CSE 6140 / CX 4140 Assignment 2

Qinyu Wang

1 Dynamic Programming: Atlanta MARTA

1) Goal: to find the minimum amount of money
   Let $dp(i)$ be the minimal cost to travel by Marta from day i to the end of travel plan.
   Base case: If $i = N, last\ day\ of\ the\ year, dp(i) = 0$
   Case 1: if i is not in the days array:
   $$dp(i) = dp(i+1) + 0$$
   Case 2: if i is in the days array:
   $dp(i)$ can be written as 3 conditions:
   $$dp(i) = \mathrm{d}p(i+1) + tickets[0]$$
   $$dp(i) = \mathrm{d}p(i+7) + tickets[1]$$
   $$dp(i) = \mathrm{d}p(i+30) + tickets[2]$$
   Thus, when $i = 365, dp(i) = 0$.
   Otherwise, $dp(i) = \min(dp(i+1) + tickets[0], dp(i+7) + tickets[1], dp(i+30) + tickets[2])$
2) $dp(i) = \begin{cases} 0 & i = N \\ \min(dp(i+1) + tickets[0], dp(i+7) + tickets[1], dp(i+30) + tickets[2]), & otherwise \end{cases}$
3) Time complexity: $O(N \times 3) = O(N)$. The algorithm loop through whole year N days, and calculate 3 times the travel plan in each iteration, $3N$. Hence, $O(N \times 3) \rightarrow O(N)$.
   Space complexity: $O(N)$. An array with length size $O(N)$ is needed to store as much as to store all the solutions

Require: i (day i to the end of the travel plan)
Ensure: $dp(1)$ (minimal cost for you to travel from day i to the end of the plan)

```
for i ← N to 1 do (N is last day travel)
    if i > 365 then
        return 0
    end if
    if memo[i] not empty then (dp(i) already calculated)
        return memo[i]
    else
        if i is in days array then
            dp(i) ← min (dp(i + 1) + ticket[0], dp(i + 7) + ticket[1], dp(i + 30) + ticket[2])
        else
            dp(i) = dp(i + 1)
        end if
        memo[i] = dp(i)
    end if
end for
return dp(1)
```

2 Dynamic Programming: Buy More

The monthly purchases are independent and non-overlapping events. Let $\{d_1, d_2, \ldots, d_n\}$ denotes the monthly demands; $\{p_1, p_2, \ldots, p_n\}$ denotes the monthly purchases.

1) Whether ith month place an order or not.

$$cost(0, i) = 0$$
$$cost(t, 0) = 0$$

Case 1: place an order

$$cost(t, i) = cost(t + 1, i + p_i - d_i) + C \times (i + p_i - d_i) + R$$

Case 2: no order

$$cost(t, i) = cost(t + 1, i - d_i) + C \times (i - d_i)$$

Thus,

$$cost(t, i) = \begin{cases} 0 & t = 0, i = 0 \\ cost(t + 1, j) + C \times j + R, & p \neq 0 \\ cost(t + 1, j) + C \times j, & p = 0 \end{cases}$$
$$(where\ j = i + p_i - d_i)$$

2)

Require: t month, i capacity, p monthly order, R delivery fee, C keep fee
Ensure: $cost(t, i)$ (minimal cost) $x(t, i)$ (optimal order)

```
for t ← n to 1 do (n is the last month in business)
    for i ← 0 to I do
        maxorder = I + d_t − i
        minorder = 2d_t − i
        for p ← minorder to maxorder do
            j = i + p_t − d_t (current storage)
            if p > 0 then (***recursion: whether ith month place an order)
                cost(t, i) = C × j + cost(t + 1, i) + R
                order(t, i) = (t + 1, p_t)
            else
                cost(t, i) = C × i + cost(t + 1, i + d_t)
                order(t, i) = (t + 1, 0)
            end if
            if C × j + cost(t + 1, i) + R > C × i + cost(t + 1, i + d_t) then
                mincost(t, i) = C × j + cost(t + 1, i) + R
                minorder(t, i) = p_t
            end if
        end for
    end for
    return mincost(t, i) = mincost
    return minorder(t, i) = minorder
```

1) Will this algorithm also work?

$OPT(i, j)$ is defined as the minimal cost in month i with leftover j.

