Problem Set 5

This problem set is due at **10:00 am** on **Tuesday, March 21st**.

Problem 5- 1: Job assignment

You are given a matrix M such that $M_{(worker,\ task)}$ = amount of time a worker can dedicate to the given task, and a list T where $T[task]$ = amount of time required to complete the task. Multiple workers can be assigned to the same task and each worker can work for a maximum of 10 hours. Decide by reducing to max-flow, if there exists an assignment of workers to tasks such that every task gets completed. Prove that your reduction is correct.
Solution:
Consider the network constructed as follows - first construct a bipartite graph where one side has a vertex for every worker and the other side has a vertex for every task. Connect each worker vertex to every task vertex with an edge of capacity = $M_{(worker,\ task)}$. Introduce a source vertex s and connect s to each worker vertex with an edge of capacity 10, and connect each task vertex to a sink vertex t with an edge of capacity $T[task]$. There exists an assignment of workers to tasks such that every task is completed if and only if the max flow value is equal to $\sum_{task} T[task]$.

Claim: If a satisfying assignment exists, then there is a max flow of value $\sum_{task} T[task]$.
Proof sketch: A satisfying assignment is an assignment of workers to tasks such that (1) Each worker works for a maximum of 10 hours, (2) Each task gets $T[task]$ many hours of attention, (3) Each worker is assigned to a task for at most $M_{(worker,\ task)}$ hours. These three properties allow us to construct a flow by just using the assignment given. This is a max flow because the total flow value is upper bounded by $\sum_{task} T[task]$ because this is the net capacity of the edges coming into the sink vertex t.

Claim: If there is a flow of value $\sum_{task} T[task]$, then there is a satisfying assignment.
Proof sketch: A flow having the value $\sum_{task} T[task]$ will be such that (1) No worker node gets more than 10 units of flow, (2) The flow from any worker vertex to a task vertex is less than or equal to $M_{(worker,\ task)}$, (3) The flow from any task vertex to the sink vertex is at most $T[task]$ units. However we actually have equality in (3) because of the value of the flow. This gives us an assignment of workers to tasks where the amount of time worker w works on task t is given by the flow going through the edge between the vertices corresponding to w and t. This assignment will satisfy the properties of a valid satisfying assignment because of the capacities in the construction of the network.

Problem 5- 2: Broken Keys

You are given an old, worn out keyboard and you would like to calculate the number of keystrokes you have to perform to change one string to another. Because the keyboard is worn out, some keys need to be pressed multiple times for the characters to be printed. Let $N[x]$ denote the number of times the character x needs to be pressed for x to be printed. Write a dynamic program to calculate the minimum number of keystrokes needed to change string1 to string2 given *N*, string1 and string2. You are allowed to delete and enter characters (Note that it might take many keystrokes to enter a character!). Changing the position of the cursor does not cost anything (imagine you can do this with a mouse). Calculate the runtime and prove the correctness of your algorithm.
Solution:

* Delete only cost 1 key press

Recurrence Relation:

Let a be the given string

Let b be the desired string

let a[0 .. 0] and b[0 .. 0] refer to the empty string.

let a[0 .. 1] and b[0 .. 1] refer to the first character of each string a and b, respectively.

Let replace(i, j) = (a[i] != b[j]) * (N[b[j]] + 1)

OPT(i, j) = minimum number of keystrokes to change from a[0 .. i] to b[0 .. j]

OPT(0, 0) = 0; OPT(0, j) = OPT(0, j-1) + N[b[j]] OPT(i, 0) = OPT(i-1) + 1

OPT(i, j) = min(OPT(i, j-1) + N[b[j]], OPT(i-1, j) + 1, OPT(i-1, j-1) + replace(i, j))

Algorithm:

Result: Minimum number of keystrokes to change from string a to string b

$N \leftarrow$ map of char to number of times the key needs to be pressed

MinKeystrokes $\leftarrow |a| \times |b|$ array of minimum keystrokes;
▷ MinKeystrokes[i, j] = minimum number of keystrokes to change from a[0 .. i] to b[0 .. j]

MinKeystrokes[0, 0] = 0;

for *i = 1 ...* $|a|$ **do**
  MinKeystrokes[i, 0] = MinKeystrokes[i - 1, 0] + 1;
end

for *j = 1 ...* $|b|$ **do**
  MinKeystrokes[0, j] = MinKeystrokes[0, j-1] + $N[b[j]]$;
end

for *i = 1 ...* $|a|$ **do**
  for *j = 1 ...* $|b|$ **do**

$$\text{MinKeystrokes}[i,j] = min(\text{MinKeystrokes}[i,j-1] + N[b[j]], \text{MinKeystrokes}(i-1,j) + 1, \text{MinKeystrokes}(i-1,j-1) + \text{replace}(i,j))$$

  end
end
return MinKeystrokes[$|a|$, $|b|$];

Proof:

Claim: MinKeystrokes[i, j] contains the minimum number of keystrokes to change from a[0 .. i] to b[0 .. j]

Proof by Induction

Base Case: i + j = 0, therefore i = j = 0. There no keystrokes are required to change from an empty string to an empty string. Thus MinKeystrokes[0, 0] is correct.

