Homework 2

CSOR W4231 - Analysis of Algorithms

Chandan Suri

UNI - CS4090

Collaborators: Arnavi Chedda (amc2476), Parth Jawale (pcj2105), Gursifath Bhasin (gb2760)

Submitted on -

March 08, 2022

Homework 2

Problem 1

Problem Description:
Input: An undirected weighted graph G = (V,E,w) with positive edge weights and a specific vertex $s \in V$
Output: A boolean array unique of size n such that unique[i] = 1 iff. there exists a unique shortest s-i path.
Solution:
Description of the Algorithm: I will be determining the unique shortest path from source node to every other node in an undirected, weighted graph G. To achieve this, I will have to modify the initialize and update functions of the Dijkstra's algorithm. As we discover paths from the source node s to all the nodes in the graph, we update the distance array with the cumulative edge weights from s to the current node under consideration in the loop. If the distance of current node v from the source node is lesser than the one we discovered previously, we update the distance of the node v to this newly found shorter distance and set $unique[v]$ to be 1. But, if we find a path whose distance is equal to the one we had already discovered, then we update $unique[v]$ to 0, since we found another path with same cumulative edge weight. This would mean that we wouldn't reach that node via a unique shortest path as there are multiple shortest paths to reach that node. We continue this process until all nodes have been visited.

For the implementation, we can either use the array or min-heap implementation of Dijkstra algorithm. This is dependent on how sparse or dense the array is. We will only be modifying $initialize()$ and $update()$ in the algorithm. Rest of the implementation will remain the same.
Pseudocode:

Algorithm 1: Find whether all the nodes in a graph G can be reached from s via a unique path.

```
Function initialize($G$, $s$):
    for $v \in V$ do
        $dist[v] \leftarrow \infty$;
        $prev[v] \leftarrow NULL$;
        $unique[v] \leftarrow 0$;
    end for
    $dist[s] \leftarrow 0$;
    $unique[s] \leftarrow 1$;
Function update($u$, $v$):
    if $dist[v] > dist[u] + w_{uv}$ then
        $dist[v] = dist[u] + w_{uv}$;
        $prev[v] = u$;
        $unique[v] = 1$;
    else if $dist[v] == dist[u] + w_{uv}$ and $prev[v] \neq u$ then
        $unique[v] = 0$;
        $prev[v] = u$;
```

Correctness of the Algorithm: As we are only updating the *initialize* and *update* functions here, the proof of correctness is only required for the part where we update the unique array. Other than that, the original Dijkstra's algorithm works as expected.
We will try and do this through proof by contradiction. Firstly, let's assume that when there is not a unique shortest path to node $v \in V$, the $unique[v] = 1$. If we consider the premise in the assumption, then that would mean that the control wouldn't go through the first condition ever. As we only update the unique array for node v to be 1 inside that if condition, that means that this shouldn't be true. Also, it would only go through the second condition when this happens when we have a node with the same distance which would also update the unique value for that node to be 1. Thus, the first assumption in our proof by contradiction wouldn't hold.
Now, we let's assume that when there is a unique shortest path to node $v \in V$, the $unique[v] = 0$. If we consider the premise in the assumption here, that would mean that we will go through the first condition for sure thus, making the unique array for node v to be 1. Thus, this assumption also fails. As for both the assumptions above, I am considering the condition which is essentially a part of the original Dijkstra's algorithm, then that should mean that we have proved that the contradiction fails for sure.
Thus, we have proved that our algorithm works correctly.

Time Complexity Analysis: The time complexities for the 2 functions above is as follows:
Initialize: For this we are initializing 3 arrays by going over all the vertices or nodes meaning

that the lines 2-5 have $O(n)$ time. Also, for lines 7-8, these are just assignment operations, so it's constant time. Thus, the total time complexity of this function is $O(n)$.
Update: Lines 10-16 have 2 if conditions with comparisons which are of constant time and others are just assignment operations which are also of constant time. Thus, this full update function is of constant time ($O(1)$).
As both these functions have the same time complexity as the ones in the original Dijkstra's algorithm, the total time complexity of our solution here will be the same. If we use the "v2" version, considering this as a dense/sparse graph, the time complexity will be "$O((n+m)(log(n))$" where n is the number of vertices and m is the number of edges. However, if we use the "v3" version, considering it as a sparse graph, then the time complexity will be "$O(m * log(n))$" where n is the number of vertices and m is the number of edges.

