

Dynamic Programming

General method

- Works the same way as divide-and-conquer, by combining solutions to subproblems
 - Divide-and-conquer partitions a problem into independent subproblems
 - Greedy method only works with the local information
- Dynamic programming is required to take into account the fact that the problems may not be partitioned into *independent* subproblems
 - The subproblem is solved only once and the answer saved in a table
 - * Applicable when the subproblems overlap, or subproblems share subsubproblems
 - Solution to a problem can be viewed as the result of a sequence of decisions
 - * Knapsack problem solution
 - Decide the values of x_i , $0 \leq i < n$
 - Make a decision on x_0 , then x_1 , and so on
 - An optimal sequence maximizes the objective function $\sum p_i x_i$ under the constraints $\sum w_i x_i \leq m$ and $0 \leq x_i \leq 1$
 - * Shortest path problem
 - Determine the shortest path from vertex i to vertex j by finding the second vertex in the path, then the third, and so on, until vertex j is reached
 - Optimal sequence of decisions is one with a path of least length
 - Optimal sequence of decisions can be found by making the decisions one at a time and never making an erroneous decision
 - * This is the idea followed in greedy algorithms
 - * Can be guaranteed by trying all possible decision sequences but the time and space costs will be prohibitive
 - * Shortest path problem
 - Find shortest path from vertex i to vertex j
 - Let A_i be the set of vertices adjacent from i
 - The selection of vertex from A_i cannot be finalized without looking further ahead
 - If we have to find a shortest path from a single source to *all* vertices in G , then at each step, a correct decision can be made
 - Dynamic programming drastically reduces the amount of enumeration by avoiding the enumeration of some decision sequences that cannot possibly be optimal; an optimal sequence of decisions is obtained by using the ***principle of optimality***
 - * Applicable when the subproblems are not entirely independent, they may have common subsubproblems
 - * Each subsubproblem is solved only once and the results used repeatedly
- The word *programming* in dynamic programming does not refer to coding but refers to building tables of intermediate results
- Typically used for optimization problems that may have many possible solutions
 - An optimal solution vs *the* optimum solution

Definition 1 The ***principle of optimality*** states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining states must constitute an optimal decision sequence with regard to the state resulting from the first decision.

- Greedy method vs Dynamic programming
 - In greedy method, only one decision sequence is ever generated
 - In dynamic programming, many decision sequences may be generated
 - Sequences containing suboptimal sequences cannot be optimal because of principle of optimality, and so, will not be generated
 - Shortest path problem
 - * Assume that $i, i_1, i_2, \dots, i_k, j$ is a shortest path from i to j
 - * Starting with vertex i , a decision is made to go to i_1
 - * Following this decision, the problem state is defined by vertex i_1 , and we need to find a path from i_1 to j
 - * The sequence i_1, i_2, \dots, i_k, j must be a shortest i_1 to j path
 - * If not, let $i_1, r_1, r_2, \dots, r_q, j$ be a shortest i_1 to j path
 - * Then, $i, i_1, r_1, r_2, \dots, r_q, j$ is an i to j path that is shorter than the path $i, i_1, i_2, \dots, i_k, j$
 - * Hence, principle of optimality is applicable
- Development of solution broken into four steps:
 1. Characterize the structure of an optimal solution
 2. Recursively define the value of an optimal solution
 3. Compute the value of an optimal solution in bottom-up fashion
 4. Construct an optimal solution from computed information

- Computing the n th Fibonacci number

```
int fib ( const int n )
{
    if ( n <= 0 ) return ( 0 );
    if ( n == 1 ) return ( 1 );
    return ( fib ( n - 1 ) + fib ( n - 2 ) );
}
```

- This code is extremely inefficient; why?