Induction Hypothesis: Suppose the claim is true for all i' and j' such that $i' + j'$ ¡ i + j

Induction Step: Prove the claim for i + j.

For i = 0:

To reach any desired string from an empty string, we can only insert letters. From the induction hypothesis, we know that MinKeystrokes[0, j - 1] holds the correct minimum number of keystrokes. Thus, the number of keystrokes to match a[0 .. 0] to b[0 .. j] must be MinKeystrokes[0, j - 1] + N[b[j]]. This is exactly what our algorithm calculates.

For j = 0:

From any given to an empty string, we can only delete letters. From the induction hypothesis, we know that MinKeystrokes[i - 1, 0] holds the correct minimum number of keystrokes. Thus, the number of keystrokes to match a[0 .. i] to b[0 .. 0] must be MinKeystrokes[i - 1, 0] + 1. This is exactly what our algorithm calculates.

For i != 0 and j != 0:

The possible choices are to delete a character, insert a character, or replace (delete and insert a character).

From the induction hypothesis we know that MinKeystrokes[i - 1, j - 1], MinKeystrokes[i, j - 1], and MinKeystrokes[i - 1, j] already hold the correct minimum number of keystrokes.

The number of keystrokes to match a[0 .. i] to b[0 .. j] if a deletion were performed is MinKeystrokes[i - 1, j] + 1. The delete button costs one keystroke, the string is now the same as a[0 .. i-1]. The minimum cost to match a[0 .. i-1] and b[0 .. j] is given by MinKeystrokes[i - 1, j].

If an insertion were performed, the number of keystrokes is MinKeystrokes[i, j - 1] + N[b[j]]. Entering the b[j] character costs N[b[j]] keystrokes. After the b[j] character is inserted in a, a[i+1] == b[j], so the cost to match all the previous characters in the string must be considered. The previous characters are a[0 .. i] and b[0 .. j - 1], the cost to match time is given by MinKeystrokes[i, j - 1].

If an replacement were to be performed the number of keystrokes is MinKeystrokes[i - 1, j - 1] plus the replacement cost. If a[i] == b[i] the actual replacement is free. If a[i] != b[i], the number of keystrokes of the replacement is a deletion plus an insertion (1 + N[b[j]]).

The overall minimum number of keystrokes to match a[0 .. i] to b[0 .. j] is the minimum of the three allowed choices. This is exactly what our algorithm computes.

Runtime:

$O(|a| \times |b|)$ because we are filling an array of size $|a| \times |b|$ that contains the minimum keystrokes to change a[0 .. i] to b[0 .. j] for all valid i and j. It takes constant time to fill

each cell of the array.

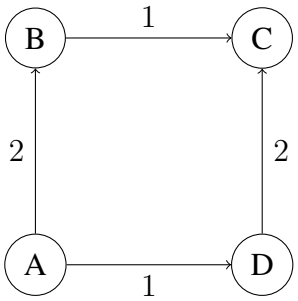
Problem 5- 3: Different paths

Let $G = (V, E)$ be a graph with nonnegative edge weights. Are the shortest paths found by Bellman-Ford the same as the ones found by Dijkstra's algorithm? If yes, prove your result. If no, give a counterexample and specify the different paths found by the algorithms. Assume that there exists an order on E and among the choices made by both algorithms, the edges are picked according to this order. This means that during the course of a run, if Dijkstra / Bellman-Ford have a choice between edges to be picked, they will pick these edges in the prespecified order.

Solution:

Different Paths

Counterexample:



Edge Ordering: (A, B), (B, C), (A, D), (D, C)

Dijkstra:

Starting from A

Iteration 1

Queue: D:1 , B: 2

Iteration 2

Pop D, shortest path is (A, D)

Queue: B:2, C:3

Iteration 3

Pop B, shortest path is (A, B)

C is already in the queue with distance 3, the first one corresponds to coming from D, the second corresponds to coming from B

Queue: C:3, C:3

iteration 4

pop C, shortest path is (A, D) (D, C)

Bellman Ford

Ending in C

$C \quad 0$
$B \quad \infty$
$D \quad \infty$
$A \quad \infty$

Iteration 1

Edge (B, C) and (D, C) will update the graph, in that order.

$C \quad 0$
$B \quad 1$
$D \quad 2$
$A \quad \infty$

Iteration 2

Edge (A, B) and (A, D) will update the graph, in that order.

$C \quad 0$
$B \quad 1$
$D \quad 2$
$A \quad 3$

The shortest path from A to C is (A, B), (B, C) since (A, B) was ordered before (A, D).