CSE 421 HW6

Rohan Mukherjee

May 15, 2024

Problem 1.

Our algorithm is as follows: The above algoritm works because the value returned by the

Algorithm 1 2-Approximation to Bounded Knapsack

```
procedure BK(w, v, W)
    Sort w and v in decreasing order of v_i / w_i
    S ← []
    w' ← 0
    for i = 1 to n do
        if w' < W then
            if w' + w_i ≤ W then
                S.append(i)
                w' ← w' + w_i
            else
                S.append((W − w')/w_i · i)
            end if
        end if
    end for
    if S[−1] is a fraction then
        if v_{S[−1]} ≥ (1/2) Σ_{i∈S} v_i then
            return non fractional variant of S[−1]
        else
            return S.remove(S[−1])
        end if
    end if
end procedure
```

ordinary fractional knapsack is an upper bound for the value of OPT in our case. Then

we see if the fractional term we add is at least 1/2 of the total value in the knapsack, and output that one if that is the case. Otherwise, we throw away the fractional term and output the rest. This is a 2-approximation because either the value of the fractional term in the knapsack is at least half the value of the total knapsack or not, and we output the one with the higher value which is guarenteed to be at least half of the fractional output, which is at least half of the optimal value by the hint. Our algorithm clearly runs in $O(n \log n)$ time since we have to sort and the rest of the for loops take $O(n)$ time.

Problem 2.

Our algorithm is as follows:

Algorithm 2 Biggest Value Rectangle

procedure BVR(A)
  $O \leftarrow n \times n \times n$ tensor of 0
  for $1 \leq j \leq n$ **do**
    for $1 \leq w \leq j$ **do**
      $O[1, j, w] \leftarrow \max\left\{0, \sum_{k=j-w+1}^{j} A[1, k]\right\}$
    end for
  end for
  for $2 \leq i \leq n$ **do**
    for $1 \leq j \leq n$ **do**
      $r \leftarrow 0$
      for $1 \leq w \leq j$ **do**
        $r \leftarrow r + A[i, j - w + 1]$
        $O[i, j, w] \leftarrow \max\{r + O[i - 1, j, w], 0\}$
      end for
    end for
  end for
  return $\max_{i,j,w} O[i, j, w]$
end procedure

Let OPT(i, j, w) be the optimal rectangle of width $w \leq j$, where the height can possibly be 0 (resulting in the empty rectangle), with (i, j) as the bottom right corner. We proceed by induction on i. In the case where $i = 0$, we know have also specified the height to be either 0 or 1, since the bottom right corner of the rectangle has index 1. It then follows immediately that for each j and $1 \leq w \leq j$, we the optimal and only rectangle of width w

with height ≤ 1 is just going to be

$$\max\left\{0, \sum_{k=j-w+1}^{j} A[1,k]\right\}$$

where the first entry in the max is taken if the only rectangle with width w and height 1 with bottom right corner $(1, j)$ has negative weight. Now, notice that the optimal rectangle with width w with (i, j) as the bottom right corner can be found by doing the following. Inductively, OPT$(i-1, j, :)$ holds the right value, so we know that the optimal rectangle of width w, and bottom right corner (i, j) is simply going to be the optimal rectangle of width w and bottom right corner $(i-1, j)$ plus the final row we have added on the bottom, where the optimal rectangle with bottom right corner of $(i-1, j)$ of width w can possibly be empty, or 0 if this sum happens to be negative. Thus we get the following formula, for $1 \leq w \leq j$,

$$\text{OPT}(i, j, w) = \max\left\{\text{OPT}(i-1, j, w) + \sum_{k=j-w+1}^{j} A[i,k], 0\right\}.$$

The last step is to take the maximum over all valid i, j, w, and since the optimal rectangle will have a bottom right corner and a width, this algorithm is correct.

We see that the first nested for loop takes $O(n^3)$ time, because the sum can certainly be found in $O(n)$ time. The second for loop has two inner steps that both take $O(1)$ time, where we save the value of the row sum $\sum_{k=j-w+1}^{j} A[i,k]$ in r and can update it to find the correct row sum for the next w in $O(1)$ time. In this way the triple nested for loop takes only $O(n^3)$ time. The last max is over $\leq n^3$ elements and hence our algorithm runs in $O(n^3)$ time.

Problem 3.