- An efficient code using an array of size n

```
int fib ( const int n )
{
    if ( n <= 0 ) return ( 0 );
    if ( n == 1 ) return ( 1 );
    int * a = new int[n+1];
    a[0] = 0;
    a[1] = 1;
    for ( int i ( 2 ); i <= n; i++ )
        a[i] = a[i-1] + a[i-2];
    int tmp ( a[n] );
    delete[] a;
    return tmp;
}
```

- If we want to save space as well, the following code is more appropriate

```

int fib ( const int n )
{
    if ( n <= 0 ) return ( 0 );
    if ( n == 1 ) return ( 1 );
    int f0 ( 0 ), f1 ( 1 ), f;
    for ( int i ( 2 ); i <= n; i++ )
    {
        f = f1 + f0;
        f0 = f1;
        f1 = f;
    }
    return f;
}

```

The above solution is known as *bottom-up dynamic programming* – we compute the smallest values first and build the solution using the solution to smaller problems; most of the real dynamic programming situations refer to *top-down dynamic programming* (also known as *memoization*) as you will see next in knapsack problem

- Knapsack problem

- Recursive solution

- * Each time you choose an item, you assume that you can optimally find a solution to pack the rest of the knapsack

```

struct item
{
    int    size;
    int    val;
};

int knapsack ( const int capacity )
{
    int t;
    // N is the number of item types
    for ( int i ( 0 ), int max ( 0 ); i < N; i++ )
        if ( ( int space = capacity - items[i].size ) >= 0 )
        {
            Remove items[i] from items;
            if ( ( t = knapsack ( space ) + items[i].val ) > max )
                max = t;
        }
    return ( max );
}

```

- Dynamic programming solution

- * Decide the values of $x_i, 1 \leq i \leq n$
- * Make a decision on x_1 , then on x_2 , and so on
- * An optimal sequence of decisions maximizes the objective function $\sum p_i x_i$, under the constraints $\sum w_i x_i \leq m$ and $0 \leq x_i \leq 1$

```

int knapsack ( const int capacity )
{
    int maxi, t;
    if ( maxknown[capacity] )
        return ( maxknown[capacity] );
    for ( int i ( 0 ), int max ( 0 ); i < N; i++ )

```

```

    if ( ( int space = capacity - items[i].size ) >= 0 )
        if ( ( t = knapsack ( space ) + items[i].val ) > max )
        {
            max = t;
            maxi = i;
        }
    maxknown[capacity] = max;
    itemknown[capacity] = items[maxi];
    return ( max );
}

```

- 0/1 knapsack

- Same as regular knapsack problem except that the x_i 's are restricted to a value of 0 or 1 (take nothing or everything for an item)
- Formally, an instance of knapsack problem $\text{knap} (1, j, y)$ is

$$\begin{array}{ll}
 \text{Maximize} & \sum_{1 \leq i \leq j} p_i x_i \\
 \text{subject to} & \sum_{1 \leq i \leq j} w_i x_i \leq y \\
 & x_i = 0 \text{ or } 1, 1 \leq i \leq j
 \end{array}$$

and the knapsack problem is represented by $\text{knap} (1, n, m)$

- Let y_1, y_2, \dots, y_n be an optimal sequence of 0/1 values for x_1, x_2, \dots, x_n , respectively
 - * If $y_1 = 0$, then y_2, y_3, \dots, y_n must constitute an optimal sequence for the problem $\text{knap} (2, n, m)$
 - * If it is not an optimal subsequence, then y_1, y_2, \dots, y_n is not an optimal sequence for $\text{knap} (1, n, m)$
 - * If $y_1 = 1$, then y_2, y_3, \dots, y_n must constitute an optimal sequence for the problem $\text{knap} (2, n, m - w_1)$
 - * If it is not an optimal subsequence, then
 - there is another 0/1 sequence z_2, z_3, \dots, z_n such that $\sum_{i=2}^n w_i z_i \leq m - w_1$, and
 - $\sum_{i=2}^n p_i z_i > \sum_{i=2}^n p_i y_i$
 - * Hence, the sequence $y_1, z_2, z_3, \dots, z_n$ gives a sequence with greater value
 - * Not possible due to principle of optimality

