass to take MARTA on day j.
  - **Option 2**: George uses a weekly pass (expiring on day j) to take MARTA on day j.
  - **Option 3**: George uses a monthly pass (expiring on day j) to take MARTA on day j.

  Now we are going to show that for each option we can calculate $OPT(j)$ from $OPT(k)$ where $k < j$.

  - **Option 1**: $OPT(j) = OPT(j-1) + tickets[0]$ since George has to purchase a daily pass on day j.
  - **Option 2**: $OPT(j) = OPT(j-7) + tickets[1]$ since George has to purchase a weekly pass on day $j - 6$.
  - **Option 3**: $OPT(j) = OPT(j-30) + tickets[2]$ since George has to purchase a monthly pass on day $j - 29$.

  So in this case,

  $$OPT(j) = \min\{OPT(j-1) + tickets[0], OPT(j-7) + tickets[1], OPT(j-30) + tickets[2]\}.$$

So we will have the recursive relation presented in part b). Now we prove that this algorithm computes correctly the optimal value for each $j = 0, 1, 2, ..., n$ by induction.

1) Base case (j = 0), $OPT(0) = 0$. True.
2) Induction hypothesis: assume $OPT(i)$ is optimal for all $i < j$.
3) On day j, there are two cases:
   - **Case 1**: $j \notin days$. OPT(j-1) is the minimum amount of money for commuting on days $1, 2, ..., j$. If not, we could take the optimal solution for days $1, 2, ..., j$ and safely apply it to days $1, 2, ..., j-1$, which gives a lower than optimal cost for days $1, 2, ..., j-1$. This contradicts with the induction hypothesis that $OPT(j-1)$ is optimal.
   - **Case 2**: $j \in days$. Similarly, we can use the same exchange argument to prove that for each of the three options listed above, $OPT(j)$ defined above is the minimum amount of cost for days $1, 2, ..., j$.

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

Solution: To simplify notation, we denote $f(j) = OPT(j)$, $t_1 = tickets[0]$, $t_7 = tickets[1]$, and $t_{30} = tickets[2]$.

$$f(j) = \begin{cases} 0 & \text{if } j <= 0, \\ \min\{f(j-1)+t_1, f(j-7)+t_7, f(j-30)+t_{30}\} & \text{if } j \in days \text{ and } j \geq 1, \\ f(j-1) & \text{if } j \notin days \text{ and } j \geq 1. \end{cases}$$

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.

The algorithm is of space complexity $O(n)$ where n is the number of days in the commuting plan. The time complexity is also of $O(n)$ since we calculate $Memo[i]$ once for each $i = 1, 2, ..., n$.

```
Memo = [0]*365
def OPT(i):
    # base case
    if i <= 0:
        return 0
    if Memo[i] == 0:
        if i not in days:
        # don't commute
            Memo[i] = OPT(i-1)
        else:
        # commute
            Memo[i] = min(OPT(i-1) + ticket[0], OPT(i-7) + ticket[1], OPT(i-30) +
                                                        ticket[2])
    return Memo[i]
```

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 K. 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.

Solution: The idea is to use two variables i and k to denote $opt(i,k)$ as the minimum cost for the first i months with k unsold computers stocked in inventory during the ith month. For $i = 1$, Chuck has no stocked computer and has to order some amount of computers with a fixed shipment cost of K. For $i > 1$, if $d[i] + k \leq I$, Chuck has two options. He can either order or not order computers in the ith month. However, if $d[i] + k > I$, Chuck has to place an order in the ith month.

$$opt(i,k) = \begin{cases} R & \text{if } i = 1, \\ \min(opt(i-1, d[i]+k) + (d[i]+k) * C, opt(i-1,0) + R) & \text{if } d[i]+k \leq I \text{ and } i \neq 1 \\ opt(i-1,0) + R & \text{if } d[i]+k > I \text{ and } i \neq 1 \end{cases}$$

for $i = 0, 1, 2, ..., n-1$ and $k = 0, 1, ..., I$.

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.