Problem 2

Problem Description:
Input: An undirected weighted graph G = (V,E,w) with positive edge weights and a origin vertex $s \in V$
Output: An array minedges of size n such that minedges[i] = minimum number of edges in a shortest path from s to i.

Solution:
Description of the Algorithm: I will determine the minimum number of edges from source node to every other node in a directed, weighted graph G. For the same, I will have to use the modified version of the Dijkstra's algorithm. We will be using an array $'minedges'$ to store the minimum number of edges required to reach every node of the graph G from source node s. Basically, We update $minedges[v]$ to this newly discovered path with minimum number of edges if and when we come across a minimum cost path that is shorter than the one discovered earlier. Thus the minedges will be updated as $minedges[prev(v)] + 1$.
Pseudocode:

Algorithm 2: Find the minimum number of edges needed to reach from s to all the nodes in the graph.

```
Function initialize($G$, $s$):
    for $v \in V$ do
        $dist[v] \leftarrow \infty$;
        $prev[v] \leftarrow NULL$;
        $minedges[v] \leftarrow \infty$;
    end for
    $dist[s] \leftarrow 0$;
Function update($u$, $v$):
    if $dist[v] > dist[u] + w_{uv}$ then
        $dist[v] = dist[u] + w_{uv}$;
        $prev[v] = u$;
        if $minedges[u] == \infty$ then
            $minedges[v] = 1$;
        else
            $minedges[v] = minedges[u] + 1$;
    else if $dist[v] == dist[u] + w_{uv}$ and $prev[v] \neq u$ then
        if $minedges[u] + 1 < minedges[v]$ then
            $minedges[v] = minedges[u] + 1$;
        end if
        $prev[v] = u$;
```

Correctness of the Algorithm:
As we are only updating the *initialize* and *update* functions here, the proof of correctness is only required for the part where we update the unique array. Other than that, the original Dijkstra's algorithm works as expected.
We will try and prove the correctness by explanation here. As the initialize function here is just initializing the *minedges* array to ∞, this function will be correct if the original Dijkstra's algorithm is correct.
Furthermore, for the update function, when we reach a new node with some distance, we simply increment the minimum number of edges by 1 considering that initially the value can be infinity. This would basically count the number of edges to reach that node. As Dijkstra's algorithm by itself tries to find the shortest path, thus, the distance to reach a node incrementally can be the same or greater than the distance to the previous node (at a current node v when all the nodes in the path s-v have been traversed). For the case, when it's equal, that could possibly mean that the minimum number of edges are less for the current node from a new previous node. For this case, we basically update the number of edges accordingly. As according to the Dijkstra's algorithm, this is the only valid case possible, then this should give us the final minimum number of edges for each of the nodes correctly.

Time Complexity Analysis: The time complexities for the 2 functions above is as follows:
Initialize: For this we are initializing 3 arrays by going over all the vertices or nodes meaning that the lines 2-5 have $O(n)$ time. Also, for line 7, this is just an assignment operation, so it's constant time. Thus, the total time complexity of this function is $O(n)$.
Update: Lines 9-20 have 4 if conditions with comparisons which are of constant time and others are just assignment or update operations which are also of constant time. Thus, this full update function is of constant time ($O(1)$).
As both these functions have the same time complexity as the ones in the original Dijkstra's algorithm, the total time complexity of our solution here will be the same. If we use the "v2" version, considering this as a dense/sparse graph, the time complexity will be "$O((n+m)(log(n))$" where n is the number of vertices and m is the number of edges. However, if we use the "v3" version, considering it as a sparse graph, then the time complexity will be "$O(m * log(n))$" where n is the number of vertices and m is the number of edges.

Problem 3

Problem Description:
Input: An array A of n triples (s_i, b_i, r_i) where s_i, b_i, r_i corresponds to expected swimming, biking and running times for contestant i.
Output: Sequence of start times (or schedule) with minimum completion times of all contestants.