QED

- General decision problem

- Let S_0 be initial problem state
- Assume that you have to make n decisions $d_i, 1 \leq i \leq n$
- Let $D_1 = \{r_1, r_2, \dots, r_j\}$ be the set of possible decision values for d_1
- Let S_i be the problem state following the choice of decision $r_i, 1 \leq i \leq j$
- Let Γ_i be an optimal sequence of decisions with respect to the problem state S_i
- Then, by principle of optimality, an optimal sequence of decisions with respect to S_0 is the best of decision sequences $r_i, \Gamma_i, 1 \leq i \leq j$

- Shortest path

- Which of the vertices in A_i (where $A_i \subset V$ is the set of vertices adjacent to i) should be the second vertex on the path?
- Let A_i be the set of vertices adjacent to vertex i
- For each vertex $k \in A_i$, let Γ_k be a shortest path from k to j
- Then, a shortest i to j path is the shortest of the paths $\{i, \Gamma_k \mid k \in A_i\}$.
- Dynamic programming solution for shortest path
 - * Let k be an intermediate vertex on a shortest i to j path $i, i_1, i_2, \dots, k, p_1, p_2, \dots, j$
 - * The paths i, i_1, \dots, k and k, p_1, \dots, j must, respectively, be shortest i to k and k to j paths.

- * Let P_j be the set of vertices adjacent to vertex j ; $k \in P_j \Leftrightarrow \langle k, j \rangle \in E(G)$
- * For each $k \in P_j$, let Γ_k be a shortest i to k path
- * By principle of optimality, a shortest i to k path is the shortest of paths $\{\Gamma_k, j | k \in P_j\}$
 - Start at vertex j and look at last decision made
 - Last decision was to use one of the edges $\langle k, j \rangle, k \in P_j$
- Dynamic programming eliminates all recomputation in any recursive program, by saving intermediate values in variables whose scope is designed to allow them to be visible in more than one local context

Property 1 *Dynamic programming reduces the running time of a recursive function to be at most the time required to evaluate the function for all arguments less than or equal to the given argument, treating the cost of a recursive call as constant.*

- Property 1 implies that the running time for the knapsack problem is $O(NM)$
- Dynamic programming becomes ineffective when the number of possible function values that may be needed is so high that we cannot afford to save or precompute all of them
- Dynamic programming solution to 0/1 knapsack

- The intermediate knapsack problem $\text{knap}(1, j, y)$ can be represented by

$$\begin{array}{ll} \text{Maximize} & \sum_{l \leq i \leq j} p_i x_i \\ \text{subject to} & \sum_{l \leq i \leq j} w_i x_i \leq y \\ & x_i \in \{0, 1\}, l \leq i \leq j \end{array}$$

- The original knapsack problem now is: $\text{knap}(0, n-1, m)$
- Let $g_j(y)$ be the value of an optimal solution to $\text{knap}(j+1, n, y)$.
 - * $g_0(m)$ is the value of an optimal solution to $\text{knap}(1, n, m)$
 - * The possible decisions for x_1 are 0 and 1 ($D_1 = \{0, 1\}$)
 - * From the principle of optimality, it follows that

$$g_0(m) = \max\{g_1(m), g_1(m - w_1) + p_1\}$$

- Let y_1, y_2, \dots, y_n be an optimal solution to $\text{knap}(1, n, m)$.
 - * For each $j, 1 \leq j \leq n, y_1, \dots, y_j$, and y_{j+1}, \dots, y_n must be optimal solutions to the problems $\text{knap}(1, j, \sum_{i=1}^j w_i y_i)$ and $\text{knap}(j+1, n, m - \sum_{i=1}^j w_i y_i)$ respectively
 - * This observation allows us to generalize the previous function $g()$ to

