Homework 2
Colorado CSCI 5454

RITHIK KUMAR ATHIGANUR SENTHIL

September 7, 2022

People I studied with for this homework: Ashwin

Other external resources used: Office hours on Sept 1, thursday and https://www.youtube.com/watch?v
AbdulBari

Problem 1 (4 points)

Give an instance of weighted shortest paths on which BFS does not return the correct answer, but Dijkstra's algorithm does. Explain how BFS goes wrong in this case and what it returns.

Solution:

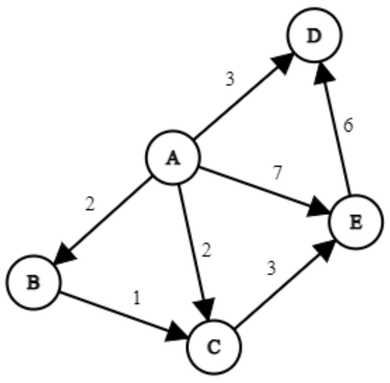

Figure 1: Graph

According to the BFS way of finding the shortest path(least sum of the edges in the path), let's take the case of traversing from vertex A to E, so we first initialize a queue to mark the visited vertices and a distance array distance[5] and set all the values to infinity first.

Now, starting from A, the weight of travelling from A to A would be 0 so set distance[0]=0.

Then push A to the queue because it is visited, now push all the out neighbours of A to the queue and update the distance[] array.
Now, go to the first element of the queue and reach out to it's out neighbours and see if they are already visited, if yes then ignore it otherwise update the weight as:

$$distance[v] = distance[u] + W\text{uv}$$

While travelling from vertex u to v.
When followed this, the BFS will give us the weight of shortest path from A to E as 7, the path is:
A → E
But, the actual total weight of shortest path from A to E is 5, the path is:
A → C → E
But when we follow the Dijkstra's algorithm, we get the weight of shortest path from A to E as 5.
Both Dijkstra's algorithm and BFS would give us the same answer if the weight of each edges are same.
So, this example shows that BFS will go wrong.

Problem 2 (4 points)

Give an instance of weighted shortest paths on which Dijkstra's algorithm does not return the correct answer, but Bellman-Ford does. Explain how Dijkstra's goes wrong in this case and what it returns.

Solution:

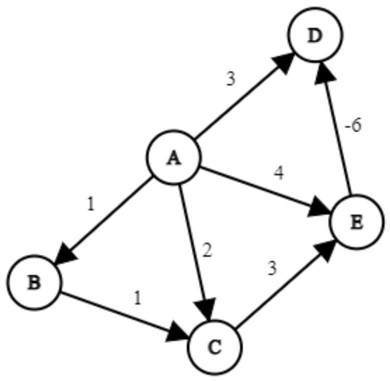

Figure 2: Graph

In case of this graph, we will try to find out the weight of the shortest path from A to all other vertices using Dijkstra's algorithm, so we will initialize the distance[] array as 0 for vertex A and infinity for all other vertices, a priority queue to get the vertex with the shortest distance every time and a queue to mark the visited nodes.
Now we will update the weight of all the vertices connected to A as follows:
distance[A]=0
distance[B]=1
distance[C]=2
distance[D]=3
distance[E]=4

Then we will pop out the vertex with the shortest weight and apply:

$$distance[v] = distance[u] + W\text{uv}$$

While travelling from vertex u to v and update only if distance[v] is lesser than it's previous value.
If a vertex is already visited then we will not try to update the weight of it again.
Now it is B, and the updated weights are:
distance[A]=0
distance[B]=1

distance[C]=2
distance[D]=3
distance[E]=4
Now A and B, both are marked as visited.

The next vertex from the priority queue will be vertex C and the updates weights are:
distance[A]=0
distance[B]=1
distance[C]=2
distance[D]=3
distance[E]=4
Now C is marked as visited.

The next vertex from the priority queue will be vertex D and the updates weights are:
distance[A]=0
distance[B]=1
distance[C]=2
distance[D]=3
distance[E]=4
Now D is marked as visited.

The next vertex from the priority queue will be vertex E and the updates weights are:
distance[A]=0
distance[B]=1
distance[C]=2
distance[D]=3
distance[E]=4
Now E is marked as visited.

