Homework 2
Colorado CSCI 5454

Alex Book

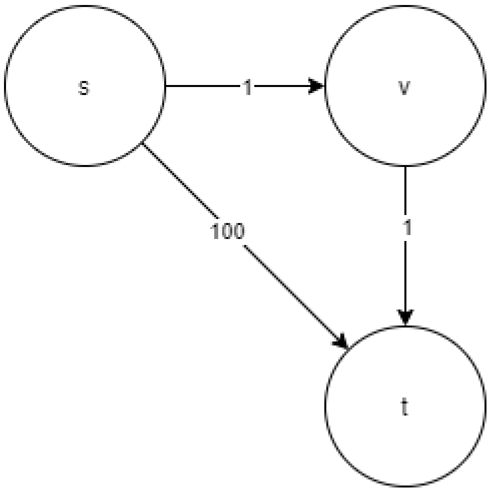
September 7, 2021

People I studied with for this homework: Cole Sturza
Other external resources used: N/A

Problem 1

Part a

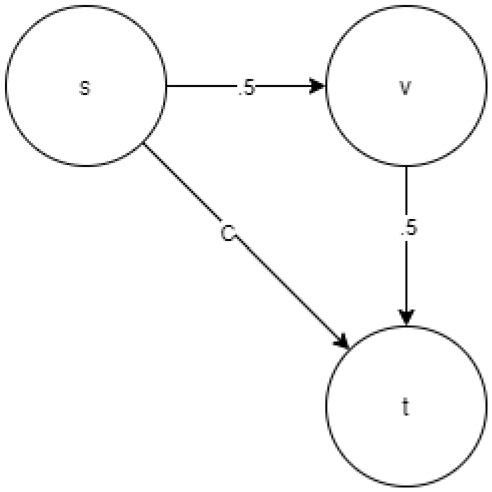
Give a weighted, directed graph G and a pair of vertices s, t such that breadth-first search fails to find a shortest path from s to t.



Breadth-first search on the above graph will always find that the shortest path from s to t to be 100 units long (only containing the edge (s, t)), while the shortest path is actually 2 units long (containing the edges (s, v) and (v, t)). BFS doesn't take into account weights, only finding the shortest path based on number of edges in path.

Part b

Given any constant $C > 1$, give a weighted, directed graph and pair of vertices s, t such that $\frac{\text{BFS length}}{\text{OPT length}} \geq C$.



Similar to what was discussed in part a, BFS will always find the shortest path based on number of edges in path. On the above graph, BFS would find the shortest path from s to t to be C units long (only containing the edge (s, t)), while the shortest path is actually 1 unit long (containing the edges (s, v) and (v, t)). Regardless of the value of C, $\frac{\text{BFS length}}{\text{OPT length}}$ will always have a value equal to C. This follows the same idea as described above. Thus, BFS can be unboundedly worse than the correct/optimal solution in weighted graphs.

Problem 2

Part a

Describe a modified version of the Bellman-Ford algorithm that, given (G, s, t, k) (where the shortest paths from s to other vertices have at most k edges), runs in time $O(k \cdot |E|)$.

Consider the Bellman-Ford algorithm given in the lecture 3 notes. Replace the outer loop's given $n - 1$ with k.

Algorithm 1 Adjusted Bellman-Ford Algorithm

```
1: Input: Weighted directed graph $G = (V, E)$ with weights $w_{uv}$, vertices $s, t$, non-negative
   integer $k$
2: Set dist[$v$] = $\infty$ for all $v$
3: Set dist[$s$] = 0
4: Set prev[$v$] = NULL for all $v$
5: for $k$ iterations do
6:    for each edge $(u, v)$ do
7:       Call UpdateDist$(u, v)$
8:    end for
9: end for
10: Run and return Reconstruct along with dist[$t$]
```

Correctness
It is proven in the lecture notes that after iteration j, we have found and finalized all shortest paths that have at most j edges. This idea directly applies to this problem. Simply replace the outer loop's $n - 1$ iterations with k iterations. The extra $n - 1 - k$ iterations that are being "chopped off" are not needed, as the shortest paths that have at most k edges have all been found after iteration k.