$$g_i(y) = \max\{g_{i+1}(y), g_{i+1}(y - w_{i+1}) + p_{i+1}\}$$

- * The recursion is solved by using $g_n(y) = 0$ for $y \geq 0$ and $g_n(y) = -\infty$ for $y < 0$
- * From $g_n(y)$, we can obtain $g_{n-1}(y)$ using the above recurrence; then, using $g_{n-1}(y)$, we can find $g_{n-2}(y)$, and so on; finally, we can determine $g_0(m)$
- Example
 - * $n = 3, w = \{2, 3, 4\}, p = \{1, 2, 5\}, m = 6$
 - * We have to compute $g_0(6)$

$$\begin{aligned} g_0(6) &= \max(g_1(6), g_1(4) + 1) \\ g_1(6) &= \max(g_2(6), g_2(3) + 2) \\ g_2(6) &= \max(g_3(6), g_3(2) + 5) \\ &= \max(0, g_3(2) + 5) \\ &= \max(0, 0 + 5) \end{aligned}$$

$$\begin{aligned}
&= 5 \\
g_2(3) &= \max(g_3(3), g_3(3-4) + 5) \\
&= \max(0, -\infty) \\
&= 0 \\
g_1(6) &= \max(5, 2) \\
&= 5 \\
g_1(4) &= \max(g_2(4), g_2(4-3) + 2) \\
g_2(4) &= \max(g_3(4), g_3(4-4) + 5) \\
&= \max(0, 0 + 5) \\
&= 5 \\
g_2(1) &= \max(g_3(1), g_3(1-4) + 5) \\
&= \max(0, -\infty) \\
&= 0 \\
g_1(4) &= \max(5, 2) \\
&= 5 \\
g_0(6) &= \max(5, 5 + 1) \\
&= 6
\end{aligned}$$

- The sequence of decisions x_1, x_2, \dots, x_n leads to

$$g_j(y) = \max(\{g_{j-1}(y), g_{j-1}(y - w_j) + p_j\})$$

where $g_j(y)$ is the value of an optimal solution to $\text{knap}(1, j, Y)$

- 0/1 knapsack

- Looking backwards at the sequence of decisions x_1, x_2, \dots, x_n , we see that

$$f_j(y) = \max\{f_{j-1}(y), f_{j-1}(y - w_j) + p_j\}$$

where $f_j(y)$ is the value of an optimal solution to $\text{knap}(1, j, Y)$

- Value of an optimal solution to $\text{knap}(1, n, m)$ is $f_n(m)$
- Solve by beginning with $f_0(y) = 0$ for all $y, y \geq 0$, and $f_0(y) = -\infty$ for all $y, y < 0$
- Successively obtain f_1, f_2, \dots, f_n
- The solution method may indicate that one has to look at all possible decision sequences to get an optimal sequence using dynamic programming
 - * Using principle of optimality, suboptimal decision sequences are discarded
 - * Although total number of decision sequences are exponential in the number of decisions, dynamic programming algorithms often have polynomial complexity
 - * Also, optimal solutions to subproblems are retained to avoid recomputing their value

Traveling Salesperson Problem

- Given a directed graph $G = (V, E)$ with edge costs c_{ij}
- c_{ij} is defined such that $c_{ij} > 0$ for all i and j and $c_{ij} = \infty$ if $\langle i, j \rangle \notin E$.
- $|V| = n$ and $n > 1$
- *Tour*

- A tour of G is a directed cycle that includes every vertex in V , and no vertex occurs more than once except for the starting vertex
- Cost of a tour is the sum of the cost of edges on the tour
- Traveling salesperson problem is to find a tour of minimum cost

- Comments

- 0/1 knapsack is a subset selection problem
- Traveling salesperson is a permutation problem
- Permutation problems are harder to solve
 - * $n!$ different permutations of n objects
 - * 2^n different subsets of n objects
 - * $n! > 2^n$

