CS 170: Efficient Algorithms and Intractable Problems

Spring 2017

●

Homework 6

Due on Tuesday, March 14th, 2017 at 11:59am

●

Solutions by

Ninh DO

25949105

In collaboration with

None

Problem 1: Hacking for Justice (shortcut question)

★★★★★ Level

In an alternate universe, the students of CS170 found a certain problem on HW6 to be extremely difficult. Initially, no one was able to find the solution. However, some subset of the students managed to download the solution PDF to their laptops. These students began to send the PDF to others via email, who then sent the PDF to others, and so on. Eventually, all of the students had the solution PDF. Uh oh!

After much effort, a TA has figured out the full history of the solution PDF sharing, and constructed a directed acyclic graph $G = (V, E)$ to represent it. V represents the students, and an edge E from v_1 to v_2 represents that v_1 sent the solution file to v_2. All sharing is one-way, and you know that there are no cycles.

If the TA could go back in time, and completely block off all communications to/from one student's laptop, which student should be blocked to minimize the number of students who received the PDF? Assume that blocking one person will not cause anyone else to share with more people than they did before. Answer this question in part (c), or try (a) and (b) for inspiration.

One more important detail: the TA notices that for any two different paths from A to B, the two paths differ by at most k vertices, where k is a small number. You can assume k is about equal to $\log(|V| + |E|)$.

A useful bound: the TA also notices that if you take a depth-k BFS search starting at any vertex, you will traverse $O(k)$ vertices in the search, and the BFS search runs in $O(k)$ time. Using this property will make the problem easier.

a. **(10% credit)** Is it always the case that the best student to block is one of the original students who first downloaded the PDF? If so, explain why in 2-3 sentences. If not, give a small counterexample

Solution

b. **(50% credit)** Write a recursive definition for $F(s)$, the number of students that won't have the PDF if you choose to block s.

Solution

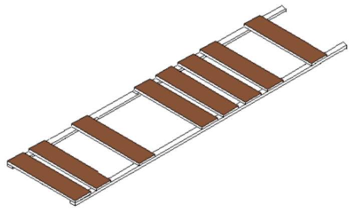
c. **(100% credit)** Design an algorithm to determine which student to block off. Give a 4-part solution.

Solution

Problem 2: Bridge Hop

★★★★ Level

You notice a bridge constructed of a single row of planks. Originally there had been n planks; unfortunately, some of them are now missing, and you're no longer sure if you can make it to the other side. For convenience, you define an array $V[1 \dots n]$ so that $V[i] = \textsc{True}$ iff the ith plank is present. You're at one side of the bridge, standing still; in other words, your *hop length* is 0 planks. Your bridge-hopping skills are as follows: with each hop, you can increase or decrease your hop length by 1, or keep it constant.



For example, the image above has planks at indices $[1, 2, 4, 7, 8, 9, 10, 12]$, and you could get to the other side with the following hops: $[0, 1, 2, 4, 7, 10, 12, 14]$.

Clarifications: You start at location 0, just before the first plank. Arriving at any location greater than n means you've successfully crossed. Due to your winged shoes, there is no maximum hop length. But you can only hop forward (hop length cannot be negative).

Devise an efficient algorithm to determine whether or not you can make it to the other side.

For this problem, you should know how to do the proof of correctness, but need not include it in your submission. You should submit the main idea, pseudocode, and runtime.

Solution

Main idea:

Imagine that we are at the i-th plank of the bridge and we have just made an l-plank hop. We want to check if there is any plank in next (l-1)-plank, l-plank or (l+1)-plank hop. Therefore, if the problem is BridgeHop(l, i) asking if we can make it to the other side from the i-th plank, it turns into three subproblems BridgeHop(l-1, i+l-1]), BridgeHop(l, i+l]) and BridgeHop(l+1, i+l+1]), where V[i+l-1], V[i+l], V[i+l+1] are True's IF these planks exist. If any of these planks does not exist, we just simply omit the subproblem. If all of these three planks do not exist, we return false immediately.

