

Unit 6.2 Dynamic Programming, II

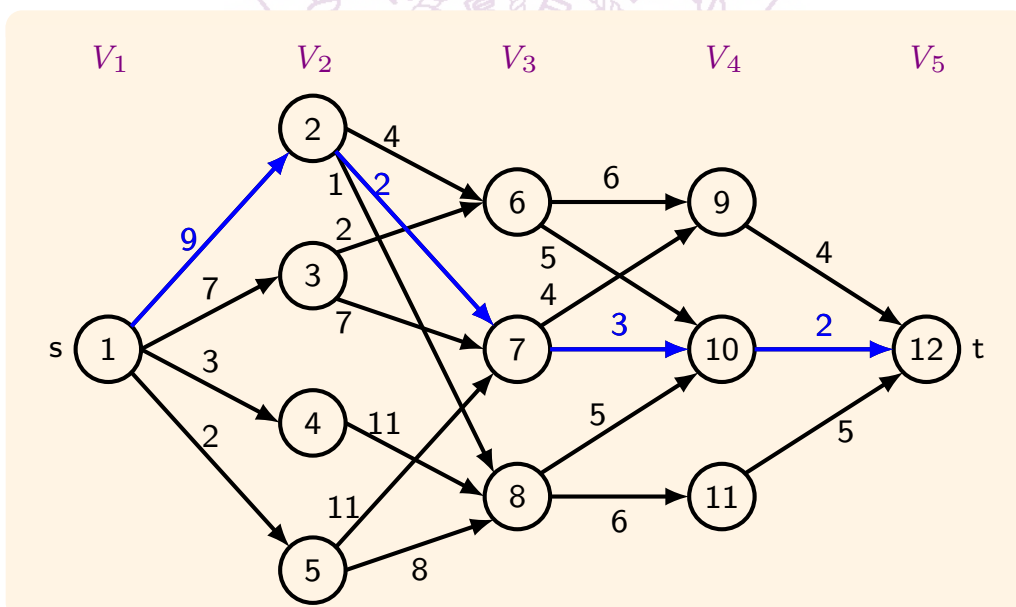
Algorithms

EE3980

May 12, 2020

Multi-Stage Graphs

- A multistage graph $G = (V, E)$ is a directed graph.
 - Vertices are partitioned into $k > 2$ disjoint sets V_i , $1 \leq i \leq k$.
 - If $\langle u, v \rangle \in E$, then $u \in V_i$ and $v \in V_{i+1}$ for some i , $1 \leq i < k$.
 - The sets V_1 and V_k both have only one vertex.
 - Vertex $s \in V_1$ is the source and $t \in V_k$ is the sink.
 - The cost of a path from s to t is the sum of the costs of the edges on the path.
 - The **multistage graph problem** is to find the minimum-cost path from s to t .

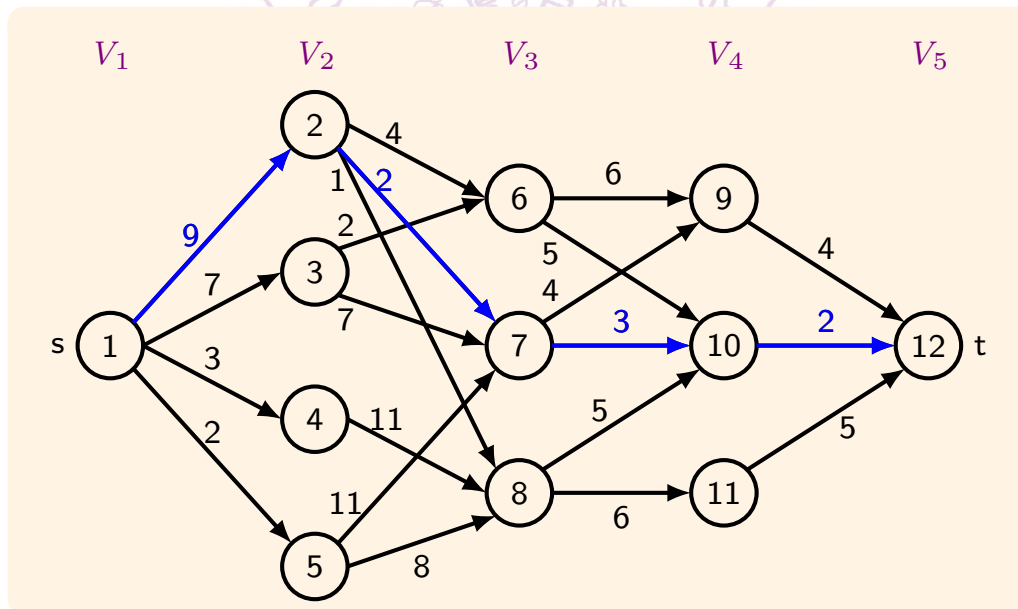


Multi-Stage Graphs — Example

- Since edges connect only consecutive stages, $\langle u, v \rangle \in E$, $u \in V_i$ and $v \in V_{i+1}$, minimum cost path from source s is

$$cost(1, 1) = \min_{\langle 1, j \rangle \in E} \{c(1, j) + cost(2, j)\} \quad (6.2.1)$$

where $cost(a, b)$ is the minimum cost of vertex b at stage a and $c(i, j)$ is the edge cost of $\langle i, j \rangle$.



