CS5800: Algorithms SEC 05 Summer Full 2024 (Seattle)
Homework 6

Payal chavan

06/28/2024

Question 1: A contiguous subsequence of a list S is a subsequence made up of consecutive elements of S. For instance, if S is

5, 15, -30, 10, -5, 40, 10,
then 15, -30, 10 is a contiguous subsequence but 5, 15, 40 is not. Give a linear-time algorithm for the following task:
Input: **A list of numbers,** $a_1, a_2, \cdots, a_n$.
Output: **The contiguous subsequence of maximum sum.**
For the preceding example, the answer would be 10, -5, 40, 10, with a sum of 55.
(*Hint:* For each $j \in 1, 2, \cdots, n$**, consider contiguous subsequences ending exactly at position j.)**

Solution:

We are required to find the contiguous subsequence with the maximum sum in a given list of integers.
For that we need to find the starting and ending indexes of the subsequence that gives the maximum sum.

The pseudocode algorithm for this is given below:

function max_subsequence(A):
Input: A list of numbers, $(a_1, a_2, \cdots, a_n)$.
Output: The contiguous subsequence of maximum sum $\rightarrow max\_sum, A(start, end)$

```
max_sum = 0
current_sum = 0
start, end = 0, 0
temp_start = 0

n = length of list A
for j = 0 to n:
current_sum += A[j]
if current_sum < 0:
current_sum = 0
temp_start = j + 1

if max_sum < current_sum:
max_sum = current_sum
start = temp_start
end = j

return max_sum, A(start,end)
```

The algorithm works as follows:

1. Initialize two variables: max_sum and current_sum to 0. With that, also set start, end, temp_start to 0. max_sum keeps track of the maximum sum found so far, and current_sum accumulates the sum of the subsequence. start and end represent starting and ending index of subsequence of maximum sum. While, temp_start is the starting index of the current subsequence.
2. Iterate through each element A[j] in the array, where 'j' denotes the current index position in the list.
3. Add A[j] to current_sum.
4. Suppose if current_sum becomes negative, reset it to 0 (i.e, current subsequence is ended). Update temp_start $= j + 1$ as new sub sequence will start from $j + 1$.
5. Otherwise, update max_sum if current_sum is greater than max_sum. In this case, start = temp_start, that means starting index of a subsequence of maximum sum is set to the starting index of the current subsequence. And end index of the subsequence of maximum sum is set to the current position in the list.
6. Return max_sum and A(start, end) i.e., (maximum_sum, subsequence of maximum sum).

To determine the time complexity of this algorithm, we can see that the algorithm contains just one for loop.
To find the optimal subsequence with the maximum sum, the algorithm efficiently processes the list A in a single pass, taking only $\mathcal{O}(n)$ time. This linear time complexity makes it an effective approach for solving this problem.

Question 2: *Pebbling a checkerboard.* We are given a checkerboard which has 4 rows and n columns, and has an integer written in each square. We are also given a set of 2n pebbles, and we want to place some or all of these on the checkerboard (each pebble can be placed on exactly one square) so as to maximize the sum of the integers in the squares that are covered by pebbles. There is one constraint: for a placement of pebbles to be legal, no two of them can be on horizontally or vertically adjacent squares (diagonal adjacency is fine).

Part (a): Determine the number of legal patterns that can occur in any column (in isolation, ignoring the pebbles in adjacent columns) and describe these patterns.
Call two patterns compatible if they can be placed on adjacent columns to form a legal placement. Let us consider subproblems consisting of the first k columns $1 \leq k \leq n$. Each subproblem can be assigned a type, which is the pattern occurring in the last column.

Solution (a):

A checkerboard with 4 rows and n columns is given in the problem. Along with that 2n pebbles are required to be placed on the checkerboard such that they form legal patterns. The condition for pebble placement is that no two pebbles can be placed on vertically or horizontally adjacent squares.

Dynamic programming is used for solving this problem.
Dynamic programming is an algorithmic approach that breaks down a problem into smaller sub-problems. The optimal solution to the overall problem is then derived from the optimal solutions of these sub-problems. Essentially, it's about solving a big problem by solving smaller related problems first.

