CS180 Homework 5

Zhehao Wang 404380075 (Dis 1B)

May 4, 2016

1 Printing minimize space square DP

The algorithm's given in alg 1. The algorithm takes in a list of word lengths l, the maximum number of characters per line M, and returns the minimum sum of cube of extra spaces except the last line, and prints the index of words that should be followed by a line break.

The algorithm first calculates the number of extra space when putting word i to word j in a line, and stores it in *space* array; the algorithm then calculates the corresponding line cost of putting word i to word j in a line, and stores it in *cost*. Finally the algorithm finds the minimum cost using the *cost* array and dynamic programming (the recursion backing this is explained in Correctness section), meanwhile keeping record of the indices of line breaks, and prints such indices using recursion.

Algorithm 1 Minimize space square DP

```
1:  function printSpace(l, M)
2:      space ← [0...0][0...0]
3:      cost ← [0...0][0...0]
4:      for 0 < i ≤ len(l) do
5:          for i < j ≤ len(l) do
6:              space[i][j] = (M − j + i − Σ_{k=i}^{j} l_k)
7:              if space[i][j] < 0 then
8:                  cost[i][j] ← ∞
9:              else if j = len(l) ∧ space[i][j] ≥ 0 then
10:                 cost[i][j] ← 0
11:             else
12:                 cost[i][j] ← space[i][j]^3
        minCost ← [0, ∞...∞] record ← []
13:     for 0 < i ≤ len(l) do
14:         for 0 ≥ j < i do
15:             if minCost[j] + cost[j + 1][i] < minCost[i] then
16:                 minCost[i] ← minCost[j] + cost[j + 1][i]
17:                 record[i] ← j + 1
18:     printResult(record, len(l))
19:     return minCost[len(l)]
20: function printResult(record, index)
21:     if record[index] = 1 then
22:         return
23:     printResult(record, record[index] − 1)
24:     print(record[index])
```

Correctness: this algorithm's correctness is established via the following recursion.

$$minCost[n] = min(minCost[j] + cost[j+1][n])$$

where $0 \leq j < n$, $minCost[n]$ denotes the minimum sum of cube of extra space from character 1 to n, and $cost[m][n]$ denotes the cube of extra space at the end of the line if words m to n are on the same line. The algorithm first calculates the *cost* matrix with respect to the max number of characters per line M, then uses the result to calculate the minimum cost of printing from the start to the end of the given string.

Time and space complexity: time complexity is $O(n^2)$, where n is the number of words. The two sequential nested for loops each will be executed $\frac{n \cdot (n-1)}{2}$ times, thus the overall time complexity is $O(n^2)$.

Space complexity is also $O(n^2)$, as we keep the $n \cdot n$ cost array and extra space array.

2 Longest palindrome substring

The algorithm's given in alg 2. The algorithm takes in string s, and returns the length of the longest palindrome substring.

The algorithm looks at each possible center of the palindrome substring, and find the longest palindrome substring among them.

Algorithm 2 Longest palindrome substring

```
1:  function longestPalindrome(s)
2:      length ← 0
3:      for 0 ≤ i < len(s) do
4:          length ← max(length, getPalindrome(s, 2i + 1), getPalindrome(s, 2i))
5:      return length
6:  function getPalindrome(s, i)
7:      j ← 0
8:      left ← 0
9:      right ← 0
10:     inc ← 0
11:     if i = 2n then
12:         left ← n
13:         right ← n + 1
14:     else if i = 2n − 1 then
15:         left ← n − 1
16:         right ← n + 1
17:         inc ← 1
18:     for 0 < j < len(s) do
19:         if s[left − j] ≠ s[right + j] ∨ left − j < 1 ∨ right + j > len(s) then
20:             return 2j + inc
21:     return 2j + inc
```

Correctness: a palindrome substring only has $2n-1$ possible centers, where n is the length of the string: namely the n characters in the string, and $n-1$ gaps between each two adjacent characters can

be centers. The algorithm starts from each of the possible centers, and expand left and right character by character till the resulting string is no longer a palindrome. The longest palindrome string among the $2n-1$ centers is then returned.

