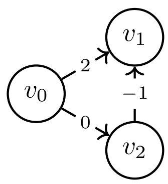
Homework 4 — Dynamic Programming

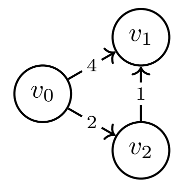
Due: November 17^{th}

A few of these questions may appear on or help you prepare for the quiz on **November 17^{th}**.
Note that these selected solutions and comments do not necessarily represent complete answers.

1. Recall that Dijkstra's algorithm greedily traverses the next lightest path leading to an unexplored vertex. Suppose that a weighted, directed graph $G = (V, E)$ may have negative weights, and consider offsetting them by some constant to produce G' in which all weights are positive. For example, given:



becomes

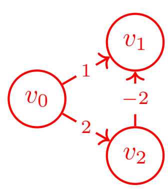


   (a) Give an example showing that SHORTESTPATH may fail to find the shortest paths in G when weights may be negative.

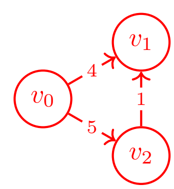
   See below: with negative weights, it is possible that a path that initially appears longer is later shortened by an edge with negative weight. Starting from v_0, SHORTESTPATH computes the path (v_0, v_1) of weight 1, even though the path (v_0, v_2, v_1) then turns out to have weight 0.

   Offsetting weights by a constant affects all edges *equally, but it may not affect all* paths *equally; paths that traverse more edges are offset by more constants, becoming longer than they originally were. After offsetting weights by 3, the path (v_0, v_1) is shorter than the path (v_0, v_2, v_1).*

   (b) Give an example showing that SHORTESTPATH may still fail to find the shortest paths in G via G', because the shortest paths of G' are not necessarily the same as those of G.



becomes



Essentially, starting from s, if the next lightest path ends at t, SHORTESTPATH greedily assumes that no lighter path from s to t will be encountered in the future. Since this assumption no longer holds in the presence of negative weights, it becomes necessary to try all possible edges leading to t:

NAÏVENEGATIVEPATH$(G = (V, E), s, t, k)$

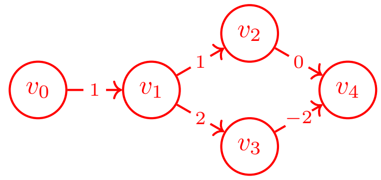
Input: A weighted, directed graph G, a starting vertex s, a target vertex t, and a threshold k, where weights may be negative but cycles may not, and initially $k = |V| - 1$
Output: The distance from s to t, where no path may traverse more than k edges

```
if s = t then
    return 0
else if k = 0 then
    return ∞
else
    let x be NaïveNegativePath(G, s, t, k − 1)
    for all edges e = (v, t) do
        let x be min{x, NaïveNegativePath(G, s, v, k − 1) + w_e}
    return x
```

Provided that no cycle has negative weight (else repeatedly traversing that cycle creates infinitely small distances), the Bellman-Ford algorithm computes paths by observing that there are two ways to make the problem smaller: by changing the target, or by reducing the number of edges traversed.

(c) Give an example showing that NAÏVENEGATIVEPATH may compute the same path multiple times.

Multiple paths can branch off of the same common path, in which case that common path must be computed multiple times. Starting from v_0 and targeting v_4, NAÏVENEGATIVEPATH tries both the path $(v_0, \ldots, v_2, v_4)$ and $(v_0, \ldots, v_3, v_4)$, thereby computing the path (v_0, v_1) twice:



(d) Give a dynamic programming algorithm that optimizes NAÏVENEGATIVEPATH.

