Arthur Dent
42

# Assignment 4

COMP 3804
December 4, 2022

## Question 1

You and two of your friends live all in different cities. Each of you has a car and can start driving right now (after determining the meeting location). You drive on a road network, i..e, you have cities as vertices and two cities are connected via a directed edge if there is a road between them; the weight of the directed edge $(u, v)$ is the time it takes you to get from $u$ to $v$. You want to meet as soon as possible. How would you select a meeting location from one of your $k$ favourite hang-out spots(also graph vertices) known to all of you? Which algorithm would you use and how is this done most efficiently? The more efficient the solution, the better the mark.

**Solution:**

Step 1: We create a MST for each of the three starting locations and store the cost for each person at each city. The most efficient way to do this is with Prim's Algorithm which runs in $O(|E| + V \log V) = O(V^2 + V \log V) = O(V^2)$ time.
Note: there is a linear time deterministic algorithm by Bernard Chazelle but it only works on undirected graphs (and therefore an incorrect answer). Karger, Klein & Tarjan have a randomized method which is linear expected time but the same worst case time as Prim's (and is not covered in this course). Djikstra's is also an acceptable answer as it has the same run time.

Step 2: Since we want to meet as soon as possible, we want to choose the city where the last person to arrive arrives soonest:

**Algorithm 1** Pick Meeting Location

**Input:** $k$ **meeting points**
**Output: Optimal meet location:** *location*

```
1: cost ← ∞
2: location ← NULL
3: for v in k meeting points do
4:     if max{v.costs} < cost then
5:         cost ← max{v.costs}
6:         location ← v
7: return location
```

The proofs are trivial (and omitted): Prim's has been proven in class, and the second part we iterate over all k points comparing it's max to the current min and updating accordingly.

# Question 2

Give a bijection between well-formed (or valid) bracket sequences and full binary trees (see definition in class) so that you can count the number of such trees. Prove that it is a bijection.

To prove a bijection we need to show two things: That there are the same number of well formed bracket sequences (2n) and full binary trees (2n+1), and that there is one and only one sequence for a given tree.

From class we know that the number of well formed bracket sequences is $C_n$.

For full trees we will prove by induction that there is also $C_n$:
**Base Case**: $n = 0$: $C_0 = 1$ and there is only one possible tree with 1 node.

**Inductive Hypothesis**: A tree of size n leaf nodes has $C_n$ possible constructions

**Inductive Step**: A tree with $n + 1$ leaf nodes has two sub trees, $k$ nodes to the left, and $n - k$ nodes to the right. By IH, there are $C_k$ possible configurations on the left and $C_{n-k}$ possible configurations on the right. The total is then:

$$C_{n+1} = \sum_{k=1}^{n} C_k C_{n-k} = C_n$$

Therefore, for a full binary tree of size $2n + 1$ there are $C_n$ possible constructions.

**Mapping binary trees to bracket sequences**:
This can be done with an Euler-Tour: On the first visit of a node, if it is a left child add "(" to the sequence, and if it is a right child add ")". The one-to-one relationship should be obvious: In order for a full binary tree to produce a different bracket sequence, the vertices must be visited in a different order. It also maintains the well ordered bracket sequence property as a left child is always visited before a right (by definition of an Euler tour).

# Question 3

There are many algorithms for finding the longest common subsequence between two strings.

- Formally define the problem.
- Illustrate the problem via an example.
- Describe a brute force algorithm, prove its correctness and analyze its time complexity and space complexity.
- Describe a dynamic programming algorithm for the problem.
- Illustrate on an example of reasonable size how the algorithm works.
- Prove that it works correctly.

- Analyze its time and space complexity.

**Formally define the problem:** A subsequence is a string of characters contained within a larger string in the same order, but not necessarily contiguous. The problem is to find the largest string of characters that can be found in the two larger strings, such that the same characters of the subsequence appear in the same order in both strings.

**Illustrate the problem via an example:** Given the following input strings:

$$S_1 = \text{'this is a string'}$$
$$S_2 = \text{'the second string'}$$

The longest common subsequence is: 'th s string'.