Efficiency
It can be seen that this adjusted Bellman-Ford algorithm runs in $O(k \cdot |E|)$ time, as initialization runs in constant time, the two loops are of k and $|E|$ lengths, respectively, and the inner operations (the call to and operations of UpdateDist) run in constant time.

The space usage is dominated by the arrays *dist* and *prev*, each of which is $O(|V|)$.

Part b

Describe an algorithm that accomplishes the same goal, but does so without being given k.

Once again consider the Bellman-Ford algorithm given in the lecture 3 notes. Imagine that we introduce a flag variable that signifies whether or not a change was made to the *dist* and *prev* arrays during the current iteration of the outer loop. We will the check flag variable after the inner loop terminates. If changes occurred during any iteration of the inner loop, we will exit both loops, proceeding to run and return Reconstruct along with $dist[t]$.

For this idea to be correct, the last changes to *dist* and *prev* must occur on or before the k^{th} iteration. As shown and argued in the lecture notes, we will never increase $dist[v]$ nor set it to a distance shorter than the shortest path. Since we know (also from lecture notes) that after k iterations, all shortest paths with less than or equal to k edges will have been found, we can say with certainty that no changes to *dist* or *prev* will be made after the k^{th} iteration. One additional iteration is needed in order for the flag variable to remain false, then the algorithm will exit the loops and begin reconstruction.

Problem 3

Considering the given max-sum contiguous subsequence problem, I propose the following dynamic programming solution:

Algorithm 2 Max-sum contiguous subsequence

```
Input: Array $A$ with $n$ elements, 1-indexed
Set $M = [0]*(n+1)$, 0-indexed to allow for the base case
for $i = 1...n$ do
    $M[i] = \max(M([i-1] + A[i], A[i])$
end for
Set best_sum = max($M$)
Set best_sum_index = index of best_sum in $M$
// decrementing loop for reconstructing the subsequence
for $j$ = best_sum_index ... 0 do
    best_sum -= $A[j]$
    if best_sum == 0 then
        return $A$[j ... best_sum_index]
    end if
end for
```

Correctness

We prove by induction that $M[j]$ is the sum of the max-sum contiguous subsequence (MSCS) taken from $A[0...j]$ that must include $A[j]$. This will prove correctness because the MSCS must end at some index j, so it is the largest of these.

Base case:
$M[0] = 0$ is correct, because the MSCS of an empty array is the empty array.

Inductive case:
Suppose $M[0], \dots, M[j-1]$ are all correct. Now at index j, one possible subsequence is just the element $A[j]$ itself, which has a sum equal to itself. The other possibility is a subsequence that starts at an earlier index and ends at j. This can only occur if $M[j-1] \geq 0$, and if so, its sum is equal to $M[j-1] + A[j]$ because we appended the element at index j. The algorithm takes the maximum over these two possibilities, so $M[j]$ is correct.

Now if each $M[j]$ is correct, then the algorithm is correct because it returns the sequence ending on the maximum value of $M[j]$ for all j, one of which must be the ending index of the MSCS of the input.

Reconstruction starts at the end of the MSCS and works backward to find the beginning.

It starts with the sum of the MSCS, and subtracts the value of one element at a time until reaching a value of 0, at which point the sum of $A[j \ldots best\_sum\_index]$ must be equal to best_sum.

Efficiency

We claim the running time is $O(n)$ and space use is $O(n)$.

Initialization runs in constant time. The first loop runs in $O(n)$ time, with its inner operations running in constant time. Finding the maximum value of an array that isn't necessarily sorted takes $O(n)$ time. The second loop (reconstruction) runs in $O(n)$ time (worst case). The $O(n)$ operations are run one after another in a non-nested fashion, so the running time remains $O(n)$.

The space usage is dominated by the array M, which is $O(n)$.