First assign T a root v, and then run the following algorithm, and then extract the first item from the returned tuple:

The below (I do not know how to typset it above) algorithm works by partioning into cases. We first root the tree T with v. Suppose that $\deg v = 3$. Then removing v from T will yield 3 connected components C_1, C_2, C_3 with roots being the neighbors w_1, w_2, w_3 of v. To find the number of connected subsets of T with k elements, we sum over $i + j + m = \ell - 1$ by putting together a path of length i from the first connected component, j from the second and m from the third, which can be done in $B[i]C[j]D[m]$ ways if we store the number of connected subsets with w_i as a root in B, C, D respectively. The last thing is to decree that the number of connected subsets with 0 elements to be 0 so that the multiplication works out. Finally we can keep a running total of the number of k sized connected subset by noticing that the number of k sized connected subset of the entire graph is either going to be a k sized connected subset of one of the connected components, or is going to use v as a bridge between two or more connected components, which we have already calculated and stored in A. The time complexity is complicated once again, but we get something of the following type:

$$T(n) = \sum_{(v,w_i)\in E} T(|C_i|) + O(k \cdot n^3)$$

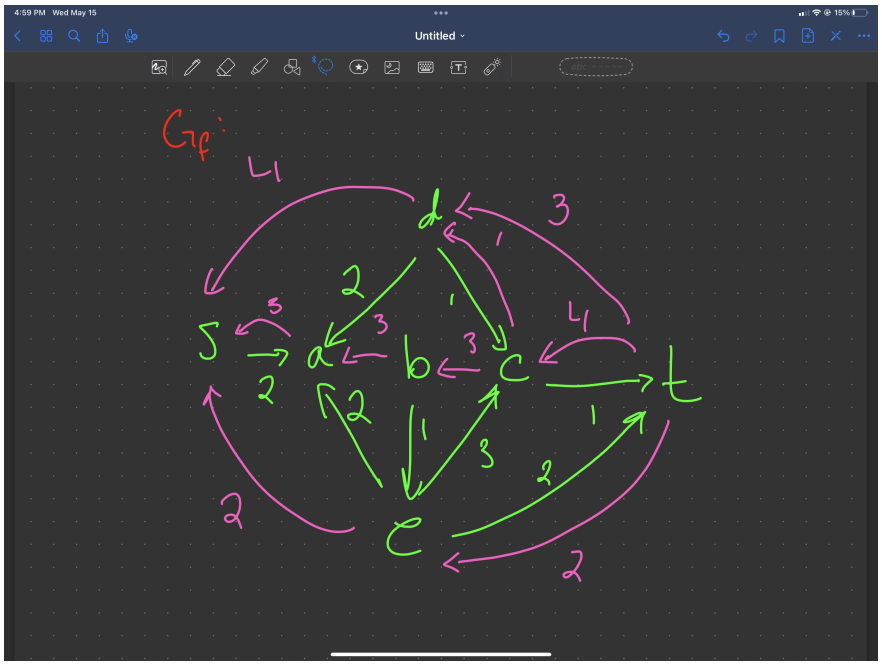
This because the third for loop is easily seen to be the most expensive part of the algorithm, and there are $\binom{n+3-1}{3} = O(n^3)$ loop iterations (by stars and bars), all of which take $O(1)$ time, which is being run inside of a bigger for loop iterating over ℓ which runs through k. Then we call the algorithm recursively on the connected components who have total number of vertices summing to $n - 1$. Thus our time complexity is as above. I have shown on a previous homework that $\sum_{(v,w_i)\in E} T(|C_i|) \leq T(n-1)$ because T is growing at least polynomially, and hence we see that $T(n) \leq T(n-1) + O(n^3)$ in any case which shows that $T(n) = O(k \cdot n^4) \leq O(n^5)$ because $k \leq n$.

Algorithm 3 Number of Connected Subsets

```
procedure Ncs(T, k)
    if T is a single vertex then
        return (1, [1] + [0] * k)
    end if
    v ← root of T
    n ← number of neighbors of v
    A ← [0] * k
    A[0] ← 1
    if deg v = 1 then
        T, B ← Ncs(T − v, k)
        for i = 1 to k do
            A[i] ← B[i − 1]
        end for
        return (T + A[k], A)
    end if
end procedure
if deg v = 2 then
    C_1, C_2 ← connected components of T − v with roots being v's neighbors
    T_1, B ← Ncs(C_1, k)
    T_2, C ← Ncs(C_2, k)
    for ℓ = 1 to k do
        for i + j = ℓ − 1 do
            A[ℓ] ← B[i]C[j]
        end for
    end for
    return (T_1 + T_2 + A[k], A)
end if
if deg v = 3 then
    C_1, C_2, C_3 ← connected components of T − v with roots being v's neighbors
    T_1, B ← Ncs(C_1, k)
    T_2, C ← Ncs(C_2, k)
    T_3, D ← Ncs(C_3, k)
    for ℓ = 1 to k do
        for i + j + m = ℓ − 1 do
            A[ℓ] ← B[i]C[j]D[m]
        end for
    end for
    return (T_1 + T_2 + T_3 + A[k], A)
end if
```

Problem 4.

The residual graph I got is:



The flow and cut I got are: