CS 577: HW 2

Daniel Ko

Summer 2020

§1 Given the coordinates of each of the n events, find a viewable subset of maximum size, subject to the requirement that it should contain event n. Such a solution will be called optimal.

§1.1 Write the iterative version of the algorithm to find the maximal size.

The main idea of this algorithm is to compute bottom up the optimal cost to view event n when we start on some event i. We do this by creating a length n list which the ith index represents the most amount of events you can view, given that you go straight from the starting coordinate, 0, to event i, and visit the most maximum amount of events from i to n. We will assume that |coordinate of n| $\leq n$ and are using zero based numbering for lists.

```
Data: crd, a list of coordinates that corresponds to events.
Result: Maximal amount of events we can view, given that we must view the last event.
def optimal(crd):
    #dp holds the maximum amount of events you can view given you visit crd[i]
    dp ← a list of 0's with the same length of crd
    dp[n-1] = 1
    for int i = length of event - 2; i ≥ 0; i–:
        possibleEvents ← a empty list
        for int j = length of event - 1; i < j; j–:
            if abs(crd[j] - crd[i]) ≤ j - i and i + 1 - abs(crd[i]) ≥ 0:
                add crd[j] to possibleEvents
        if possibleEvents is not empty:
            dp[i] = 1 + max element of possibleEvents
    return max element of dp
```

§1.2 Show the algorithm for tracing the events selected.

This algorithm looks through the list, dp, generated in *optimal* and returns the right most value for each number starting from the max to 1. This will provide which events to view such that the number of views is maximized.

Data: dp, the list used for memoization
crd, a list of coordinates that corresponds to events
max, the maximal amount of events we can view
Result: List coordiantes of events to view.
def *events(dp, crd, max)*:
  view ← a empty list
  for *each num in dp*:
    find the right most max and add its index to view
    decrement max
  return *view*

§1.3 Give a brief argument of correctness.

$$\text{MaxEvent}(i) = \begin{cases} 1 & \textbf{when } i = n-1 \\ 1 + \max(\text{MaxEvent}[j_1], \text{MaxEvent}[j_2], \cdots) & \textbf{if } \exists j > i \text{ s.t. } \mid \text{MaxEvent}[i] - \text{MaxEvent}[j] \mid \leq i - j \\ & \text{and } i + 1 - \mid \text{MaxEvent}[i] \mid \geq 0 \\ 0 & \textbf{otherwise} \end{cases}$$

Note that MaxEvent$[n-1]$ corresponds to the nth event because of our zero based numbering for lists. We compute the recurrence relation descending, i.e. from $i = n-1$ to $i = 0$.

Theorem
The recurrence relation, MaxEvent(i) is most amount of events you can visit given that the first event you visit is at i and you end on nth event.

Proof. We show by strong backwards induction that our recurrence relation is correct. Let $P(i)$ be the predicate, "MaxEvent correctly computes the most amount of events you can visit given that the first event you visit is at i and you end on nth event". We define $i \in \mathbb{N}$.
Base Case: When $i = n-1$, MaxEvent returns 1 which is the correct output because if you start at the nth event, the last event, you can only visit one event. Hence, $P(n-1)$ holds.
Inductive step: Suppose $P(\alpha)$ holds for all $\alpha \leq n-1$. We show that $P(\alpha - 1)$ holds. Our recurrence relation looks for all $j > i = \alpha - 1$ such that if you start at event i, you can visit event j. This part is given by,

$$\exists j > i \text{ s.t. } \mid \text{MaxEvent}[i] - \text{MaxEvent}[j] \mid \leq i - j$$

The second condition of the recurrence checks if it is possible to view event i, given you start at coordinate 0 and can move 1 distance before the first event, i.e the second assumption given in the homework. This part is given by,

$$i + 1 - \mid \text{MaxEvent}[i] \mid \geq 0$$

If both conditions hold, the most amount of events you can visit is the event at i, which is 1 event, plus the biggest MaxEvent from the j's. By our strong backwards inductive hypothesis, MaxEvent(j) returns the correct amount of events because we defined $j > i = \alpha - 1 \Leftrightarrow j \geq \alpha$. This will give us the correct amount of events you can visit in the case where we can visit i and j.
In all other cases, we cannot visit i or j which means the max amount of events we can visit is 0.
By strong backwards induction, we have proven that $P(i)$ is true for all $i \in \mathbb{N}$. Therefore, MaxEvent is correct. □

Corollary

The algorithm, optimal, returns the maximal amount of events we can view, given that we must view the last event.

