COMP 4190 Assignment 1 Answer

Fidelio Ciandy, 7934456

Problem 1

For my first question, I have two functions with different purposes.

- **TransformWord**
  - This function performs the BFS.
  - Starting from the begin word, it performs BFS by comparing the current word with the words in the word list.
  - If the current word differs by a single letter, the new word is inserted into the queue and the sequence counter is incremented by 1.
  - The visited word is added to a set to prevent circular loops.
  - The loop continues until the end word is reached or the queue is empty.
- **CheckAdjacentWords**
  - This function will compare two strings and check whether those two strings differ by 1 letter or not

Problem 2

For the second question, I also have two different functions.

- **CountProvinces**
  - This function will iterate through the isConnected array and check whether the current city has been visited or no
  - If the city hasn't been visited, we will visit that city and find the connection through DFS, by calling the DFS function
  - for unvisited city, we will increment the province count by 1
- **DFS**
  - This function will iterate through one of the isConnected entry
  - if indeed, the city is connected (==1) with another city and that city hasn't been visited previously
  - we will recursively go into that city and repeat the same process (this is the DFS process)
  - Otherwise, we will go into the next iteration of the isConnected

Problem 3

For the third question, I also have two different functions.

- **HikeDijkstra**
  - This is where the Dijkstra's algorithm is being used.
  - It will have two main data structures,
    * queue[(row, col, current effort)], where current effort is the min effort to reach the cell thus far
    * distance[][], to record each node effort thus far (this is how we use the Dijkstra algorithm, where we will always compare with the lower distance)

    Two conditions for the loop to stop:
    * the queue is empty
    * or, we have reached the end of the grid
  - Starting from the top left grid, for each node, it will find all the possible neighbors
  - for each of those neighbors, we will calculate the height of the neighbor with the current node
  - get the maximum absolute difference with the previous height
  - and we would only append to the queue only if the new max difference is lower than the last recorded distance (this is where the Dijkstra's algorithm is happening)
- **FindNeighbors**
  - Given a specific entry (row, col), this function will calculate the neighbors for that specific (row, col)

Problem 4

- **FindGoal**
  - this is where the iterative deepening is being handled
  - This function is using a while loop, it will keep calling the DepthGoalSearch until we found the result or until the maximum depth is reached
  - first, it will start with curr max depth = 0, the loop will keep incrementing the current max depth until it reach the max depth
  - if the current max depth has reached the max depth but result hasn't been found, it will return -1
  - Otherwise, it will keep trying curr max depth + 1
- **DepthGoalSearch**
  - This is where the DFS is happening
  - if the current node is equal to the goal, we return the current depth
  - if current depth is curr max depth, we return -1 (iterative deepening will be handled in the FindGoal function)
  - if children doesn't exist, return -1
  - else, it will use a depth first search, iterating through all its children first

Problem 5

(a) The state $dp[i][j]$ represents the minimum number of edit operations to transform s into t. For each value in i and j, the table will save the previous minimum comparison between prefixes of $s[1..i]$ and $t[1..j]$, and dp[m][n] will get the minimum value of the edit operations between the two strings.

For recurrence, where $i \geq 1, j \geq 1$:

- $dp[i-1][j] + 1 \rightarrow$ this specify the delete operation, where last character of s_i is deleted from $s[1..i]$, to transform $s[1..i-1]$ into $t[1..j]$
- $dp[i][j-1] + 1 \rightarrow$ this specify the insert operation, where we would like to insert character t_j into string s, transforming $s[1..i]$ into $t[1..j-1]$.
- $dp[i-1][j-1] + l \rightarrow$ this specify the substitution operation,
  - where if $s_i = t_j$ we won't add 1, so cost is 0
  - otherwise, if $s_i \neq t_j$, we will add 1 to substitute the value

(b) The time complexity is $O(m \times n)$.

Since the two strings s and t have lengths m and n, i ranges from 0 to m, and j ranges from 0 to n the dp table has $(m+1)(n+1)$ cells.

Futhermore, the calculation of each dp entry using the three recurrences (insertion, deletion, substitution), all of those operations are just accessing the array, thus this can be considered to be $O(1)$ as the time complexity.

Thus, since each entry is filled using constant number of operations $O(1)$ and each result is being stored in an $(m+1)(n+1)$ array. Therefore, the time complexity is $O(m \times n)$.

(c) The memory/ space complexity of this DP implementation is $O(m \times n)$.