The problem with the above algorithm is that the shortest path from A to D is -2 but once we pop out E from the priority queue and go to the vertex D, we cannot modify it because D is already marked as visited and according to Dijkstra's algorithm we cannot modify a vertex which was already visited, so here the algorithm fails.

But if we use Bellman-Ford algorithm to the above graph, we will be able to get the shortest path for A to D as -2 in 4 iterations.

Problem 3 (4 points)

Professor F. Lake suggests the following algorithm for finding the shortest path from node s to node t in a directed graph with some negative edges: add a large constant to each edge weight so that all the weights become positive, then run Dijkstra's algorithm starting at node s, and return the shortest path found to node t. Is this a valid method? Either prove that it works correctly, or give a counterexample.

Solution:

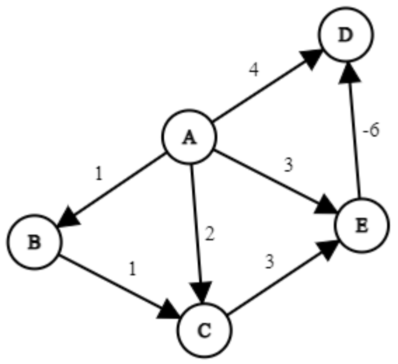

Figure 3: Graph

For this above graph, when we apply Dijkstra's algorithm, the shortest path taken when we try to reach from vertex **A** to **D** is **A** $\rightarrow$ **E** $\rightarrow$ **D** with a weight of -3.
Now according to Professor F.Lake, if we add a constant to all the edges of the graph, here let's add 10 to all the edges, the graph now becomes:

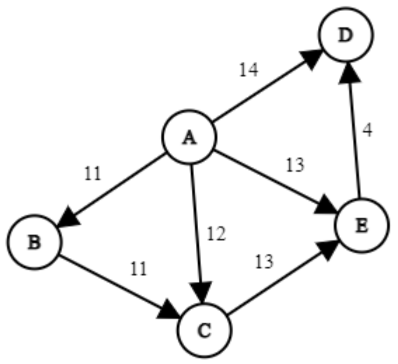

Figure 4: Graph

Now, when we apply Dijkstra's algorithm, the shortest path taken when we try to reach

from vertex **A** to **D** is **A** $\rightarrow$ **D** with a weight of 14.
Now, this is different from the one before.
So, this shows that the claim made by Professor F.Lake is wrong.

Problem 4 (10 points)

In the Max-Sum Contiguous Subsequence problem, we are given a sequence of integers A of length n. A contiguous subsequence from i to j is the sequence A[i], A[i + 1], . . . , A[j]. The problem is to return the maximum sum of any contiguous subsequence. Note: the empty sequence is a valid contiguous subsequence, and its sum is zero. You will consider the following proposed subproblem definition: S[j] = the maximum sum of any contiguous subsequence that ends with element j, inclusive.

0.1 Part a (2 points)

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

Solution:
So, by using Dynamic Programming to solve this problem by following the order of solving the sub problems.
The sub problem here is that any $S[j]$ should give the maximum sum of any contiguous sub sequence that ends with element j, we have to prove this:
Base Case: Let the given array be A and S be the array to store the maximum sum of any contiguous sub sequence till that index.
And $S[1] = A[1]$, given 1 indexing instead of 0.
This is valid because if the size of the array is 1 then it can be the sub sequence itself.
Inductive case: When we apply Dynamic Programming and reach any index j, the maximum sum can be either the element at that index itself or the maximum sum till the $(j-1)th$ index plus that element itself.
if:

$$\text{S[j-1]+A[j]>A[j]} \quad then \quad \text{S[j]=S[j-1]+A[j]}$$

When we follow this, each of the element in array S gives the right answer for maximum sum of contiguous sub sequence till that index.
Hence, the algorithm is correct.

0.2 Part b (2 points)

Give the recurrence for S[j], including base case and recursive case. Prove your answer.

Solution:
Base Case: Let the given array be A and S be the array to store the maximum sum of any contiguous sub sequence till that index.
And $S[1] = A[1]$, given 1 indexing instead of 0.
This is valid because if the size of the array is 1 then it can be the sub sequence itself.
Recursive case: For every element starting from the second element we have,
if:

$$\text{S[j-1]+A[j]>A[j]} \quad \textit{then} \quad \text{S[j]=S[j-1]+A[j]}$$

else:

$$\text{S[j]=A[j]}$$

Since, the maximum sum can be either the element at that index itself or the maximum sum till the $(j-1)th$ index plus that element itself, the above recurrence holds true.