**Brute Force Algorithm:**

**Algorithm 2** Brute_Force($S_1$, $S_2$)

**Input: string:** $S_1$, $S_2$
**Output: string:** *longest*

```
1:  subsequences_1 ← Get_All_Subsequences(S_1)
2:  subsequences_2 ← Get_All_Subsequences(S_2)
3:  longest_length ← 0
4:  longest ← NULL
5:  for sequnce_1 in subsequences_1 do
6:      for sequnce_2 in subsequences_2 do
7:          if sequnce_1 = sequence_2 then
8:              if sequnce_1.length > longest_length then
9:                  longest_length ← sequence_1.length
10:                 longest ← sequence_1.length
11: return longest
```

**Analysis**:
If $S_1$ is of length $n$ and $S_2$ is of length $m$:
Time Complexity:
Lines 1 & 2) Creating the subsequences takes $O(2^n + 2^m)$ time
Lines 3 & 4) Constant time
Lines 5 - 7) Looping takes $O(2^n) * O(2^m)$ time. Each comparison takes $O(n * m)$ worse case.
Total time is $O(2^n * 2^m * n * m)$
Lines 9 - 11) Constant
Total time complexity: $O(n * m * 2^n * 2^m)$
Space:
There are $2^n$ subsequences of $S_1$ each taking $O(n)$ space.
There are $2^m$ subsequences of $S_2$ each taking $O(m)$ space.
Total Space: $O(n * 2^n + m * 2^m)$

**Correctness:**
Outer Loop:

Invariant: At the beginning of the $i^{th}$ loop the longest of the first $i-1$ subsequences is stored.

Initialization: Before entering the loop, no string is stored

Maintenance: During the $i^{th}$ iteration, the $ith$ subsequence will be compared with all subsequences from the second string. If a longer matching string is found, the longest subsequence will be updated.

Termination: Upon completing the $(2^n)^{th}$ iteration, the longest subsequence will have been found.

Inner Loop:

Invariant: At the beginning of the $j^{th}$ loop the longest matching of: 1) the first $j-1$ subsequences of $S_2$ matching the $i^{th}$ subsequence of $S_1$, or 2) the $(i-1)$ subsequence of $S_1$ matching any subsequence in $S_2$, will have been stored.

Initialization: Before entering the loop, no comparisons have been made to the $i^{th}$ subsequence of $S_1$

Maintenance: During the $j^{th}$ iteration, the $j^{th}$ subsequence from $S_2$ will be compared with the $i^{th}$ subsequence from $S_1$. If they match and are longer than the current longest, the longest subsequence will be updated.

Termination: Upon completing the $(2^m)^{th}$ iteration, all subsequences in $S_2$ will have been compared to the $i^{th}$ subserquence of $S_1$.

**Note:** This is one of several Brute Force algorithms. All should have roughly the same time/space complexity (remember recursive calls take space on the stack) **Describe a dynamic programming algorithm for the problem**

On Page 394 of the textbook, you can find the solution to this problem. It requires two 2d arrays, 1 for computing the length, and 1 for recreating the longest subsequence. They have been repoduced below:

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

Figure 1: LCS-Length creates two tables from which the LCS can be derived by running PRINT-LCS on the outputs

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2      return
3  if b[i, j] == "↖"
4      PRINT-LCS(b, X, i − 1, j − 1)
5      print x_i
6  elseif b[i, j] == "↑"
7      PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

Figure 2: Will produce the LCS based on the output of LCS-Length

# Assignment 4

**Illustrate on an example of reasonable size how the algorithm works:**
Again from the textbook $S_1 = ABCBDAB$ & $S_2 = BDCABA$:

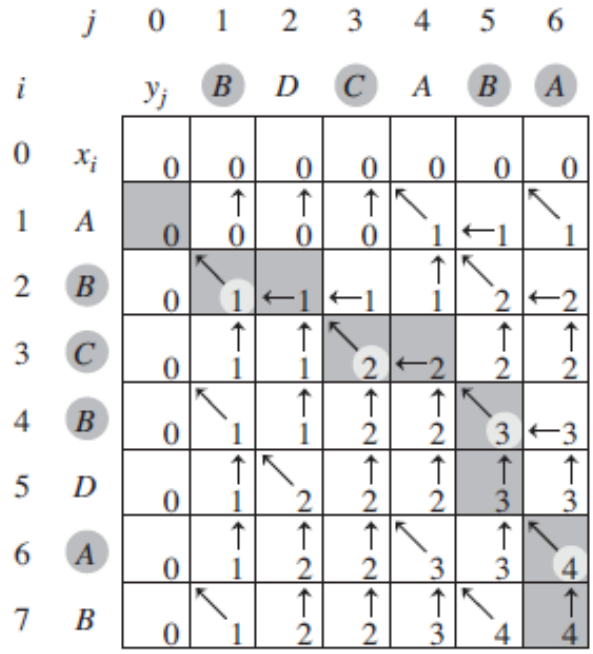


Figure 3: Example of the table based solution

**Prove that it works correctly**
By loop invariant:
Invariant: At the beginning of the $(i,j)^{th}$ loop $C[i-1,j-1]$ contains the length of the current longest subsequence and $B[i-1,j-1]$ points to the previous largest