Solution:
Description of the Algorithm: We are sorting our array consisting of triples, example:(s_i, b_i, r_i), according to the decreasing order of the sum of their biking and running times $(b_i + r_i)$. To achieve this, we'll be using merge sort since it is a stable sort. This takes $O(n * (logn))$ time where n is the number of triples in the input.

Pseudocode:

Algorithm 3: Find optimum schedule for a mini-triathlon event.

```
1 Function findOptimumSchedule($A$):
      // Sort array A (stable sort) in descending order of ($b_i$+$r_i$)
2     $optimumSch = sort(A)$;
3     return $optimumSch$;
```

Correctness of the Algorithm:

Consider two triples with indices i and j such that $b_i + r_i \leq b_j + r_j$. Moreover, I will assume that these two triples are scheduled next to each other. Following that, the following two cases arise here:

Case 1: The triplet with index i is scheduled before the triplet with index j.

In this case, the total time taken to finish both tasks will be:

$Time_1$= max $(s_i + b_i + r_i, s_i + s_j + b_j + r_j)$

Basically, first term in the max function is for the triplet with index i and second term is for the triplet with index j which has delayed start time by s_i.

Since $b_i + r_i \leq b_j + r_j$, therefore we have: $Time_1 = s_i + s_j + b_j + r_j$

Case 2: The triplet with index i is scheduled after the triplet with index j.
In this case, the total time taken to finish both is:

$Time_2 =$ max $(s_i + s_j + b_i + r_i, s_j + b_j + r_j)$

Since $b_i + r_i \leq b_j + r_j$ and both the terms of $Time_2$ are $\leq T_1$, thus,

$T_1 \geq T_2$

Also, we can see that even without assuming that these two triples are ordered next to each other, we could have reached the same conclusion that our argument holds. For that case, we will also have to take into consideration the extra time in between these two triples. Precisely, we just took this assumption for simplicity in proving that our argument holds. Hence, an array of triples sorted in decreasing order w.r.t. $b_i + r_i$ will always give us the optimal schedule.
Thus, we have successfully proved the correctness of our algorithm.

Time Complexity Analysis: Since we are just sorting the array of triples using Merge Sort (stable sorting) , the worst case time complexity of our algorithm will be $O(n * (logn))$.

Problem 4

Problem Description:
Input: A string of n characters s[1, . . . , n].
Output: Check whether the string s can be reconstituted as a sequence of valid words. If yes, output the sequence of words.

Solution:
Sub-problem: We need to find a values of K such that for any value of i, s[i:k-1] is a valid word and s[k:n] can be broken down into valid words further or s[k:n] is a valid word itself.

Recurrence: If we consider a sub-string in the text s, if we are able to find an index k such that s[k:n] can be broken down into valid words given that s[i:k-1] is a valid word, we can say that the whole text s is formed from valid words. This can be formalized for any index i such that we are considering s[i:n] as a word.
Now, suppose that we have a function $createUncorruptedText$(idx) which basically gives us a valid k at which the sub-string s[k:n] can be broken into valid words and sub-string s[i:k-1] is a valid word or it returns n if s[i:n] is a valid word itself. Thus, we can form the recurrence as follows:
createUncorruptedText(idx) =

$$\begin{cases} k & \text{if } k > i, createUncorruptedText(k) \neq -1, dict(s[i:k-1] == 1 \\ n & \text{if } dict(s[i:n] == 1 \\ -1 & \text{Otherwise} \end{cases}$$

As the recurrence above only depends on the values for the indices when k is greater than i, we can maintain a single dimensional array and do the same in a bottom-up fashion. We can fill in the values directly by looping in this fashion which would give us the possible ending indices of the valid words using which we can create a sequence of words from the text if it's possible to do so.

Boundary Conditions: We initialize the array $possibleWordEndings$ to -1 for all the indices. Also, during reconstitution step, we can start from 0th index, and go up to n following the indices stored in the array. This can be be from from the start of the array rather than backtracking or bottom-up fashion.