Proof. Our algorithm evaluates the recurrence MaxEvent(i) bottom-up by storing MaxEvent(i) in dp[i]. All values for subproblems referenced by the recurrence for MaxEvent(i) will have already been computed because we evaluate dp[i] as i decreases from n-1 to 0. After all the values in dp are computed, the algorithm returns the max element, which corresponds to the most amount of events you can visit somewhere in the list crd, which in turn is the most amount of events you can visit in the entire list crd. □

§1.4 Analyze the running time.

We claim that the running time of *optimal* is $O(n^2)$.

Proof. We have a nested for loop, which run at most n times each. In the outer for loop, we search for the max element of an array which at most is size n. The rest of the computations in the outer for loop take constant time. Additionaly, after all the loops have been run, we search for the max element of dp which is size n. This leads to a total time complexity of $n(2n) + n = 2n^2 + n$. Thus, our total time complexity of our algorithm is $O(n^2)$ □

We claim that the running time of *events* is $O(n)$.

Proof. We have a for loop, which run at most n times each. In the for loop, we check if the current value is the current max element of an array takes constant time. Thus, our total time complexity of our algorithm is $O(n)$ □

§2 Design an $O(n^3)$ algorithm using dynamic programming methodology to find an optimal (least total testing cost) assembly order.

§2.1 Use iterative implementation for the algorithm to find the optimal cost.

We define α and ω to each be some piece between $[1, n]$ parts. Each piece consists of an interval $[a, b]$. The main idea of this algorithm is to compute bottom up the optimal cost. We do this by creating a nxn table which represents the optimal cost of combining $[\alpha, \omega]$. We know that it costs nothing to combine the same piece, i.e. when $\alpha = \omega$ so we initialize those values to be zero. Then, we start computing the optimal cost of each possible interval in the linear structure. We know that intervals that consist of 2 or more parts can be broken down two sections, from $[\alpha, \phi]$ and $[\phi + 1, \omega]$. We search through our table given all possible intervals and pick the smallest—most optimal—sum. Finally, we add the cost of assembling from $[\alpha, \omega]$ and add it to our sum. This will give us the most optimal cost of combining $[\alpha, \omega]$, and we save this value to our table so that subsequent subproblems will be able to use this value.

Data: pieces, a size n list containing ordered pairs representing the interval of parts, i.e. $[a, b]$ where some part begins at a and ends at b.

Result: Optimal cost to assemble the linear structure.

def *assemble(pieces)*:
    dp $\leftarrow$ a nxn matrix used for memoization, initialized with 0's
    for α *decreasing from n to 1*:
        for ω *increasing from* $\alpha + 1$ *to n*:

$$\text{dp}[\alpha, \omega] = f(\text{pieces}[\alpha][a], \text{pieces}[\omega][b]) + \min\left(\left\{\text{dp}[\alpha, \phi] + \text{dp}[\phi+1, \omega] \mid \alpha \leq \phi < \omega\right\}\right)$$

    return dp$[1, n]$

§2.2 Show the algorithm for finding the optimal order.

This algorithm performs a depth traversal to find the optimal order. We begin at dp$[\alpha, \omega]$. If the interval of this part α, ω is not 1, i.e. contains 2 or more pieces we subtract calculate the next optimal value to search for which would be dp$[\alpha, \omega] - f([\alpha][a], [\omega][b])$. We find the index of this value and call the algorithm using the index as the new parameter. Once we hit the bottom of this recursion, we print out the first two parts we want to assemble, then the next two parts, etc. until we finally have two parts remaining and we assemble the last two parts.