- Greedy algorithm

- Start with vertex v_1 ; call it v_i
- Visit the vertex v_j that is *nearest* to v_i , or can be reached from v_i with least cost
- Repeat the above starting at vertex v_j (call it as new v_i) taking care never to visit a vertex already visited

- Dynamic programming algorithm

- Regard the tour to be a simple path that starts and ends at vertex 1
- Every tour consists of an edge $\langle 1, k \rangle$ for some $k \in V - \{1\}$ and a path from vertex k to vertex 1
- The path from vertex k to vertex 1 goes through each vertex in $V - \{1, k\}$ exactly once
- If the tour is optimal, then the path from k to 1 must be a shortest k to 1 path going through all vertices in $V - \{1, k\}$
- Let $g(i, S)$ be the length of a shortest path starting at vertex i , going through all vertices in S , and terminating at vertex 1
- $g(1, V - \{1\})$ is the length of an optimal salesperson tour
- From the principle of optimality

$$g(1, V - \{1\}) = \min_{2 \leq k \leq n} \{c_{1k} + g(k, V - \{1, k\})\} \quad (1)$$

- Generalizing (for $i \notin S$)

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\} \quad (2)$$

- Equation 1 may be solved for $g(1, V - \{1\})$ if we know $g(k, V - \{1, k\})$ for all values of k
- The g values may be obtained by using Equation 2
 - * $g(i, \phi) = c_{i,1}, 1 \leq i \leq n$
 - * We can use Equation 2 to obtain $g(i, S)$ for all S of size 1
 - * Then we can obtain $g(i, S)$ for S with $|S| = 2$
 - * When $|S| < n - 1$, the values of i and S for which $g(i, S)$ is needed are such that $i \neq 1, 1 \notin S$, and $i \notin S$

- Solving traveling salesperson problem with dynamic programming – example

- Consider the directed graph presented below

0	10	15	20
5	0	9	10
6	13	0	12
8	8	9	0

- Solving for 2, 3, 4

$$\begin{aligned} g(2, \phi) &= c_{21} = 5 \\ g(3, \phi) &= c_{31} = 6 \\ g(4, \phi) &= c_{41} = 8 \end{aligned}$$

- Using Equation 2, we get

$$\begin{aligned} g(2, \{3\}) &= c_{23} + g(3, \phi) = 15 & g(2, \{4\}) &= 18 \\ g(3, \{2\}) &= 18 & g(3, \{4\}) &= 20 \\ g(4, \{2\}) &= 13 & g(4, \{3\}) &= 15 \end{aligned}$$

- Next, we compute $g(i, S)$ with $|S| = 2$, $i \neq 1$, $1 \notin S$, and $i \notin S$

$$\begin{aligned} g(2, \{3, 4\}) &= \min\{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\} = 25 \\ g(3, \{2, 4\}) &= \min\{c_{32} + g(2, \{4\}), c_{34} + g(4, \{2\})\} = 25 \\ g(4, \{2, 3\}) &= \min\{c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})\} = 23 \end{aligned}$$

- Finally, from Equation 1, we obtain

$$\begin{aligned} g(1, \{2, 3, 4\}) &= \min\{c_{12} + g(2, \{3, 4\}), c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\} \\ &= \min\{35, 40, 43\} \\ &= 35 \end{aligned}$$

- Optimal tour

- * Has cost 35
- * A tour of this length may be constructed if we retain with each $g(i, S)$ the value of j that minimizes the right hand side of Equation 2
- * Let this value be called $J(i, S)$
- * Then, $J(1, \{2, 3, 4\}) = 2$
- * Thus the tour starts from 1 and goes to 2
- * The remaining tour may be obtained from $g(2, \{3, 4\})$
- * Now, $J(2, \{3, 4\}) = 4$
- * Thus the next edge is $\langle 2, 4 \rangle$
- * The remaining tour is for $g(4, \{3\})$
- * $J(4, \{3\}) = 3$
- * The optimal tour is 1, 2, 4, 3, 1