Order of Array Filling: As explained above and also shown in the pseudocode below, the value of $possibleWordEndings$ at index i only depends on the values of this array at indices j such that j is greater than i. Hence, we fill the array in reverse fashion (bottom-up approach), essentially starting at the last index and going till the 0th index.

Pseudocode:

Algorithm 4: Find the corresponding sequence of valid words from the corrupted text, if it's possible.

```
1  Function createUncorruptedText($s$):
2      for $idx \leftarrow 0$ to $n-1$ do
3          $possibleWordEndings[idx] \leftarrow -1$;
4      end for
5      for $startIdx \leftarrow n-1$ to $0$ do
6          for $endIdx \leftarrow startIdx+1$ to $n$ do
7              if $dict[s[startIdx : n]]$ then
8                  $possibleWordEndings[startIdx] = n$;
9              else if $possibleWordEndings[endIdx] \neq -1$ and
                    $dict[s[startIdx : endIdx]]$ then
10                 $possibleWordEndings[startIdx] = endIdx$;
11         end for
12     end for
13     $currIdx \leftarrow 0$;
14     $prevIdx \leftarrow 0$;
15     $wordsInText \leftarrow Array()$;
16     while $currIdx \neq n$ do
17         $prevIdx = currIdx$;
18         $currIdx = possibleWordEndings[currIdx]$;
19         if $currIdx == -1$ then
20             return $(False, Array(empty))$;
21         end if
22         $wordsInText = wordsInText \cup s[prevIdx : currIdx]$;
23     end while
24     return $(True, wordsInText)$;
```

Time Complexity Analysis:
Lines 2-4 take $O(n)$ time as we are going over all the full length of the text which is the size of the array $possibleWordEndings$.
Also, for the loop on line 5, we loop over the same n times, and the inner loop runs for n times in the worst case but incrementally reaches there one by one. As this looping construct essentially runs n*n times at the most, the worst case complexity for this would be $O(n^2)$.
All the lines from 7-11 run in constant time as those are just comparison or assignment operations.
Also, lines 13-15 are just initialization operations and thus runs in constant time. Following this, the while loop on line 16 could run for n times in the worst case when all the words in the text are just single character words and thus, could run in $O(n)$ time. Also, all the operations in the lines from 17-23 are just assignment, update or comparison operations which

would run in constant time. Thus, the while loop runs in $O(n)$ time.
Finally the return statement would also run in $O(1)$ time. Thus, the final time complexity for the whole algorithm is: $O(n^2)$ after adding all the time complexities stated above.

Space Complexity Analysis:
Regarding the space considerations, all the indices for the loops takes constant space. Other than that, we only keep an array of n size naming $possibleWordEndings$ and thus takes $O(n)$ space. Therefore, the total space complexity for it will be $O(n)$.

Problem 5

Problem Description:
Input: The number of games won by Alice i, the number of games won by Bob j and the total number of games n to be won.
Output: The probability that Alice will win n games first thus winning the match.

Solution:
Sub-problem: As we need to calculate the probability of Alice winning the game if she has already won i games and Bob has already won j games. This can be subdivided into 2 sub problems as follows:

1. The probability of Alice winning n games before Bob when Bob has won one game.
2. The probability of Bob winning n games before Alice when Alice has won one game.

Recurrence: Let's make some assumptions first. Let $probs[i][j]$ be the probability that Alice wins the game when Alice has already when i games and Bob has won j games. This will be the sum of the two mentioned sub problems above.

1. The probability of Alice winning one game multiplied by the probability of Alice winning the entire game (n games). This is shown as such (in the recurrence below - 1st term) because after the current game, Alice would have won i+1 games and Bob would have only won j games.
2. The probability of Bob winning one game multiplied by the probability of Bob winning the entire game (n games). This is shown as such (in the recurrence below - 2nd term) because after the current game, Bob would have won j+1 games and Alice would have only won i games.

After adding the probabilities above, the recurrence can be shown as:
$probs[i][j] = probAliceWins * probs[i+1][j] + probBobWins * probs[i][j+1]$.