Using dynamic programming, we start with 1-plank hop from the beginning of the bridge and call the function recursively with the current plank position.

Pseudocode:

```
for i in range(len(V), 2*len(V)):
        V.append(True)
for i in range(len(V)):
        for j in range(0, 2*len(V)):
                cache[i][j] = None
print(BridgeHop(0,0))

def BridgeHop(l, i):
        if i >= n+1:
                return True
        else:
                if cache[l][i] != None:
                        return cache[l][i]
                else:
                        ret = V[i] and (BridgeHop(l-1, i+l-1) or \
                            BridgeHop(l, i + l) or BridgeHop(l+1, i+l+1))
```

Solution (cont.)

```
                    cache[l][i] = ret
                    return ret
```

Runtime:

Without using cache, the runtime would be $O(3^n)$ but with using cache, the runtime will depend on the size of cache which is $O(n^2)$

Problem 3: Longest Palindrome Substring

★★★★ Level

A substring is *palindromic* if it is the same whether read left to right or right to left. For example, "bob" and "racecar" are palindromes, but "cat" is not. Devise an algorithm that takes a sequence $x[1 \dots n]$ and returns the length of the longest palindromic substring. Its running time should be $O(n^2)$.

For this problem, you should know how to do the proof of correctness, but need not include it in your submission. You should submit the main idea, pseudocode, and runtime.

Solution

Main idea:

Check both ends of the substring, if they are identical characters, the length of the palindrome P in the string S is len(P in S) = 2 + len(P[1:end-1] in S[1:end-1]), where P[1:end-1] , S[1:end-1] are the substrings of P, S truncated at both ends, respectively. If they are not identical, the palindrome must lie either in S[1:] or S[:end-1], so len(P in S) = max(len(P in S[1:]), len(P in S[:end-1]). We use dynamic programming.

Similar to the previous problem, in order to reduce the runtime we use cache to avoid duplicated calculation.

Pseudocode:

```
for i in range(len(S)):
        for j in range(len(S)):
                cache[i][j] = None

def LenOfPalind(i,j,S):
        if len(S) == 0:
                return 0
        elif len(S) == 1:
                return 1
        else:
                if S[0] == S[end]:
                        if cache[i][j] != None:
                                return cache[i][j]
                        else:
                                ret = 2 + (i+1, j-1, S[1:end-1])
                                cache[i][j] = ret
                                return cache
                else:
                        if cache[i][j] != None:
                                return cache[i][j]
                        else:
                                ret = max(LenOfPalind(i+1,j,S[1:]), \
                                        LenOfPalind(i,j-1,S[:end-1]))
                                cache[i][j] = ret
                                return ret
```

Runtime:

In worst case, the runtime is the time to fill the array cache which is $O(n^2)$

Problem 4: A Sisyphean Task

★★★★ Level

Suppose that you have n boulders, each with a positive integer weight w_i. You'd like to determine if there is any set of boulders that together weigh exactly k pounds. You may want to review the solution to the Knapsack Problem for inspiration.

For this problem, you should know how to do the proof of correctness, but need not include it in your submission. You should submit the main idea, pseudocode, and runtime.

a. Design an algorithm to do this.

Solution

Main idea:

We want to know if there are any number of boulders among n ones summing up to k pounds. Take an arbitrary weight w_i, there are two scenarios:

1. Some boulders among the remaining $n-1$ boulders sum up k pounds.
2. Some boulders among the remaining $n-1$ boulders sum up $k - w_i$ pounds

Thus, the recursive equation looks like:
$Sum(n,k) = Sum(n-1,k)$ *or* $Sum(n-1,k-w_i)$ We use dynamic programming. The main problem will recurse to the base case problems. If k drops below 0, return false. If k == 0, then it will be false if we still have weight(s), or true if we do not have any weight left.