- Analysis of traveling salesperson

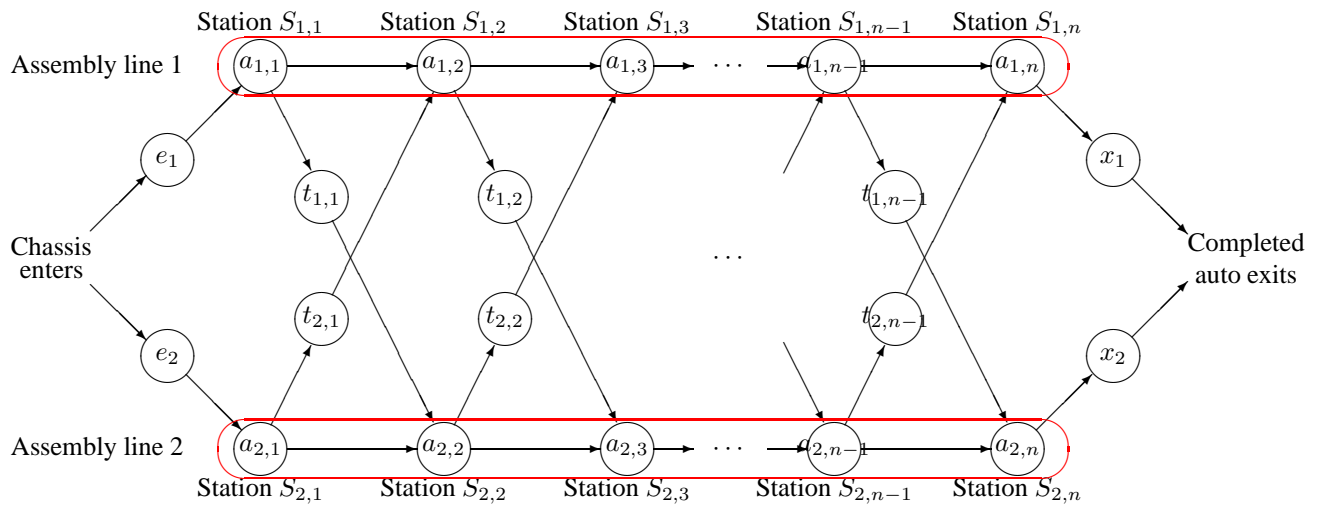
- Let N be the number of $g(i, S)$ s that have to be computed before Equation 1 may be used to compute $g(1, V - \{1\})$
- For each value of $|S|$, there are $n - 1$ choices of i
- The number of distinct sets S of size k not including 1 and i is $\binom{n-2}{k}$
- Hence,