Boundary Conditions: We initialize the probabilities with -1 for all the places in the (n+1)*(n+1) matrix.
Following that, as each row suggest a win for Alice, the final winning probability for Alice should be 1.0 for once she wins the game by winning n games. Thus, for the last nth row, I fill 1.0 suggesting the probability of Alice winning as 1.
After that, I also initialize the probability of Bob winning n games as 0. That's why I initialize all the values in the last column of the matrix as 0.
Other than that, I am taking the probability of a win for Alice or Bob as 0.5 as provided in the problem statement.

Order of Matrix Filling: As we can observe from the recurrence above, the values of $probs[i][j]$ only depends on the values of $probs[i+1][j]$ and $probs[i][j+1]$. Hence, we can fill up the matrix in the bottom-up fashion. (as shown in the pseudocode)

Pseudocode:

Algorithm 5: Compute the probability of Alice winning n games first.

```
Function computeProbAliceWins(i, j, n):
    $probs \leftarrow Array((n+1, n+1), init: -1)$;
    $probAliceWins \leftarrow 0.5$;
    $probBobWins \leftarrow 0.5$;
    for $wins \leftarrow 0$ to $n-1$ do
        $probs[n][wins] \leftarrow 1.0$;
        $probs[wins][n] \leftarrow 0.0$;
    end for
    for $wins1 \leftarrow n-1$ to $0$ do
        for $wins2 \leftarrow n-1$ to $0$ do
            $probs[wins1][wins2] = (probAliceWins * probs[wins1+1][wins2]) + (probBobWins * probs[wins1][wins2+1])$;
        end for
    end for
    return $probs[i][j]$;
```

Second probable solution could be (Better Space Complexity):

Algorithm 5: Compute the probability of Alice winning n games first.

```
1  Function computeProbAliceWins($i$, $j$, $n$):
2      $probs \leftarrow Array((2,2), init: -1)$;
3      $probAliceWins \leftarrow 0.5$;
4      $probBobWins \leftarrow 0.5$;
5      for $wins \leftarrow 0$ to $1$ do
6          $probs[n][wins] \leftarrow 1.0$;
7          $probs[wins][n] \leftarrow 0.0$;
8      end for
       // (mod) here represents the modulo operator.
9      for $wins1 \leftarrow n-1$ to $0$ do
10         for $wins2 \leftarrow n-1$ to $0$ do
11             $probs[wins1(mod)2][wins2(mod)2] =$
                 $(probAliceWins * probs[(wins1+1)(mod)2][wins2(mod)2]) +$
                 $(probBobWins * probs[wins1(mod)2][(wins2+1)(mod)2])$;
12         end for
13     end for
14     return $probs[i][j]$;
```

Time Complexity Analysis: This analysis is for the first solution above:
Line 2 is basically initialization of the matrix holding all the probabilities as -1 for which we would have to go over each cell in the matrix thus, taking $O(n^2)$ time.
Lines 3 and 4 will take constant time as these are just assignment operations.
Following that, the loop from lines 5-8 will run in $O(n)$ time as we are going over the last column and the last row and initializing them with zeros and ones respectively.
For lines 9-12, the outer loop runs n times and the inner loop also runs n times. Also, the statement on line 11 is basically an assignment operation which will take constant time. Thus, lines 9-12 would have a worst case time complexity of $O(n^2)$.
Also, the return statement will run in $O(1)$ time. After adding up all the time complexities above, the total worst case time complexity for the algorithm will be $O(n^2)$.

Space Complexity Analysis: This is for the first solution above:
Regarding the space considerations for the problem, other than the indices needed for looping and the variables holding the probabilities, I take the matrix of probabilities as an n*n matrix. As this should hold the probability for the worst case as well, that's why I am taking the probability matrix to be of size n*n. Thus, the final space complexity of the solution is $O(n^2)$.
However, we could just take a 2*2 matrix to hold the probabilities under consideration which is directly visible from the recurrence above. This is what has been done in the second solution above. If we follow that solution, the time complexity remains the same but the space

complexity goes from being $O(n^2)$ to $O(1)$ as we will always take a 2*2 matrix for holding the probabilities no matter what.