Homework 2
Colorado CSCI 5454

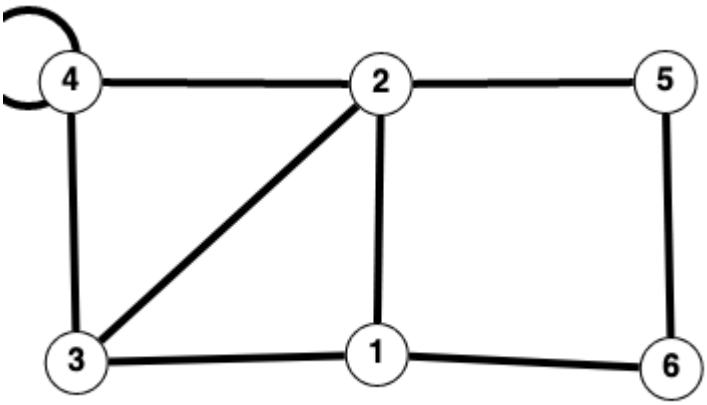
Sai Siddhi Akhilesh Appala

September 6, 2022

People I studied with for this homework: None

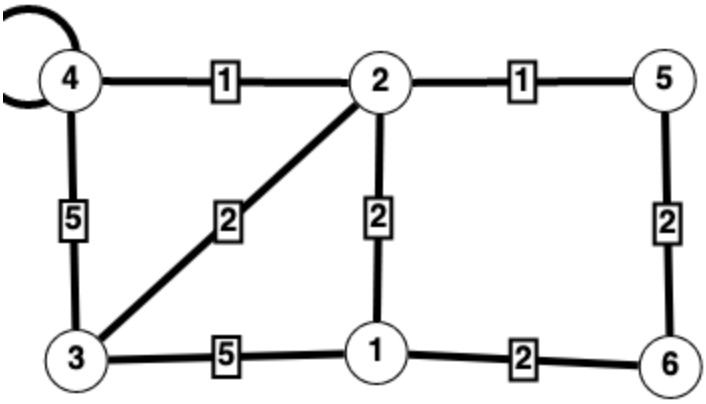
Problem 1

Example 1 :



we use a breadth-first search to identify the shortest routes in graph G from a source vertex to every other vertex. Since each edge had an equal weight (Above example), the shortest path between any two vertices was the one with the less number of edges. But the weighted graph has an edge-to-edge weight difference in the cost of traversing edges. here in the weighted graph, the shortest path definition itself is different. The path with the lowest edge weight sum is said to be the shortest path between two vertices. The path with the fewest edges may not be the shortest if the edges it contains are expensive, hence BFS will not operate on weighted graphs.

Find the below example with an explanation which is the same graph as example 1

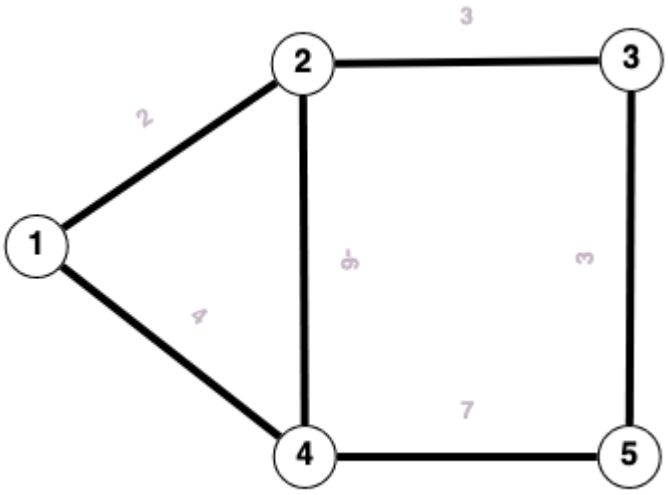


If you observe in the above example if we want to find the shortest path from 3 to 6 For BFS $\longrightarrow$ the path is 3-1-6
as the length is 2 which is the shortest for BFS as it doesn't deal with the weights.

For Dijkstra's $\longrightarrow$ 3-2-5-6
Dijkstra will take the weights into consideration rather than the number of edges. in the above example, we have seen BFS failed but Dijkstra worked for the weighted graph.