Data: dp, matrix used for memoization from *assemble*
pieces
Result: The optimal order to assemble the linear structure.
def *optOrder*(α, ω):
    min $\leftarrow$ dp$[\alpha, \omega]$
    if $\omega - \alpha == 0$:
        output nothing
    else:
        nextMin $\leftarrow$ min - $f([\alpha][a], [\omega][b])$
        $\alpha', \omega' \leftarrow$ search for nextMin in the column and row that min belongs, find its index
        optOrder(α', ω')
        output $\left[\text{pieces}[\alpha][a], \text{pieces}[\omega][b]\right]$

Call *optOrder(1,n)*

§2.3 Give a brief argument of correctness.

a_α represents where the piece α begins. Likewise, b_ω represents where the piece ω ends.

$$\text{MinCost}(\alpha, \omega) = \begin{cases} f(a_\alpha, b_\omega) + \min\left(\left\{\text{MinCost}(\alpha, \phi) + \text{MinCost}(\phi + 1, \omega) \mid \alpha \leq \phi < \omega\right\}\right) & \textbf{if } \omega > \alpha \\ 0 & \textbf{otherwise} \end{cases}$$

Theorem

The recurrence relation, MinCost(α, ω) is the minimum cost to assemble the linear structure from α to ω.

Proof. Let $\psi = \omega - \alpha$. We show by strong induction that our recurrence relation is correct. Let $P(\psi)$ be the predicate, "MinCost correctly computes the minimum cost to assemble the linear structure that has ψ pieces". We define $\psi \in \mathbb{N}$.
Base Case: When $\psi = 0$, MinCost correctly returns 0 because there is nothing to join when you have no elements. Hence, $P(0)$ holds.
Inductive step: Suppose $P(\psi)$ holds for all $0 \leq \psi \leq k$. We show that $P(k+1)$ holds. For all ϕ where $\alpha \leq \phi \leq \omega$, we know that $\phi - \alpha \leq k$ and $\omega - (\phi + 1) \leq k$. It follows by our strong inductive hypothesis that all values contained in the set inside the min function will be correct. The min function then returns the minimum cost to assemble two adjacent pieces linear structure of combined size $k+1$. This value is then added to the cost needed to merge remaining these two pieces, $f(a_\alpha, b_\omega)$.

By strong induction, we have proven that $P(\psi)$ is true for all $\psi \in \mathbb{N}$. Therefore, MinCost is correct. □

Corollary

The algorithm, *assemble*, correctly returns the optimal cost to assemble the linear structure.

Proof. Our algorithm evaluates the recurrence MinCost bottom-up by storing MinCost(α, ω) in dp$[\alpha, \omega]$. All values for subproblems referenced by the recurrence for MinCost(α, ω) will have already been computed because we evaluate dp$[\alpha, \omega]$ as α decreases from n to 1 and ω increasing from $\alpha + 1$ to n. After all the values in dp are computed, the algorithm returns dp$[1, n]$, which corresponds to the the minimum cost to assemble the linear structure from 1 to n, i.e. the entire linear structure. □

§2.4 Analyze the running time.

We claim that the running time of *assemble* is $O(n^3)$. We assume computing $f(a, b)$ takes constant time.

Proof. We have a nested for loop, which run at most n times each. In the inner for loop, we calculate dp$[\alpha, \omega]$ which requires computing the minimum of at most $n - 1$ values in dp. The rest of the computations take constant time. Thus, our total time complexity of our algorithm is $O(n^3)$ □

We claim that the running time of *optOrder* is $O(n^2)$. We assume computing $f(a, b)$ takes constant time.

Proof. We have a recursive call which makes at most 1 recursive call. These recursive call ends once the size for the subproblem becomes 1, i.e. when ω and α are the same value. Since the new $\omega' - \alpha'$ generated are strictly decreasing, these recursive calls happen at most n times. Inside the recursive call, we search a row and column which takes $2n$ time. The rest of the computations take constant time. Thus, our total time complexity of our algorithm is $n * 2n = O(n^2)$ □