$$N = \sum_{k=0}^{n-2} (n-k-1) \binom{n-2}{k} = (n-1)2^{n-2}$$

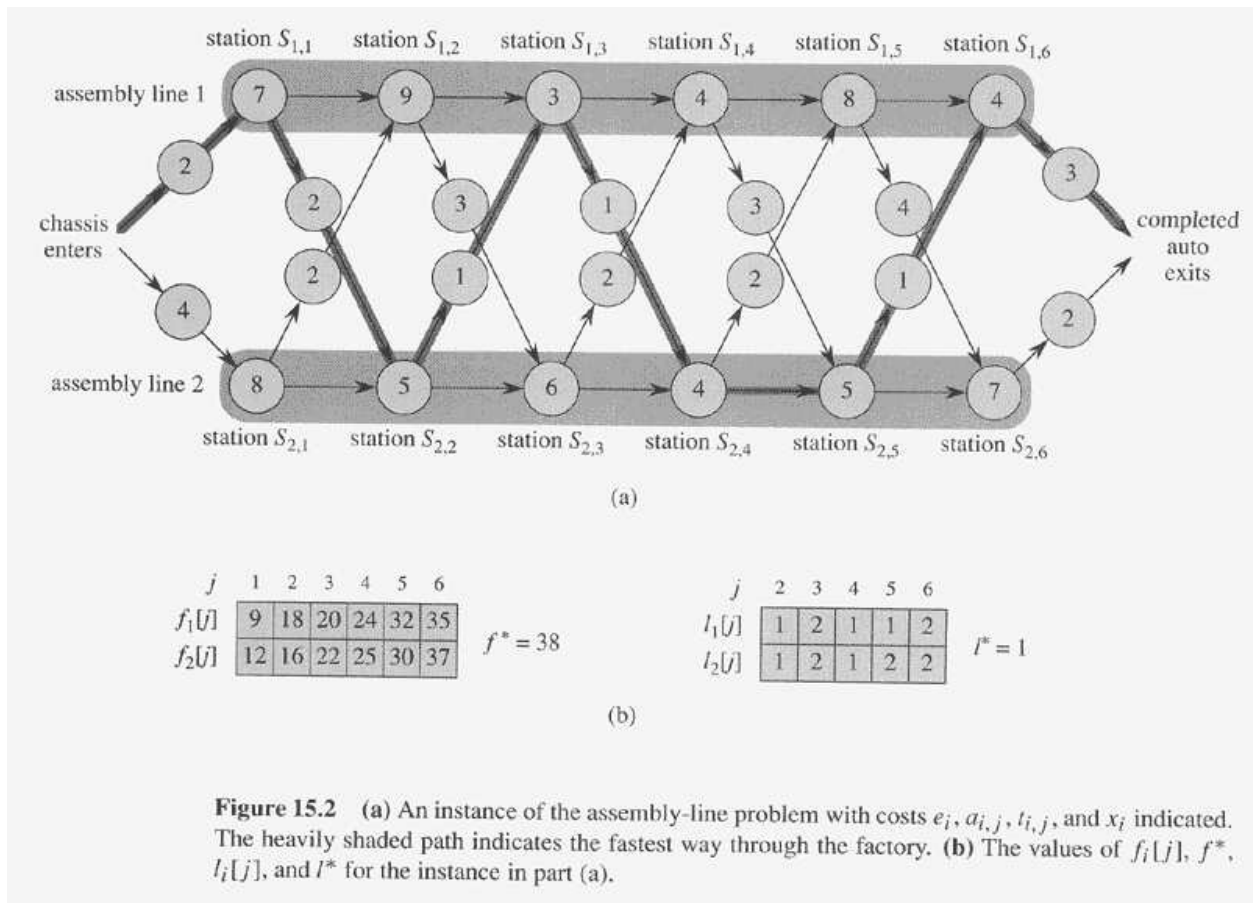
- An algorithm that finds an optimal tour using Equations 1 and 2 will require $\Theta(n^2 2^n)$ time as the computation of $g(i, S)$ with $|S| = k$ requires $k - 1$ comparisons when solving Equation 2
- Better than enumerating all $n!$ different tours to find the best one
- The most serious drawback of the dynamic programming solution is the space needed ($O(n2^n)$)
 - * This can be too large even for modest values of n .

Assembly line scheduling

- Manufacturing problem with two assembly lines
 - Chassis enters an assembly line
 - Parts added to chassis at each station
 - Finished product exits the assembly line
 - Each assembly line i has exactly n stations – S_{i1}, \dots, S_{in}
 - * Corresponding stations perform exactly the same function but may take different amount of time
 - * Assembly time required at station S_{ij} is denoted by a_{ij}
 - Shown in following figure