Multi-Stage Graphs – Recursive Algorithm

- Note that Eq. (6.2.1) can be generalized to

$$cost(r, i) = \min_{\langle i, j \rangle \in E} \{c[i, j] + cost(r + 1, j)\} \quad (6.2.2)$$

- Therefore a recursive algorithm to solve the multistage graph problem is

Algorithm 6.2.1. Recursive Multistage Graph

```
// Find minimum cost path p of n-vertices multistage graph for vertex i.
// Input: n, cost matrix c, vertex i
// Output: mincost, path p.
1 Algorithm MSGraph_R(n, c, i, p)
2 {
3     if (i = n) then { // sink vertex
4         p[i] := 0;
5         return 0;
6     } // Otherwise, find the minimum cost path to the sink.
7     mincost := ∞; // initialize.
8     for all j such that ⟨i, j⟩ ∈ E do { // check all out-going edges
9         if (c[i, j] + MSGraph_R(n, c, j, p) < mincost) then { // smaller cost.
10             mincost := c[i, j] + MSGraph_R(n, c, j, p); p[i] := j;
11         }
12     }
13     return mincost;
14 }
```

- The vertices of the graph is assumed to be ordered from 1 to n .
 - Vertex 1 is the source vertex and n is the sink vertex.
- Matrix $c[i, j]$ is the cost of the edge $\langle i, j \rangle$.
- After completion the array $p[1 : n]$ is the minimum-cost path from source vertex to sink vertex.
- This function is invoked by `MSGraph_R($n, c, 1, p$)` at the top level and it returns the minimum path cost and the path array p .
- Though coding of this recursive version of the algorithm is straightforward, the execution efficiency can be improved.
 - For any vertex $j, j \neq 1$, with more than one edge $\langle i, j \rangle \in E$, `MSGraph_R(n, c, j, p)` can be called more than once.
 - This inefficiency can be corrected by the following algorithms.

Multi-Stage Graphs — Top-Down Approach

Algorithm 6.2.2. Multistage Graph Top-Down Approach

```
// Find minimum cost path  $p$  of  $n$ -vertices multistage graph for vertex  $i$ .
// Input:  $n$ , cost matrix  $c$ , vertex  $i$ 
// Output:  $mincost$ , path  $p$ , mincost table  $d$ .
1 Algorithm MSGraph_TD( $n, c, i, d, p$ )
2 {
3     if ( $i = n$ ) then { // sink vertex
4          $p[i] := 0$ ;
5          $d[i] := 0$ ;
6         return 0;
7     }
8     // Otherwise, find the minimum cost path to the sink.
9      $mincost := \infty$ ; // initialize.
10    for all  $j$  such that  $\langle i, j \rangle \in E$  do { // check all out-going edges
11        if ( $d[j] < 0$ ) then
12             $d[j] := \text{MSGraph\_TD}(n, c, j, d, p)$ ; // eval min cost for  $j$ .
13        if ( $c[i, j] + d[j] < mincost$ ) then { // smaller cost.
14             $mincost := c[i, j] + d[j]$ ;
15             $p[i] := j$ ;
16        }
17    }
18     $d[i] := mincost$ ; // record min cost for vertex  $i$ .
19    return  $mincost$ ;
20 }
```

Multi-Stage Graphs — Top-down Approach, II

- Before the top-down multistage algorithm is called, the array $d[i]$, which stores the minimum cost from vertex i to sink, should be initialized to $-\infty$.
- The algorithm should be called from main function by `MSGraph_TD($n, c, 1, d, p$)`;
where n is the number of vertices of the graph,
 $c[1:n, 1:n]$ is a matrix such that $c[i, j]$ is the edge cost connecting vertices i and j ,
 1 is the source vertex,
 $d[1:n]$ is an array such that $d[i]$ records the min cost from vertex i to sink,
 $p[1:n]$ is an array such that $p[i]$ records the next vertex from vertex i along the min cost path to the sink.
- In this top-down algorithm each vertex is processed once on lines 11-12.
- Each edge should be visited once, line 10
- The overall time complexity is $O(|V| + |E|)$
- This is more efficient than the recursive version.
- The array (or table) d reduces the number of recursive calls and improves the efficiency significantly.
 - This is one of the key in dynamic programming approach.

