Chapter 7: Dynamic Programming

Chin Lung Lu

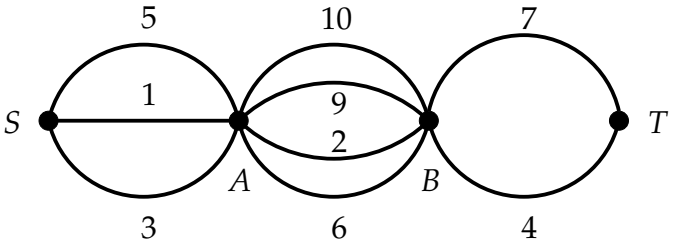
Department of Computer Science

National Tsing Hua University

Shortest path problem

Example 1:

Find a shortest path from S to T in the following graph.

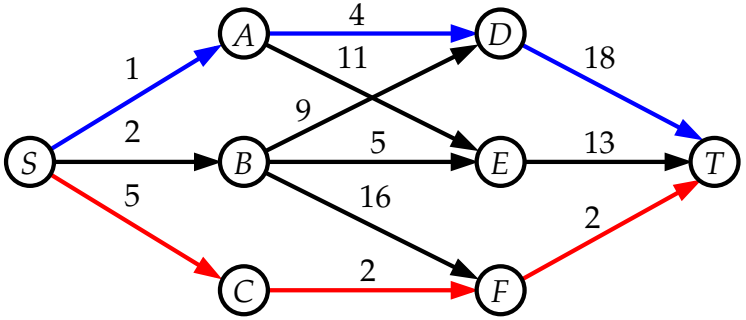


- Since the shortest path from S to T must pass through A and B, we can use a greedy method to find it.
- Hence, the shortest path from S to T is $shortest\_path(S,A) + shortest\_path(A,B) + shortest\_path(B,T)$.

Shortest path problem (cont'd)

Example 2:

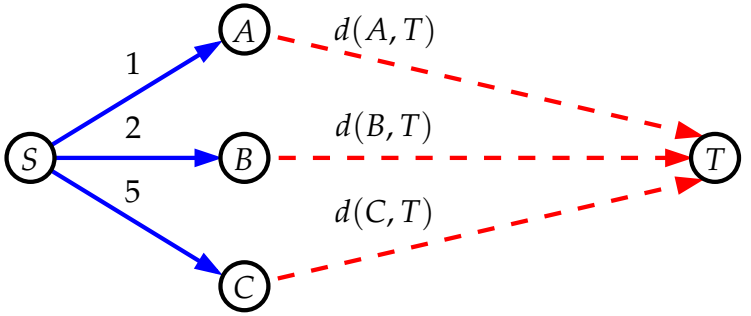
Find the shortest path from S to T in the following graph.



- The greedy method cannot work, because we don't know, among A, B and C, which vertex the shortest path will go through.

Shortest path problem (cont'd)

Dynamic programming approach

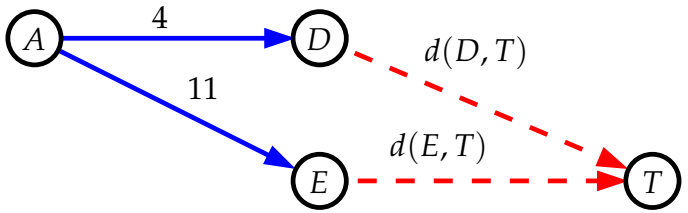


- $d(S,T) = \min\left\{\begin{array}{l} 1 + d(A,T), \\ 2 + d(B,T), \\ 5 + d(C,T) \end{array}\right\}$

where $d(X,Y)$ is the length of the shortest path from X to Y.

Shortest path problem (cont'd)

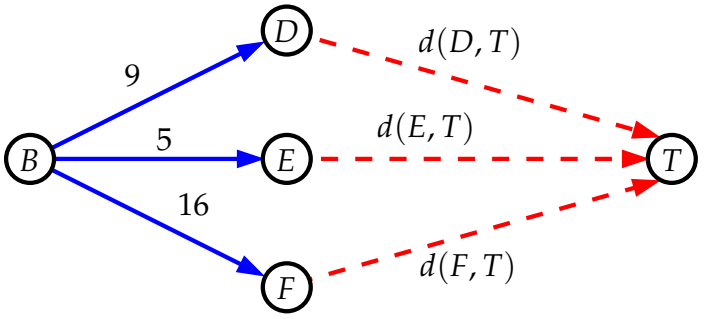
Dynamic programming approach



- $d(A,T) = \min\left\{ \begin{array}{l} 4 + d(D,T), \\ 11 + d(E,T) \end{array} \right\} = \min\left\{ \begin{array}{l} 4 + 18, \\ 11 + 13 \end{array} \right\} = 22$
- Actually, the shortest paths from D, E and F to T are already known.

Shortest path problem (cont'd)

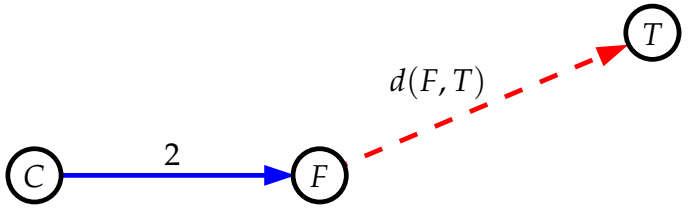
Dynamic programming approach



- $d(B,T) = \min\left\{ \begin{array}{l} 9 + d(D,T), \\ 5 + d(E,T), \\ 16 + d(F,T) \end{array} \right\} = \min\left\{ \begin{array}{l} 9 + 18, \\ 5 + 13, \\ 16 + 2 \end{array} \right\} = 18$

Shortest path problem (cont'd)

Dynamic programming approach



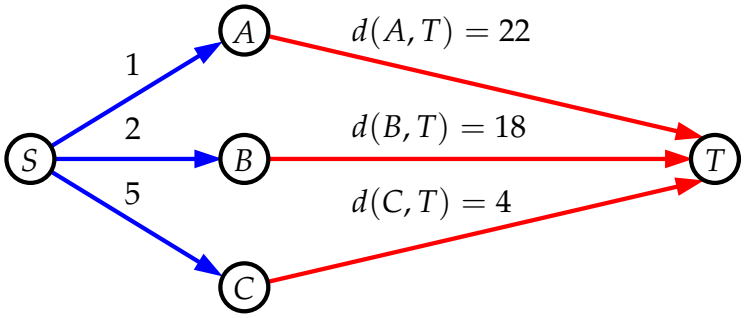
- $d(C,T) = 2 + d(F,T) = 2 + 2 = 4$

Shortest path problem (cont'd)

Dynamic programming approach

- Having found $d(A,T) = 22, d(B,T) = 18$ and $d(C,T) = 4$, we can find $d(S,T)$ by using the following formula:

$$d(S,T) = \min\left\{ \begin{array}{l} 1 + d(A,T), \\ 2 + d(B,T), \\ 5 + d(C,T) \end{array} \right\} = \min\left\{ \begin{array}{l} 1 + 22, \\ 2 + 18, \\ 5 + 4 \end{array} \right\} = 9$$



Shortest path problem (cont'd)

Dynamic programming approach

We summarize the DP approach for the shortest path problem:

1. To find the shortest path from S to T, we find shortest paths from A, B and C to T.
2. To find the shortest path from A to T, we find shortest paths from D and E to T.
   To find the shortest path from B to T, we find shortest paths from D, E and F to T.
   To find the shortest path from C to T, we find the shortest path from F to T.
3. Shortest paths from D, E and F to T can be found easily.
4. Having found shortest paths from D, E and F to T, we can find shortest paths from A, B and C to T.
5. Having found shortest paths from A, B and C to T, we can find the shortest path from S to T.

Shortest path problem (cont'd)

Dynamic programming approach

- The basic principle of the dynamic programming approach is to decompose a problem into subproblems and each subproblem is solved by the same approach recursively.
- The above reasoning way for the shortest path problem is a backward reasoning.
- In fact, we can also solve the shortest path problem by forward reasoning.

DP method for shortest path problem (cont'd)

1. We first find $d(S,A) = 1, d(S,B) = 2$ and $d(S,C) = 5$.
2. We then determine $d(S,D), d(S,E)$ and $d(S,F)$ as follows:

$$\begin{aligned} d(S,D) &= \min\{d(A,D)+d(S,A), d(B,D)+d(S,B)\} \\ &= \min\{4+1, 9+2\} \\ &= \min\{5, 11\} \\ &= 5 \end{aligned}$$

$$\begin{aligned} d(S,E) &= \min\{d(A,E)+d(S,A), d(B,E)+d(S,B)\} \\ &= \min\{11+1, 5+2\} \\ &= \min\{12, 7\} \\ &= 7 \end{aligned}$$

$$\begin{aligned} d(S,F) &= \min\{d(B,F)+d(S,B), d(C,F)+d(S,C)\} \\ &= \min\{16+2, 2+5\} \\ &= \min\{18, 7\} \\ &= 7 \end{aligned}$$

Shortest path problem (cont'd)

Dynamic programming approach

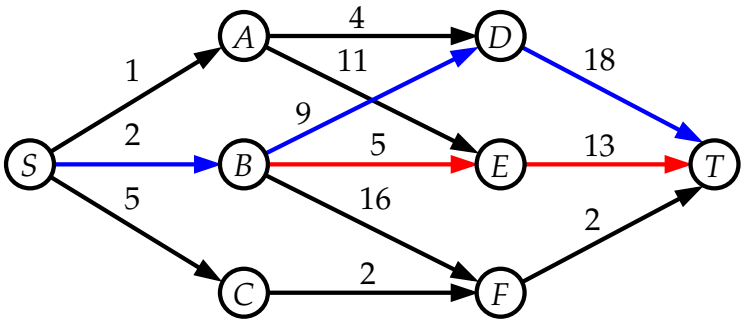
3. The shortest distance from S to T can be determined as:

$$\begin{aligned} d(S,T) &= \min\left\{\begin{array}{l} d(D,T)+d(S,D), \\ d(E,T)+d(S,E), \\ d(F,T)+d(D,F) \end{array}\right\} \\ &= \min\{18+5, 13+7, 2+7\} \\ &= \min\{23, 20, 9\} \\ &= 9 \end{aligned}$$

Shortest path problem (cont'd)

Dynamic programming approach

- The dynamic programming approach can save computations.