Initialization: Before the loop $C$ is empty with every value initialized to 0 and $B$ is empty

Maintenance: If $S_1[i] = S_2[j]$, $C[i,j] = C[i-1,j-1]+1$ and $B[i,j]$ points to $B[i-1,j-1]$. Otherwise $C[i,j] = max\{C[i,j-1], C[i-1,j]\}$ and $B[i,j]$ points to the same max (and this the same character in one of the two strings), and the loop invariant is maintained.

Termination: At the end of the $(i,j)^{th}$ iteration $C[i,j]$ contains the length of the longest subsequence and $B[i,j]$ points to either the next character, or the same.

**Time Complexity:**
Since there are two nested loops of length $n$ and $m$ and constant work being done within the nesting, the overall time complexity is $O(n*m)$

# Question 4

Find an optimal parameterization of a matrix-chain product whose sequence of dimensions is $(3, 10, 2, 12, 5, 50, 4)$. You must use the table form discussed in class (the diagram on page 62 of the class notes takes from the CLRS textbook). Describe how you obtained these values of $m[i, j]$: $m[1, 6]$, $m[2, 5]$, $m[3, 5]$, and $m[2, 3]$. The remaining values of $m[i, j]$ just need to be filled in.

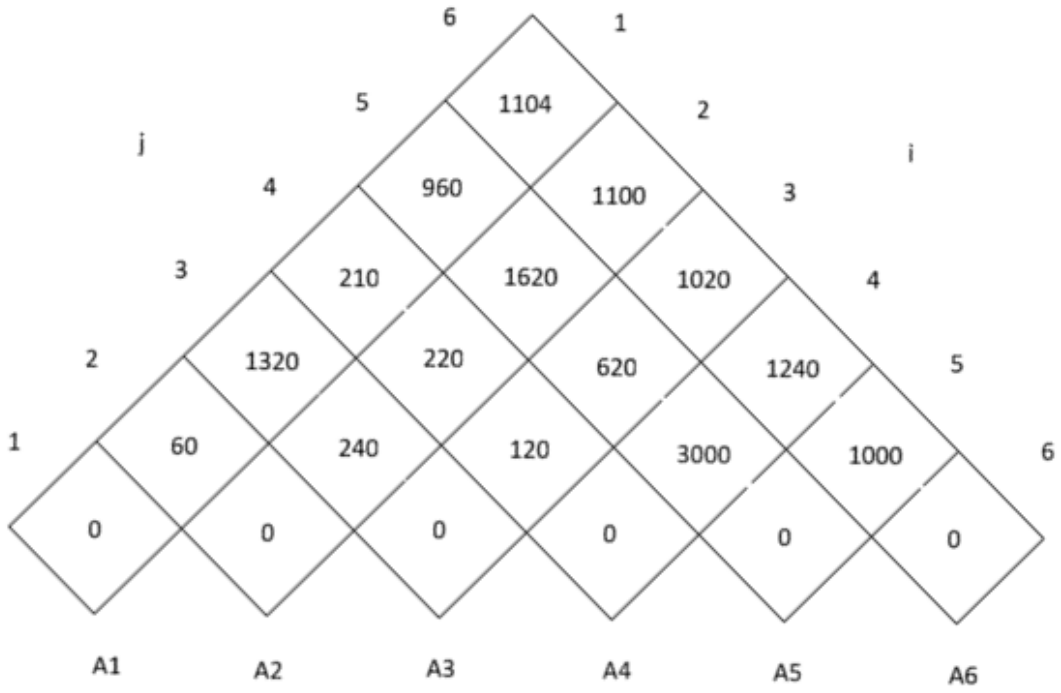


Figure 4: M table for Question 4

$$m[1,6] = min \begin{cases} m[1,1] + m[2,6] + 3 * 10 * 4 = 1220 \\ m[1,2] + m[3,6] + 3 * 2 * 4 = 1104 \\ m[1,3] + m[4,6] + 3 * 12 * 4 = 1516 \\ m[1,4] + m[5,6] + 3 * 50 * 4 = 1560 \end{cases}$$

$$m[2,5] = min \begin{cases} m[2,2] + m[3,5] + 10 * 2 * 50 = 1620 \\ m[2,3] + m[4,5] + 10 * 12 * 50 = 9240 \\ m[2,4] + m[5,5] + 10 * 5 * 50 = 2720 \end{cases}$$