0.3 Part c (2 points)

Give the full DP algorithm for max-sum contiguous subsequence.

Solution:
Following 1 indexing instead of 0:

```
1: Input a non-empty array 'A' of n integers
2: if N = 0, then then
3:     Return 0
4: end if
5: Create an array 'S' of length n and set S[1]=A[1]
6: for i in 2,3,...,n do
7:     Update S[i] = max(A[i],A[i]+S[i-1])
8: end for
9: Return maximum element in array 'S'
```

0.4 Part d (2 points)

Explain briefly how to modify your solution to return the sub sequence itself. A proof is not required.

Solution:

When:

$$S[j-1]+A[j]>A[j]$$

This is satisfied, then will store the index of (j-1) at j in the prev[] array to keep track of the previous elements.

Algorithm 1 Maximum Sum Contiguous sub sequence

```
take input of a non-empty array 'A' of n integers
if n = 0, then then
    return 0
end if
create an array 'S' of length n and set S[1]=A[1]
create an array prev and set prev[i]=NULL for all i
for i in 2,3,...,n do
    if A[i] < A[i]+S[i-1] then
        set S[i] = A[i]+S[i-1])
        prev[i] = i-1
    else
        set S[i] = A[i]
    end if
end for
initialize k as pointer to index (max(S[j]))
return S[k*] along with Reconstruct - Max Sum Contiguous Sub sequence.
```

Algorithm 2 Reconstruct the Maximum Sum Contiguous sub sequence

```
create an empty list S
add A[k] to S
while prev[k] is not NULL do
    set k = prev[k]
    add A[k] to start of S
end while
return S
```

Problem 5 (10 points)

In the Rod Cutting problem, we are given a rod of valuable material of integer length m. We can cut the rod into pieces of integer lenghts. We are given the profit from selling a piece of each length, v[i] for each i = 1, . . . , m. The problem: what is the total profit we can get from cutting up the rod and selling the pieces? And to achieve that profit, how many pieces should we cut, and of what lengths?

Question: Give a dynamic programming solution to Rod Cutting: Identify all of the components of your dynamic programming solution, including reconstruction. Briefly prove correctness.

Solution:

The sub problem for this looks as:

Sub Problem: Form an array S, such that $S[j]$ = the maximum profit we can obtain by selling a rod of length j.

Computing final answer: for every value of index j in S, the value stored is the maximum profit that can be obtained by selling rod of length j, so we can return $s[j$ as the final answer for length j.

Recurrence:

Base Case: Let array B be the input array, then $S[1] = V[1]$, because maximum profit that can be obtained from length 1 is $B[1]$.

Inductive: let n be the length of the given rod, then for every j in S where
2 ≤j≤n, S[j]= max(S[j-i]+S[i],B[j]).

Reconstruction of object:

We will create an array named prev[], which we will use to store values like prev[j]=i, where j is the length of the rod and i is the index of that length of rod which will give us a maximum profit for that length i and prev[j]=j when the whole rod's length is maximum, in order to get the combinations contributing to the maximum profit of length j, we will have to parse the prev[] array till prev[i]=i.

We will try to prove correctness using induction:

Correctness:

S[n], where n is the last element of S will contain the highest profit.

Base case: $S[1]$=$B[1]$, because the highest profit for a rod of length 1 will be B[1].

Inductive: For every index in the array S, we have to compute the combination of different lengths of rods that would give us the maximum profit. When we try to compute the value at index j, we assume that the value of S till the $(j-1)th$ index is already computed and is correct. Now using the previously computed values, we can use:

$S[j] = \max(B[j], S[j-1] + S[i])$

Like this we can obtain the maximum profit for every length of the rod.

Algorithm:

```
 1: take input of an array 'B' of n integers
 2: initialize A[i] = 0, for i = 0,1......,n
 3: set A[1] = B[1]
 4: for i in 2,3,...,n do
 5:     set $maxValue$ = B[i]
 6:     for j in 1,2,3....i do
            $maxValue = max($A[i-j] + A[j],$maxValue)$
 7:     end for
 8:     A[i] = $maxValue$
 9: end for
10: return A[n]
```