Time complexity: the given algorithm is $O(n^2)$, where n is the length of the string. It tries all n characters in the string as the middle point of the palindrome, as well as all $n-1$ gaps between each two characters in the string. Each trial takes at most n visits to the characters in the string. Thus the total number of execution is $O(n(2n-1))$, which is $O(n^2)$.

Note: Manacher's algorithm solves this problem with $O(n)$ time.

3 Cut string DP

The algorithm's given in alg 3. The algorithm takes in the string S, cut array P, and returns the minimal cost of cutting, and prints the cut points in order.

The algorithm uses dynamic programming: the indices of cut points are first extracted into the array *cutPoints*, and *cost*[*i*][*j*] is used to denote the minimum total cost of cutting all the *cutPoints* between the i^{th} and j^{th} cut points. In the first iteration, the algorithm calculates the *cost* of all pairs of cut points with one cut point in between; using the result from the first iteration, the algorithm then calculatetes the *cost* of all pairs of cut points with two cut point between them, choosing the cut point that yields smaller total cost to cut...and eventually the algorithm gets the *cost* from the start of the array to its end.

The algorithm uses *nextCut* to keep track of the *index* of the cut point that results in smallest total cost when cutting between a *start* and *end* point in the string. The printing algorithm looks up the *index*, and recursively looks up the next cut points by checking *nextCut*[*start*][*index*] and *nextCut*[*index*][*end*], until all cut points are addressed.

Algorithm 3 Cut string DP

```
1:  function CUTSTRING(S, P)
2:      cutPoints ← [0].append(index of 1s in P).append(len(S))
3:      cost ← [0...0][0...0]
4:      nextCut ← [0...0][0...0]
5:      for i = 2..len(cutPoints) − 1 do
6:          for j = 0..len(cutPoints) − 1 − i do
7:              cost[j][j + i] ← cutPoints[j + i] − cutPoints[j]
8:              for k = 1..i − 1 do
9:                  min_index, min_value ← min(cost[j][j + k] + cost[j + k][j + i])
10:             cost[j][j + i] ← cost[j][j + i] + min_value
11:             nextCut[j][j + i] ← min_index
12:     printResult(cutPoints, nextCut, 0, len(cutPoints))
13:     return cost[0][len(cutPoints)]
14: function PRINTRESULT(cutPoints, nextCut, start, end)
15:     cut ← nextCut[start][end]
16:     if cutPoints[cut] ≠ 0 then
17:         print(cutPoints[cut])
18:     if cut = 0 then
19:         return
20:     printResult(cutPoints, nextCut, start, cut)
21:     printResult(cutPoints, nextCut, cut, end)
```

Correctness: the algorithm's correctness is established via the following recursion:

$$cost(s,t) = \begin{cases} min(cost(s,t') + cost(t',t) + (t-s), t > s \\ 0, t = s \end{cases}$$

where s and t are starting and ending indices in the string, and t' iterates over the cut points between s and t. With this recursion, we build the algorithm from bottom-up: cost of starts and ends with no cut points in between are assigned first, then cost of starts and ends with one cut point in between, and eventually the cost of cutting the entire string.

Time complexity: with the specific description above, this algorithm is $O(m^3)$, where m is the number of cut points in the array; which makes the algorithm $O(n^3)$ at worst, where n is the number of characters in the array.

The triple nested for loops takes $O(n^3)$, while printing the cut points takes $O(n)$ as each cut point is visited exactly once in the entire recursion. Thus the algorithm is $O(n^3)$.

4 Job scheduling DP

(a) Prove that there exists an optimal schedule in which jobs are ordered by their deadlines increasing.