Pseudocode:

```
for i in range(n):
        for j in range(k):
                cache[i][j] = None

def Sum(n,k,W):
        if n == 0:
                if k != 0:
                        cache[n][k] = false
                        return false
                else:
                        cache[n][k] = true
                        return true
        else:
                if k < 0:
                        cache[n][k] = false
                        return false
                for i in range(n):
                        ret = Sum(n-1,k-W[i],W) or Sum(n-1,k,W)
                        cache[n][k] = ret
                        return ret
```

Pseudocode:

Again, the strategy is to use cache to reduce the runtime into the time to populate cache which is O(nk)

b. Is your algorithm polynomial in the *size* of the input? Remember that size is in terms of how many bits we need.

Solution

NO. The algorithm does not mention anything about the size of the input. It is proportional to k which is the magnitude of the input rather than the size of the input which is the number of bits or logarithm of the magnitude.

Problem 5: Advertising Network

★★★★★ Level

You are an advertising network tasked with making bids on Facebook's mobile ad units for your customers. On each day, you make a bid of $\$\theta, \theta \in [0, 1]$, and win that day's auction with probability θ. If you don't win, then you don't spend any money. As per your contract with your customers, you *need* to win at least k out of n auctions that will occur this fiscal year.

An obvious approach may be to bid \$1 for each of the first k days, then nothing for the rest of the $n - k$ days — this strategy will cost you $\$k$. However, it may not be optimal! Consider this alternative for $n = 30, k = 4$:

- Bid \$0.50 for each of the first 26 days, or until you've won 4 auctions.
- Bid \$1.00 for each of the next 4 days, if you didn't win them yet.
- This strategy will have expected cost \$2.00 (round to the nearest cent), and worst case cost \$4.00 – a strictly superior strategy.

Give an optimal strategy to minimize expected cost while maintaining your contract, for any k, n. You only need to explain the main idea; no need for proof, runtime, or pseudocode.

Here are some hints:

- You'll need to use probability; specifically, the linearity of expectation.
- Parameterize your strategy as a set of variables $\theta_{n,k}$, and notice that you need to minimize some function that can be written in terms of $\theta_{n,k}$.
- To actually solve said optimization problem, you can assume you have access to a quadratic program solver that can minimize any quadratic function of a single variable, in time that is efficient.

Solution
Given n days and k bid wins, we have to minimize the expected cost $E[n, k]$. There are two scenarios:

1. We win today with the probability $\theta_{n,k}$, we pay $\theta_{n,k}$. The expected cost becomes $\theta_{n,k} + E[n-1, k-1]$
2. We lose today with the probability $1 - \theta_{n,k}$, we pay nothing. The expected cost becomes $E[n-1, k]$

We come up with the recursive equation for the expected cost:

$$E[n, k] = \theta_{n,k} \left(\theta_{n,k} + E[n-1, k-1]\right) + \left(1 - \theta_{n,k}\right) E[n-1, k] \tag{1}$$

The equation recurses the expected cost $E[n, k]$ to two base cases which are:

$$E[i, i] = i \qquad \text{and} \qquad E[j, 0] = 0 \tag{2}$$

for whatever $i \leq k$ and $j \leq n - k$.
With dynamic programming, the problem $E[n, k]$ turns into the subproblems $E[n-1, k]$ and $E[n-1, k-1]$. When it hits the base cases, we can use the quadratic program solver to minimize the expected cost that takes the form of a quadratic function. In each recursion call, the quadratic program solver is applied to minimize the return until it gets back to the main problem.

Problem 6: Knightmare (extra credit question)

★★★★★★ Level

Give an algorithm to find the number of ways you can place knights on an N by M chessboard such that no two knights can attack each other (there can be any number of knights on the board, including zero knights). The runtime should be $O(2^{3M} \cdot N)$ (or symmetrically, switch the variables).

Note that even though this question is extra credit (and thus only worth 1 pt), the staff *strongly recommends* that you at least attempt it and understand the solution. It will be *very* helpful practice for your upcoming midterm, which we remind you will be Monday March 20, 2017.

Solution