Same reasoning as the previous part, since the two strings s and t have lengths m and n, i ranges from 0 to m, and j ranges from 0 to n the dp table has $(m+1)(n+1)$ cells. Thus, the memory complexity is $O(m \times n)$.

(d) From the recurrence formula, to fill a row in the DP array, we require only one row from the previous calculation. For instance for i=5, we only need to access

- $dp[4][5] \rightarrow$ previous row, current column
- $dp[5][4] \rightarrow$ current row, previous column
- $dp[4][4] \rightarrow$ previous row, previous column

Knowing how these calculation work, we could just create two separate array prev[] and curr[]. Where prev[] stores the values of row-1 and curr[] stores the current row i. Thus, this approach reduces the space complexity to $O(n)$.

Problem 6

(a) $\nabla_x f(x) = \frac{1}{2}(A + A^T)x - b$

Since A is symmetric, thus, $A = A^T$

$\nabla_x f(x) = Ax - b$

(b) $x^* : \nabla f(x^*) = 0$

$Ax^* - b = 0$
$Ax^* = b$
$A^{-1}(Ax^*) = A^{-1}b$
$Ix^* = A^{-1}b$
$x^* = A^{-1}b$

(c) From the function $f(x)$ we know that the curve of the graph is upward since $x^T Ax > 0$ for all non-zero x..(1)

Furthermore, A is also symmetric as given in the question..(2)

Based on (1) and (2), we can infer that $det(A) > 0$, thus A is invertible, thus, it has a unique solution..(3)

From part (b), we also have calculated the value of x^* for the matrix A..(4)

Thus, we can conclude that the solution is unique and minimum based on (3) and (4)

Problem 7

(a) $\nabla f(x) = 2(x - 2)$
(b) $x_{k+1} = x_k - \alpha \nabla f(X_k)$
$x_{k+1} = x_k - \alpha 2(x - 2)$
(c) $x^* : \nabla f(x^*) = 0$
So, $x^* = 2$
(d) if step size α is too large: oscillations (no convergence)
if step size α is too small: slow convergence
(e) $x_{k+1} = x_k - \alpha 2(x - 2)$
$= x_k - 2\alpha x_k + 4\alpha$
$= (1 - 2\alpha)x_k + 4\alpha$
Example, let $(1 - 2\alpha) = b$
$x_1 = x_0 b + 4\alpha$

$x_2 = x_1 b + 4\alpha$
$= (x_0 b + 4\alpha)b + 4\alpha$
$= x_0 b^2 + 4\alpha b + 4\alpha$
$= x_0 b^2 + 4\alpha(b + 1)$

$x_3 = x_2 b + 4\alpha$
$= (x_0 b^2 + 4\alpha(b + 1))b + 4\alpha$
$= (x_0 b^2 + 4\alpha b + 4\alpha)b + 4\alpha$
$= x_0 b^3 + 4\alpha b^2 + 4\alpha b + 4\alpha$
$= x_0 b^3 + 4\alpha(b^2 + b + 1)$

So, in general for x_k
$x_k = x_0 b^k + 4\alpha(b^k + b^{k-1} + ... + 1)$

$(b^k + b^{k-1} + ... + 1)$ is a geometric sum

Thus,

$x_k = x_0 b^k + 4\alpha \left(\frac{1-b^k}{1-b}\right)$

$= x_0 b^k + 4\alpha \left(\frac{1-b^k}{2\alpha}\right)$

$= x_0 b^k + 2(1-b^k)$

$= x_0 b^k + 2 - 2b^k$

$= (x_0 - 2)b^k + 2$

$= (1-2\alpha)^k (x_0 - 2) + 2$ ... sub in $b = (1-2\alpha)$

x_k converges to x^* == x_k converges to 2, since $x^* = 2$

Thus, $x_k - 2$ converges to 0

In other word, $(1-2\alpha)^k (x_0 - 2) \to 0$

Furthermore, $x_0 \neq x^* \to x_0 - 2 \neq 0$

Therefore, $(1-2\alpha)^k \to 0$

To make, $(1-2\alpha)^k \to 0,$ we need to make $|(1-2\alpha)| < 1$

Solve for, $|(1-2\alpha)| < 1$

$-1 < 1 - 2\alpha < 1$

$-1 - 1 < 1 - 2 - 1\alpha < 1 - 1$

$-2 < -2\alpha < 0$

$0 < \alpha < 1$

Thus, the condition of α under which it converges to x^* is $0 < \alpha < 1$