- Consider a feasible solution: $S \to B \to D \to T$.
- Its length $d(S,B) + d(B,D) + d(D,T)$ is never calculated, if the backward reasoning of dynamic programming approach is used.

Shortest path problem (cont'd)

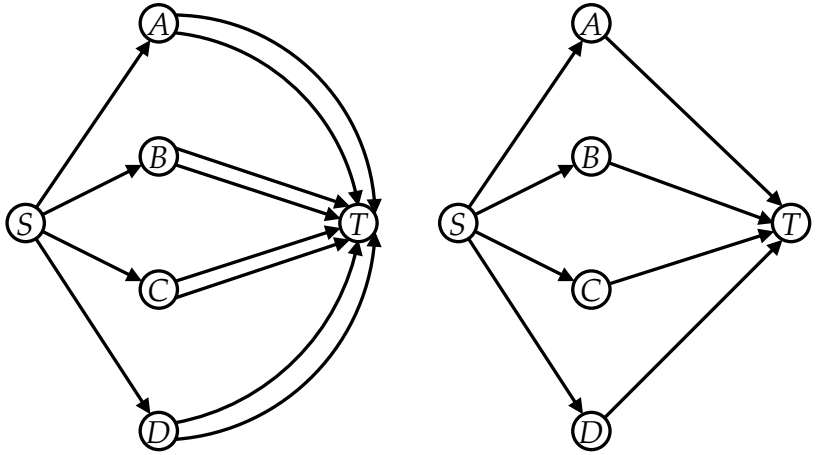
Dynamic programming approach

- Using the backward reasoning approach, we will find:

$$5 + 13 = d(B,E) + d(E,T) \leq d(B,D) + d(D,T) = 9 + 18$$

- That is, we need not consider the solution $S \to B \to D \to T$ because $B \to E \to T$ is shorter than $B \to D \to T$.
- Hence, like the branch and bound approach, the dynamic programming approach helps us avoids exhaustively searching the solution space.
- We may say that the dynamic programming approach is an elimination by stage approach, because after a stage is considered, many subsolutions are eliminated.

DP method for shortest path problem (cont'd)



- There are originally eight solutions (left), while using the dynamic programming approach, the number of solutions is reduced to four (right).

Principle of optimality

Principle of optimality:

- Suppose that in solving a problem, we need to make a sequence of decisions $D_1, D_2, \ldots, D_n$.
- If this sequence is optimal, then the last k decisions must be optimal, where $1 \leq k \leq n$.

Two advantages of dynamic programming:

1. Save computational time by eliminating solutions and avoiding computing the same subproblems repeatedly.
2. Solve the problem stage by stage systematically.

Two-sequence alignment

- Let $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$ be two sequences over an alphabet set Σ.
- A sequence alignment between A and B is a $2 \times k$ matrix M of characters over $\Sigma \cup \{-\}$, where $k \geq \max\{m, n\}$, such that (1) no column of M consists entirely of "–", and (2) the result sequences by removing all "–" in the 1st and 2nd rows of M equals to A and B, respectively.
- For example, let $A = abcd$ and $B = cbd$.

Example 1: ✓ Example 2: ✓ Example 3: ✗

a	b	c	–	d
–	–	c	b	d

a	b	c	d
c	b	–	d

a	b	c	–	d
c	b	–	–	d

Two-sequence alignment (cont'd)

- The second alignment seems to be better than the first one.
- To be precise, we need to define a scoring function which can be used to measure the performance of an alignment.
- Let $f(x, y)$ denote the score for aligning x with y.

Definition of $f(x, y)$:

- Suppose that both x and y are characters.
- Then $f(x, y) = 2$ if $x = y$ and $f(x, y) = 1$ if $x \neq y$.
- If x or y is "–", then $f(x, y) = -1$.

Two-sequence alignment (cont'd)

- The score of an alignment is the total sum of the scores of columns.

Example 1:

a	b	c	–	d
–	–	c	b	d

$score = -1 - 1 + 2 - 1 + 2 = 1$

Example 2:

a	b	c	d
c	b	–	d

$score = 1 + 2 - 1 + 2 = 4$

- Thus, the second alignment is better than the first one.

Two-sequence alignment problem

Dynamic programming approach: Recursive formula

Definition:

The two-sequence alignment problem for sequences A and B is to find an optimal alignment with the maximum score.

- Let $A_{i,j}$ denote the optimal alignment score between $a_1 a_2 \ldots a_i$ and $b_1 b_2 \ldots b_j$, where $1 \leq i \leq m$ and $1 \leq j \leq n$.

Initial conditions:

- $A_{0,0} = 0$.
- $A_{i,0} = i \times f(a_i, -)$.
- $A_{0,j} = j \times f(-, b_j)$.

Two-sequence alignment problem (cont'd)

Dynamic programming approach: Recursive formula

$$A_{i,j} = \max \begin{cases} A_{i-1,j-1} + f(a_i, b_j) \\ A_{i-1,j} + f(a_i, -) \\ A_{i,j-1} + f(-, b_j) \end{cases}$$

❶ $a_1 a_2 \ldots a_{i-1}$ | a_i ; $b_1 b_2 \ldots b_{j-1}$ | b_j } $A_{i-1,j-1} + f(a_i, b_j)$

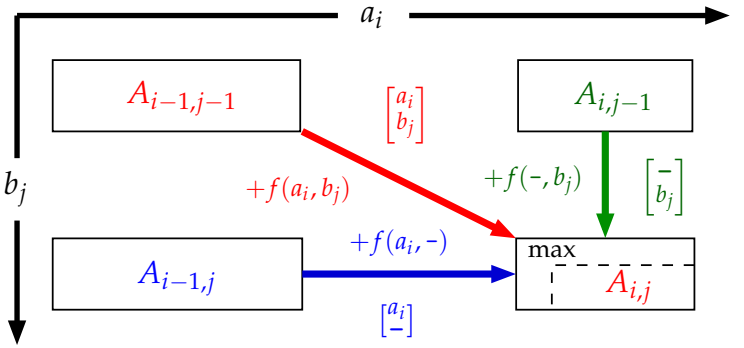
❷ $a_1 a_2 \ldots a_{i-1}$ | a_i ; $b_1 b_2 \ldots b_j$ | $-$ } $A_{i-1,j} + f(a_i, -)$

❸ $a_1 a_2 \ldots a_i$ | $-$; $b_1 b_2 \ldots b_{j-1}$ | b_j } $A_{i,j-1} + f(-, b_j)$

Two-sequence alignment problem (cont'd)

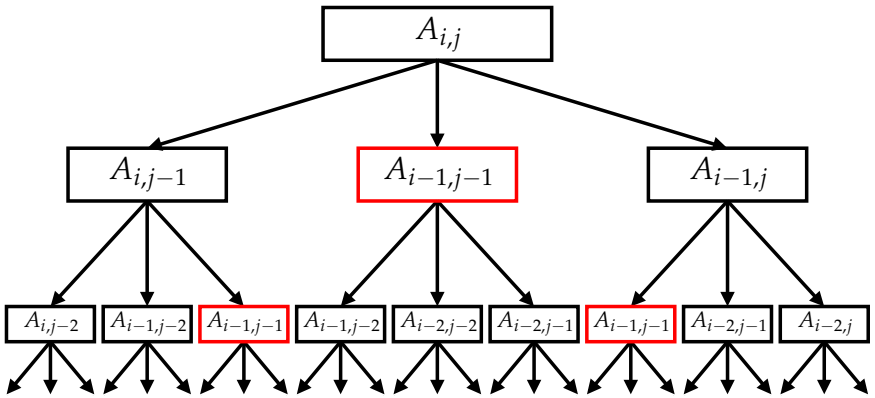
Dynamic programming approach: Recursive formula

$$A_{i,j} = \max \begin{cases} A_{i-1,j-1} + f(a_i, b_j) \\ A_{i-1,j} + f(a_i, -) \\ A_{i,j-1} + f(-, b_j) \end{cases}$$



Two-sequence alignment problem (cont'd)

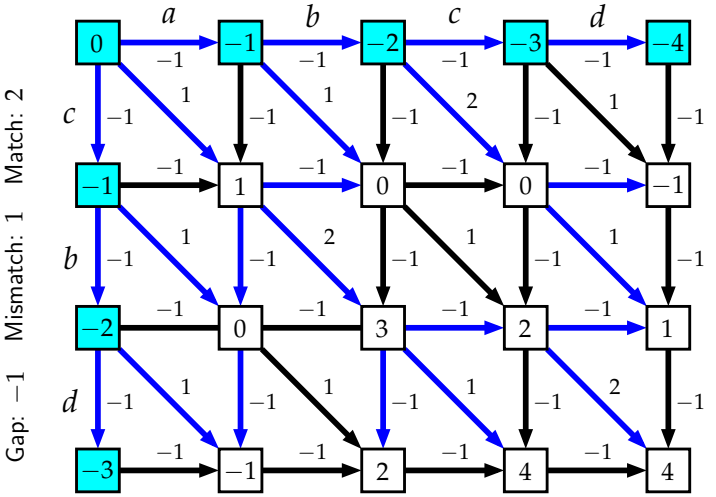
Dynamic programming approach: Top-down implementation



Overlapping between subproblems

Two-sequence alignment problem (cont'd)

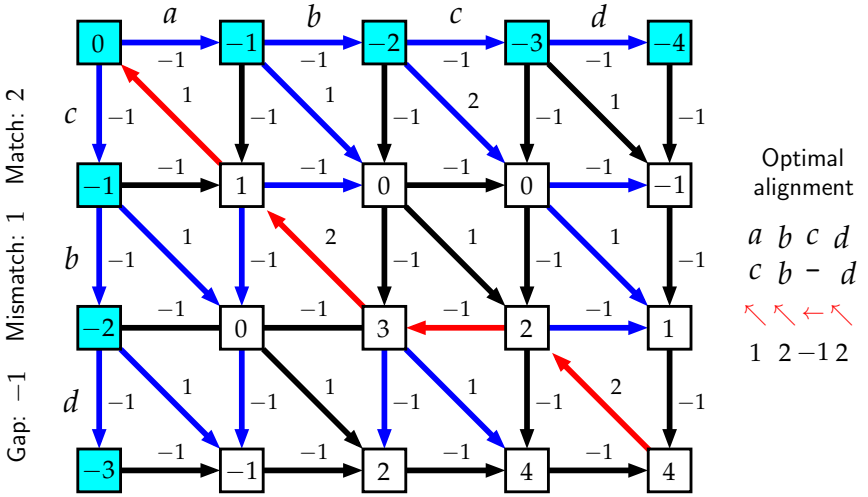
Dynamic programming approach: Bottom-up implementation



Computation of optimal alignment score

Two-sequence alignment problem (cont'd)

Dynamic programming approach: Bottom-up implementation



Backtracking for finding optimal alignment

Two-sequence alignment problem (cont'd)

Dynamic programming approach: Time complexity

- The time and space complexities of the dynamic programming approach are proportion to the size of the matrix A, both of which are $\mathcal{O}(mn)$.

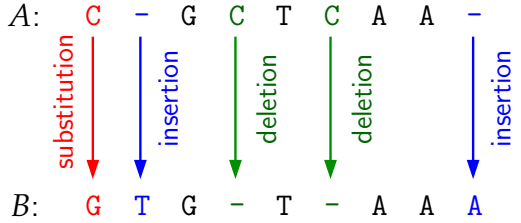
Edit operations

- Let $A = a_1 a_2 \dots a_m$ and $B = b_1 b_2 \dots b_n$ be two sequences.

Three edit operations:

- Deletion: delete a character from A.
- Insertion: insert a character into A.
- Substitution: replace a character in A with another character.

- Example Let $A =$ CGCTCAA and $B =$ GTGTAAA.



Edit distance

- We can associate an cost with each edit operation.
- The edit distance is the minimum cost associated with the edit operations needed to transform A to B.
- If the cost is one for each edit operation, then the edit distance becomes the minimum number of the edit operations needed to transform A to B.

Edit distance (cont'd)

Dynamic programming approach: Recursive formula

- Let α be the cost of insertion.
- Let β be the cost of deletion.
- Let γ be the cost of substitution.
- Let $A_{i,j}$ denote the edit distance between between $a_1 a_2 \dots a_i$ and $b_1 b_2 \dots b_j$.

Initial conditions:

- $A_{0,0} = 0$.
- $A_{i,0} = i\beta$.
- $A_{0,j} = j\alpha$.

Edit distance (cont'd)

Dynamic programming approach: Recursive formula

Recursive formula:

$$A_{i,j} = \min \begin{cases} A_{i,j-1} + \alpha, \\ A_{i-1,j} + \beta, \\ A_{i-1,j-1} + \gamma \times \delta \end{cases}$$

where $\delta = 0$ if $a_i = b_j$; otherwise, $\delta = 1$.

- Actually, the edit distance problem is equivalent to the optimal alignment problem.
- A table with $(m+1) \times (n+1)$ entries is needed to record $A_{i,j}$'s.
- Therefore, it takes $\mathcal{O}(mn)$ time to find an edit distance for two sequences $A = a_1 a_2 \dots a_m$ and $B = b_1 b_2 \dots b_n$.

Longest common subsequence

- Consider a string $A = abaade$.
- A subsequence of A is obtained by deleting zero or more (not necessarily consecutive) characters from A.
- Example Both ab and bde are subsequences of A.
- A common subsequence between two strings A and B is a subsequence of both A and B.
- Example Let $A = abaadec$ and $B = caacedc$. Then ad and $aaec$ are common subsequences of A and B.

Longest common subsequence problem

Definition of longest common subsequence (LCS) problem:

Given two strings A and B, the problem is to find a longest common subsequence between A and B.

- It is solvable by an exhaustive search (in exponential time):

Example: Let $A = abaadec$ and $B = bafc$

- The length of LCS must be less than or equal to that of the shorter string.
- We may try to determine if $bafc$ is a common subsequence.
- We then try subsequences chosen from B with length 3.
- Since bac is a common subsequence, it must also be an LCS of A and B because we started from the longest possible one.

Longest common subsequence problem (cont'd)

Dynamic programming approach

- Let $A = a_1 a_2 \dots a_m$ and $B = b_1 b_2 \dots b_n$.
- Let's pay attention to the last two characters: a_m and b_n.

Case 1: $a_m = b_n$

- In this case, the LCS must contain a_m.
- Hence, we only have to find the LCS of $a_1 a_2 \dots a_{m-1}$ and $b_1 b_2 \dots b_{n-1}$.

Case 2: $a_m \neq b_n$

- In this case, we have to match $a_1 a_2 \dots a_m$ with $b_1 b_2 \dots b_{n-1}$ and also $a_1 a_2 \dots a_{m-1}$ with $b_1 b_2 \dots b_n$.
- Whatever produces a longer LCS, this will be our LCS.

Longest common subsequence problem (cont'd)

Dynamic programming approach: Recursive formula

- Let $L_{i,j}$ be the length of LCS of $a_1 a_2 \dots a_i$ and $b_1 b_2 \dots b_j$.

Initial conditions:

- $L_{0,0} = L_{0,j} = L_{i,0} = 0$ for $1 \leq i \leq m$ and $1 \leq j \leq n$.

Recursive formula:

$$L_{i,j} = \begin{cases} L_{i-1,j-1} + 1 & \text{if } a_i = b_j \\ \max\{L_{i-1,j}, L_{i,j-1}\} & \text{if } a_i \neq b_j \end{cases}$$

- The time complexity of the algorithm is $\mathcal{O}(mn)$.

Resource allocation problem

- We are given m resources and n projects, where a profit $P(i,j)$ will be obtained if j resources are allocated to project i, where $1 \leq j \leq m$.
- Then the resource allocation problem is to find an allocation of resources to maximize the total profit.

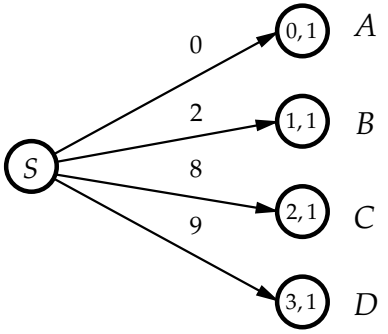
Example:

P(project, resource)	1	2	3
1	2	8	9
2	5	6	7
3	4	4	4
4	2	4	5

Resource allocation problem (cont'd)

Dynamic programming approach

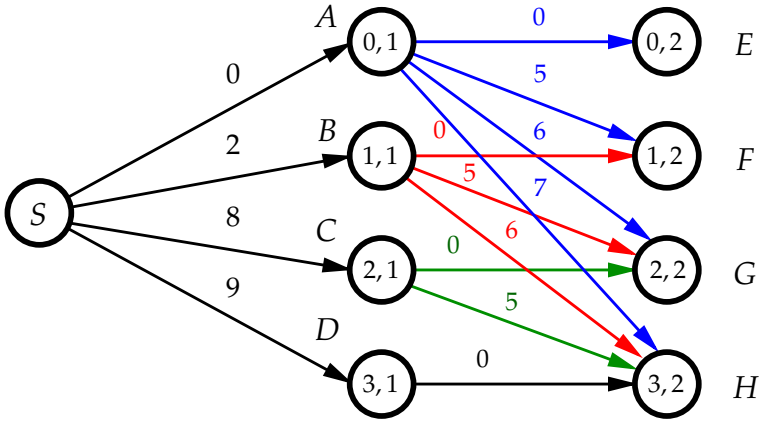
- Transform the problem into a problem of finding a longest path in a multi-stage graph.
- Let (i,j) be the state where i resources have been allocated to projects $1, 2, \dots, j$.
- Initially, we have only 4 states:



Resource allocation problem (cont'd)

Dynamic programming approach

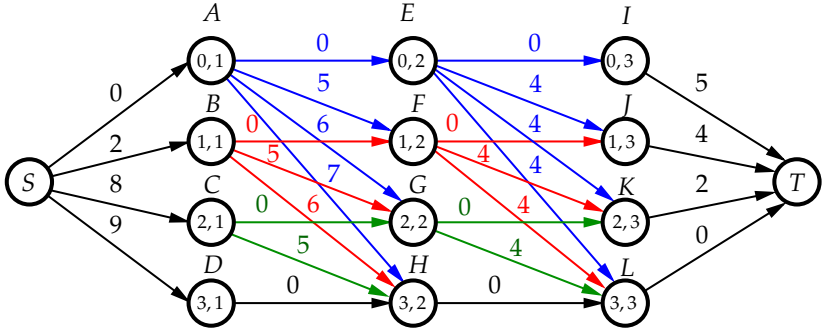
- After allocating i resources to project 1, we can allocate at most $3-i$ resource to project 2.



Resource allocation problem (cont'd)

Dynamic programming approach

- Finally, the problem can be described as follows.

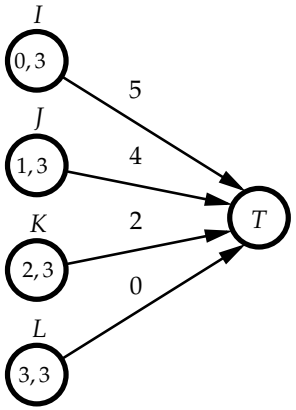


- Then the longest path $(S \to (2,1) \to (3,2) \to (3,3) \to T)$ corresponds to an optimal solution.
- To find the longest path from S to T, we use the backward reasoning approach:

Resource allocation problem (cont'd)

Dynamic programming approach

Step 1: The longest paths from I, J, K and L to T are:



Resource allocation problem (cont'd)

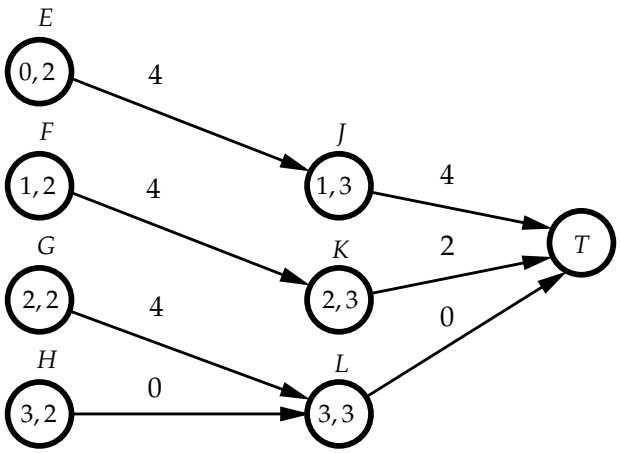
Dynamic programming approach

Step 2: After that, we can obtain the longest paths from E, F, G and H to T easily.

$$
\begin{aligned}
d(E,T) &= \max\left\{\begin{array}{l} d(E,I)+d(I,T) \\ d(E,J)+d(J,T) \\ d(E,K)+d(K,T) \\ d(E,L)+d(L,T) \end{array}\right\} \\
&= \max\{0+5, 4+4, 4+2, 4+0\} \\
&= \max\{5,8,6,4\} \\
&= 8
\end{aligned}
$$

Resource allocation problem (cont'd)

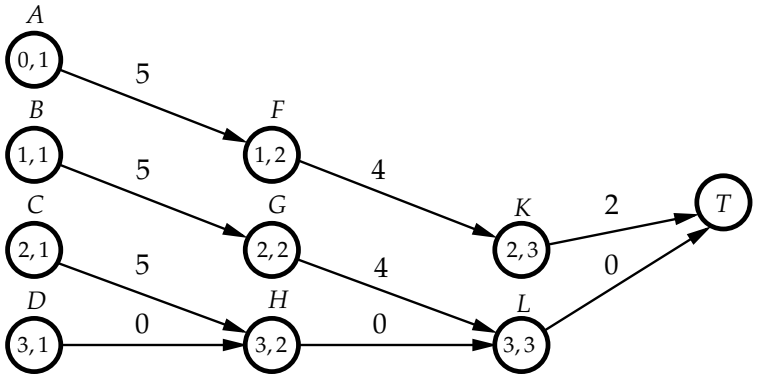
Dynamic programming approach



Resource allocation problem (cont'd)

Dynamic programming approach

Step 3: The longest paths from A, B, C and D to T can be found by the same method.



Resource allocation problem (cont'd)

Dynamic programming approach

Step 4: Finally, the longest path from S to T is obtained:

$$\begin{aligned} d(S,T) &= \max\left\{\begin{array}{l} d(S,A)+d(A,T) \\ d(S,B)+d(B,T) \\ d(S,C)+d(C,T) \\ d(S,D)+d(D,T) \end{array}\right\} \\ &= \max\{0+11, 2+9, 8+5, 9+0\} \\ &= \max\{11, 11, 13, 9\} \\ &= 13 \end{aligned}$$

As a result, the longest path is $S \to C \to H \to L \to T$.

- 2 resources allocated to project 1.
- 1 resources allocated to project 2.
- 0 resources allocated to project 3.
- 0 resources allocated to project 4.

0/1 Knapsack problem

Definition:

- We are given n objects and a knapsack.
- Object i has a weight W_i and the knapsack has a capacity M.
- If object i is placed into the knapsack, we will obtain a profit P_i.
- Then the 0/1 knapsack problem is to maximize the total profit under the constraint that the total weight of all chosen objects is at most M.

Example:

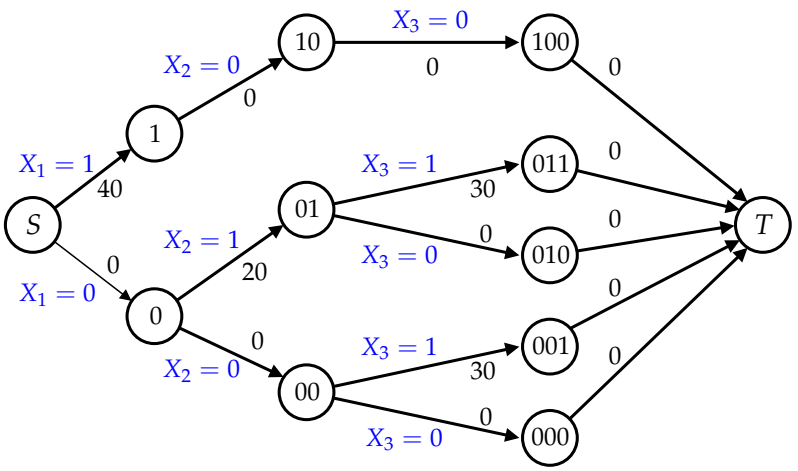
Suppose we are given 3 objects, whose weights and profits are shown in the right table, and a knapsack with capacity 10.

i	W_i	P_i
1	10	40
2	3	20
3	5	30

0/1 Knapsack problem (cont'd)

Dynamic programming approach

- Actually, the 0/1 knapsack problem can be represented by a multi-stage graph problem.



0/1 Knapsack problem (cont'd)

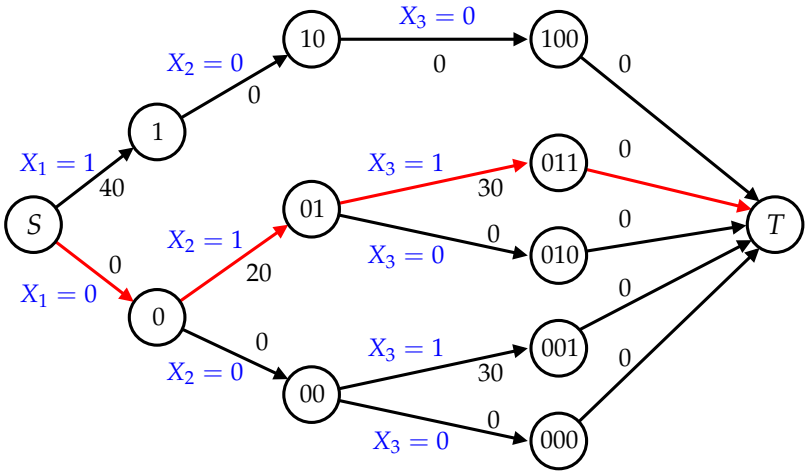
Dynamic programming approach

- Let X_i be the variable denoting whether object i is chosen.
- Let $X_1 = 1$ if object i is chosen and 0 if it is not.
- In each node, we have a label specifying the decision already made up to this node.
- Example 011 means $X_1 = 0, X_2 = 1, X_3 = 1$.
- A path from start node to stop node corresponds to a feasible solution of the 0/1 knapsack problem.
- The longest path $S \to 0 \to 01 \to 011 \to T$ corresponds to an optimal solution with cost 50.

0/1 Knapsack problem (cont'd)

Dynamic programming approach

- The longest path $S \to 0 \to 01 \to 011 \to T$ corresponds to $X_1 = 0, X_2 = 1$ and $X_3 = 1$.

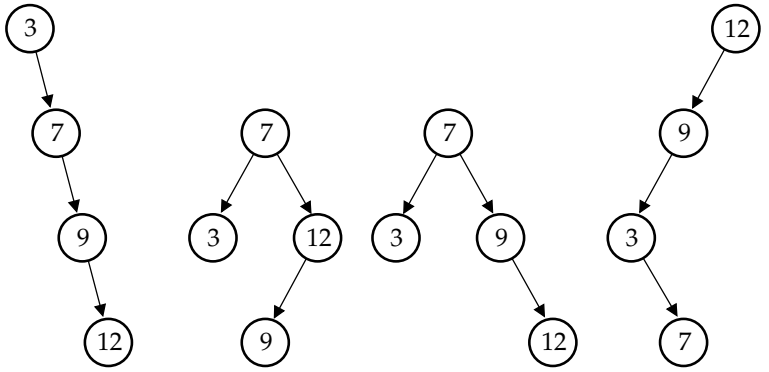


Binary search tree

- Binary search tree is a binary tree in which for any node x, the identifiers (or keys/values) in the left subtree of x are at most the identifier of x and the identifiers in the right subtree of x are at least the identifier of x.

Binary search tree (cont'd)

- For example, four distinct binary search trees are shown below for a set of four identifiers 3, 7, 9 and 12.



Optimal binary search tree problem

- For a given binary tree, the identifiers stored close to the root of the tree can be searched rather quickly, while it takes more steps to retrieve data stored far away from the root.
- For each identifier a_i, we associate with it a probability P_i with which a_i will be searched.
- For an identifier not stored in the tree, we assume that there is a probability associated with it.

Optimal binary search tree problem (cont'd)

- Suppose that we are given n different identifiers $a_1 < a_2 < \ldots < a_n$.
- Then we can partition the remaining identifiers into $n+1$ equivalence classes as follows, where we assume $a_0 = -\infty$ and $a_{n+1} = \infty$.

$$-\infty \equiv a_0 \overset{\text{class } 1}{\uparrow} a_1 \overset{\text{class } 2}{\uparrow} a_2 \overset{\text{class } 3}{\uparrow} \ldots \overset{\text{class } n}{\uparrow} a_n \overset{\text{class } n+1}{\uparrow} a_{n+1} \equiv \infty$$

- Let Q_i denote the probability that X will be searched, where $a_i < X < a_{i+1}$.
- The probabilities satisfy the following equation:

$$\sum_{i=1}^{n} P_i + \sum_{i=0}^{n} Q_i = 1$$

Optimal binary search tree problem (cont'd)

Question 1:

How to determine the number of steps needed for a successful search?

- Let $L(a_i)$ denote the level of the binary search tree where a_i is stored.
- Then the retrieval of a_i needs $L(a_i)$ steps if we let $L(root) = 1$.

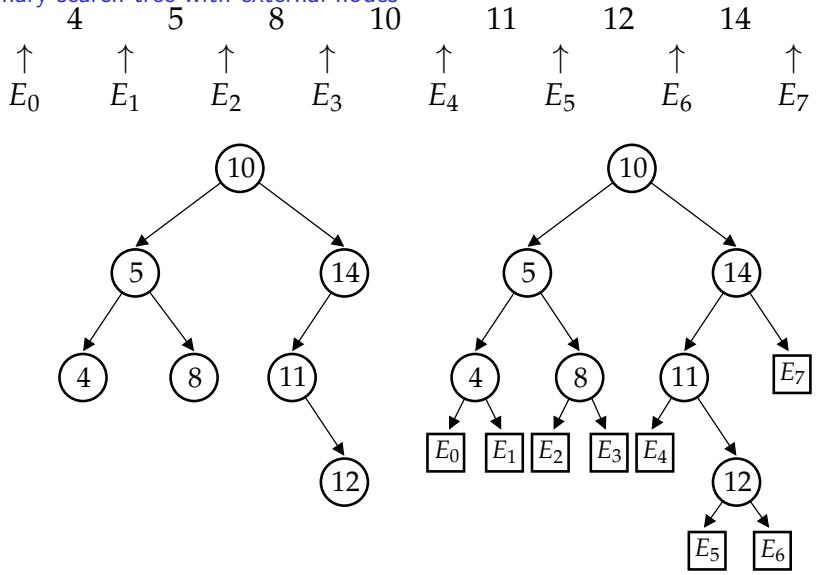
Question 2:

How to determine the number of steps needed for an unsuccessful search?

- The best way is to add external nodes to the binary search tree.

Optimal binary search tree problem (cont'd)

Binary search tree with external nodes



Optimal binary search tree problem (cont'd)

- The circle nodes are called internal nodes and the square nodes are called external nodes.
- Clearly, successful searches always terminate at internal nodes, while unsuccessful searches terminate at external nodes.
- Hence, the expected cost of a binary search tree with external nodes can be expressed as:

$$\sum_{i=1}^{n} P_i L(a_i) + \sum_{i=0}^{n} Q_i (L(E_i) - 1)$$

Optimal binary tree problem (cont'd)

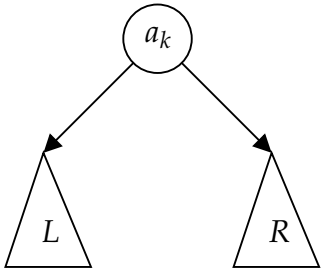
- An optimal binary search tree is a binary search tree with the minimum cost.
- Note that given n identifiers, the number of all distinct binary search trees is $\frac{1}{n+1}\binom{2n}{n}$[2], which is approximately $\frac{4^n}{\sqrt{\pi} n^{1.5}}$.
- It indicates that the exhaustive approach is time consuming.
- Actually, an optimal binary search tree can be found by the dynamic programming approach in a more efficient way.

[2] D. E. Knuth, Optimum binary search trees, Acta Informatica, vol. 1, pp. 14–25, 1971.

Optimal binary search tree problem (cont'd)

Dynamic programming approach

- A critical first step is to select an identifier, say a_k, to be the root of the optimal binary search tree.
- After a_k is selected, all identifiers smaller than a_k will constitute the left descendants of a_k and all identifiers greater than a_k will constitute the right descendants.



Optimal binary search tree problem (cont'd)

Dynamic programming approach

- We cannot determine which a_k should be selected as the root of the binary search tree in advance, and hence we need to examine all possible a_k's.
- However, once a_k is selected as the root, both subtrees L and R must be optimal.
- That is, after a_k is selected, our job reduces to finding optimal binary search trees for identifiers smaller than a_k and identifiers larger than a_k.

Optimal binary search tree problem (cont'd)

Basic idea of dynamic programming approach

- The dynamic programming approach would solve the problem by working bottom up.
- We start by building small optimal binary search trees.
- Using these small optimal binary search trees, we can build larger and larger optimal binary search trees.
- We reach our goal when an optimal binary search tree containing all identifiers has been found.

Optimal binary search tree problem (cont'd)

Dynamic programming approach

- Assume that we are given n different identifiers $a_1 < a_2 < \ldots < a_n$ and a_k is one of them.
- Suppose that a_k is selected as the root of binary search tree.
- Then it is clear that the left subtree contains $a_1, \ldots, a_{k-1}$ and the right subtree contains $a_{k+1}, \ldots, a_n$.
- The retrieval of a_k needs one step.
- All other successful searches and all the unsuccessful searches need one plus the number of steps needed to search either the left or the right subtree.

Optimal binary search tree problem (cont'd)

Dynamic programming approach

- Let $C(i,j)$ denote the cost of an optimal binary search tree containing identifiers a_i to a_j.
- The cost of an optimal binary search tree containing all identifiers is:

$$C(1,n) = \min_{1 \le k \le n} \{❶ + ❷ + ❸\}$$

❶ $= P_k$.

That is, the cost of searching for root.

❷ $= Q_0 + \sum_{l=1}^{k-1}(P_l + Q_l) + C(1,k-1)$.

That is, the cost of searching the left subtree.

❸ $= Q_k + \sum_{l=k+1}^{n}(P_l + Q_l) + C(k+1,n)$.

That is, the cost of searching the right subtree.

Optimal binary search tree problem (cont'd)

Dynamic programming approach

- The above formula can be generalized to obtain any $C(i,j)$ as follows.

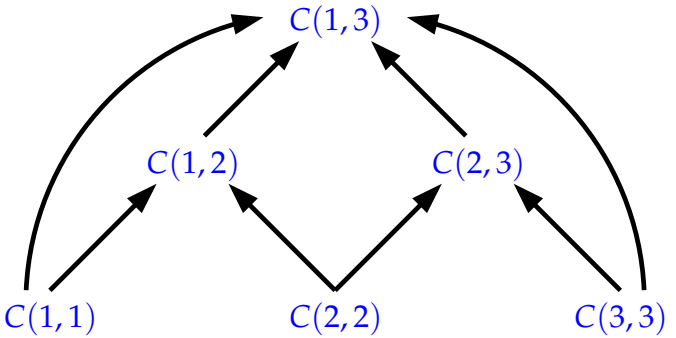
$$C(i,j) = \min_{i \le k \le j} \left\{ P_k + \left[Q_{i-1} + \sum_{l=i}^{k-1} (P_l + Q_l) + C(i,k-1) \right] + \left[Q_k + \sum_{l=k+1}^{j} (P_l + Q_l) + C(k+1,j) \right] \right\}$$

$$= \min_{i \le k \le j} \left\{ C(i,k-1) + C(k+1,j) + \sum_{l=i}^{j} (P_l + Q_l) + Q_{i-1} \right\}$$

Optimal binary search tree problem (cont'd)

Computational relationships of subtrees

- For example, if we have four identifiers a_1, a_2 and a_3, then the objective will be finding $C(1,3)$.



Optimal binary search tree problem (cont'd)

Time complexity of dynamic programming method

- To compute $C(1,n)$, we proceed by first computing all $C(i,j)$'s with $j-i=0$, then all $C(i,j)$'s with $j-i=1$, and so on.
- In other words, this procedure requires to compute $C(i,j)$ for all $j-i=0,1,\ldots,n-1$.
- When $j-i=m$, there are $(n-m)$ $C(i,j)$'s to compute.
- Example If $m=0$, there $n-m=n$ $C(i,j)$'s to compute.
- Computing each $C(i,j)$ with $j-i=m$ requires us to find the minimum of $m+1$ quantities.
- In other words, the total time for computing all $C(i,j)$'s with $j-i=m$ is $\mathcal{O}(mn-m^2)$.

Optimal binary search tree problem (cont'd)

Time complexity of dynamic programming method

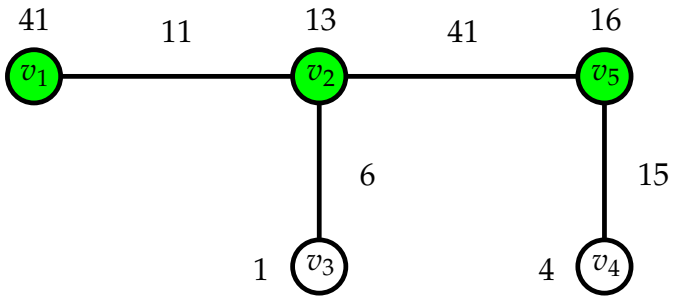
- Hence, the total time complexity of the dynamic programming algorithm to solve the optimal binary search tree problem is:

$$\mathcal{O}\left(\sum_{0 \le m \le n-1} \left(mn - m^2 \right) \right) = \mathcal{O}(n^3)$$

Perfect domination set

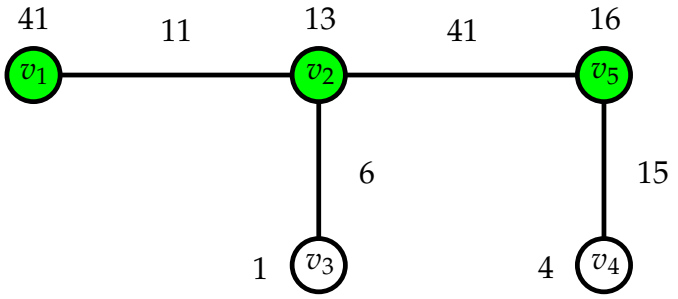
Definition:

- Suppose that we are given a graph $G = (V, E)$, where every vertex $v \in V$ has a cost $c(v)$ and every edge $e \in E$ also has a cost of $c(e)$.
- A perfect domination set of G is a subset D of V such that every vertex not in D is adjacent to exactly one vertex in D.



Cost of a perfect domination set

- The cost of a perfect dominating set D includes all the costs of the vertices in D and the cost of $c(u, v)$ if $u \in D$, $v \notin D$ and $(u, v) \in E$.
- Example $\{v_1, v_2, v_5\}$ is a perfect dominating set and its cost is 91.

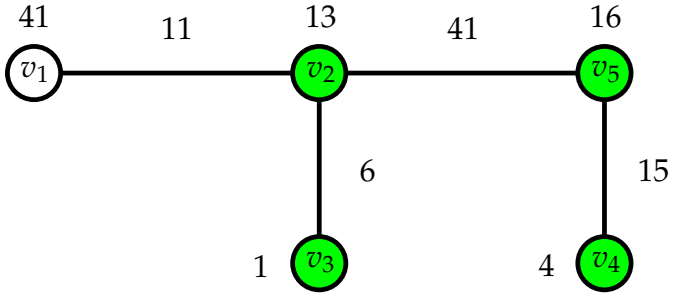


Weighted perfect domination problem

Definition:

The weighted perfect domination problem is to find a perfect domination set with minimum cost.

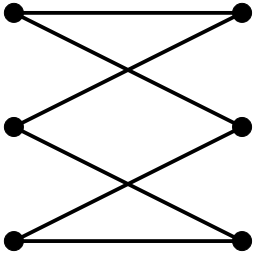
- Example $\{v_2, v_3, v_4, v_5\}$ is a minimal cost perfect dominating set whose cost is 45.



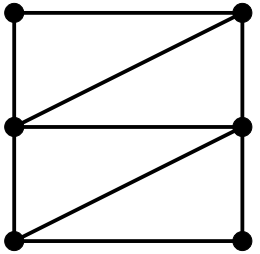
Weighted perfect domination problem (cont'd)

Bipartite and chordal graphs

- A bipartite graph is a graph whose vertices can be partitioned into two disjoint sets A and B such that every edge connects a vertex in A to one in B.
- A graph is chordal if each of its cycles of four or more nodes has a chord, which is an edge joining two nodes that are not adjacent in the cycle.



Bipartite graph



Chordal graph

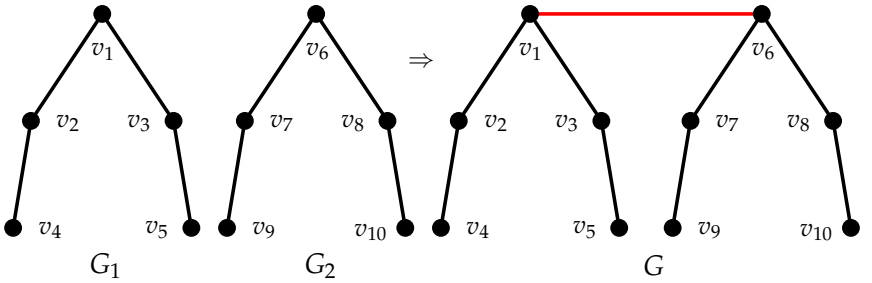
Weighted perfect domination problem (cont'd)

- It is well known that the weighted perfect domination problem is NP-hard for bipartite graphs and chordal graphs.
- However, this problem on trees is solvable in polynomial time.
- Below, we shall show how to utilize the dynamic programming approach to solve the weighted perfect domination problem on trees.

Weighted perfect domination on trees

Dynamic programming approach

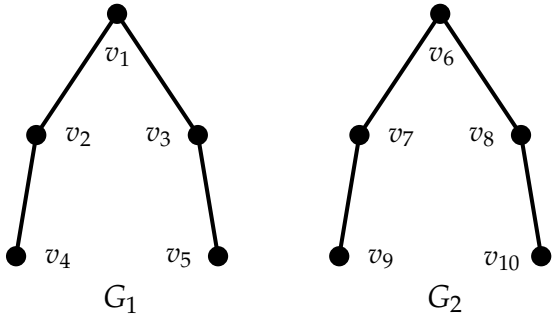
- The main job of the dynamic programming strategy is to merge two feasible solutions into a new feasible solution.
- Example Suppose that G_1 and G_2 can be merged into G by linking v_1 and v_6.



Weighted perfect domination on trees (cont'd)

Dynamic programming approach

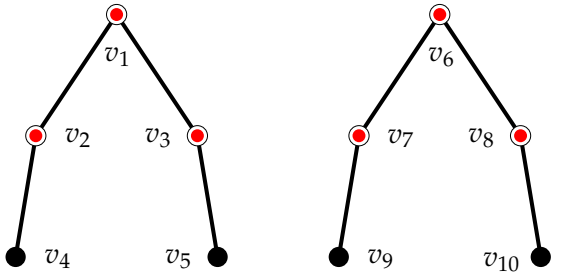
- Let's consider six vertex subsets as follows:
  - $D_{11} = \{v_1, v_2, v_3\}$
  - $D_{12} = \{v_3, v_4\}$
  - $D_{13} = \{v_4, v_5\}$
  - $D_{21} = \{v_6, v_7, v_8\}$
  - $D_{22} = \{v_7, v_{10}\}$
  - $D_{23} = \{v_9, v_{10}$



Weighted perfect domination on trees (cont'd)

Perfect domination sets D_{11} and D_{21}

- D_{11} is a perfect dominating set of G_1 under the condition that the root v_1 of G_1 is included in it.
- D_{21} is a perfect dominating set of G_2 under the condition that the root v_6 of G_2 is included in it.

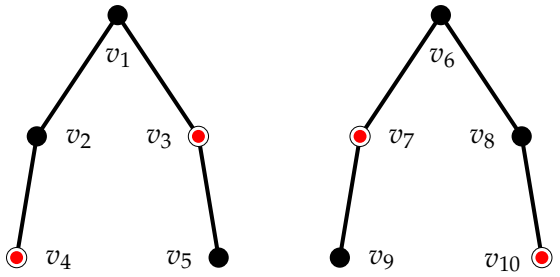


$D_{11} = \{v_1, v_2, v_3\}$ $D_{21} = \{v_6, v_7, v_8\}$

Weighted perfect domination on trees (cont'd)

Perfect domination sets D_{12} and D_{22}

- D_{12} is a perfect dominating set of G_1 under the condition that the root v_1 of G_1 is not included in it.
- D_{22} is a perfect dominating set of G_2 under the condition that the root v_6 of G_2 is not included in it.

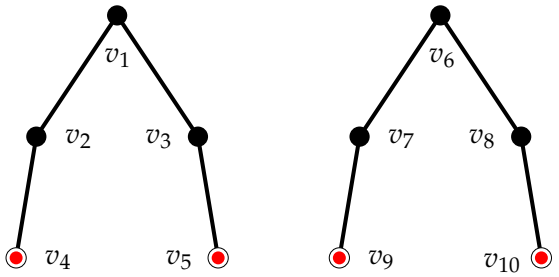


$D_{12} = \{v_3, v_4\}$ $D_{22} = \{v_7, v_{10}\}$

Weighted perfect domination on trees (cont'd)

Special perfect domination sets D_{13} and D_{23}

- D_{13} is a perfect dominating set of $G_1 \setminus \{v_1\}$ under the condition that none of v_1's neighbors are not included in it.
- D_{23} is a perfect dominating set of $G_2 \setminus \{v_6\}$ under the condition that none of v_6's neighbors are not included in it.

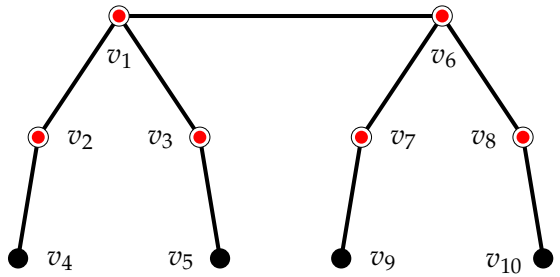


$D_{13} = \{v_4, v_5\}$ $D_{23} = \{v_9, v_{10}\}$

Weighted perfect domination on trees (cont'd)

How to merge two sub-solutions to a bigger solution?

- Actually, we can produce perfect domination sets for G by combining the above six perfect domination sets.
- Case 1 $D_{11} \cup D_{21} = \{v_1, v_2, v_3, v_6, v_7, v_8\}$ is a perfect domination set on G that includes both v_1 and v_6.
- The cost of $D_{11} \cup D_{21}$ is the sum of costs of D_{11} and D_{12}.

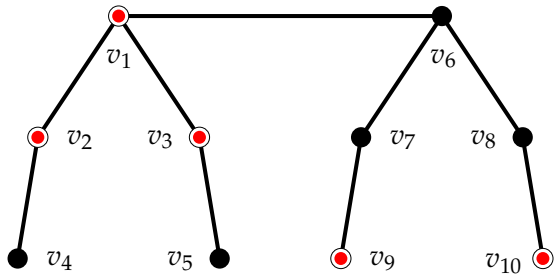


$D_{11} \cup D_{12} = \{v_1, v_2, v_3, v_6, v_7, v_8\}$

Weighted perfect domination on trees (cont'd)

How to merge two sub-solutions to a bigger solution?

- Case 2 $D_{11} \cup D_{23} = \{v_1, v_2, v_3, v_9, v_{10}\}$ is a perfect domination set on G that includes v_1 but not v_6.
- The cost of $D_{11} \cup D_{23}$ is the sum of costs of D_{11} and D_{23} plus $c(v_1, v_6)$.

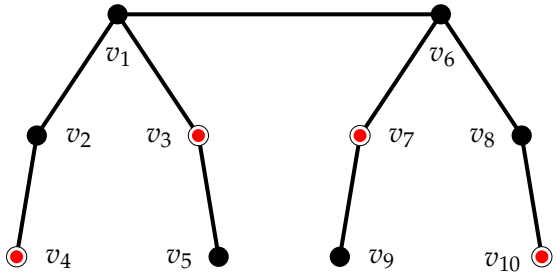


$D_{11} \cup D_{23} = \{v_1, v_2, v_3, v_9, v_{10}\}$

Weighted perfect domination on trees (cont'd)

How to merge two sub-solutions to a bigger solution?

- Case 3 $D_{12} \cup D_{22} = \{v_3, v_4, v_7, v_{10}\}$ is a perfect domination set on G that includes neither v_1 nor v_6.
- The cost of $D_{12} \cup D_{22}$ is the sum of costs of D_{12} and D_{22}.

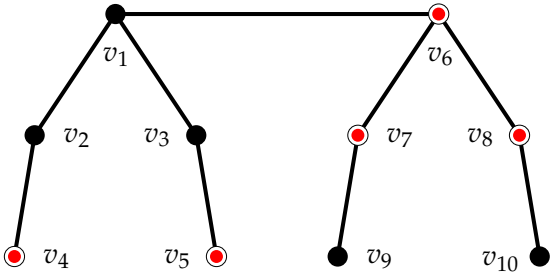


$$D_{12} \cup D_{22} = \{v_3, v_4, v_7, v_{10}\}$$

Weighted perfect domination on trees (cont'd)

How to merge two sub-solutions to a bigger solution?

- Case 4 $D_{13} \cup D_{21} = \{v_4, v_5, v_6, v_7, v_8\}$ is a perfect domination set on G that includes v_6 but not v_1.
- The cost of $D_{13} \cup D_{21}$ is the sum of costs of D_{13} and D_{21} plus $c(v_1, v_6)$.



$$D_{13} \cup D_{21} = \{v_4, v_5, v_6, v_7, v_8\}$$

Weighted perfect domination on trees (cont'd)

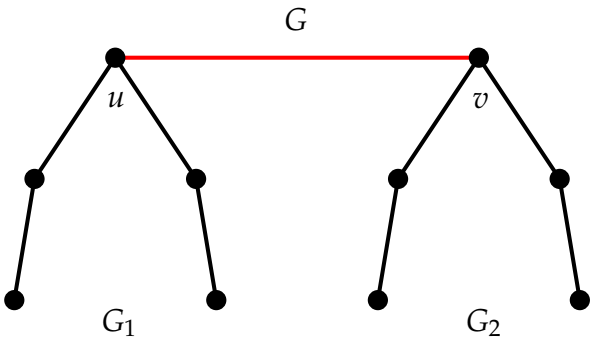
Dynamic programming approach

- Let G be a tree rooted at a certain vertex u.
- Let $D_1(G, u)$ denote an optimal perfect domination set for G under the condition that it includes u.
- The cost of $D_1(G, u)$ is denoted as $\delta_1(G, u)$.
- Let $D_2(G, u)$ denote an optimal perfect domination set for G under the condition that it doest not include u.
- The cost of $D_2(G, u)$ is denoted as $\delta_2(G, u)$.
- Let $D_3(G, u)$ be an optimal perfect domination set for $G \setminus \{u\}$ under the condition that it does not include any neighboring vertex of u.
- The cost of $D_3(G, u)$ is denoted as $\delta_3(G, u)$.

Weighted perfect domination on trees (cont'd)

Dynamic programming approach

- Given two trees G_1 and G_2 rooted at u and v, respectively, let G denote the tree obtained by linking u and v.

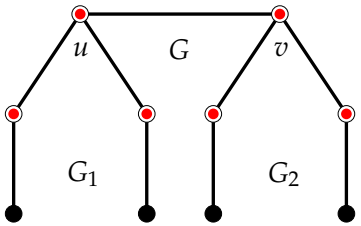


- Let $D(G)$ denote an optimal perfect domination set of G and its cost be $\delta(G)$.

Weighted perfect domination on trees (cont'd)

Dynamic programming approach

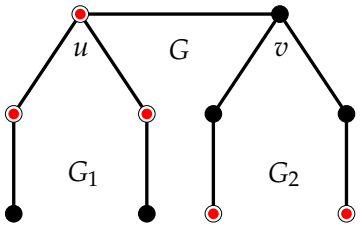
- Rule 1 Both $D_1(G_1,u) \cup D_1(G_2,v)$ and $D_1(G_1,u) \cup D_3(G_2,v)$ are perfect domination sets of G which include u.



$D_1(G_1,u) \cup D_1(G_2,v)$

$\delta_1(G_1,u) + \delta_1(G_2,v)$

Rule 1.1



$D_1(G_1,u) \cup D_3(G_2,v)$

$\delta_1(G_1,u) + \delta_3(G_2,v) + c(u,v)$

Rule 1.2

Weighted perfect domination on trees (cont'd)

Dynamic programming approach

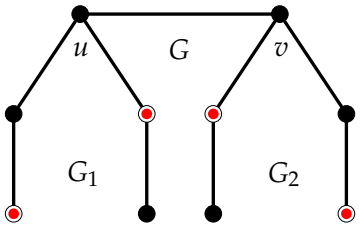
Computation of $D_1(G,u)$ and $\delta_1(G,u)$:

1. **if** $\delta_1(G_1,u) + \delta_1(G_2,v) < \delta_1(G_1,u) + \delta_3(G_2,v) + c(u,v)$ **then**
2. $\quad D_1(G,u) = D_1(G_1,u) \cup D_1(G_2,v)$.
3. $\quad \delta_1(G,u) = \delta_1(G_1,u) + \delta_1(G_2,v)$.
4. **else**
5. $\quad D_1(G,u) = D_1(G_1,u) \cup D_3(G_2,v)$.
6. $\quad \delta_1(G,u) = \delta_1(G_1,u) + \delta_3(G_2,v) + c(u,v)$.
7. **end if**

Weighted perfect domination on trees (cont'd)

Dynamic programming approach

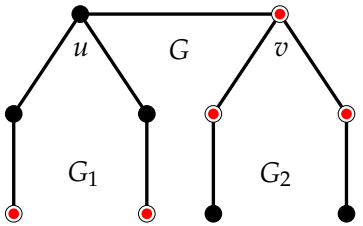
- Rule 2 Both $D_2(G_1,u) \cup D_2(G_2,v)$ and $D_3(G_1,u) \cup D_1(G_2,v)$ are perfect domination sets of G which does not include u.



$D_2(G_1,u) \cup D_2(G_2,v)$

$\delta_2(G_1,u) + \delta_2(G_2,v)$

Rule 2.1



$D_3(G_1,u) \cup D_1(G_2,v)$

$\delta_3(G_1,u) + \delta_1(G_2,v) + c(u,v)$

Rule 2.2

Weighted perfect domination on trees (cont'd)

Dynamic programming approach

Computation of $D_2(G,u)$ and $\delta_2(G,u)$:

1. **if** $\delta_2(G_1,u) + \delta_2(G_2,v) < \delta_3(G_1,u) + \delta_1(G_2,v) + c(u,v)$ **then**
2. $\quad D_2(G,u) = D_2(G_1,u) \cup D_2(G_2,v)$
3. $\quad \delta_2(G,u) = \delta_2(G_1,u) + \delta_2(G_2,v)$
4. **else**
5. $\quad D_2(G,u) = D_3(G_1,u) \cup D_1(G_2,v)$
6. $\quad \delta_2(G,u) = \delta_3(G_1,u) + \delta_1(G_2,v) + c(u,v)$
7. **end if**

Weighted perfect domination on trees (cont'd)

Dynamic programming approach

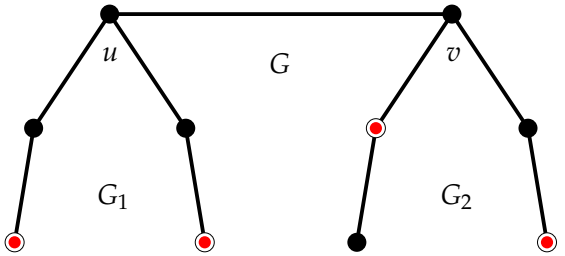
Computation of $D(G)$ and $\delta(G)$:

1. **if** $\delta_1(G,u) < \delta_2(G,u)$ **then**
2. $\quad \delta(G) = \delta_1(G,u)$
3. $\quad D(G) = D_1(G,u)$
4. **else**
5. $\quad \delta(G) = \delta_2(G,u)$
6. $\quad D(G) = D_2(G,u)$
7. **end if**

Weighted perfect domination on trees (cont'd)

Dynamic programming approach

- $D_3(G,u) = D_3(G_1,u) \cup D_2(G_2,v)$ is a perfect domination sets of $G \setminus \{u\}$.



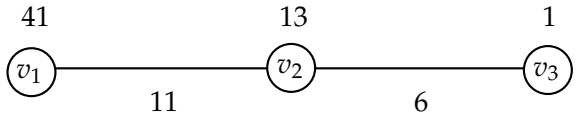
Computation of $D_3(G,u)$ and $\delta_3(G,u)$:

1. $D_3(G,u) = D_3(G_1,u) \cup D_2(G_2,v)$.
2. $\delta_3(G,u) = \delta_3(G_1,u) + \delta_2(G_2,v)$.

Weighted perfect domination on trees (cont'd)

Dynamic programming approach

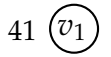
- For example, consider the following weighted tree:



Weighted perfect domination on trees (cont'd)

Dynamic programming approach

- Let's start from leaf node v_1 only.

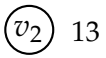


- Its perfect domination set either contains v_1 or does not contain v_1.
  - $D_1(\{v_1\},v_1) = \{v_1\}$
  - $D_2(\{v_1\},v_1)$ does not exist.
  - $D_3(\{v_1\},v_1) = \varnothing$
  - $\delta_1(\{v_1\},v_1) = c(v_1) = 41$
  - $\delta_2(\{v_1\},v_1) = \infty$
  - $\delta_3(\{v_1\},v_1) = 0$

Weighted perfect domination on trees (cont'd)

Dynamic programming approach

- Consider the subtree containing v_2 only.

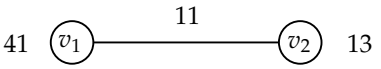


- Again, we have the following result:
  - $D_1(\{v_2\}, v_2) = \{v_2\}$
  - $D_2(\{v_2\}, v_2)$ does not exist.
  - $D_3(\{v_2\}, v_2) = \varnothing$
  - $\delta_1(\{v_2\}, v_2) = c(v_2) = 13$
  - $\delta_2(\{v_2\}, v_2) = \infty$
  - $\delta_3(\{v_2\}, v_2) = 0$

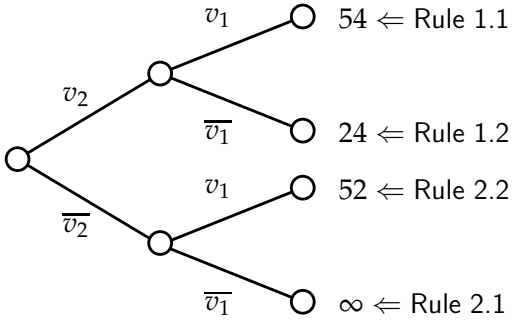
Weighted perfect domination on trees (cont'd)

Dynamic programming approach

- Now, consider the subtree containing v_1 and v_2:



- The computation of the perfect domination sets for $\{v_1, v_2\}$ can be illustrated as follows:



Weighted perfect domination on trees (cont'd)

Dynamic programming algorithm

- Since $\min\{54, 24\} = 24$, we have:
  - $D_1(\{v_1, v_2\}, v_2) = \{v_2\}$
  - $\delta_1(\{v_1, v_2\}, v_2) = 24$
- Since $\min\{\infty, 52\} = 52$, we have:
  - $D_2(\{v_1, v_2\}, v_2) = \{v_1\}$
  - $\delta_2(\{v_1, v_2\}, v_2) = 52$
- Besides, we have:
  - $D_3(\{v_1, v_2\}, v_2)$ does not exist.
  - $\delta_3(\{v_1, v_2\}, v_2) = \infty$

Weighted perfect domination on trees (cont'd)

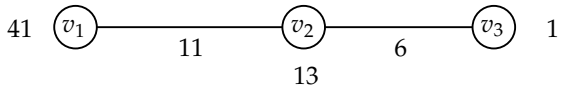
Dynamic programming approach

- Consider the subtree containing v_3 only.

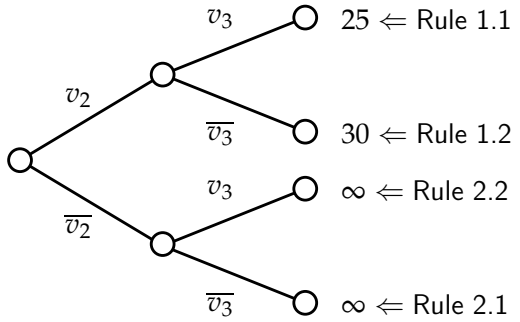
- Then we have the following result:
  - $D_1(\{v_3\}, v_3) = \{v_3\}$
  - $D_2(\{v_3\}, v_3)$ does not exist.
  - $D_3(\{v_3\}, v_3) = \varnothing$
  - $\delta_1(\{v_3\}, v_3) = c(v_3) = 1$
  - $\delta_2(\{v_3\}, v_3) = \infty$
  - $\delta_3(\{v_3\}, v_3) = 0$

Weighted perfect domination on trees (cont'd)

- We add v_3 to the subtree containing v_1 and v_2:



- The computation of the perfect domination sets for $\{v_1, v_2, v_3\}$ can be illustrated as follows:



Weighted perfect domination on trees (cont'd)

Dynamic programming approach

- Finally, we have the following result:
  - $D_1(\{v_1, v_2, v_3\}, v_2) = \{v_2, v_3\}$
  - $D_2(\{v_1, v_2, v_3\}, v_2)$ does not exist.
  - $D_3(\{v_1, v_2, v_3\}, v_2)$ does not exist.
  - $\delta_1(\{v_1, v_2, v_3\}, v_2) = c(v_2) + c(v_3) + c(v_2, v_3) = 25$
  - $\delta_2(\{v_1, v_2, v_3\}, v_2) = \infty$
  - $\delta_3(\{v_1, v_2, v_3\}, v_2) = \infty$