Dynamic Programming - II

Elements of Dynamic Programming

Instructor: Ashok Singh Sairam

Lecture Plan

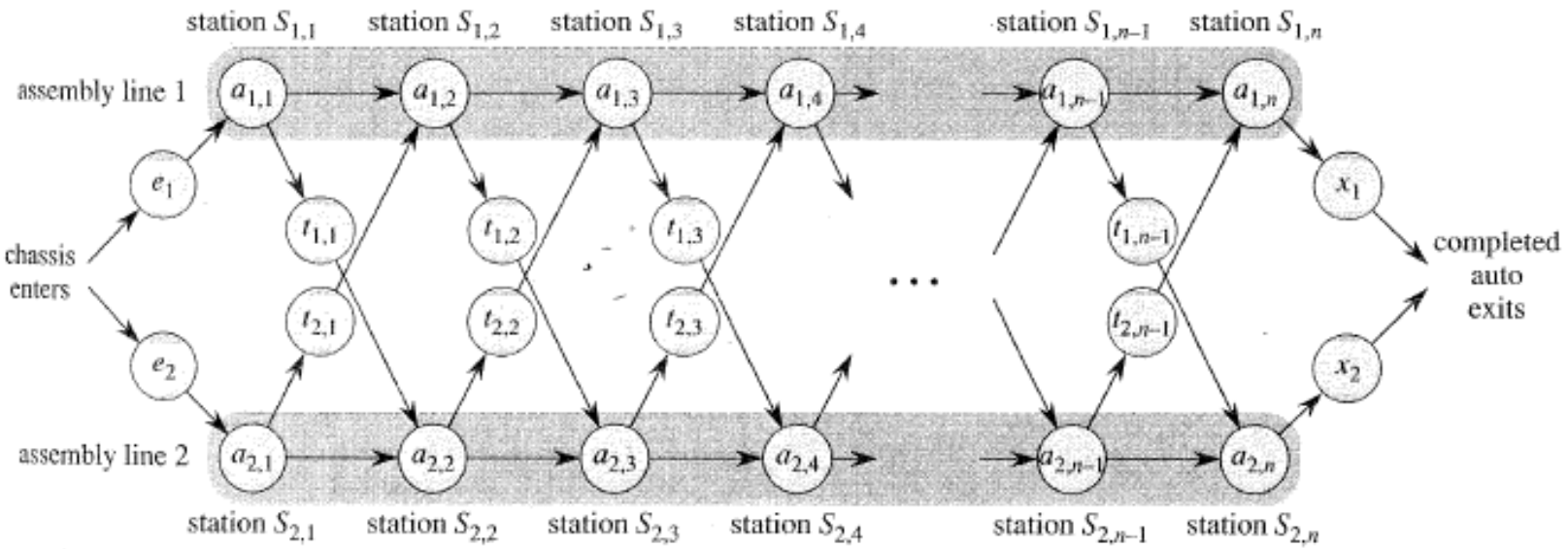
- When we should use dynamic programming?
  - Optimal substructure
  - Overlapping subproblems
- Memoization

Optimal Substructure

- Optimal substructure: Optimal solution contains within it optimal solutions to sub problems.
  - Ex: In matrix-chain multiplication optimally doing $A_1, A_2, A_3, \ldots, A_n$ required $A_{1\ldots k}$ and $A_{k+1\ldots n}$ to be optimal.
  - Ex: Assembly line scheduling, Find the fastest way through $S_{1,j}$ contains fastest way through $S_{1,j-1}$ or $S_{2,j-1}$