The following bottom-up dynamic programming algorithm has space and time complexity $O(n(I+1))$ since we used an array of size $(n, I+1)$ to store the results. Also, since when we compute the ith row in Memo, we only need the result from the $(i-1)$th row in Memo. So the space complexity can be reduced to $O(I)$.
The algorithm for finding the solution has space and time complexity $O(n)$ since we only need to track back for a plan of a total of n days.
Overall, the complete algorithm has space and time complexity $O(nI)$.

```
## bottom-up dynamic programming algorithm

n = len(d)
Memo = [[0 for _ in range(I+1)] for _ in range(n)]
# base case
for k in range(I+1):
    Memo[1][k] = R

# bottom up
for i in range(2, n):
    for k in range(I+1):
        if d[i] + k <= I:
        # two options available
            Memo[i][k] = min(Memo[i - 1][d[i] + k] + (d[i] + k) * C, Memo[i - 1][0] +
                                                                    R)
        else:
        # has to place an order
            Memo[i][k] = Memo[i - 1][0] + R

optimal = Memo[-1][0]
```

```
## find the optimal solution

buy_computer = [0]*n
def find_solution(i,k):
    # base case, place an order
    if i == 1:
        buy_computer[i] = d[i] + k
        return

    # place an order
    if Memo[i][k] == Memo[i - 1][0] + R:
        buy_computer[i] = d[i] + k
        find_solution(i-1,0)
    else:
    # don't place an order
        buy_computer[i] = 0
        find_solution(i-1, d[i] + k)

find_solution(n-1,0)
## Optimal solution: Chuck should order buy_computer[i] computers in the ith month.
```

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).

Data structure: there are two data structures, **vertex** object and **graph** object. These two data structure make it very cheap (constant time) and convenient to add, search, and remove edges in the graph. Suppose there are $|V|$ vertices and $|E|$ edges in a graph.

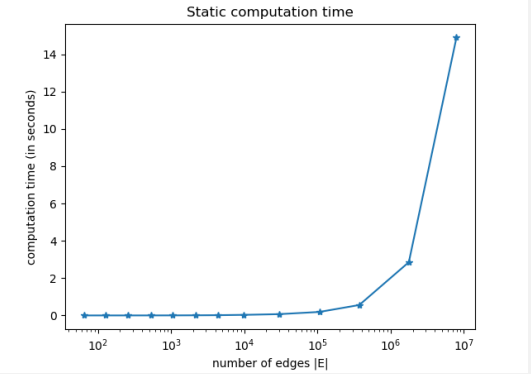
Runtime of computeMST: O($|V||E|\log(|E|)$).
Runtime of recomputeMST: O($|V|$).

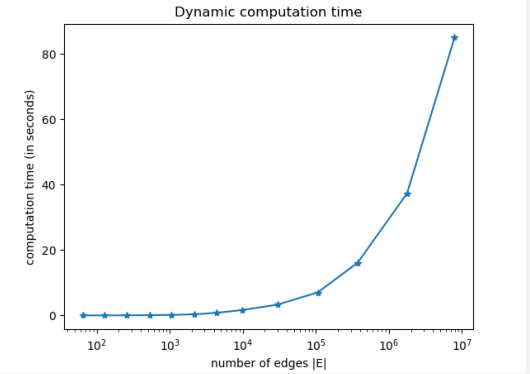
- **vertex**(*id*)
  Returns a new vertex object initialized with id *id*.
  Vertex objects support the following methods:
  - **add_neighbor**(*neighbor, weight*)
    Add an edge between **vertex**(*id*) and *neighbor* with weight *weight*. If there is an edge exists, only update the weight when the new weight *weight* is smaller. Runtime: O(1).
  - **remove_neighbor**(*neighbor*)
    Remove the edge between **vertex**(*id*) and *neighbor* if it exist. Runtime: O(1).
  - **is_connected**(*neighbor*)
    Reture True if there is an edge between **vertex**(*id*) and *neighbor*. Otherweise, return False. Runtime: O(1).
  - **get_connections**()
    Return a list of all the neighbors of **vertex**(*id*). Runtime: O($|V|$).
  - **get_weight**(*neighbor*)
    Return the weight between **vertex**(*id*) and *neighbor*. Runtime: O(1).
- **graph**()
  Returns a new graph object initialized with 0 vertex and 0 edge.
  Graph objects support the following methods:
  - **add_vertex**(*id*)
    Add **vertex**(*id*) to the graph. Runtime: O(1).
  - **get_vertex**(*id*)
    Return **vertex**(*id*) if it is in the graph. Otherwise, return None. Runtime: O(1).
  - **add_edge**(*from, to, weight*)
    Add an edge between **vertex**(*from*) and **vertex**(*to*) with weight *weight*. Runtime: O(1).
  - **remove_edge**(*from, to*)
    Remove the edge between **vertex**(*from*) and **vertex**(*to*) if it exists. Runtime: O(1).
  - **get_vertices**()
    Return a list of all the vertices in the graph. Runtime: O($|V|$).

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?

For computeMST, the theoretical worst time complexity is $O(|V||E|\log(|E|))$. The empirical computation times on the left plot shows that as we increase $|E|$ from 10^6 to 10^7, the computation time also increased to 10 times as before. However, we also doubled $|V|$ in this process. So the empirical computation time is slower than the theoretical computation time.

For recomputeMST, the theoretical worst time complexity is $O(|V|)$, independent of $|E|$. Since each time we doubled the size of $|V|$, the plot on the right shows that the runtime was also doubled. In this case, the empirical computation times reflected the theoretical computation complexity.





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.