Problem 2

Example 1 :



If you observe in the above example if we want to find the shortest path from 1 to 3 we will see both approaches for finding the shortest paths.

For Dijkstra's $\longrightarrow$ 1-2-3
as per algorithms iterations as follows

1. At the starting of iteration all distances are ∞ except starting point as 0. In this iteration we check which one is the lowest and append it to the priority queue, now 2 is inside the priority as it has the lowest value as it has distance[2] = 2. we pull 2 and check for its neighbors now 3 goes into the queue and we assign the distance of the 3 as 5, so, the distance from 1 to 3 is 5 from one path

2. Once 2 is pulled from the priority queue, 4 is pulled from the queue, and update the distance of 2 as 4-6 = -2 which is less than 2. now 2 is not pushed into the queue as it is already visited.

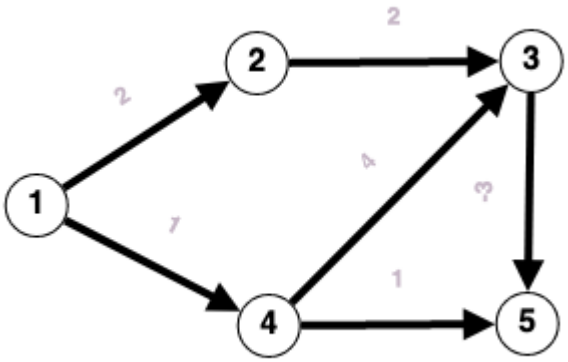
3. But the distance from 1 to 3 will be updated via 4 as 1$\rightarrow$ 4 $\rightarrow$ 2 $\rightarrow$ 3as 4-6+3 = 1$whichislessthanthepreviousone$.

The above is not the actual path if we consider the negative weights.
For Bellman-Ford$\longrightarrow$ 1-4-2-3 $\rightarrow$ which gives the path as 2 as the shortest path.

So, based on the above example Dijkstra's algorithm fails, and bellman-ford succeeds.

Problem 3

Example 1 :



If we take the above example for the given claim. For example, we have 2 paths from 1 to 5 which are
1$\rightarrow$ 2 $\rightarrow$ 3 $\rightarrow$ 5 – path 1 – value = 1
1 $\rightarrow$ 4 $\rightarrow$ 5 – path 2 – value= 2

From the above, the shortest is path 1 Now we are adding a constant let's say 4 to the graph

1$\rightarrow$ 2 $\rightarrow$ 3 $\rightarrow$ 5 – path 1 – value = 13

$1 \rightarrow 4 \rightarrow 5$ – path 2 – value= 10

Now the shortest path has changed from path 1 to path 2, which is false before adding a constant. So we cannot prove the shortest path for negative edges by adding the constants.

Problem 4

0.1 Part a

Question: Explain how to compute the final solution assuming we have solved S[1], . . . , S[n]. Prove your answer.

Solution and Proof :

- Here we will be breaking an array based on the last value in the iteration.
- Now we will iterate over the array and update the S[i] of the array till that point and keep on iterating by comparing with arr[i] and storing s[i]
- we will iterate through the array s[i] and return the maximum of the array which would be the final solution.
- We will be storing the contiguous sum till that point of the array to s[i], in the question it is asked to return the maximum contiguous sum, whereas our s[i] stores the max sum till that point. To which we need to the return maximum of s[i].

0.2 Part b

Please find the following criteria for solving the problem
Base Case: Let's say if there are no elements in the set then the S[0]=0 will be 0 i.e arr[i]= 0, maximum = 0 and if all the elements are negative then, we will return the greater value as it is a corner case.
Inductive case:
$\boldsymbol{S[i] = max(S[i-1] + arr[i], arr[i])}$.

Every time we will be calculating currSum+arr[i] and comparing it with currSum so that we can decide to move forward or stop the algorithm.

0.3 Part C

In the above 2 approaches in part a, solution 1 is the efficient one with iterative dynamic programming.

