CSE 417 Algorithms

Lecture 21, Autumn 2020
Dynamic Programming
Subset Sum etc.

Announcements

- Final exam, Monday, December 14
  - Tentatively, 24 hour take-home exam

Reading

- Dynamic Programming Examples:
  - 6.1-6.2, Weighted Interval Scheduling
  - 6.3 Segmented Least Squares
  - 6.4 Knapsack and Subset Sum
  - Exercises: Billboard placement, Paragraphing
  - 6.6 String Alignment
    - 6.7* String Alignment in linear space
  - 6.8 Shortest Paths (again)
  - 6.9 Negative cost cycles
    - How to make an infinite amount of money

Subset Sum Problem

- Given integers $\{w_1,\ldots,w_n\}$ and an integer K
- Find a subset that is as large as possible that does not exceed K
- Opt[j, K] the largest subset of $\{w_1, \ldots, w_j\}$ that sums to at most K
- Opt[j, K] = max(Opt[j – 1, K], Opt[j – 1, K – w_j] + w_j)

Subset Sum Grid

Opt[j, K] = max(Opt[j – 1, K], Opt[j – 1, K – w_j] + w_j)

4	0	2	2	4	4	6	7	7	9	10	11	12	13	14	14	16	17
3	0	2	2	4	4	6	7	7	9	9	11	11	13	13	13	13	13
2	0	2	2	4	4	6	6	6	6	6	6	6	6	6	6	6	6
1	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

{2, 4, 7, 10}

Subset Sum Code

```
for j = 1 to n
    for k = 1 to W
        Opt[j, k] = max(Opt[j-1, k], Opt[j-1, k-wj] + wj)
```

Knapsack Problem

- Items have weights and values
- The problem is to maximize total value subject to a bound on weght
- Items $\{I_1, I_2, \ldots I_n\}$
  - Weights $\{w_1, w_2, \ldots, w_n\}$
  - Values $\{v_1, v_2, \ldots, v_n\}$
  - Bound K
- Find set S of indices to:
  - Maximize $\sum_{i \in S} v_i$ such that $\sum_{i \in S} w_i <= K$

Knapsack Recurrence

Subset Sum Recurrence:

Opt[j, K] = max(Opt[j – 1, K], Opt[j – 1, K – w_j] + w_j)

Knapsack Recurrence:

Knapsack Grid

Opt[j, K] = max(Opt[j – 1, K], Opt[j – 1, K – w_j] + v_j)

4																
3																
2																
1																
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Weights {2, 4, 7, 10} Values: {3, 5, 9, 16}

Knapsack Grid

Opt[j, K] = max(Opt[j – 1, K], Opt[j – 1, K – w_j] + v_j)

4	0	3	3	5	5	8	9	9	12	16	16	18	18	21	21	24	25
3	0	3	3	5	5	8	9	9	12	12	14	14	17	17	17	17	17
2	0	3	3	5	5	8	8	8	8	8	8	8	8	8	8	8	8
1	0	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Weights {2, 4, 7, 10} Values: {3, 5, 9, 16}

Alternate approach for Subset Sum

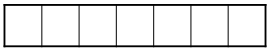
- Alternate formulation of Subset Sum dynamic programming algorithm
  - Sum[i, K] = true if there is a subset of $\{w_1, \ldots w_i\}$ that sums to exactly K, false otherwise
  - Sum [i, K] = Sum [i -1, K] **OR** Sum[i - 1, K - w_i]
  - Sum [0, 0] = true; Sum[i, 0] = false for i != 0
- To allow for negative numbers, we need to fill in the array between K_{min} and K_{max}

Run time for Subset Sum

- With n items and target sum K, the run time is O(nK)
- If K is 1,000,000,000,000,000,000,000,000 this is very slow
- Alternate brute force algorithm: examine all subsets: $O(n2^n)$

One dimensional dynamic programming: Interval scheduling

Opt[j] = max (Opt[j – 1], w_j + Opt[p[j]])



Billboard Placement

- Maximize income in placing billboards
  - $b_i = (p_i, v_i)$, v_i: value of placing billboard at position p_i
- Constraint:
  - At most one billboard every five miles
- Example
  - {(6,5), (8,6), (12, 5), (14, 1)}

Design a Dynamic Programming Algorithm for Billboard Placement

- Compute Opt[1], Opt[2], . . ., Opt[n]
- What is Opt[k]?

Input b_1, …, b_n, where $b_i = (p_i, v_i)$, position and value of billboard i

Opt[k] = fun(Opt[0],…,Opt[k-1])

- How is the solution determined from sub problems?

Input b_1, …, b_n, where $b_i = (p_i, v_i)$, position and value of billboard i

Solution

```
j = 0; // j is five miles behind the current position
       // the last valid location for a billboard, if one placed at P[k]
for k := 1 to n
       while (P[ j ] < P[ k ] – 5)
              j := j + 1;
       j := j – 1;
       Opt[ k ] := Max(Opt[ k-1 ] , V[ k ] + Opt[ j ]);
```

Two dimensional dynamic programming

Subset sum and knapsack

Opt[j, K] = max(Opt[j – 1, K], Opt[j – 1, K – w_j] + w_j)

Opt[j, K] = max(Opt[j – 1, K], Opt[j – 1, K – w_j] + v_j)

4	0															
3	0															
2	0															
1	0															
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Optimal line breaking

Element distinctness has been a particular focus of lower bound analysis. The first time-space tradeoff lower bounds for the problem apply to structured algorithms. Borodin et al. [13] gave a time-space tradeoff lower bound for computing ED on *comparison* branching programs of $T \in \Omega(n^{3/2}/S^{1/2})$ and, since $S \geq \log_2 n$, $T \in \Omega(n^{3/2}\sqrt{\log n}/S)$. Yao [32] improved this to a near-optimal $T \in \Omega(n^{2-\epsilon(n)}/S)$, where $\epsilon(n) = 5/(\ln n)^{1/2}$. Since these lower bounds apply to the average case for randomly ordered inputs, by Yao's lemma, they also apply to randomized comparison branching programs. These bounds also trivially apply to all frequency moments since, for $k \neq 1$, $ED(x) = n$ iff $F_k(x) = n$. This near-quadratic lower bound seemed to suggest that the complexity of ED and F_k should closely track that of sorting.

Optimal Line Breaking

- Words have length w_i, line length L
- Penalty related to white space or overflow of the line
  - Quadratic measure often used
- Pen(i, j): Penalty for putting w_i, $w_{i+1}, \ldots, w_j$ on the same line
- Opt[k, m]: minimum penalty for ending line k with w_m

Optimal Line Breaking

Optimal score for ending line k on w_m

$\mathrm{Opt}_k[m] = \min_i \{ \mathrm{Opt}_{k-1}[\, i \,] + \mathrm{Pen}(i+1,m)\}$ for $0 < i < m$

4																	
3																	
2																	
1																	
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0