- Entry time for assembly line i denoted by e_i
- Exit time for assembly line i denoted by x_i
- Chassis can go from one station to another
 - Within the same assembly line in negligible time, or at no cost
 - To the other assembly line at some cost
 - * Cost to go from one assembly line to another after having gone through station S_{ij} is t_{ij}
- Problem is to schedule the assembly line such that the selection of stations from each assembly line minimizes the overall assembly cost
 - Need to determine the stations to choose from assembly line 1 and 2 to minimize the assembly time
 - In the following example, choose stations 1, 3, 6 from line 1 and 2, 4, 5 from line 2



- Brute force solution
 - Enumerate all possibilities for stations
 - Compute how long does each one take, and pick the best
 - Problem hard because there are 2^n ways to choose the stations
 - Time required is given by $\Omega(2^n)$ which is infeasible for large
- Dynamic programming solution: Step 1: Structure of fastest way through factory, or structure of optimal solution
 - Fastest possible way for a chassis to get from starting point through station $S_{1,j}$
 - Only one possible way to go from starting point through station $S_{1,1}$
 - Two ways to arrive at each station $S_{i,j}$, $j > 1$
 1. Station on same line, from station $S_{i,j-1}$
 2. Station on other line, from station $S_{i',j-1}$, $i' \neq i$, at a cost $t_{i',j-1}$
 - Assume that fastest way through station $S_{1,j}$ is through station $S_{1,j-1}$
 - * The chassis must have taken the fastest way from starting point to station $S_{1,j-1}$
 - * If there were a faster way to get through station $S_{1,j-1}$, we could substitute this faster way to get through station $S_{1,j}$ – contradiction
 - Assume that fastest way through station $S_{1,j}$ is through station $S_{2,j-1}$
 - * The chassis must have taken the fastest way from starting point to station $S_{2,j-1}$
 - * If there were a faster way to get through station $S_{2,j-1}$, we could substitute this faster way to get through station $S_{1,j}$ – contradiction

- Optimal substructure
 - * Optimal solution to a problem contains the optimal solution to the subproblems within it
 - * Fastest way to a station requires that the chassis must have taken the fastest way to the previous station in the line
 - * To find the fastest way to a station $S_{i,j}$, solve the subproblem to compute the fastest way to the two previous stations – $S_{i,j-1}$ and $S_{i',j-1}$

• Step 2: A recursive solution

- Define the value of an optimal solution recursively in terms of optimal solution to subproblems
 - * Subproblems will be defined as the problem of finding the fastest way through station j on both lines, for $j = 1, 2, \dots, n$
- Let $f_i[j]$ be the fastest possible time to get a chassis from starting point through station $S_{i,j}$
- Let f^* be the fastest time to get the chassis all the way through the factory

$$f^* = \min(f_1[n] + x_1, f_2[n] + x_2)$$

- Time to go through the first station in each line is given by

$$\begin{aligned} f_1[1] &= e_1 + a_{1,1} \\ f_2[1] &= e_2 + a_{2,1} \end{aligned}$$

- Compute $f_i[j]$ for $j = 2, 3, \dots, n$ and $i = 1, 2$
- Adding the recursive step, it is easy to see that

$$\begin{aligned} f_1[j] &= \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) \\ f_2[j] &= \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) \end{aligned}$$

- Define $l_i[j]$ to keep track of line number whose station $j-1$ is used to get to station $S_{i,j}$
 - * No need to define $l_1[j]$ because no station precedes station 1 on either line
- Define l^* to be the line whose station n is used as the last station to get through the assembly line
- Starting with $l^* = 1$, use station $S_{1,6}$
- $l_1[6] = 2 \Rightarrow$ station $S_{2,5}$
- $l_2[5] = 2 \Rightarrow S_{2,4}$
- $l_2[4] = 1 \Rightarrow S_{1,3}$
- $l_1[3] = 2 \Rightarrow S_{2,2}$
- $l_2[2] = 1 \Rightarrow S_{1,1}$

• Step 3: Computing the fastest times

- Simple to write recursive algorithm but its running time is exponential in n
- Compute $f_i[j]$ values in increasing order of station numbers
 - * Leads to $\Theta(n)$ time