$$m[3,5] = min \begin{cases} m[3,3] + m[4,5] + 2 * 12 * 50 = 4200 \\ m[3,4] + m[5,5] + 2 * 5 * 50 = 620 \end{cases}$$

$$m[2,3] = m[2,2] + m[3,3] + 10 * 2 * 12 = 240$$

The optimal paraameterization is then:

$$(A_1 \cdot A_2) \cdot (((A_3 \cdot A_4) \cdot A_5) \cdot A_6)$$

# Question 5

Consider the enclosed graph G = (V, E) on n vertices. Apply DFS to it (starting at node 1) and describe the intervals $[pre, post]$ for all nodes. List all back-edges and cross-edges.

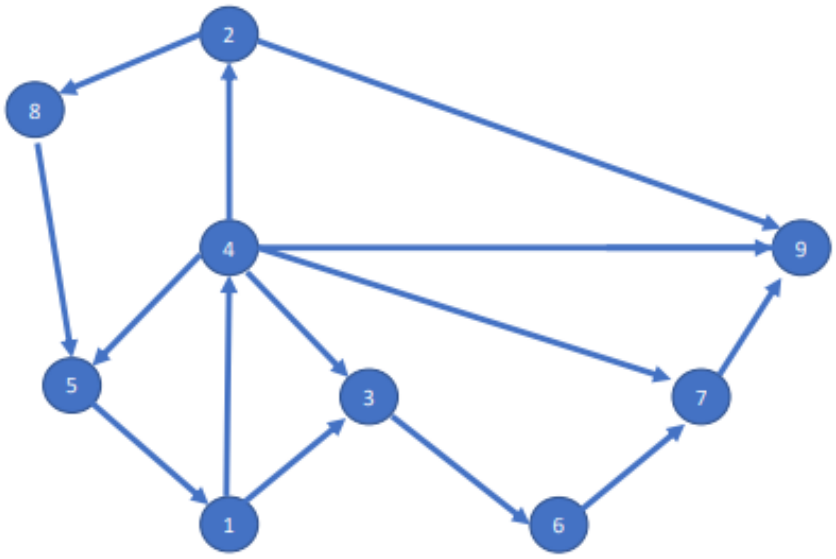


Figure 5: The input graph for Question 5

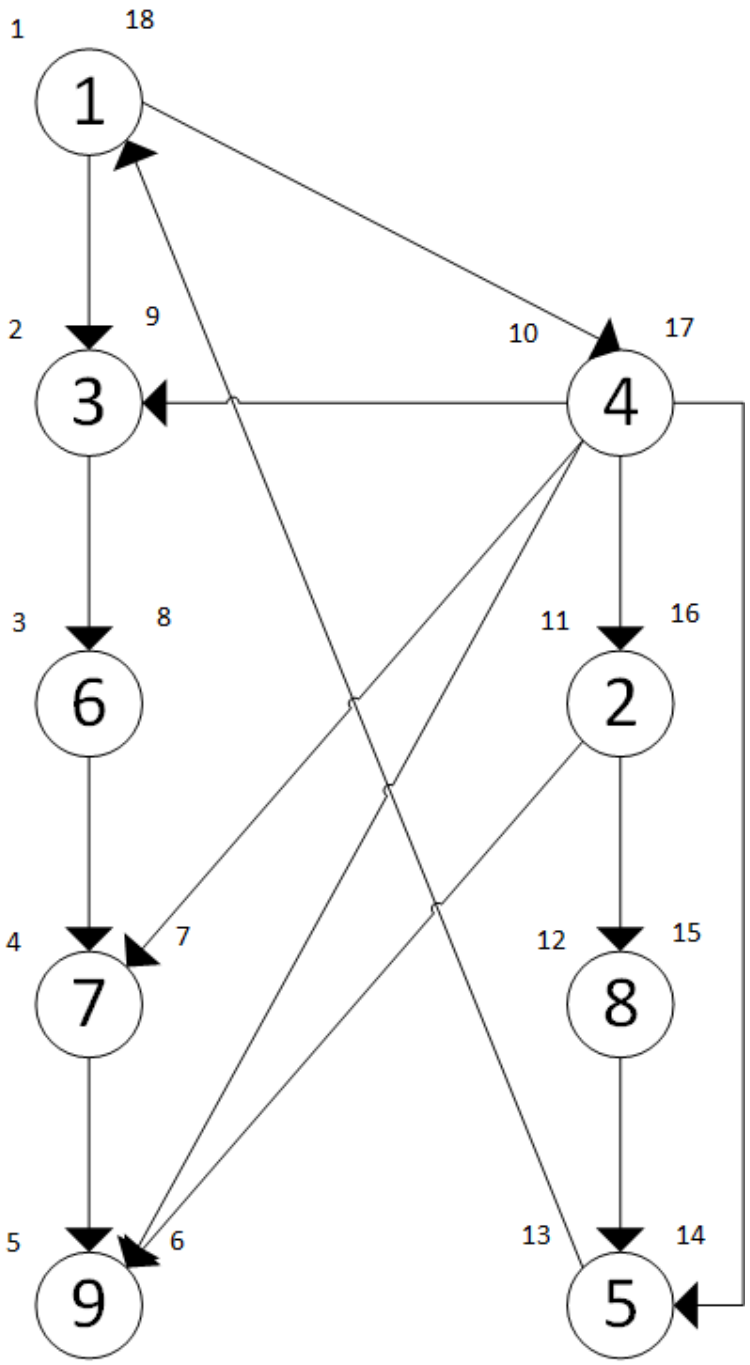


Figure 6: Result of DFS on the input graph

### Edge Types:

| Edge | Type |
|---|---|
| 13 | Tree |
| 14 | Tree |
| 28 | Tree |
| 29 | Cross |
| 36 | Tree |
| 43 | Cross |
| 47 | Cross |
| 45 | Forward |
| 49 | Cross |
| 51 | Back |
| 67 | Tree |
| 79 | Tree |
| 85 | Tree |

Suppose you have a graph in which, after a DFS traversal, every pair of intervals $[pre, post]$ have a non-empty intersection. What can you conclude about the graph (if anything)?