Now, let's determine the legal patterns and given below is the table that shows the possible patterns based on the given condition:

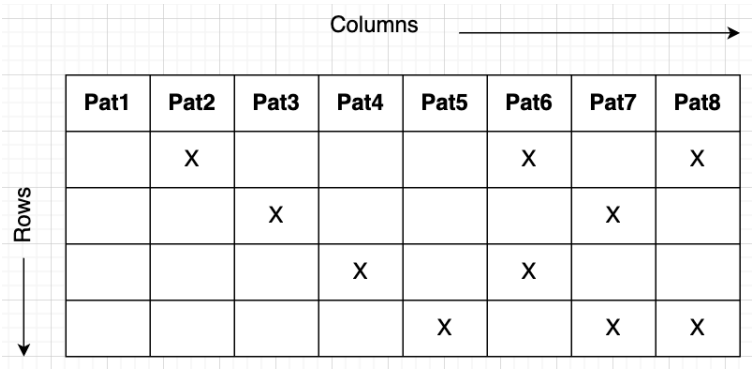

Figure 1: Possible Legal Patterns for Pebbles

	Pat1	Pat2	Pat3	Pat4	Pat5	Pat6	Pat7	Pat8
Pat1	1	1	1	1	1	1	1	1
Pat2	1	0	1	1	1	0	1	0
Pat3	1	1	0	1	1	1	0	1
Pat4	1	1	1	0	1	0	1	1
Pat5	1	1	1	1	0	1	0	0
Pat6	1	0	1	0	1	0	1	0
Pat7	1	1	0	1	0	1	0	0
Pat8	1	0	1	1	0	0	0	0

Figure 2: Compatibility Matrix

In the above fig1, 'X' is the Pebble position, and ' ' is the empty space.

So, we can have 8 possible legal patterns.

Part (b): Using the notions of compatibility and type, give an $\mathcal{O}(n)$-time dynamic programming algorithm for computing an optimal placement.

Solution (b):

Now, let's see how dynamic programming is used to solve this problem:

1. To simplify the problem, we can reduce the size of the checkerboard. Consider the subproblems involving pebbling the first k columns of the checkerboard (where, $1 \leq k \leq n$).
2. Define an array $C_j[i]$ to represent the optimal value of pebbling columns 1 to k, with column having pattern type j ($j \in \{1, 2, \cdots, 8\}$).
3. The base case for this problem is:
   $C_j[0] = 0$ for all j.

4. Create separate arrays for each of the eight pattern types.
5. Compute the isolated scores for each pattern i in each column j ($S_j[i]$).
6. For each pattern i and compatible pattern x, maximize the total score Cj[x] as follows:
   $C_j[x] = C_{j-1}[i] + S_j[x]$
7. Store the pebbling pattern that led to each score ($P_j[x] = i$).
8. By following this approach, we can efficiently find the optimal placement in $\mathcal{O}(n)$ time.

Question 3: Solve the following bipartite graph maximum matching problem using the algorithm in Goddard D. The lower nodes (classrooms) are X, and the upper nodes (courses) are Y. Show the current matching M at each iteration and the augmenting path found each time findAugmentingPath is called. Start the BFS search with a node in X (classrooms), and use the ordering of edges from left to right. For example, the search starting at 60/AM would process the edge to 5004A/25 first.

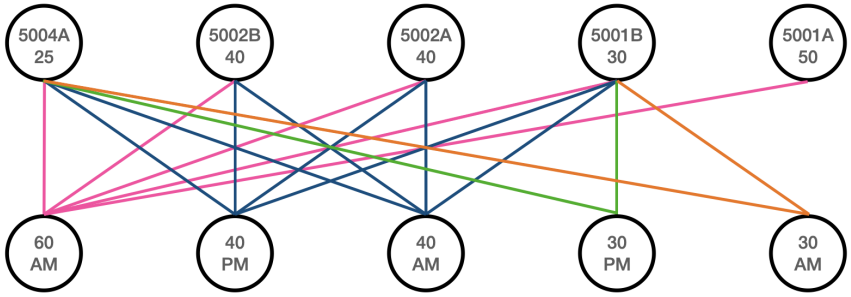

Figure 3: Matching Graph

Solution:

Given Problem Setup:
Classrooms (X):
$60/AM, 40/PM, 40/AM, 30/PM, 30/AM$

Courses (Y):
$5004A/25, 5002B/40, 5002A/40, 5001B/30, 5001A/50$

Edges (Classroom to Courses):
$60/AM \rightarrow 5004A/25, 5002B/40, 5002A/40, 5001B/30, 5001A/50$
$40/PM \rightarrow 5004A/25, 5002B/40, 5002A/40, 5001B/30$
$40/AM \rightarrow 5004A/25, 5002B/40, 5002A/40, 5001B/30$
$30/PM \rightarrow 5004A/25, 5001B/30$
$30/AM \rightarrow 5004A/25, 5001B/30$

Given below are the steps to follow to find the maximum matching in a graph:

1. Initialization: Begin with an empty matching, denoted as M.

2. Find Augmenting Paths: Utilize a Breadth-First Search (BFS) to discover the shortest augmenting path.
3. Repeat: Continue this process until no more augmenting paths can be found.

