John Miller
Shane Barrantes
2016-10-22
CS325 - Group Assignment 2

Group Assignment 2 Report

2016-10-22

John Miller, Shane Barrantes

Problem (1) Suppose P, Q and a real number *l* are given. Let J(P[1 · · m], Q[1 · · n], *l*) be true if and only if the frogs can traverse P and Q with a leash of length *l*. Come up with a recursive relation for J(P[1 · · m], Q[1 · · n], *l*). Argue that your recursive relation is correct.

Pseudocode: Recursive solution for a given leash length

```
J(P, Q, leash):
    P_len ← len(P) - 1
    Q_len ← len(Q) - 1

    if distance(P[P_len], Q[Q_len]) > leash:
        return False

    if P_len = 0 and Q_len = 0:
        return True

    if (P_len > 0 and Q_len > 0) and distance(P[P_len - 1], Q[Q_len - 1]) <= leash:
        return J(P[0...P_len], Q[0...Q_len], leash)

    if P_len > 0 and distance(P[P_len - 1], Q[Q_len]) <= leash:
        return J(P[0...P_len], Q, leash)

    if Q_len > 0 and distance(P[P_len], Q[Q_len - 1]) <= leash:
        return J(P, Q[0...Q_len], leash)

    return False
```

The function *len* returns the length of a given list

```
distance(i, j)
    return ceil(hypot(|i[0] - j[0]|, |i[1] - j[1]|))
```

The function *distance* makes use of two black box algorithms of which the inner workings are irrelevant.
The function *hypot* finds the hypotenuse length given two lengths assumed to be the x and y of a right triangle.
The *ceil* function simply rounds the result up to the nearest integer. We do this because all leashes given are integers.

Proof by induction: Let P and Q represent arbitrary arrays of size n and m respectively and let *l* represent a leash of arbitrary integer length. Assume that for some points i < n and j < m there is a path that exists such that we can reach P and Q without breaking *l*. There are seven cases to consider:

Base Case: P and Q are of length 1 and close enough such that the leash is not broken therefore P[n] and Q[m] have been reached and the leash is intact.

(1) The distance between P[0] and Q[0] is greater than *l*
    (a) There is no viable path.
(2) P[0] and Q[0] have been reached without *l* breaking.
    (a) There is a viable path.
(3) The distance between P[i+1] and Q[j+1] is less than or equal to *l.*
    (a) The index of P and Q move forward.
(4) The distance between P[i+1] and Q[j] is less than or equal to *l.*
    (a) The index of P moves forward.
(5) The distance between P[i] and Q[j+1] is less than or equal to *l.*
    (a) The index of Q moves forward.
(6) No valid case (move) was found.
    (a) The process steps back to the previous step and attempts the next case (move) until finds a viable path to the end or it reaches the initial state and there are no viable cases (moves) left to attempt

Thus, if there is a path P[0] to P[n] and Q[0] to Q[m] such that *l* is not broken, the recursive relation will allow it to be be found

John Miller
Shane Barrantes

Problem (2) Explain why the algorithm of part **(1)** is slow, and how can you speed it up (substantially) using Memoization.

The algorithm for part **(1)** is slow because there are situations where illegitimate paths can be repeated because we don't check if we have walked down a path before and/or if the path itself does not connect to a viable path, instead we just keep trying variations until we get to the ending values or find there is no path via brute force.

By utilizing memoization we would store the paths that we had previously walked down so that we can do a simple check to avoid them altogether if they are not viable instead of re-walking the entire path which could lead to a dead end and would substantially increase our algorithm's run time. This reduction of re-walking paths via memoization leads to a substantially sped up run time compared to part **(1)**.

This solution is implemented (iteratively) and examined further in **(3)**
We are able to "skip" unviable paths by keeping track of the ends of current viable paths and simply starting there.

Problem (3) Change your memoized algorithm of part (2) into an iterative dynamic programming algorithm, and analyze its running time.

Let *P* and *Q* be lists of size *n* and *m* respectively and each list item containing a tuple representing coordinates on matrix. Let *L* be an unordered list of unique leash sizes.

Pseudocode - Memoized dynamic programming solution

```
J(P, Q, l)
    m   len(P) # rows
    n ← len(Q) # columns
    truthTable ← [m] * n # Boolean 2D list initialized to False

    for i in 1...m
        for j in 1...n
            if validMove(i, j, truthTable) and distance(P[i], Q[j]) <= leash
                lastTrue = [i,j]
                truthTable[i][j] ← True
            else
               # row above has T in column to the right of j, skip to the closest one and
              continue
                if lastTrue[0] = (i - 1) and lastTrue[1] > j
                    j ← lastTrue[1]
                    continue
                else
                    break
        if lastTrue[0] < i # Checks to see if there is any T in the current row
            break # if not then there is no viable path

    return truthTable[m][n] # Will be False if there is no viable path
```

The distance function refers to the one defined in problem 1

Psuedocode - Checks if a move is valid given the current state of our truth table and the current position we are attempting a move on

```
validMove(i, j, truthTable)
    valid ← False

    if truthTable[i - 1][j] # if value to left is true
        valid ← True
    if truthTable[i][j - 1] # if value above is true
        valid ← True
    if truthTable[i - 1][j - 1] # if value diagonal is true
        valid ← True

    return valid
```

Running Time Analysis
Without any memoization the runtime of this algorithm would consistently be $O(mn)$. With memoization the average case would be significantly faster than $O(mn)$ however, the worst case scenario would still be $O(mn)$. This scenario would be present if the leash that is given allows every possible move to be made causing every value in the table to fill with true preventing any optimizations from being made.

Problem (4) Now, suppose that you are given a list of lengths L = {1, . . . , t}, instead of just one number. Describe an efficient algorithm to pick the shortest useful leash from L. (Hint: use the algorithm of part (3) as a black box)

Let *P* and *Q* be lists of size *n* and *m* respectively and each list item containing a tuple representing coordinates on matrix. Let *L* be an unordered list of unique leash sizes.

An efficient algorithm to pick the shortest useful leash from L is to sort L and utilize a binary tree to find the smallest possible leash length in L.

```
bestLeash(L, P, Q)
    i ← 0
    leash ← False
    L_length ← len(L)

    while(L_length >= 1)
        L_length ← len(L)
        i ← ceil(L_length / 2)
        result ← J(P, Q, L[i])

        if result = False:
            L ← L[i...L_length]
        else:
            leash ← L[i]
            L ← L[0...i]
        L_length ← (L_length - len(L))
```

The function *J* is a black box defined in problem number 3