Iterative Dynamic Programming Approach 1

```
1: Input: An array of length arr[n] elements
2: lets initiate array S[n+1] = 0
3: Set maximum sum as Min_Value(Integer)
4: for Iterate arr[i] starting from 0 to n-1 do
5:    S[i] = max(S[i-1] + arr[i], arr[i]).
6:    if S[i-1] + arr[i]  > arr[i] then
7:       S[i] = S[i-1] + arr[i]// for reconstruction
8:    else
         S[i]= arr[i]
9:    end if
10:   if s[i] < 0 then
11:      S[i] =0 ;
12:   end if
13: end for
14: Reconstruction-maxsum(i)
15: Return i and S[i];
```

0.4 Part d

Search for the maximum in the array s, let's say S[i] from that point keep going backward by subtracting. the elements from right to left till you get 0. The elements will be our sequence.

Algorithm 2 Reconstruction-maxsum(i)

```
1: Input: integer of length n
2: prev: List of n + 1 elements, initialize all to 0.
3: sum = s[i]
4: while sum> 0 do
5:    Sum = sum - A[i];
6:    add A[i] beginning to the list
7:    i- -;
8: end while
9: return list
```

Problem 5

Solution explanation :

Sub-Problem Definition: mtvalue[m+1] is the array with the max profit of a rod with length L.
Computing final solution: mtvalue[m] is the final answer for the rod of given length m

which is in the array.
Recurrence: To come up with an optimal solution which will be decreasing the computations every time and removing repeated computations. so we divide this problem optimal way and apply the same to the other sub-problems. The below points explain how we can solve overlapping sub-problems
1. Base case / initialization: we initialize C[0] = 0, because this is the only possible solution for rod of length 0
2. Inductive case: for the rod length m, ***maxvalue*** = maximum of ***maxvalue*** and Profit[j] + mtvalue[i-j+1]

- The naive approach will generate all the variations of the rod and calculate the highest-priced variation. If we follow such an approach we will be calculating the price of the same length multiple times in the native approach.
- we will be going to use an array that store the intermediate results so that we can pick and calculate the price for the different size of rods. we will pre-compute and store the max value of the different sizes of rod in the array.
- The array contains the max profit for that size of the rod, so it will be easy to calculate for bigger variations.

DP-based approach for finding the max profit 3

```
1: Given an Input array of profits v[] and length of the rod as m
2: create an array for calculating max profits of different lengths less than given m
3: Create an array prev[] for reconstruction and length m+1 and initialize to 0
4: mtvalue[m + 1] − −max table values for different sizes.
5: for Iterate i from 0 to m-1 do
6:    maxvalue = Integer min value
7:    for Iterate j from 0 to i do
8:       maxvalue = maximum of the maxvalue and Profit[j] + mtvalue[i-j+1] – //
9:       calculates the max profit for different sizes of rod and stores the previous value//
         Set prev[i]=j
10:   end for
      mtvalue[i] = maxvalue
11: end for
12: Reconstruction-trace(m)
13: return mtvalue[m];
```

Reconstruction:

- Whenever we find maxprofit while calculating the maximum profit for each size of the rod. We are saving the previous rod length in an array prev[] so that we can trace

Algorithm 4 Reconstruction-trace(m)

```
1: Input: integer of length
2: if prev[m]==m or m==0 then
3:    add n to the list L
4:    return
5: end if
6: Reconstruction-trace(m-prev[m])
7: return list L
```

back and find the way, from where we combined the profits or rod lengths to get max profit for that particular size of the rod. And then we iterate backward by removing the rod lengths from m until it length is equal to 0.

Proof and correctness:

- we have solved 2 main issues here optimal subproblem where we are dividing the problem into substructures. and we are solving the overlapping problem where we removed repeated computations of the same problem by finding the max profit table.
- let's say if we get a rod of length 4, we will be calculating the $\mathbf{2^4}$ variations.
- for 1 : loop from o to 1 , for 2 counting 0 to 2 – (1+1),(2), for 3 – (1+2),(1+1+1),(2+1),(3), for 4 – (1+1+1+1), (1+3), (2+2), (3+1), (1+1+2), (1+2+1), (2+1+1), (4)
- for each size we calculate the max of its subproblems and save it in the array.
- for calculating 1 – it's given as 1, for calculating 2 - we have (1+1),(2) – where (1+1) is calculated from the previous solution. in the same way for 3 - we use values of 2 and 1, for 4 we use the values 3,2,1 of the previous value.
- Finally we use that max profit array and return the requested length.
- TC for this approach is $\boldsymbol{o^2}$