CSEP521 : Applied Algorithms: Homework 3

Karuna Sagar Krishna

December 4, 2022

Question 1

Idea

As hinted in the question, we'll define $p(i, j)$ as the length of the longest palindrome subsequence between x_i and x_j inclusive. So $p(1, n)$ answers the question by providing the length of the longest palindrome subsequence in the given sequence of characters. To compute $p(i, j)$, we can have the following 2 cases:

1. if $x_i = x_j$, then we can combine this with the longest palindrome from $p(i + 1, j - 1)$
2. if $x_i \neq x_j$, then we have 2 choices - remove the first character - $p(i + 1, j)$ or remove the last character - $p(i, j-1)$ and the maximum of these choices would be $p(i, j)$

We get the following recurrence relation. Finally, to return the longest palindrome subsequence we can look at how $p(1, n)$ was computed and include appropriate characters recursively.

$$p(i, i) = 1$$

$$p(i, i + 1) = \begin{cases} 2 & \text{if x[i] = x[i+1]} \\ 1 & \text{otherwise} \end{cases}$$

$$p(i, j) = max \begin{cases} 2 + p(i + 1, j - 1) & \text{if x[i] = x[j]} \\ max(p(i + 1, j), p(i, j - 1)) & \text{otherwise} \end{cases}$$

Algorithm

```
procedure LongestPalindromeSubsequence(x)
    n = x.length()
    ComputeLPSSubproblems(x)
    return ConstructLPS(x, 1, n, "")
procedure ComputeLPSSubproblems(x)
    for i ∈ {1 ... n} do
        p[i][i] = 1
    for i ∈ {1 ... n − 1} do
        if x[i] = x[i + 1] then
            p[i][i + 1] = 2
        else
            p[i][i + 1] = 1
    for k ∈ {2 ... n − i} do
        for i ∈ {1 ... n} do
            j = i + k
            if x[i] = x[j] then
                option1 = 2 + p[i + 1][j − 1]
            option2 = p[i + 1][j]
            option3 = p[i][j − 1]
            p[i][j] = max(option1, option2, option3)
procedure ConstructLPS(x, i, j, partialLPS)
    if i = j then
        return partialLPS + x[i] + partialLPS.reverse()
    if i + 1 = j then
        if x[i] = x[j] then
            return partialLPS + x[i] + x[j] + partialLPS.reverse()
        else
            return partialLPS + x[i] + partialLPS.reverse()
    if x[i] = x[j] ∧ p[i][j] = 2 + p[i + 1][j − 1] then
        return ConstructLPS(x, i + 1, j − 1, partialLPS + x[i])
    else
        if p[i][j] = p[i + 1][j] then
            return ConstructLPS(x, i + 1, j, partialLPS)
        else
            return ConstructLPS(x, i, j − 1, partialLPS)
```

Correctness

ComputeLPSSubproblems(x) algorithm is the same as the recurrence relation mentioned above. The base case $p[i][i]$ is correct because a single character is a palindrome by itself. The base case $p[i][i+1]$ has two sub-cases - when $x[i] = x[i+1]$ then we have a maximum palindrome of length 2 otherwise we have a maximum palindrome of length 1. Note, the base cases compute all

subproblems of length 1 and 2. The other subproblems are computed in bottom-up fashion - in increasing order of interval length. As defined in the recurrence relation, if $x[i] = x[j]$, then we should consider the palindrome formed by $x[i]$, longest palindrome subsequence between $(i+1, j-1)$ and $x[j]$. Otherwise, we have 3 choices to form a palindrome:

1. consider the longest palindrome subsequence between $(i+1, j)$ i.e. remove i character
2. consider the longest palindrome subsequence between $(i, j-1)$ i.e. remove j character
3. consider the longest palindrome subsequence between $(i+1, j-1)$ i.e. remove both i and j characters

Note, the third choice above is already captured by other 2 choices due to overlapping interval and longest palindrome subsequence is cummulative (there is no negation when interval length is increased). Hence choice 3 can be ignored. This makes $p[i][j]$ equal to maximum of $2+p[i+1][j-1]$ (if $x[i] = x[j]$), $p[i+1][j]$ and $p[i][j-1]$. Note, $p[i][j]$ depends on subproblems of smaller interval length which would have been already computed in previous iterations of outer loop. This proves $ComputeLPSSubproblems(x)$ is correct.