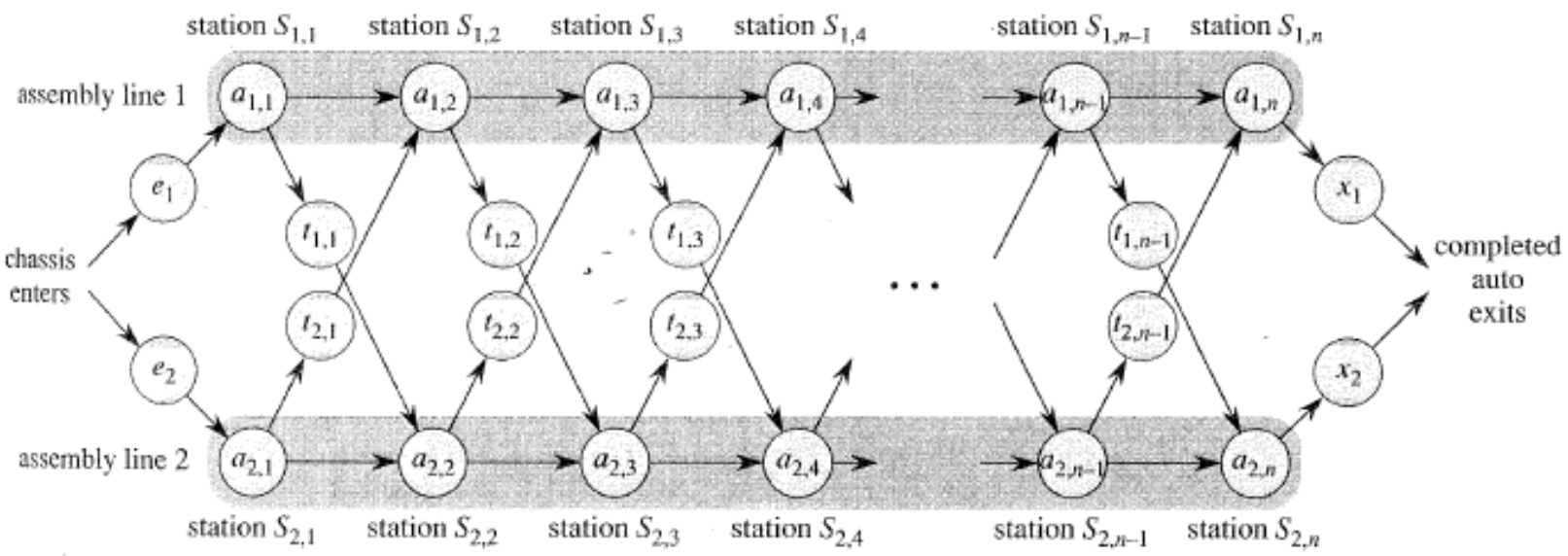
Assembly Line Scheduling

- Automobile factory with two assembly lines
  - Each line has n stations: $S_{1,1}, \ldots, S_{1,n}$ and $S_{2,1}, \ldots, S_{2,n}$
  - Corresponding stations $S_{1,j}$ and $S_{2,j}$ perform the same function but can take different amounts of time $a_{1,j}$ and $a_{2,j}$
  - Entry times are: e_1 and e_2; exit times are: x_1 and x_2



