Homework 15

Brian Knotten, Brett Schreiber, Brian Falkenstein

September 30, 2018

23

This problem is a variation of the set partition problem where we are partitioning a set of size n (the n request times) into k subsets (each subset is the collection of requests satisfied if the information is sent at one of the k broadcasts). There are a max of $\binom{n-1}{k-1}$ possible partitions: there are n possible times for each of the k broadcasts (time 2 to time $n+1$), but the final broadcast must go after the last nonzero request time, so there are a max of $n-1$ possible times for each of the $k-1$ broadcasts.
Let B be the list of k broadcasts and let m be the time of the last nonzero request.The algorithm starts by placing each of the k broadcasts at the farthest feasible time i.e.:

```
for broadcast $i = 1$ to $k$:
    place partition i at slot m+2-i
```

so that the first (assigned) broadcast is at time $m+1$, the second broadcast is at time m, etc.
Then: let $minSum = \infty$ and let $bTimes =$ the current positions of the k broadcasts.

```
for broadcasts $i = k$ to 1:
    let $\ell$ be the number of broadcasts sent thus far (i.e. if $i = k$, one sent if $i = k-1$, two sent, etc.)
    for each $\binom{n-i+1}{\ell}$ possible broadcast time:
        if $totalWaitTime \leq minSum$:
            $minSum = totalWaitTime$
            $bTimes =$ current positions of the $k$ broadcasts
```

The first for loop iterates over the k broadcasts. The second for loop iterates over the k broadcasts, and for each loop it considers a max of $\binom{n}{k}$ combinations. Note that due to the symmetry of binomial coefficients, $\binom{n}{k} = \binom{n}{n-k}$. Therefore $\binom{n}{k}$ reaches is max value when $k = \lfloor \frac{n}{2} \rfloor$ or $\lceil \frac{n}{2} \rceil$. Therefore the algorithm is $O(min(n^k, n^{n-k}))$

23

This problem is a variation of the set partition problem where we are partitioning a set of size n (the n request times) into k subsets (each subset is the collection of requests satisfied if the information is sent at one of the k broadcasts). The decision tree for this problem can be thought of as:

- Each node has a branching factor of n and stores the cumulative waiting times given the partitions applied up until that point.
- The children of each node represent placing a partition at that position in R. So, the first child represents placing a partition at R_0, the second child a partition at R_1, etc.
- At depth i, i partitions have been placed.

Thus, the root of the tree would be null, and the first level would contain the waiting times resulting from placing a partition at $R_0 ... R_n$. Note that the sum of the waiting times in solutions that do not broadcast pages to users (that is, there are users requesting the page after the last partition) is ∞, because a valid solution must have every request met.
This will cover all $\binom{n}{k}$ possible ways to partition R k times. This tree can then be pruned and turned into a dynamic programming algorithm with the following rules:

1. If a node has more than k partitions, prune it. (this limits the height of the tree to k).
2. For nodes at the same depth (IE have made the same number of partitions), prune all but the one with the shortest cumulative wait time.

This limits the height of the tree to k, and the number of nodes at each depth to n (the sums must be calculated, and then pruned). The solution would then be found at the leaves, and would be the minimum sum over all the leaves.
This tree can then be imagined as an $n \times k$ table, where the rows represent the number of partitions (the depth of the tree) and the columns represent the places you can place the next partition (the children of each node in the tree). Position $A[i, j]$ can be described as follows:

- The cumulative waiting time if adding the $j'th$ partition at time i.

This algorithm is polynomial, as it takes time nk to traverse the table, and n time to compute the sum at each index, resulting in a runtime of $O(n^3)$.

```
A = [n, k]
Broadcast = [k]                       #array containing the times of the broacasts
for i = 1 to k do:
     Broadcast[i] = -1

for i = n to 1 do:                      #initialize first row. Set all solutions that do not meet all
     if R[i-1] != 0:                       #requests to infinity. IE where to place first partition,
          R[i] = waitTime(i)                 #which must alwaysbe immediately
          Broadcast[1] = i                   #after the last non-zero request
          for j = 1 to i do:
               R[j] = infinity

for j = 2 to k do:                      #iterate over the rest of the possible k partitions
     for i = n to 1 do:
          A[i,j] = waitTime(Broadcast[1..j], i)       #calculate wait time given the previous
                                                       #partitions points,with a new partition at time i
output min(A[:, k])
```

$min(A[:, i])$ can be defined as the minimum value over all columns in the $i'th$ row. $partSum(i...)$ is the sum of wait times of partitioning R at times i.... The time for one partition can be computed as:

$$\sum_{x=1}^{i}(i-x)R_x + \sum_{y=i}^{n}(n-y)R_y$$

This equation would be extended with more summations for further partitions. The runtime of this function is n, as it just sums over the values in the array.

26