Proof: given jobs $j_1 ... j_m$, assume we have an optimal schedule $S = j_{w_1} ... j_{w_n}$, and the scheduled jobs have deadlines $d_{w_1} ... d_{w_n}$, and execution times $t_{w_1} ... t_{w_n}$.

If $d_{w_1} ... d_{w_n}$ is increasingly ordered, then we have the conclusion.

If not, there exists some w_k such that $d_{w_k} > d_{w_{k+1}}$; since j_{w_k} and $j_{w_{k+1}}$ are both in schedule S, we have $\Sigma_{i=1}^{k-1} t_{w_i} + t_{w_k} \leq \Sigma_{i=1}^{k-1} t_{w_i} + t_{w_k} + t_{w_{k+1}} \leq d_{w_{k+1}} < d_{w_k}$. And we have $d_{w_k} > \Sigma_{i=1}^{k-1} t_{w_i} + t_{w_k}$ and $d_{w_{k+1}} \geq \Sigma_{i=1}^{k-1} t_{w_i} + t_{w_k} + t_{w_{k+1}}$. Thus we can schedule S' which is S with the sequence of jobs w_k and w_{k+1} reversed, and S' is no worse than S. Switching each adjacent pair satisfying $d_{w_k} > d_{w_{k+1}}$ in the optimal S, we have a schedule S'' which is optimal, and the jobs in S'' are ordered by increasing deadline.

Summarizing the two cases, we have the conclusion that there exists an optimal schedule whose jobs are ordered by their deadlines increasing.

(b) Algorithm is given in alg 4. It takes in deadline array d and execution time array t, and returns the schedule that has the maximum amount of jobs, and prints the jobs in the schedule. As proved in (a) there exists an optimal schedule whose jobs are ordered by increasing deadline, we pass in a list of jobs ordered by deadline to this algorithm. Its idea is not unlike the solution for knapsack problem (building from bottom-up, for each job and time, see if the job can be scheduled or not; and if it can be scheduled, see if scheduling it or not results in a larger schedule.).

Algorithm 4 Job scheduling DP

```
1: function scheduleJob(d, t)
2:     schedule ← [0...0][0...0]
3:     D ← max(d)
4:     for t[0] ≤ i < D do
5:         schedule[1][i] = 1
6:     for 0 < i < len(d) do
7:         for 0 ≤ j < D do
8:             time ← min(j, d[i]) − t[i]
9:             if time < 0 then
10:                schedule[i][j] = schedule[i − 1][j]
11:            else
12:                schedule[i][j] = max(schedule[i − 1][j], 1 + schedule[i − 1][time])
13:    printResult(len(d), D, schedule, d, t)
14:    return schedule[len(d)][D]
15: function printResult(i, j, schedule, d, t)
16:    if i ≤ 0 then
17:        return
18:    if schedule[i][j] = schedule[i − 1][j] then
19:        printResult(i − 1, j, schedule, d, t)
20:    else
21:        time ← min(j, d[i]) − t[i]
22:        printResult(i − 1, time, schedule, d, t)
23:        print(i)
```

Correctness: the algorithm's correctness is established via the following recursion:

$$schedule(n,t) = \begin{cases} schedule(n-1,t), min(d_n,t) < t_n \\ max(schedule(n-1,t), schedule(n-1, min(d_n,t) - t_n) + 1), min(d_n,t) \geq t_n \end{cases}$$

where n is the job we are considering (we've already considered the list which consists of $job_1..job_{n-1}$), and t is the remaining time. If job i's deadline cannot be met, we have the first branch of the recursion; otherwise the recursion chooses the larger schedule among the ones with or without the job i. Thus $schedule(len(jobs), D)$ gives the max schedule among all the jobs.

Time complexity: this algorithm is $O(nD)$, where n is the number of jobs, and D is maximum deadline among the jobs. The most costly part of the program is the nested loop in which the inner loop runs D times, and the outer loop runs n times (similar with 0-1 knapsack). Thus the complexity is $O(nD)$.