If for every pair of intervals $[pre, post]$ have a non-empty intersection, we can conclude that there are no cross edges. (see slide 33 in graphProblems in the Class slides).

## Question 6

Consider the following linear program:

$$\begin{aligned} \text{minimize:} \quad & 2x_1 + 3x_2 \\ \text{subject to:} \quad & x_1 + 1.5x_2 \leq 10 \\ & x_1 - x_2 \leq 3 \\ & x_2 \leq 3 \\ & x_1, x_2 \geq 0 \end{aligned}$$

- Show the feasible region by plotting the constraints on the $(x_1, x_2)$-Cartesian coordinate system.
- Using your feasible region, find the optimal solution for this linear program. Is this the only solution? If yes, then explain why. If no, then state how many optimal solutions are there and justify your answer.

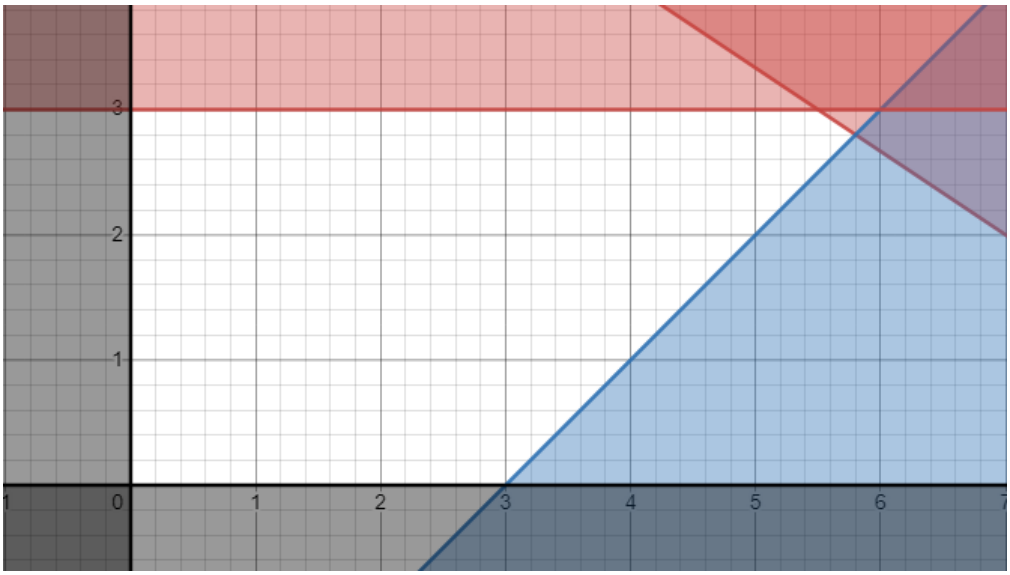


Figure 7: Feasibility region shown in white

As the LP asked us to minimize, there is only one possible solution at $x_1 = x_2 = 0$.