Iteration Steps:
Let's break down each iteration, identifying augmenting paths and updating the matching M.

Initial Matching $(M = \{\})$

1. **Iteration 1:** Starting with BFS from $60/AM$.
   BFS finds augmenting path: $60/AM \rightarrow 5004A/25$.
   Update Matching (M): $\{(60/AM, 5004A/25)\}$
2. **Iteration 2:** Starting with BFS from $40/PM$.
   BFS finds augmenting path: $40/PM \rightarrow 5002B/40$.
   Update Matching (M): $\{(60/AM, 5004A/25), (40/PM, 5002B/40)\}$
3. **Iteration 3:** Starting with BFS from $40/AM$.
   BFS finds augmenting path: $40/AM \rightarrow 5002A/40$.
   Update Matching (M): $\{(60/AM, 5004A/25), (40/PM, 5002B/40), (40/AM, 5002A/40)\}$
4. **Iteration 4:** Starting with BFS from $30/PM$.
   BFS finds augmenting path: $30/PM \rightarrow 5001B/30$.
   Update Matching (M): $\{(60/AM, 5004A/25), (40/PM, 5002B/40), (40/AM, 5002A/40), (30/PM, 5001B/30)\}$
5. **Iteration 5:** Starting with BFS from $30/AM$.
   BFS finds augmenting path:
   As $30/AM$ has path to $5004A/25$, we are breaking the initially made path from $60/AM$ to $5004A/25$. And connecting the node $60/AM$ to $5001A/50$, as there is a possible alternative path.
   $30/AM \rightarrow 5004A/25$
   $60/AM \rightarrow 5001A/50$
   Update Matching (M): $\{(60/AM, 5001A/50), (40/PM, 5002B/40), (40/AM, 5002A/40), (30/PM, 5001B/30), (30/AM, 500$

Iteration	Current Matching (M)	Augmenting Path
0.	{ }	
1.	{(60/AM, 5004A/25)}	60/AM --> 5004A/25
2.	{(60/AM, 5004A/25), (40/PM, 5002B/40)}	40/PM --> 5002B/40
3.	{(60/AM, 5004A/25), (40/PM, 5002B/40), (40/AM, 5002A/40)}	40/AM --> 5002A/40
4.	{(60/AM, 5004A/25), (40/PM, 5002B/40), (40/AM, 5002A/40), (30/PM, 5001B/30)}	30/PM --> 5001B/30
5.	{(60/AM, 5001A/50), (40/PM, 5002B/40), (40/AM, 5002A/40), (30/PM, 5001B/30), (30/AM, 5004A/25)}	30/AM --> 5004A/25 60/AM --> 5001A/50

Figure 4: Current Matching at Each Step

Therefore, the final maximum matching is:
$(60/AM \rightarrow 5001A/50)$
$(40/PM \rightarrow 5002B/40)$

$(40/AM \rightarrow 5002A/40)$
$(30/PM \rightarrow 5001B/30)$
$(30/AM \rightarrow 5004A/25)$

Reflection:

In this exercise, I gained insights into several key concepts. Firstly, I learned how to find the maximum sum within a contiguous sub-sequence. Secondly, I explored practical pattern-detection techniques using Dynamic Programming. Lastly, I tackled the problem of finding maximum matchings in bipartite graphs.

Acknowledgements:

I would like to express my sincere gratitude to the following individuals and resources for their invaluable contributions to this assignment:
1) Prof. Bruce Maxwell: Thank you, Professor Bruce Maxwell for your guidance in this assignment. Your expertise in algorithms greatly influenced my work.
2) Classmates and TA's: I appreciate the discussions of my classmates, and TA's who clarified my doubts.
3) DPV Algorithms Textbook: This textbook was a useful resource for me to understand the basics of algorithms.
4) https://people.computing.clemson.edu/ goddard/texts/algor/D1.pdf: This document helped me with the understanding of the concept "Maximum Matching in Bipartite Graphs".
5) https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/: This was a useful website to understand how to solve maximum sum sub-sequence problem.