Multi-Stage Graphs – Bottom-Up Approach

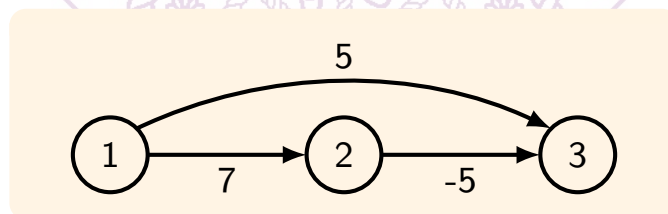
Algorithm 6.2.3. Multistage Graph Bottom-Up Approach

```
// Find minimum cost path  $p$  of  $n$ -vertices multistage graph.
// Input:  $n$ , cost matrix  $c$ 
// Output: path  $p$ , mincost table  $d$ .
1 Algorithm MSGraph_BU( $n, c, d, p$ )
2 {
3      $d[n] := 0$ ; // sink vertex.
4     for  $r := n - 1$  to 1 step  $-1$  do { // for  $n - 1$  stages.
5         for each vertex  $i \in V_r$  do { // All vertices in stage  $r$ .
6              $d[i] := \infty$ ;
7             for each  $\langle i, j \rangle \in E$  do { // All edges from vertex  $i$ .
8                 if  $(c[i, j] + d[j] < d[i])$  { // Smaller cost.
9                      $d[i] := c[i, j] + d[j]$ ; // Record min cost.
10                     $p[r] := j$ ; // Record path.
11                }
12            }
13        }
14    }
15 }
```

- This bottom-up multistage algorithm is non-recursive.
- It should be called by `MSGraph_BU(n, c, d, p)`, where n is the number of vertices of the graph, $c[1:n, 1:n]$ is a matrix such that $c[i, j]$ is the edge cost connecting vertices i and j , $d[1:n]$ is an array such that $d[i]$ records the min cost from vertex i to sink, $p[1:n]$ is an array such that $p[i]$ records the next vertex from vertex i along the min cost path to the sink.
- This algorithm has the same complexities, time and space, as the top-down approach.
- Similar table, array d , is used to improve the efficiency of the algorithm.

Single-Source Shortest Paths: General Weights

- The single-source shortest paths problem is revisited to allow negative weights for some edges.
 - However, no cycle of negative length is allowed.
 - Cycle of negative length can lead to $-\infty$ path length.
- Example



- The greedy algorithm `ShortestPaths` can fail in this case.
 - If vertex 1 is the source
 - It generates path $\langle 1, 3 \rangle$ with weight 5 as the shortest path
 - But path $\langle 1, 2, 3 \rangle$ has the weight of 2.
 - This example shows that we need consider paths through other intermediate vertices.

Single-Source Shortest Paths: General Weights

- With the possibility of negative weights, paths with more segments may have smaller weights, and thus we need to try all paths between a pairs of vertices.
- A shortest path should not include a positive cycle either, since the cycle can be removed to obtain a shorter path.
- A shortest path should not include a cycle with 0 weight, again this cycle can be removed to obtain a shortest path.
 - Thus, a shortest path should not have any cycles.
- Any shortest paths has at most $n - 1$ edges, $n = |V|$.
- Let $d^{(k)}[u]$ be the path weight from source vertex v_0 to vertex u through k edges.
 - Note that $d^{(1)}[u] = W[v_0, u]$ if $\langle v_0, u \rangle \in E$ and $W[v_0, u]$ is the weight of the edge.
- Then we have

$$d^{(k)}[u] = \min \left\{ d^{(k-1)}[u], \min_{i \in V} \{ d^{(k-1)}[i] + W[i, u] \} \right\}. \quad (6.2.3)$$

And $k \leq n - 1$.

- This leads to the dynamic programming algorithm shown next.

Bellman and Ford Algorithm

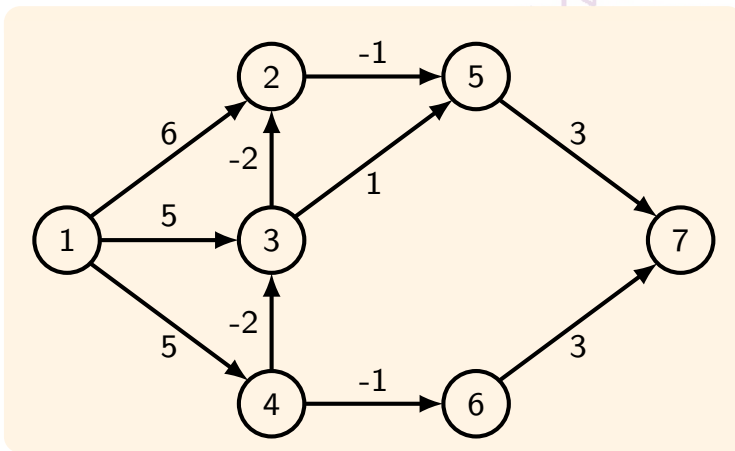
Algorithm 6.2.4. BellmanFord

```
// Generate shortest paths,  $d[1 : n]$ , from  $v$  with edge weight  $W[1 : n, 1 : n]$ .
// Input:  $n: |V|$ , source  $v$ , weight  $W$ 
// Output: distance  $d[1 : n]$ .
1 Algorithm BellmanFord( $n, v, W, d$ )
2 {
3     for  $i := 1$  to  $n$  do
4          $d[i] := W[v, i]$ ;
5     for  $k := 2$  to  $n - 1$  do
6         for each  $u$  such that  $u \neq v$  and  $u$  has incoming edges do
7             for each  $\langle i, u \rangle \in E$  do
8                 if  $(d[u] > d[i] + W[i, u])$  then
9                      $d[u] := d[i] + W[i, u]$ ;
10 }
```

- If W is kept in a matrix form
 - Lines 6-9 takes $\mathcal{O}(n^2)$ time
 - Overall complexity is $\mathcal{O}(n^3)$
- If W is kept in a list form
 - Lines 6-9 takes $\mathcal{O}(e)$ time ($e = |E|$)
 - Overall complexity is $\mathcal{O}(ne)$
 - Efficiency can still be improved further.