Assembly Line Scheduling

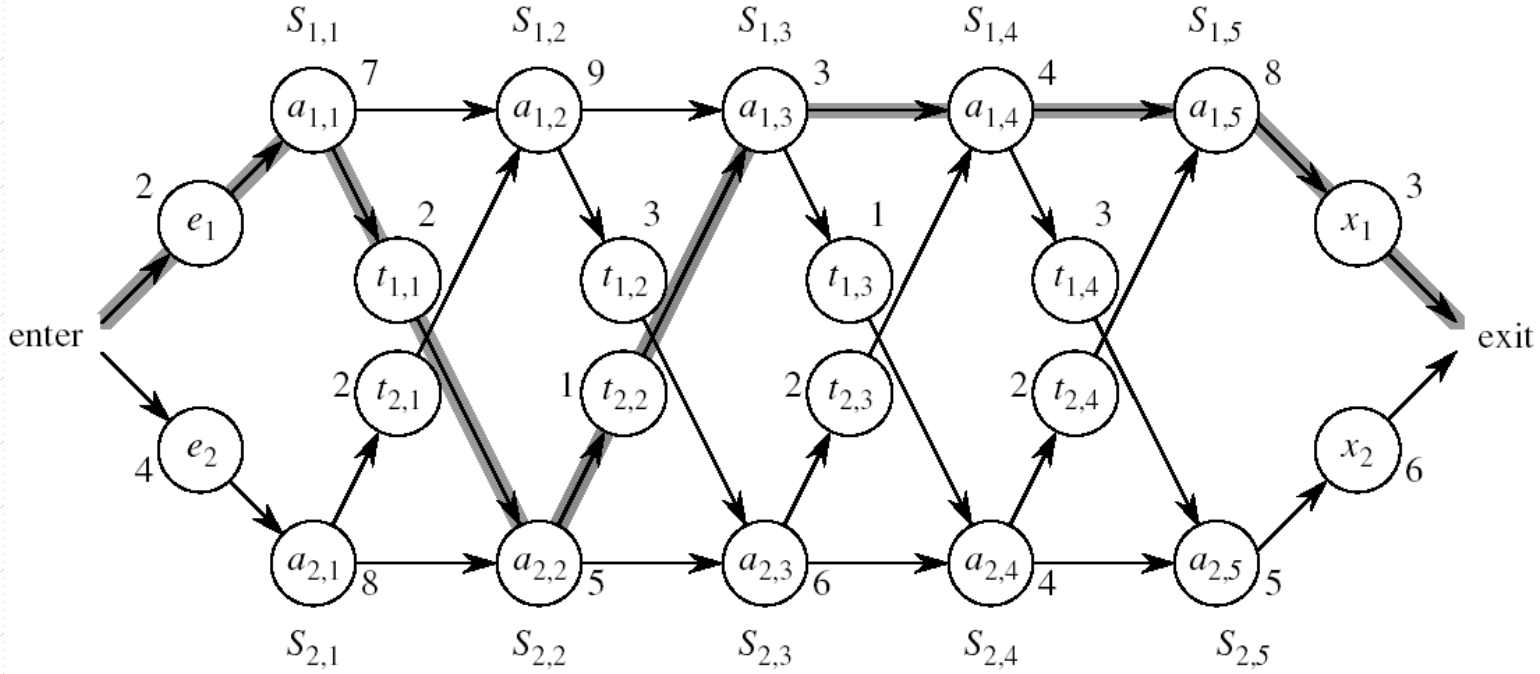
- After going through a station, can either:
  - stay on same line at no cost, or
  - transfer to other line: cost after $S_{i,j}$ is $t_{i,j}$, $j = 1, \ldots, n - 1$



Assembly Line Scheduling

- Problem:

  what stations should be chosen from line 1 and which from line 2 in order to minimize the total time through the factory for one car?



Optimal Substructure - contd.

- It is often easy to show the optimal sub problem property as follows:
  - Split problem into sub-problems
  - Sub-problems must be optimal, otherwise the optimal splitting would not have been optimal.

Characterize subproblem space

- Try to keep the space as simple as possible and expand it as necessary
  - For example, in matrix chain multiplication, subproblem space $A_1A_2…A_j$ will not work. Instead $A_iA_{i+1}…A_j$ (vary at both ends) work
  - In assembly line, subproblem is $S_{1,j}$

Optimal substructure variants

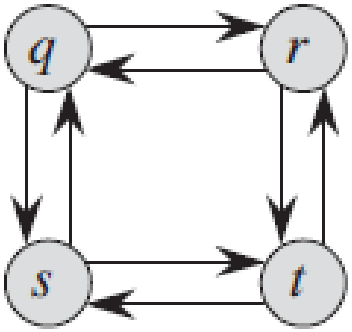
- Optimal substructure varies across problem domains in two ways
  - How many subproblems?
    - In matrix chain multiplication two subproblems
    - In assembly-line schedule: one subproblem
  - How many choices to use subproblem in optimal soln?
    - In matrix chain multiplication: j-i choices
    - In assembly line schedule: two choices
- DP often solves the optimal substructure in bottom-up manner