NEGATIVEPATH$(G = (V, E), s)$

Input: A weighted, directed graph G and a starting vertex s, where weights may be negative but cycles may not

Output: The distances from s to all vertices

- *Definition:* Let V be indexed such that $s = v_0$, and let $T(i, j)$ be the distance from s to v_i, where no path may traverse more than j edges.
- *Base Cases:* $T(0, j) = 0$, $T(i, 0) = \infty$
- *Formula:* $T(i, j) = \min \begin{Bmatrix} T(i, j-1) \\ \min\limits_{(v_k, v_i) \in E} \{T(k, j-1) + w_{ki}\} \end{Bmatrix}$
- *Solution:* $T(i, |V| - 1)$ for all $v_i \in V$

Note that this algorithm requires $O(|V|^2)$ space, since the number of edges traversed can be capped at $|V| - 1$. A path could not traverse $\geq |V|$ edges without passing through vertices multiple times, so such a path would not be shorter unless there existed a cycle of negative weight.

	0	1	2	3	4
v_0	0	0	0	0	**0**
v_1	∞	1	1	1	**1**
v_2	∞	∞	2	2	**2**
v_3	∞	∞	3	3	**3**
v_4	∞	∞	∞	1	**1**

Although populating a single cell potentially requires considering multiple incident edges, populating every cell in a column collectively considers each edge once. With $|V|$ vertices and $|E|$ edges considered when populating each of the $|V|$ columns, this algorithm thus requires $O((|V| + |E|)|V|)$ time, or $O(|E||V|)$ if the graph is connected.

Further note that this could be optimized to $O(|V|)$ space in practice, as each column depends only on the column to its left: once column j has been populated, column $j - 1$ is no longer needed. In fact, in many presentations of the Bellman-Ford algorithm (often written for those who have not yet studied dynamic programming), the "dist" mapping plays the part of the current column(s).

2. Suppose that you are given a finite set $L = \{l_0, l_1, \ldots, l_{n-1}\}$ of n possible locations for restaurants along a highway, where each location $l_i = (m_i, p_i)$ lies at mile marker m_i with projected profit p_i. Further suppose that opening two restaurants within k miles of each other will decrease profit by $(30-k)^2$.

   (a) Give a dynamic programming algorithm that maximizes the projected profit.

   MaxProfit$(S = (l_0, l_1, \ldots, l_{n-1}))$

   Input: A finite, non-empty sequence S of n locations, where $l_i = (m_i, p_i)$, sorted in increasing order by mile marker

   Output: The maximum projected profit

   - *Definition:* Let $T(i)$ be the maximum projected profit, drawing from $(l_0, l_1, \ldots, l_i)$ and including location l_i.
   - *Base Cases:* $T(0) = p_0$
   - *Formula:* $T(i) = \max\Big\{p_i,\ \max\limits_{0 \le j < i}\Big\{T(j) + p_i - (30 - (m_i - m_j))^2\Big\}\Big\}$
   - *Solution:* $\max\{T\}$

   Given a location l_i, i possibilities exist: each of the locations at which the previous restaurant l_j could have been opened (in which case profit would be decreased by $(30-(m_i-m_j))^2$, where $m_i - m_j$ is the distance between locations l_i and l_j).

   Note that location l_i must *be opened when populating cell i, because computing profit requires knowing where the last location was opened: the above computation would be inaccurate if location l_j was not actually opened.*

   (b) Give the time and space complexities of your algorithm.

   $O(n^2)$ time, $O(n)$ space

3. A *palindrome* is a string that is equal to itself when reversed. For example, within the string `acbabcac`, the substrings `acbabca`, `cbabc`, `bab`, and `cac` are palindromes (along with all of the 1-symbol substrings, which are trivially palindromes), whereas the substring `abca` is not.

   (a) Give a dynamic programming algorithm that finds the longest palindromic substring.

   Given a substring $x = x_i x_{i+1} \ldots x_{j-1} x_j$, two possibilities exist: either $x_i = x_j$ (in which case the substring $x_{i+1} x_{i+2} \ldots x_{j-2} x_{j-1}$ must be a palindrome in order for x to be a palindrome) or $x_i \ne x_j$ (in which case x is not a palindrome).

   Note that this is not *as simple as finding the longest common substring between a string and its reverse, because the string could contain reverses of its own substrings. For example, the longest common substring between `abcdba` and its reverse `abdcba` is `ab`, which is not a palindrome.*

   (b) Give the time and space complexities of your algorithm.

4. The *Damerau-Levenshtein distance* between two strings x and y is the minimum number of substitutions, insertions, deletions, and transpositions required to transform x into y. For example, given $x =$ `abc` and $y =$ `acb`, the Damerau-Levenshtein distance is 1: transpose `b` and `c`.

   (a) Give a dynamic programming algorithm that finds two strings' Damerau-Levenshtein distance.

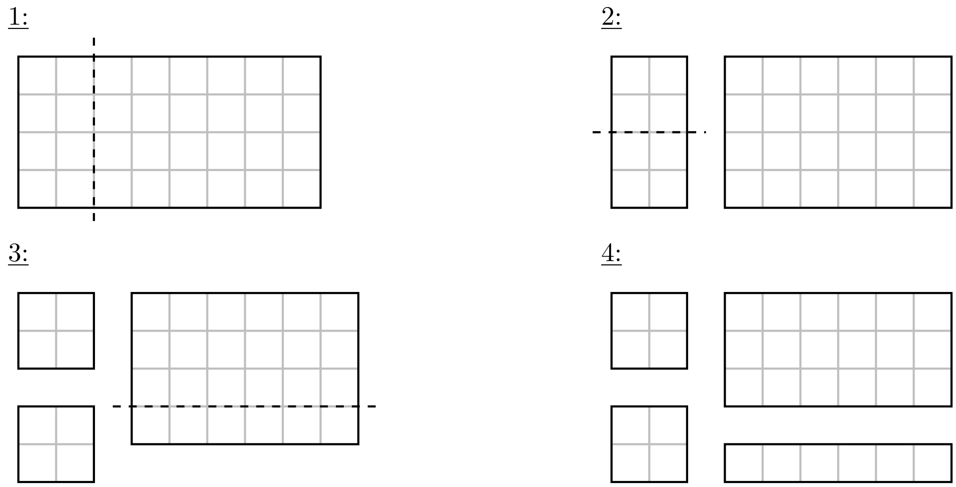
   Given two strings $x = x_0 x_1 \ldots x_i$ and $y = y_0 y_1 \ldots y_j$, there is only one possibility in which it makes sense to perform a transposition in order to transform x into y: if $x_i = y_{j-1}$ and $x_{i-1} = y_j$ (in which case transposing $x_{i-1} x_i$ leaves the task of transforming $x_0 x_1 \ldots x_{i-2}$ into $y_0 y_1 \ldots y_{j-2}$).

   (b) Give the time and space complexities of your algorithm.

5. Suppose that you are given a rectangular chocolate bar consisting of $X \times Y$ individual squares. The bar — or any smaller, rectangular, piece of the bar — can *only* be broken into two pieces along the horizontal or vertical lines separating its squares.

   Further suppose that you are given a finite set $A = \{a_0, a_1, \ldots, a_{n-1}\}$ of n possible products, where each product a_i has a value v_i and requires an unbroken chocolate bar with dimensions $x_i \times y_i$. You may manufacture each product as many times as desired.

   For example, a bar of size 8×4 can be broken up as follows:



   ...producing two bars of size 2×2, one bar of size 6×3, and one bar of size 6×1. These four bars could then be used to manufacture (among others) two products requiring a bar with dimensions 2×2, but none with dimensions 2×4.

   (a) Give a dynamic programming algorithm that maximizes the value of products manufactured.

   Given a product a_i and a bar of dimensions $j \times k$, two possibilities exist: either the product is excluded (in which case this is no different from considering product a_{i-1}), or the product is included (in which case up to two breaks have to be made to isolate a bar of dimensions $x_i \times y_i$, creating up to two smaller unused bars).

   (b) Give the time and space complexities of your algorithm.