$ConstructLPS(x, i, j, partialLPS)$ is a recursive algorithm used to return one of the longest palindrome subsequence. Roughly, the algorithm does a reverse of $ComputeLPSSubproblems(x)$ i.e. it starts by looking at how $p[1][n]$ ($ConstructLPS(x, 1, n, "")$ is invoked initially) was computed, based on which $partialLPS$ string is updated and recurses on the appropriate subproblem. $partialLPS$ represents the first half of palindrome subsequence captured so far. Since, $p[i][j]$ is computed from 3 options, so we check to figure out which of the options match and recurse on the first matching option (note, we are only returning one of the longest palindrome subsequence). So, if $x[i] = x[j] \wedge p[i][j] = 2 + p[i+1][j-1]$, we recuse on the interval $(i+1, j-1)$ and the $partialLPS$ is updated to capture $x[i]$ character. Otherwise, we recurse on interval $(i+1, j)$ or $(i, j-1)$ accordingly without any changes to $partialLPS$. Note, this corresponds to the recurrence captured by $ComputeLPSSubproblems(x)$. There are 2 base case where $ConstructLPS$ will stop. When $i = j$, we can return the full palindrome as concatenation of $partialLPS$, $x[i]$ and reverse of $partialLPS$ (note, $partialLPS$ represents first half of the palindrome and by definition of palindrome, the second half is the reverse of the first half). The other base case is when $i + 1 = j$, under which if $x[i] = x[j]$, then the full palindrome is concatenation of $partialLPS$, $x[i]$, $x[j]$ and reverse of $partialLPS$, otherwise it is a concatenation of $partialLPS$, $x[i]$ and reverse of $partialLPS$.

Hence $LongestPalindromeSubsequence(x)$ is correct.

Analysis

The running time of $ComputeLPSSubproblems(x)$ is $O(n^2)$ - the first for loop runs in $O(n)$, second loop runs in $O(n)$ and the third loop runs in $O(n^2)$.

The running time of $ConstructLPS(x, i, j, partialLPS)$ is $O(n)$. This is because the longest palindrome subsequence has an upper bound of n and at each recursive call we reduce the interval length by atleast 1 until we reach the base case.

So, the overall running time of $LongestPalindromeSubsequence(x)$ is $O(n^2)$

Question 2

Idea

This problem seems similar to Knapsack problem discussed in class though there are some major differences - we are dealing in 2 dimensions and we can make as many copies of the product. We could think about cutting a rectangle piece of cloth for some product and then recursing on the remaining piece of cloth. However, to make any cut we are told that the cutting machine can cut the entire piece of cloth horizontally or vertically. So we have 3 choice - don't cut, cut horizontally or cut vertically. If we don't cut, then we can make one product which has the highest selling price. If we cut either horizontally or vertically, then we have two smaller rectangular cloth pieces to solve (two smaller subproblems). Since X, Y, a_i and b_i are all positive integers it makes sense to cut only at integer intervals hence we have a finite number of subproblems.

Let $f(i, j)$ represent the maximum price that can be obtained by using a cloth of size $i \times j$, which is computed using the following recurrence. $f(X, Y)$ answers the question i.e. it provides the best return on $X \times Y$ piece of cloth.

Finally, we construct the strategy for cutting that leads to the max price. This is achieved by looking at how $f(i, j)$ was computed to find the appropriate cut and recurse on the appropriate subproblem.

$$f(i,j) = max \begin{cases} max(f(p,j) + f(i-p,j)) & p \in \{1 \dots i-1\} \\ max(f(i,q) + f(i,j-q)) & q \in \{1 \dots j-1\} \\ max(c_r) & (a_r \leq i \wedge b_r \leq j) \vee (a_r \leq j \wedge b_r \leq i) \end{cases}$$

Algorithm

```
procedure MaxPriceStrategy(X, Y, a, b, c)
    n = c.length()
    CalcMaxPrice(X, Y, a, b, c)
    return GetMaxPriceCuts(X, Y)
procedure CalcMaxPrice(X, Y, a, b, c)
    for i ∈ {1 ... X} do
        for j ∈ {1 ... Y} do
            CalcMaxPriceNoCut(i, j)
            CalcMaxPriceHCut(i, j)
            CalcMaxPriceVCut(i, j)
            f[i][j] = max(fn[i][j], fh[i][j], fv[i][j])
    return f[X][Y]
procedure CalcMaxPriceNoCut(i, j)
    m = 0
    for r ∈ {1 ... n} do
        if (a[r] <= i ∧ b[r] <= j) ∨ (a[r] <= j ∧ b[r] <= i) then
            if c[r] > m then
                m = c[r]
                fn[i][j] = (m, r)
    return m
procedure CalcMaxPriceHCut(i, j)
    m = 0
    if j > 1 then
        for q ∈ {1 ... j − 1} do
            if f[i][q] + f[i][j − q] > m then
                m = f[i][q] + f[i][j − q]
                fh[i][j] = (m, q)
    return m
procedure CalcMaxPriceVCut(i, j,)
    m = 0
    if i > 1 then
        for p ∈ {1 ... i − 1} do
            if f[p][j] + f[i − p][j] > m then
                m = f[p][j] + f[i − p][j]
                fv[i][j] = (m, p)
    return m
procedure GetMaxPriceCuts(i, j)
    if f[i][j] = 0 then
        return
    if f[i][j] = fn[i][j].first then
        return (i, j,′ N′, fn[i][j].second)
    if f[i][j] = fh[i][j].first then
        cuts1 = GetMaxPriceCuts(i, fh[i][j].second)
        cuts2 = GetMaxPriceCuts(i, j − fh[i][j].second)
        return (i, j,′ H′, fh[i][j].second) + cuts1 + cuts2
    else
        cuts1 = GetMaxPriceCuts(fv[i][j].second, j)
        cuts2 = GetMaxPriceCuts(i − fv[i][j].second, j)
        return (i, j,′ V′, fv[i][j].second) + cuts1 + cuts2
```

Correctness

Since X, Y, a_i and b_i are all positive integers, we cut the cloth at integer lengths. Suppose we cut at fractional lengths, this would not help us in any way because a_i, b_i are integers so irrespective of how we place these products we cannot utilize the fractional portion of the cloth. So cutting at integer intervals is enough which also limits the number of subproblems.

$CalcMaxPrice$ computes $f[i][j]$ in bottom-up fashion. It begins by computing $f[1][1]$; notice that a cloth of size 1×1 cannot be cut and hence both $CalcMaxPriceHCut$ and $CalcMaxPriceVCut$ are effectively a no-op and returns 0. So $f[1][1]$ is computed based on the product that can fit in 1×1 cloth otherwise it is 0. To compute, $f[i][j]$, $CalcMaxPrice$ explores all 3 options - without cut, with horizontal cut and with vertical cut and takes the maximum of these options.

$CalcMaxPriceNoCut$ is correct and straightforward, since we create exactly one product that can fit. So the algorithm considers the maximum selling price of all products which fit in either orientation - $a_i \times b_i$ or $b_i \times a_i$. We save a tuple in $fn[i][j]$ whose first element indicates the maximum price without any cuts using cloth of dimension $i \times j$ and the second element indicates the product that achieves this. Information in $fn[i][j]$ is used to later construct and describe the cuts.

$CalcMaxPriceHCut$ computes the maximum over all possible horizontal cuts at integer intervals. Consider any horizontal cut, we have two smaller rectangular pieces of cloth so the price we can get from this cut is the sum of maximum price for each of these smaller rectangles. Also, note that the value of f for these smaller rectangles were computed in previous iteration. We save a tuple in $fh[i][j]$ whose first element indicates the maximum price earned using an horizontal cut on cloth $i \times j$ and the second element indicates where the horizontal cut is performed. Similarly, we can prove the correctness of $CalcMaxPriceVCut$.

$GetMaxPriceCuts$ computes the description of the cuts to be performed on cloth using the provided machine. It returns a collection of tuples describing the cut. Each tuple is of the form $(i, j, cut, value)$ and it describes that we perform an operation on cloth $i \times j$ as:

- $cut =' N'$ indicates no cut be made and instead create product $a_{value} \times b_{value}$
- $cut =' H'$ indicates horizontal cut be made at position $value$
- $cut =' V'$ indicates vertical cut be made at position $value$

$GetMaxPriceCuts$ is a recursive algorithm that starts with a cloth $X \times Y$. It looks at $f[i][j]$ to determine how its value was computed by comparing the value of $f[i][j]$ with $fn[i][j]$, $fh[i][j]$ and $fv[i][j]$. This comparison indicates what operation needs to be performed on cloth $i \times j$ and also indicates what are the subproblems. We recursively call $GetMaxPriceCuts$ on the smaller subproblems. This eventually terminates when $f[i][j] = 0$ indicating that the

cloth is too small to create any product or when $f[i][j]$ matches $fn[i][j].first$ indicating that the entire cloth was used to create a single product. Note, it is possible to have different cut strategy achieving the same maximum overall price, $GetMaxPriceCuts$ returns just one such cut strategy.

Analysis

The running time of $CalcMaxPriceNoCut$ is $O(n)$ as we loop over all the products. The running time of $CalcMaxPriceHCut$ is $O(Y)$ as we loop over all integer between 1 and Y. Similarly, running time of $CalcMaxPriceVCut$ is $O(X)$.

$CalcMaxPrice$ performs $O(XY)$ iterations and each iteration runs in $O(X+Y+n)$. The maximum cuts that we can perform is $O(XY)$ i.e. if we cut at every possible integer interval. So $GetMaxPriceCuts$ runs in $O(XY)$ since it returns all the cuts.

So, the overall running time of $MaxPriceStrategy$ is $O(XY(X+Y+n))$

Question 3

Idea

Following the hints provided in the question - for a given bipartite graph we'll construct a flow network exactly as we did in class for maximum matching algorithm and then we'll prove that min cut (which is equivalent max flow and hence equivalent to max matching) provides the min vertex cover. The algorithm to find the min vertex cover for bipartite graph is the same as finding the min cut and only retaining the vertices from bipartite graph G. A min cut $\{S, T\}$ is computed by finding the connected component starting from s in the residual graph when no more augmenting paths can be found. We'll assume $FordFulkerson$ algorithm is provided as we have explored this in class and that it returns the residual graph G_f. As discussed in earlier class, we'll also assume $ConnectedComponent$ algorithm is provided that finds the connected component using BFS in given graph starting from a given vertex.

Algorithm

---

procedure BIPARTITEMINVERTEXCOVER(G)

$G' = \{V', E'\}$

$V' = L \cup R \cup \{s, t\}$

for each (l, r) in E add a directed edge (l, r) with capacity $= \infty$

add directed edge (s, l) for each $l \in L$ with capacity $= 1$

add directed edge (r, t) for each $r \in R$ with capacity $= 1$

$G_f = FordFulkerson(G')$

G_f is the residual graph at the end of max flow algorithm

$S = ConnectedComponent(G_f, s)$

$T = V' - S$

finds connected component starting at given vertex

$C = \{L \cap T\} \cup \{R \cap S\}$

return C

Correctness

Given bipartite graph $G = \{L \cup R, E\}$, we construct a flow network as $G' = \{L \cup R \cup \{s, t\}, E'\}$ exactly the same way we did in class for bipartite max matching problem. So, G' is a flow network representing the original bipartite graph G.

Claim - min cut is equal to the cardinality of min vertex cover. We prove this in 2 parts - min cut produces a vertex cover with cardinality matching the min cut and min vertex cover produces a matching of the same size.

Consider a min cut, clearly it cannot cut any edges between L and R. If it did cut one or more such edges, then by max flow min cut theorem this implies that max flow is infinite. But this cannot be true because the sum of all the outgoing capacities from s is finite. So, the min cut would have to cut some edges between s, L and some edges between R, t. In other words, S consists of s, some vertices in L and some vertices in R while T contains the remaining vertices. As hinted, somehow the min cut $\{S, T\}$ indicates a vertex cover in G. There are few ways to construct a set of vertices given $\{S, T\}$; consider the set of vertices in L that belong to T plus the set of vertices in R that belong to S i.e. $C = \{L \cap T\} \cup \{R \cap S\}$. We can show that this forms a vertex cover in the original bipartite graph. Proving this by contradiction, assume that there is some edge $(l, r) \in E$ that is not covered by C. This implies that both $l \notin C$ and $r \notin C$ which means that $l \notin T$ and $r \notin S$. Since $\{S, T\}$ partitions all the vertices in G' it implies that $l \in S$ and $r \in T$. This implies that edge

(l, r) is cut, but this edge has has infinite capacity and hence cannot be cut (as proved previously). Hence, there cannot be any (l, r) not covered by C and C is a vertex cover. Further, each edge cut is of the form (s, l) or (r, t) where $l \in \{L \cap T\}$ and $r \in \{R \cap S\}$. Also note, there is no common vertex in l and r i.e. $\{L \cap T\} \cap \{R \cap S\} = \emptyset$. If the min cut is of capacity k i.e. it cuts k edges with capacity 1, then sum of all such l and r vertices should be k i.e. $|\{L \cap T\}| + |\{R \cap S\}| = |\{L \cap T\} \cup \{R \cap S\}| = |C| = k$. Hence we've proved that given a min cut of capacity k we can construct a vertex cover of size k.

Consider a min vertex cover C^* in G and a max matching M of G. The max matching of G is obtained by computing the max flow in G' as shown in class. Clearly, C^* should cover all the edges in M. Since M is disjoint set of edges, we need exactly $|M|$ vertices to cover M. And $|E| \geq |M|$ i.e. there can be more edges in G that are not in M implying we might need more vertices to cover these edges not in M. Hence $|C^*| \geq |M|$. We obtained max matching M from max flow algorithm and applying max flow min cut theorem, we can conclude that $|M|$ is equal to the min cut capacity. As shown above, every min cut produces a vertex cover of the same size. Hence, $|C^*|$ should be equal min cut capacity. In other words, a min vertex cover should be of the same size as min cut capacity.

In summary, the algorithm $BipartiteMinVertexCover$ constructs the flow network as we did for max matching problem. It then runs max flow algorithm which returns the residual graph which has no more augmenting paths. We then find the connected component in this residual graph starting from vertex s which gives us the min cut $\{S, T\}$ as we explored in class. From this we construct the vertex cover $C = \{L \cap T\} \cup \{R \cap S\}$ which we proved above is the min vertex cover.

Analysis

Say, G has n vertices and m edges, then G' has $n' = n+2$ vertices and $m' = m+n$ edges. The residual graph of G' has the same number of vertices as G' and can have a maximum of $2m'$ edges as each edge in G' can contribute as a forward and backward edge.

We need $O(m + n)$ time to construct the flow network - we add a directed edge for each of the m edges in G and n additional edges from s to L and R to t.

As discussed in class, $FordFulkerson$ algorithm with capacity scaling to choose good augmenting paths can be implemented in $O(m'^2 \log K) = O((m + n)^2 \log K)$ where K is the sum capacities of edges starting from s in flow network. Since we are using unit capacities, $K \leq n$, so we need $O((m + n)^2 \log n)$ time to compute the residual graph.

$ConnectedComponent$ algorithm runs in $O(m' + n') = O(m + n + n + 2) = O(m+n)$ as it essentially runs BFS on the residual graph. Computing T requires $O(n)$ time.

And finally, to compute vertex cover C we need $O(n)$ time as it involves 3 operations - 2 intersections and 1 union. Each of these operations can be

computed in $O(n)$.

Overall, the $BipartiteMinVertexCover$ algorithm requires $O(m+n)+O((m+n)^2 \log n)+O(m+n)+O(n)+O(n) = O((m+n)^2 \log n)$.

Question 4

maximize

$$1000v_1 + 2000v_2 + 1500v_3$$

subject to

$$v_1 + v_2 + v_3 \leq 30$$
$$2v_1 + 5v_2 + 7v_3 \leq 100$$
$$v_1 \leq 40$$
$$v_2 \leq 20$$
$$v_3 \leq 15$$
$$v_1 \geq 0$$
$$v_2 \geq 0$$
$$v_3 \geq 0$$

Variables v_1, v_2, v_3 represents the volume of each item that our truck can carry. $1000v_1 + 2000v_2 + 1500v_3$ represents the total cost which is the objective function that we want to maximize. All the constraints mentioned in the question are covered by the following:

- $v_1 + v_2 + v_3 \leq 30$ - constraint on the total volume that truck can carry
- $2v_1 + 5v_2 + 7v_3 \leq 100$ - constraint on the total weight that truck can carry
- $v_1 \leq 40$, $v_2 \leq 20$ and $v_3 \leq 15$ - constraint on the maximum available volume for each
- $v_1 \geq 0$, $v_2 \geq 0$ and $v_3 \geq 0$ - constraint that we want each variable to non-negative