Complexity analysis of DP

- Running time = #overall subproblems * #choices
  - In matrix chain: $O(n^2) * O(n) = O(n^3)$
  - In assembly line scheduling: O(n) X O(1) = O(n)
- Cost = cost of solving subproblem * cost of making choices
  - In matrix chain choice cost: $p_{i-1}p_ip_j$
  - In assembly line scheduling:
    - a_{ij} if stays in same line
    - $t_{i'j-1} + a_{ij}$ ($i' \neq i$) otherwise

Subtleties when determining optimal substructure

- Optimal substructure may not apply though it may seem to apply at first sight
  - Unweighted shortest path:
    - Find a path from u to v containing fewest edges
    - Can be proved to have optimal substructures
  - Unweighted longest simple path
    - Find a simple path from u to v containing most edges
    - Does not satisfy optimal substructure
  - Independence (no sharing of resources) if a problem has optimal substructure



Overlapping subproblems

- Space of sub-problems must be small
  - recursive solution re-solves the same sub-problem many times rather than generating new subproblems
  - In contrast, a divide-and-conquer approach generates new subproblems at each step
- Usually there are polynomially many sub-problems, and we revisit the same ones over and over again ⇒ overlapping sub-problems.

Recursive-Matrix-Chain

- A recursive algorithm for matrix-chain multiplication

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i == j
2      return 0
3  m[i, j] = ∞
4  for k = i to j − 1
5      q = RECURSIVE-MATRIX-CHAIN(p, i, k)
           + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
           + p_{i−1} p_k p_j
6      if q < m[i, j]
7          m[i, j] = q
8  return m[i, j]
```

- Draw the recursion tree for (p,1,4).

Running Time: Recursive-Matrix-Chain

- *T(n)*: Time taken to compute matrix chain of size *n*

$$T(1) \geq 1 ,$$

$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \qquad \text{for } n > 1 .$$

- Each term T(i) appears twice once as T(k) and the next time as T(n-k)

$$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n .$$

- We can show by induction that

$$T(n) = \Omega(2^n)$$

DP vs. Divide-and-conquer

	Dynamic Programming	**Divide-and-conqeur**
Optimal Substructure	Yes	Yes
Overlapping subproblems	Yes	No
Top-down or bottom-up	Bottom-up	Top-down

Memoization

- Partial results are recorded (memo) and reused later without having to recompute them
- What if we stored sub-problems and used the stored solutions in a recursive algorithm?
  - This is like divide-and-conquer, top down, but should benefit like DP which is bottom-up.
- Memoized version maintains an entry in a table. One can use a fixed table or a hash table.

```
MEMOIZED-MATRIX-CHAIN(p)
1  n = p.length − 1
2  let m[1..n, 1..n] be a new table
3  for i = 1 to n
4      for j = i to n
5          m[i, j] = ∞
6  return LOOKUP-CHAIN(m, p, 1, n)

LOOKUP-CHAIN(m, p, i, j)
1  if m[i, j] < ∞
2      return m[i, j]
3  if i == j
4      m[i, j] = 0
5  else for k = i to j − 1
6          q = LOOKUP-CHAIN(m, p, i, k)
               + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
7          if q < m[i, j]
8              m[i, j] = q
9  return m[i, j]
```

Running Time: Memoized-Matrix-Chain

- Each call to Lookup-Chain is either
  - O(1) time if already computed earlier
  - O(n), otherwise
- Each m[i, j] called many times but initialized only once
  - There are $O(n^2)$ calls of the second type, one per table entry
- A given call of Lookup-Chain makes O(n) recursive calls
  - Total time $O(n^3)$

Exercise

15.3-2

Draw the recursion tree for the MERGE-SORT procedure from Section 2.3.1 on an array of 16 elements. Explain why memoization fails to speed up a good divide-and-conquer algorithm such as MERGE-SORT.

Review

- Write the recurrence for assembly line scheduling?
- Compare top-down with memoization and bottom-up tabulation approaches. In what scenarios is one preferable over the other?