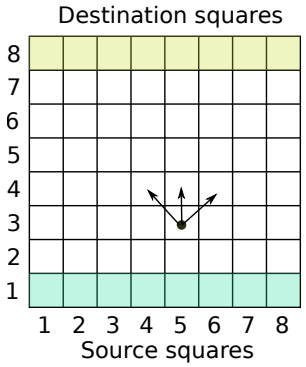
ECE 406, Fall 2020, Assignment 3

Marks: every problem is worth the same, with an equal split across sub-parts. This assignment is worth 50/4 towards your marks in the course.

1. A robot needs to get from a source to a destination in an $n \times n$ grid as shown below.



The robot is initially placed in one of the source squares, which are at the bottom of the grid shown in green. That is, the robot starts off in some square $\langle 1, \cdot \rangle$. The robot needs to end up in one of the destination squares at the top, i.e., at some square $\langle n, \cdot \rangle$. The robot can make only the kinds of moves shown in the picture. Specifically:

- It can move up one square, or diagonally up to the left or right one square each time.
- It cannot fall off the edge of the grid. That is, if the robot is in some square $\langle \cdot, 1 \rangle$, it cannot move diagonally up to the left. Similarly, if the robot is in some square $\langle \cdot, n \rangle$, it cannot move diagonally up to the right.
- There are no other moves. For example, the robot cannot move laterally along a row nor can it take any steps back down.

For every legal move the robot can make, there is a cost, $c[x, y] \in \mathbb{Z}_0^+$, where x, y are squares. That is, c can be perceived as a function $c\colon \{1, n\}^4 \to \mathbb{Z}_0^+ \cup \{\infty\}$, where we use ∞ to identify illegal moves. The domain is $\{1, n\}^4$ because each square is represented as a pair of numbers between 1 and n. E.g., in the grid in the picture above $c[\langle 3, 5 \rangle, \langle 2, 4 \rangle] = \infty$ because that would be a move backwards. Similarly, $c[\langle 3, 5 \rangle, \langle 5, 5 \rangle] = \infty$ because we cannot get to $\langle 5, 5 \rangle$ from $\langle 3, 5 \rangle$ in a single move. But $c[\langle 5, 6 \rangle, \langle 6, 7 \rangle]$ would be some non-negative integer.

Our goal is to identify a sequence of moves of least total cost the robot should take along the grid given as input: (i) n, (ii) the source square, and, (iii) c.

This problem possesses optimal substructure.

(a) Suppose $L[\langle i, j \rangle]$ represents the least total cost of going from the source square to square $\langle i, j \rangle$ in the grid. Write down a recurrence for $L[\langle i, j \rangle]$. You can assume that c is a

function as discussed above, with ∞ for disallowed moves. I have started the recurrence for you.

$$L[\langle i,j\rangle] = \begin{cases} 0 & \text{if } \langle i,j\rangle \text{ is the source square} \\ \infty & \text{if } i = 1 \text{ and } \langle i,j\rangle \text{ is not the source square} \\ \min\limits_{\substack{j' \geq j-1 \\ j' \leq j+1 \\ 1 \leq j' \leq n}} \left\{ \boxed{\textit{fill this part}} \right\} & \text{otherwise} \end{cases}$$

(b) $L[\cdot]$ from Part (a) above gives us the least total cost only. We also want a least cost path. Suppose $T[\langle i,j\rangle]$ represents such a path to get from the source square to $\langle i,j\rangle$. Adopt the mnemonic NIL if no path exists or if $\langle i,j\rangle$ is the source square. Also observe that each entry in $T[\cdot]$ which is not NIL can be represented by a single square $\langle i',j'\rangle$, which is the immediately previous square from which the robot would have moved to the square $\langle i,j\rangle$ in its least-cost path from the source square. That is, $T[\cdot]$ is similar to the prev value in our shortest-paths algorithms.

Write a recurrence for $T[\langle i,j\rangle]$. You can (and should) refer to $L[\cdot]$ in your recurrence. I have started it for you.

$$T[\langle i,j\rangle] = \begin{cases} \text{NIL} & \text{if } L[\langle i,j\rangle] = 0 \text{ or } \infty \\ \boxed{\begin{array}{c}\textit{fill this part} \\ \textit{you may need} \\ \textit{more than one case}\end{array}} & \end{cases}$$

(c) **[python3]** Code your recurrence for $T[\langle i,j\rangle]$ in python3 as a bottom-up algorithm using dynamic programming towards determining a cheapest path for the robot. Your inputs are: (i) n, (ii) the source-square as a list of size two, and, (iii) c encoded as a python3 `dict` data structure. For (iii), the key to the `dict` is a string. The string, in turn, is simply the result of invoking `str()` on a list of two lists, where each inner list is a list of the two integers that represent a square. E.g., to query the `dict` for the cost $c[\langle 3,4\rangle, \langle 4,4\rangle]$, one would use the output of `str()` on the list $[[3,4],[4,4]]$ as the key. The `dict` has an entry for a particular key if and only if the move is valid. There is a skeleton `a3p1c.py` and tester `tester-a3p1c.py` on Learn.

2. Consider Problem 3 from Assignment 2 on driving from A to B during daytime only, with the fewest stops, from amongst a set of acceptable stops, to rest by night. Suppose our n acceptable stops, in sequence starting closest to A, are $s_1, \ldots, s_n$. Think of A as stop s_0, B as stop s_{n+1}.

(a) The problem possesses optimal substructure. What is it? (In a couple of crisp sentences perhaps with some mathematical notation.)

(b) A more basic property than optimality given an instance of this problem is what we can call *feasibility*. Indeed, it does not even make sense to talk of an optimal solution unless the input instance of the problem is feasible.

For this problem, we say that an input $I = \langle s_0, \ldots, s_{n+1}\rangle$ is feasible if there exists a sequence $S = \langle s_{i_0}, s_{i_1}, s_{i_2}, \ldots, s_{i_k}, s_{i_{k+1}}\rangle$, where (i) $s_{i_0} = s_0$, (ii) $s_{i_{k+1}} = s_{n+1}$ and (iii) $i_l < i_{l+1}$ for all l, such that: given any consecutive pair $\langle s_{i_j}, s_{i_{j+1}}\rangle$ in the sequence S, it is possible to drive from s_{i_j} to $s_{i_{j+1}}$ within a day.

Prove:

An instance $s_0, \ldots, s_{n+1}$ of the problem is feasible if and only if for every consecutive pair $\langle s_i, s_{i+1} \rangle$ in the input, we can drive from s_i to s_{i+1} within a day.

Note: not needed for the assignment, but I want to point out the importance of this claim. It becomes really easy, in an algorithm, to check for feasibility. We could do that at the outset, and only then search for an optimal solution. This would be a bit like checking whether an array is already sorted before doing the hard work of sorting it.

(c) Let $m[j]$ represent the fewest stops we have to make to get from s_0 to s_j, s_0 and s_j not included. Write a recurrence for $m[j]$.

Adopt the notation $s \mapsto t$ to represent that we can get from stop s to stop t within a day, and $s \not\mapsto t$ if we cannot. The input $s_0, s_1, \ldots, s_{n+1}$ may not be feasible — use the mnemonic "∞" as the value for any $m[i]$ for which no feasible sequence exists, with the customary assumptions that $\infty >$ any real number, and $r + \infty = \infty$ for any real number r.

(d) Write pseudo-code for an algorithm, NumMinStops$(s_0, \ldots, s_{n+1})$ to compute $m[0, \ldots, n+1]$ and return $m[n+1]$, based on dynamic programming for the recurrence in Part (c) above. You are allowed to use the notation $\mapsto$ and $\not\mapsto$, and the mnemonic ∞ as in Part (c) in your pseudo-code.

Give characterizations for the worst-case time- and space-efficiencies of your algorithm both in $\Theta(\cdot)$ notation, as a function of n, the number of acceptable stops in the input. Justify each briefly.

3. Consider the edit distance problem (see Lecture 7c, Texbook section 6.3). Suppose, we are given as input two strings $x = x_1\, x_2\, \ldots\, x_n$ and $y = y_1\, y_2\, \ldots\, y_m$ where each $s_i, t_i \in$ some alphabet.

(a) In this portion of the problem, we ask: what are lower- and upper-bounds on the alignment length? For the upper-bound, assume that the worst we can do is delete every x_i and insert every y_j.

(b) For this part of the problem, assume $n = m$ for simplicity and simply denote the length of both strings as n. We want to know what the worst-case number of possible alignments is. Your textbook says, in this context: "In general, there are so many possible alignments between two strings that it would be terribly inefficient to search through all of them for the best one."

Prove, assuming $n = m$, that the number of possible alignments between the lower- and upper-bounds of Part (a) is $\Omega(2^n)$.

Hints: for a lower-bound, observe that all the characters from the string x must appear in sequence (perhaps broken-up by "–" in between) in the top row, and all the characters from the string y must appear in sequence in the bottom row. For the upper-bound, just write down the alignment that corresponds to deleting every character in x and inserting every character in y. For Part (b), suppose you consider an alignment where the top and bottom rows are each some length l. Then, you must have $l - n$ "–" characters in each of the top and bottom rows. How many different ways do we have for these "–" characters to appear? The answer to that question tells us how many alignments we have of length l. Now, we need to sum over all possible lengths l, and lower-bound that sum.