Let k denotes the left over in $(i-1)$ month. Month {1, 2, 3, …, i, …, n}; Leftover {0, 1, …, j, …, I}

Base case: $OPT(1, j) = 0 + Cj + R$

Case 1: if $k < j + d_i$, ith month needs to place an order

$$OPT(i, j) = OPT(i-1, k) + Cj + R$$

Case 2: if $k = j + d_i$, ith month no order

$$OPT(i, j) = OPT(i-1, j+d_i) + Cj$$

Recurrence:

$$OPT(i, j) = \min(OPT(i-1, k) + Cj + R, OPT(i-1, j+d_i) + Cj)$$

2)

Require: t month, i capacity, p monthly order, R delivery fee, C keep fee
Ensure: $OPT(i,j)$ (minimal cost) $Traceback(i,j)$ (optimal order)

```
for j ← 0 to I do //n is the last month in business)
    OPT(1, j) = C × j + R //no stock
    Traceback(1, j) = (0, 0)
    for i ← 1 to n do
        for j ← 0 to I do
            if k < j + d_i then
                OPT(i, j) = OPT(i − 1, k) + C × j + R
                Traceback(i, j) = (i − 1, k)
            else
                OPT(i, j) = min(OPT(i−1, k)+C×j+R, OPT(i−1, j+d_i)+C×j)
                if OPT(i, j) = OPT(i − 1, k) + C × j + R then
                    Traceback(i, j) = (i − 1, k)
                else
                    Traceback(i, j) = (i − 1, i + d_i)
                end if
            end if
        end for
    end for
end for
return OPT(i, j), Traceback(i, j)
```

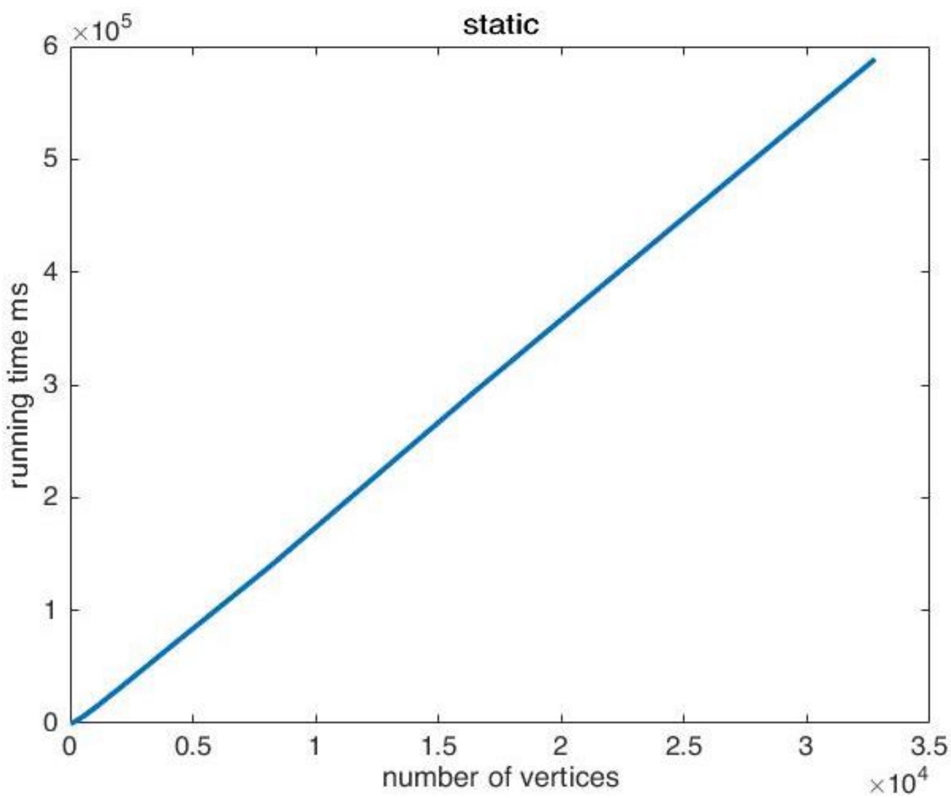
3 Programming Assignment

I implemented Kruskal's algorithm in designing undirected minimal spanning tree. Theoretically, for each graph G(N,E), after computing the MST, the newMST can be found by adding next lowest-weighted edge {e}.