Bellman and Ford Algorithm — Example

- Given the graph on the left, and $v = 1$ then we have shortest paths to all other vertices as shown on the right.

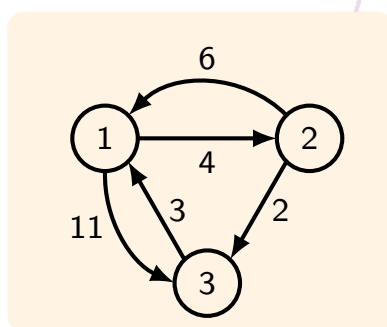


	$d^{(k)}[]$						
k	1	2	3	4	5	6	7
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

- Note that when $d^{(5)}[.] = d^{(6)}[.]$, hence the loop can be terminated.
- The shortest paths can also be printed, if the path information is kept.
- Correctness of the Bellman and Ford algorithm can be found in textbook [Cormen], pp. 652-654.

All-Pairs Shortest Paths

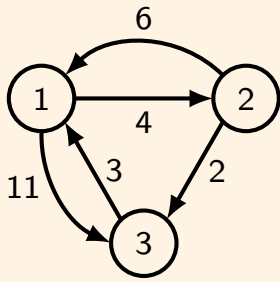
- Given a directed graph $G = (V, E)$ with n vertices and a weight function $w: E \rightarrow \mathbb{R}$, define the weight matrix, $W[1:n, 1:n]$, as
 - $W[i, i] = 0, 1 \leq i \leq n$,
 - $W[i, j] = w(i, j)$, if $\langle i, j \rangle \in E$,
 - $W[i, j] = \infty$, if $\langle i, j \rangle \notin E$.
- The **all-pairs shortest path problem** is to determine a matrix D such that $D[i, j]$ is the weight of the shortest path from vertex i to vertex j .
- One can apply the single source shortest path algorithm n times to find all-pairs shortest paths.
 - Time complexity is $\mathcal{O}(n^4)$ since the single source shortest path algorithm has the complexity of $\mathcal{O}(n^3)$.
- $w[i, j]$ can be negative but no negative cycle exists.



$$\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

Weight matrix, W .

All-Pairs Shortest Paths – Formulation



- As shown on the left, the edge weight from vertex 2 to 1 is 6.
- However, there is a path $\langle 2, 3, 1 \rangle$ with small path weight, 5.
- Thus, to find the minimum path we need consider paths through all intermediate vertices.

- Let $D^{(0)} = W$, where W is the weight matrix defined above.
- Let $D^{(k)}[i, j]$ be the minimum cost path with intermediate vertices no more than vertex k , then

$$D^{(k)}[i, j] = \min\{D^{(k-1)}[i, j], D^{(k-1)}[i, k] + D^{(k-1)}[k, j]\}. \quad (6.2.4)$$

- Since there are only $n = |V|$ vertices in the graph, $D^{(n)}[i, j]$ is the minimum weight between any pair of vertices, i and j , $1 \leq i, j \leq n$.
- This formulation lends itself to a dynamic programming approach to solve the all-pair shortest path problem.

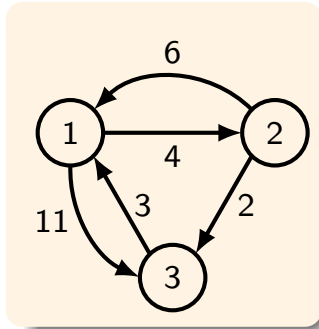
All-Pairs Shortest Paths – Algorithm

Algorithm 6.2.5. All-Pairs Shortest Paths

```
// Find all-pairs shortest paths and store them in matrix  $D[1 : n, 1 : n]$ .
// Input:  $n: |V|$ , weight  $W$ 
// Output: distance  $D$ .
1 Algorithm AllPairs( $n, W, D$ )
2 {
3     for  $i := 1$  to  $n$  do // Create  $D^{(0)}$ .
4         for  $j := 1$  to  $n$  do
5              $D[i, j] := W[i, j]$ ;
6     for  $k := 1$  to  $n$  do // Loop through all  $D^{(k)}$ .
7         for  $i := 1$  to  $n$  do
8             for  $j := 1$  to  $n$  do
9                 if  $(D[i, j] > D[i, k] + D[k, j])$  then
10                      $D[i, j] := D[i, k] + D[k, j]$ ;
11 }
```

- Using D to store all $D^{(k)}$ for better space efficiency.
- Space complexity remains as $\Theta(n^2)$.
- The time complexity is $\mathcal{O}(n^3)$.
 - Triple loop on lines 6-10.

All-Pairs Shortest Paths – Example



$$\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

$D^{(0)}$.

$$\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

$D^{(1)}$.

$$\begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

$D^{(2)}$.

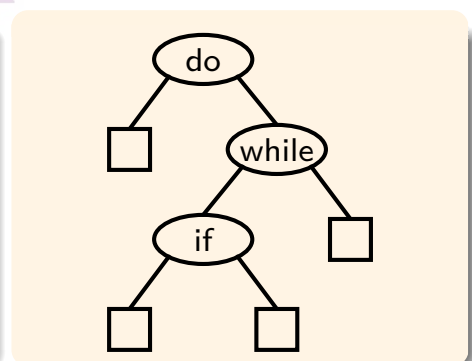
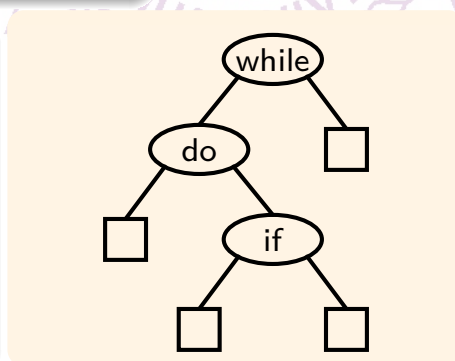
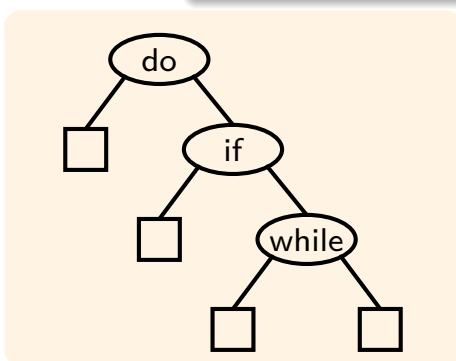
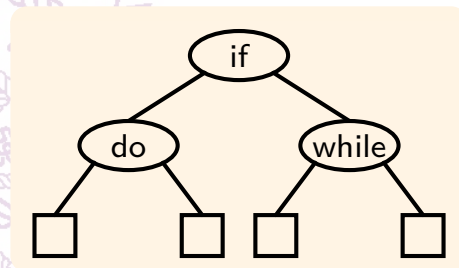
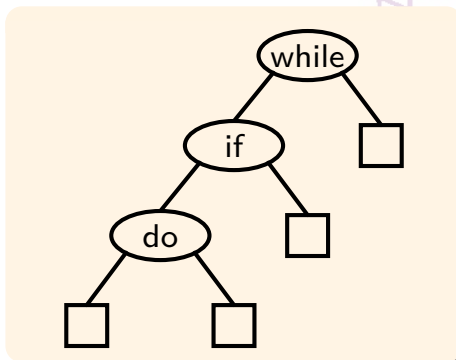
$$\begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

$D^{(3)}$.

- The minimum cost between all vertices, i and j , is given by $D^{(3)}[i, j]$, $1 \leq i, j \leq 3$.
- To print out the shortest paths for each pair of vertices, the intermediate vertex k on line 10 should be memorized to another matrix $P[1 : n, 1 : n]$.
- Using matrix P the shortest paths can be printed out.
- Correctness of the algorithm can be found in textbooks, [Horowitz], pp. 284-287, and [Cormen], pp. 693-695.

Optimal Binary Search Tree

- Possible binary search trees for three identifiers
 - Successful searches terminate at an internal node, shown in ellipse
 - Unsuccessful searches terminate at an external node, shown in square
 - n internal nodes and $n + 1$ external nodes



Optimal Binary Search Tree — cost

- For each identifier, a_i , at $level(a_i)$ in the tree, each successful search needs $level(a_i)$ comparisons.
- Note that for n identifiers there are $n + 1$ possible unsuccessful searches.
 - Name these unsuccessful events, E_j , $0 \leq j \leq n$.
 - For each unsuccessful search E_i at $level(E_i)$ of the binary tree, there are $level(E_i) - 1$ comparisons.
- Let p_i be the probability of searching for identifier a_i and q_i be the probability of searching for E_i .

$$\sum_{i=1}^n p_i + \sum_{j=0}^n q_j = 1. \quad (6.2.5)$$

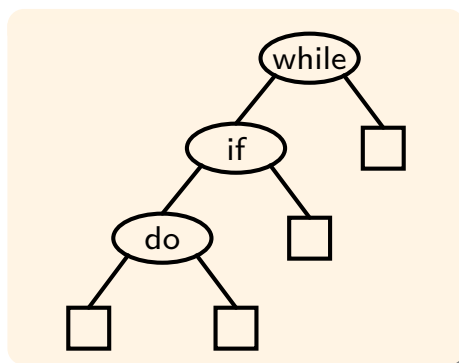
- The cost of the binary search tree is the expected value of the number of comparisons

$$cost(t) = \sum_{i=1}^n p_i \times level(a_i) + \sum_{j=0}^n q_j \times (level(E_j) - 1). \quad (6.2.6)$$

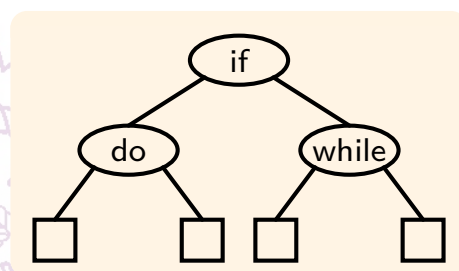
- The **optimal binary search tree** is the binary tree such that the cost of the tree is minimum.

Optimal Binary Search Tree — Example

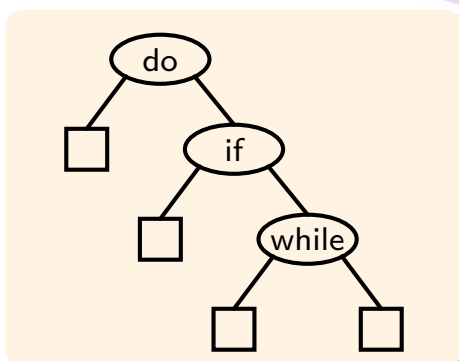
- Suppose $p_i = q_i = 1/7$ then



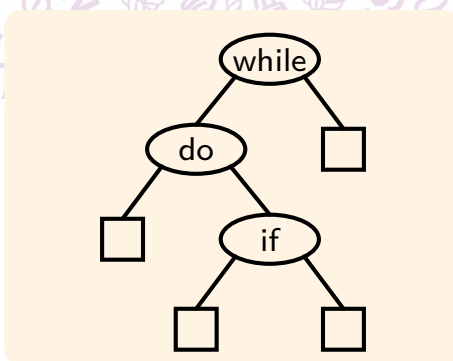
$cost = 15/7$



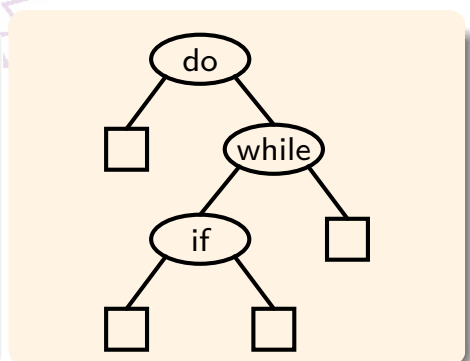
$cost = 13/7$, **optimal**



$cost = 15/7$



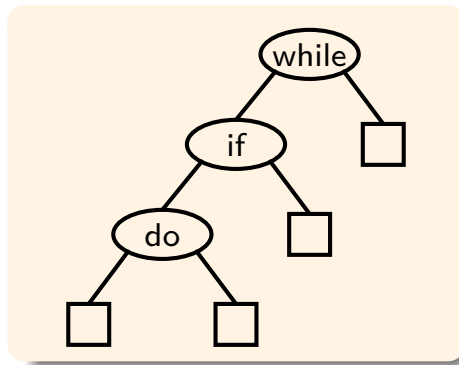
$cost = 15/7$



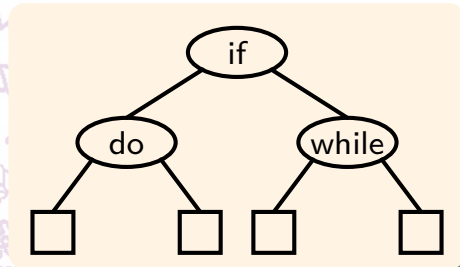
$cost = 15/7$

Optimal Binary Search Tree — Example II

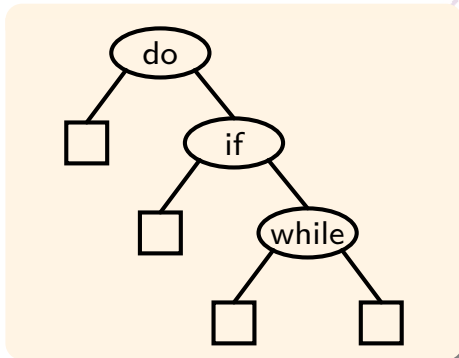
- Suppose $p_1 = 0.5(\text{do})$, $p_2 = 0.1(\text{if})$, $p_3 = 0.05(\text{while})$, $q_0 = 0.15$, $q_1 = 0.1$, $q_2 = 0.05$, $q_3 = 0.05$, then



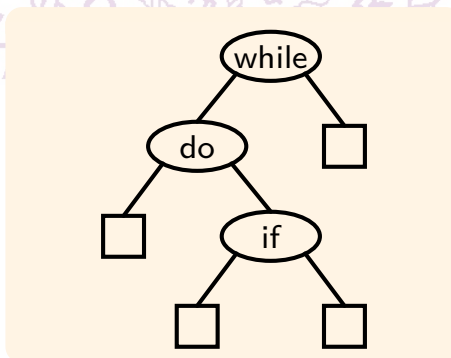
$cost = 2.65$



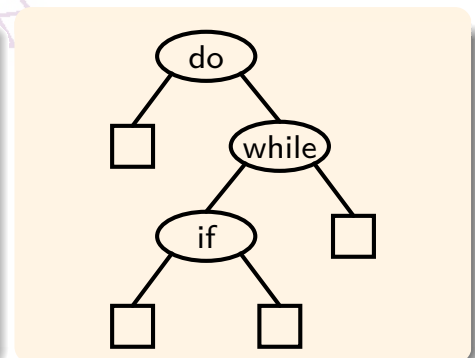
$cost = 1.9$



$cost = 1.5$, **optimal**



$cost = 2.15$



$cost = 1.6$

Optimal Binary Search Tree — Properties

- Given internal nodes $\{a_1, a_2, \dots, a_n\}$ with probabilities $\{p_1, p_2, \dots, p_n\}$ and the external nodes with probabilities $\{q_0, q_1, \dots, q_n\}$.
- If a_k is the root of a binary search tree, then its left subtree consists of internal nodes $\{a_1, a_2, \dots, a_{k-1}\}$ and external nodes $\{q_0, q_1, \dots, q_{k-1}\}$.
- The right subtree consists of internal nodes $\{a_{k+1}, \dots, a_n\}$ and external nodes $\{q_k, \dots, q_n\}$.
- Let the cost of the left subtree be c_l and the cost of the right subtree be c_r , then the cost of the tree with a_k as the root is

$$c(a_k) = c_l + c_r + w(1, n) \quad (6.2.7)$$

where

$$w(1, n) = \sum_{i=1}^n p_i + \sum_{i=0}^n q_i \quad (6.2.8)$$

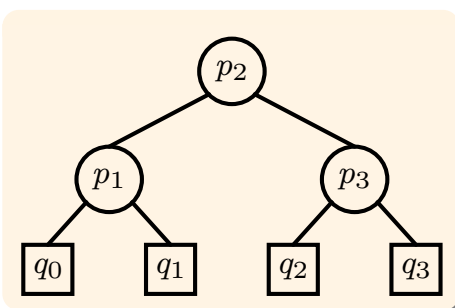
- Example

$$c_l = p_1 + q_0 + q_1$$

$$c_r = p_3 + q_2 + q_3$$

$$c(p_2) = p_2 + 2(p_1 + q_0 + q_1) + 2(p_3 + q_2 + q_3)$$

$$= c_l + c_r + p_1 + p_2 + p_3 + q_0 + q_1 + q_2 + q_3$$



Algorithm 6.2.6. Recursive OBST

```
// Find the root  $r$  of the optimal binary search tree for nodes  $a_i$  to  $a_j$ .
// Input: range:  $i, j$ ; probabilities:  $p[1 : n]$ ,  $q[0 : n]$ 
// Output:  $cost$ , root  $r$ .
1 Algorithm OBSTr( $i, j, p, q, r$ )
2 {
3     if ( $i = j$ ) then { // single vertex
4          $r := i$ ;
5         return  $q[i - 1] + q[i] + p[i]$ ;
6     }
7      $cost := \infty$ ;
8      $w := q[i - 1]$ ;
9     for  $k := i$  to  $j$  do  $w := w + p[k] + q[k]$ ; // calculate  $w(i, j)$ 
10    for  $k := i + 1$  to  $j - 1$  do { // try every vertex and find the minimum cost one
11         $cL := OBSTr(i, k - 1, p, q, rL)$ ; // find minimum cost left subtree
12         $cR := OBSTr(k + 1, j, p, q, rR)$ ; // find minimum cost right subtree
13        if ( $cL + cR + w < cost$ ) then {
14             $cost := cL + cR + w$ ;
15             $r := k$ ;
16        }
17    }
18    return  $cost$ ;
19 }
```

Optimal Binary Search Tree — Recursive Algorithm, II

- This algorithm finds the minimum-cost left subtree and right subtree and combines those two to form the minimum-cost binary search tree.
- The recursive algorithm is invoked by $OBSTr(1, n, p, q, r)$, where p is the array for the internal node probabilities, q is the array for the external nodes probabilities.
- It then finds the root r of the minimum-cost binary search tree.
 - The roots of the left and right subtrees should be found by calling $OBSTr(1, r - 1, p, q, rL)$ and $OBSTr(r + 1, n, p, q, rR)$ recursively.
- As most of the recursive function, the time complexity can be improved.

Optimal Binary Search Tree — Improved Algorithm

Algorithm 6.2.7. Optimal Binary Search Tree

```
// Find the matrix  $r$ . Each  $r[i, j]$  is the optimal root for  $a_i$  to  $a_j$ .
// Input: int  $n$ , probabilities:  $p[1 : n]$ ,  $q[0 : n]$ 
// Output:  $r$ : optimal root matrix.
1 Algorithm OBST( $n, p, q, r$ )
2 {
3     for  $i := 0$  to  $n - 1$  do {
4          $w[i, i] := q[i]$ ;
5          $r[i, i] := 0$ ;
6          $c[i, i] := 0$ ;
7          $w[i, i + 1] := q[i] + p[i + 1] + q[i + 1]$ ; // one node trees
8          $r[i, i + 1] := i + 1$ ;
9          $c[i, i + 1] := q[i] + p[i + 1] + q[i + 1]$ ;
10    }
11     $w[n, n] := q[n]$ ;
12     $r[n, n] := 0$ ;
13     $c[n, n] := 0$ ;
14    for  $m := 2$  to  $n$  do { // Find optimal trees with  $m$  nodes
15        for  $i := 0$  to  $n - m$  do {
16             $j := i + m$ ;
17             $w[i, j] := w[i, j - 1] + p[j] + q[j]$ ;
18             $k := \text{KnuthFind}(c, r, i, j)$ ; // root with min cost of  $m$ -node tree
19             $r[i, j] := k$ ; // root for tree  $a_i$  to  $a_j$ 
20             $c[i, j] := w[i, j] + c[i, k - 1] + c[k, j]$ ; // record min cost
21        }
22    } // When done,  $r[0, n]$  is the root,  $c[0, n]$  is the min cost
23 }
```

Optimal Binary Search Tree — KnuthFind

Algorithm 6.2.8. Knuth Find

```
// Find the min-cost root for tree  $a_i$  to  $a_j$ .
// Input:  $c[0 : n]$ : min cost,  $r[0 : n]$ : min cost root matrix
// Output: min cost root.
1 Algorithm KnuthFind( $c, r, i, j$ )
2 {
3      $min := \infty$ ;
4     for  $m := r[i, j - 1]$  to  $r[i + 1, j]$  do {
5         if  $((c[i, m - 1] + c[m, j]) < min)$  then {
6              $min := c[i, m - 1] + c[m, j]$ ;
7              $l := m$ ;
8         }
9     }
10    return  $l$ ;
11 }
```

• In the OBST Algorithm

- $r[i, j]$ is the min-cost root for tree a_i to a_j
 - $p[i, j]$ is the probabilities of the internal nodes a_i to a_j
 - $q[i - 1, j]$ is the probabilities of the external nodes
- $c[i, j]$ is the cost of the optimal search tree
- $w[i, j]$ is the sum of all the probabilities for internal and external nodes from a_i to a_j .

- After completion of the algorithm
 - The root of the optimal tree is given by $r[0, n]$
 - Let $k = r[0, n]$, then
 - The root of the left subtree is $r[0, k - 1]$
 - And the root of the right subtree is $r[k + 1, n]$
 - Repeating this process the entire tree can be built.
- Using **KnuthFind** function in **OBST** algorithm, the time complexity is $\mathcal{O}(n^2)$
 - Exercise
- And the complexity of using resulting $r[0, n]$ to build the optimal binary search tree is $\mathcal{O}(n)$

OBST – Execution Example

- Example: $(a_1, a_2, a_3, a_4) = (\text{do}, \text{if}, \text{int}, \text{while})$,
 $p = (3/16, 3/16, 1/16, 1/16)$,
 $q = (2/16, 3/16, 1/16, 1/16, 1/16)$.
 - **OBST** iterations:

	$w_{00} = 2/16$	$w_{11} = 3/16$	$w_{22} = 1/16$	$w_{33} = 1/16$	$w_{44} = 1/16$
iter 0	$c_{00} = 0$	$c_{11} = 0$	$c_{22} = 0$	$c_{33} = 0$	$c_{44} = 0$
	$r_{00} = 0$	$r_{11} = 0$	$r_{22} = 0$	$r_{33} = 0$	$r_{44} = 0$
	$w_{01} = 8/16$	$w_{12} = 7/16$	$w_{23} = 3/16$	$w_{34} = 3/16$	
iter 1	$c_{01} = 8/16$	$c_{12} = 7/16$	$c_{23} = 3/16$	$c_{34} = 3/16$	
	$r_{01} = 1$	$r_{12} = 2$	$r_{23} = 3$	$r_{34} = 4$	
	$w_{02} = 12/16$	$w_{13} = 9/16$	$w_{24} = 5/16$		
iter 2	$c_{02} = 19/16$	$c_{13} = 12/16$	$c_{24} = 8/16$		
	$r_{02} = 1$	$r_{13} = 2$	$r_{24} = 3$		
	$w_{03} = 14/16$	$w_{14} = 11/16$			
iter 3	$c_{03} = 25/16$	$c_{14} = 19/16$			
	$r_{03} = 2$	$r_{14} = 2$			
	$w_{04} = 16/16$				
iter 4	$c_{04} = 32/16$				
	$r_{04} = 2$				

- Multistage graph problem
- Single-source shortest path
- All-pairs shortest paths
- Optimal binary search tree