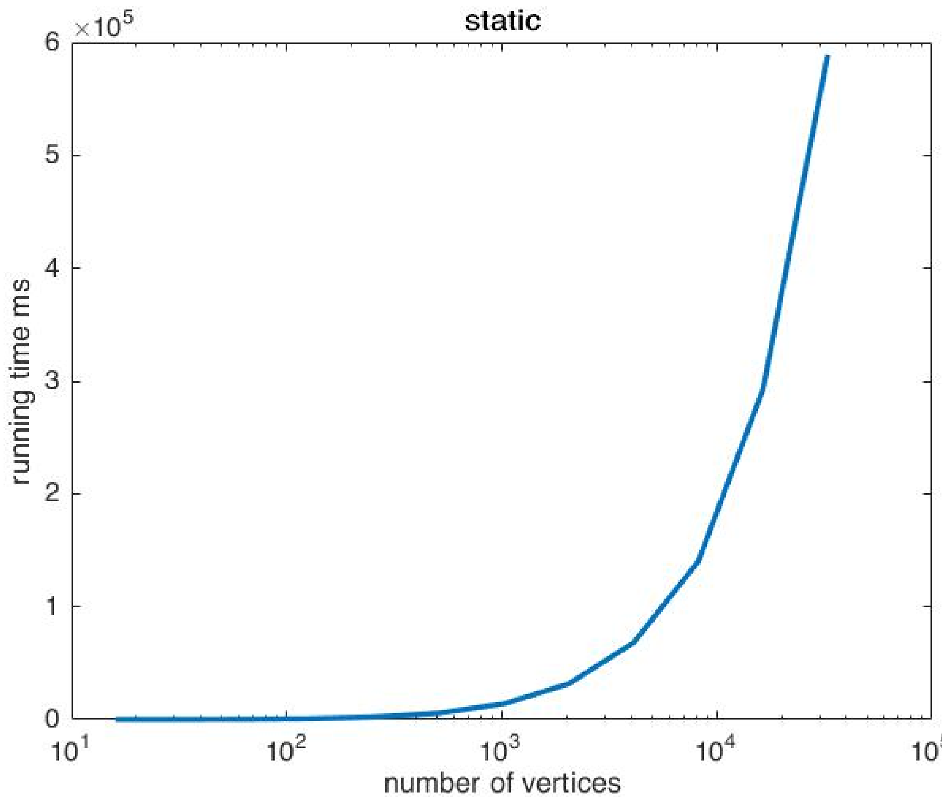
Basically, I used tuples to store the $\{u, v, w\}$ values, sort the edges in terms of ascending weights, and adds the next lowest-weight edge that will not form a cycle to the minimum spanning tree. For each edge, I used dictionary to store its root, and a recursion to assist finding the ultimate parent root. The union find algorithm runs in space complexity is $O(E)$, which is dominated by the number of vertices. And then, I attach the component having fewer nodes to the component having larger nodes in terms of rank. For a graph $G(N, E)$, parsing the graph runs in $O(E)$ time. However, the time complexity is largely depending on sorting, which runs in $O(E \log N)$ time or $O(N \log N)$ equivalently. As for recomputeMST function, I used .copy() to dynamically copy the edge_list and call the computeMST function to add next lowest-weighted edge and recompute the minimal spanning tree, which runs in $O(E)$ time. Hence, the whole algorithm run in $O(E + N \log N)$ time.

Choice:
Generally, prim's algorithm performs better in a dense graph. Kruskal performs better in sparse graphs and is easier to implement because it uses disjoint sets and simpler data structures. Thus, Kruskal is better in this case.

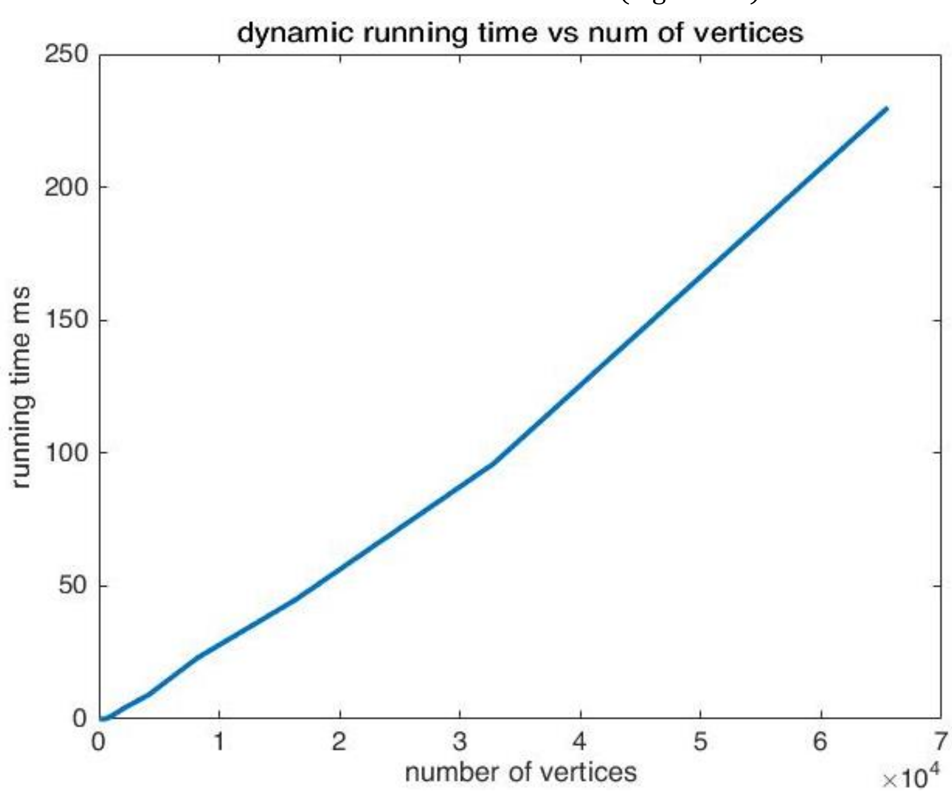
Running time is as follows:

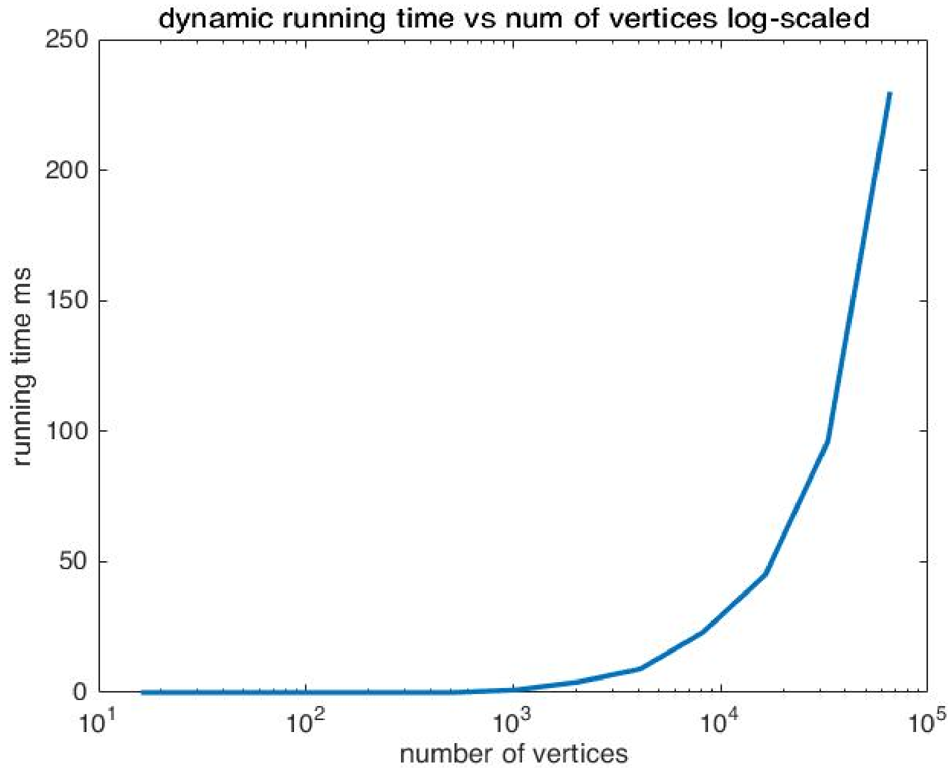


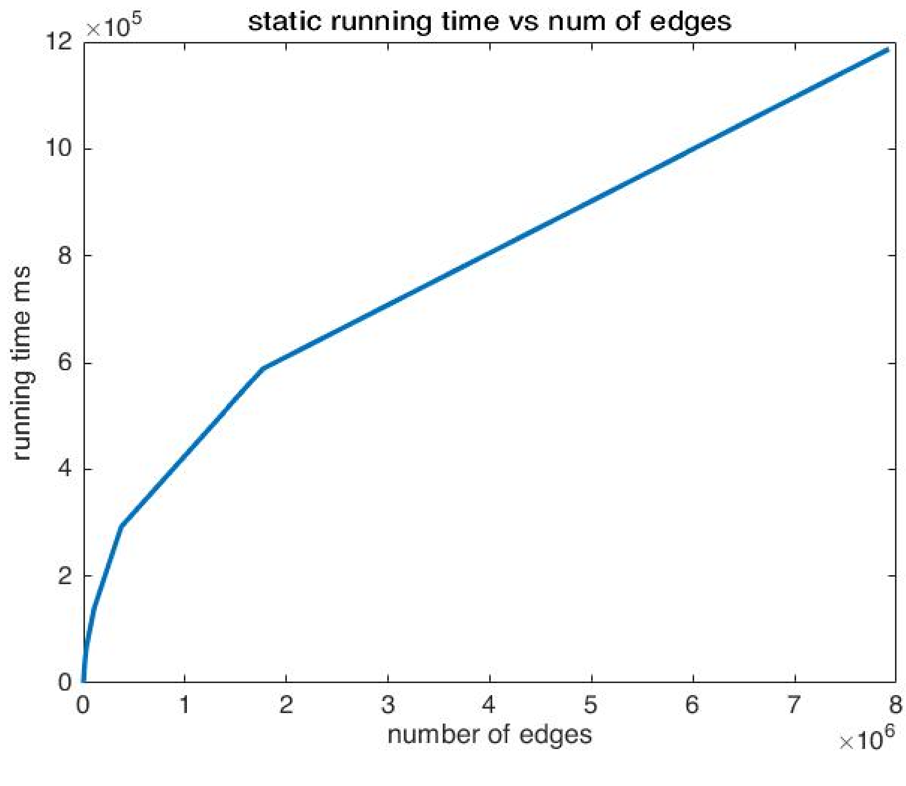
Plot of Time vs Number of vertices (normal scaled)

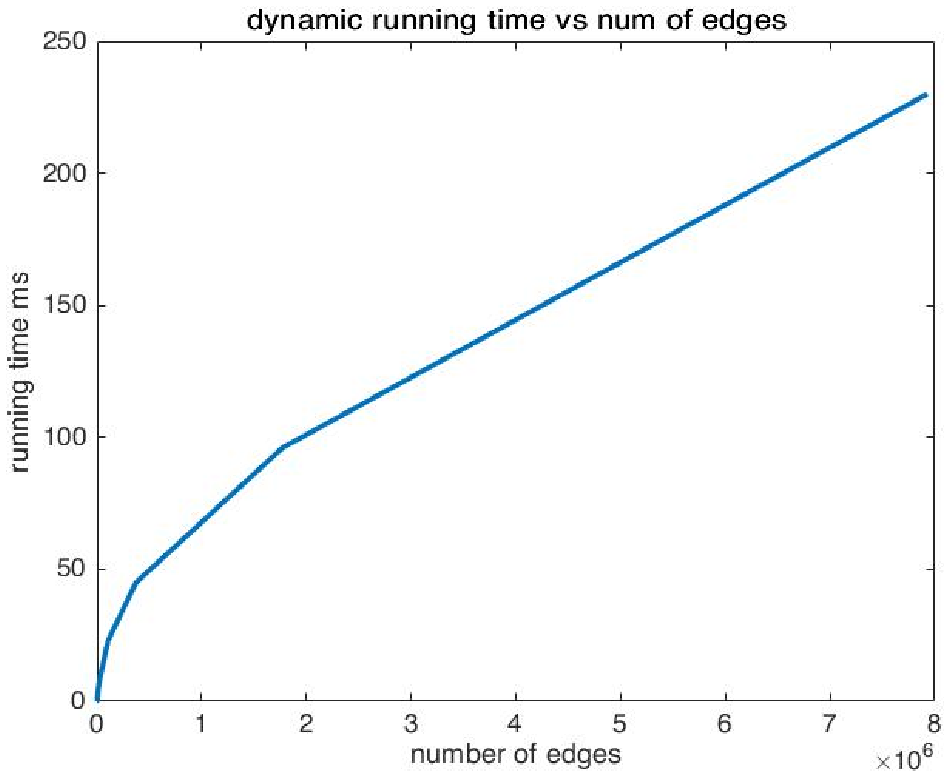


Plot of Time vs Number of vertices (log-scaled)




dynamic running time vs num of vertices log-scaled
250
200
150
100
50
0
running time ms
10^1
10^2
10^3
10^4
10^5
number of vertices


static running time vs num of edges
$\times 10^5$
12
10
8
6
4
2
0
running time ms
0
1
2
3
4
5
6
7
8
number of edges
$\times 10^6$


dynamic running time vs num of edges
running time ms
number of edges
250
200
150
100
50
0
0
1
2
3
4
5
6
7
8
$\times 10^6$