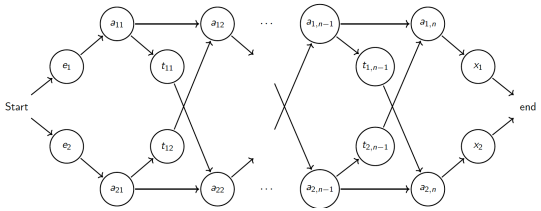


Dynamic programming 1
Lecture 23

COSC 242 – Algorithms and Data Structures

Today’s outline

1. Dynamic programming
2. Assembly line scheduling
3. Fastest path and a recursive solution
4. Example
5. Dynamic programming iterative solution

Today’s outline

1. Dynamic programming
2. Assembly line scheduling
3. Fastest path and a recursive solution
4. Example
5. Dynamic programming iterative solution

Dynamic programming

The iterative and memoised algorithms for solving the 0-1 knapsack problem are examples of dynamic programming solutions to problems.

Dynamic programming:

- is used for **optimisation problems**, where we want to find the “best way” of doing something;
- is a recursive approach that involves breaking a global problem down into more local subproblems;

List continues…

Dynamic programming

Dynamic programming:

- requires **optimal substructure**, i.e. a simple way to combine optimal solutions of smaller problems to get optimal solutions of larger problems;
- avoids the inefficiency that subproblem overlap causes for straightforward recursion (the same subproblems occurring often and thus being solved many times);
- does this by **memoisation** (i.e. storing the solutions of subproblems in a table and then looking them up).

Dynamic programming vs Greedy

Dynamic programming and greedy algorithms are applied to **optimisation problems**. They are related techniques with different approaches.

Greedy algorithms solve a problem by running forward, and greedily choosing the locally best item as the next one. Only one path or solution is ever considered. It’s a top-down approach.

Dynamic programming algorithms solve a problem by considering multiple paths at each decision point. Dynamic programming solutions often (but not always) work backwards and define the optimal solution to a problem as a choice over optimal solutions to sub-problems.

Example problems

There are many problems that can be solved efficiently using dynamic programming including:

- 0-1 knapsack problem
- assembly-line scheduling (we'll look at this next)
- matrix chain multiplication (in what order to multiply matrices)
- longest common subsequence of two strings (useful for finding commonalities in genomes)
- constructing optimal binary search trees given a known distribution of search keys.

Today’s outline

1. Dynamic programming
2. Assembly line scheduling
3. Fastest path and a recursive solution
4. Example
5. Dynamic programming iterative solution

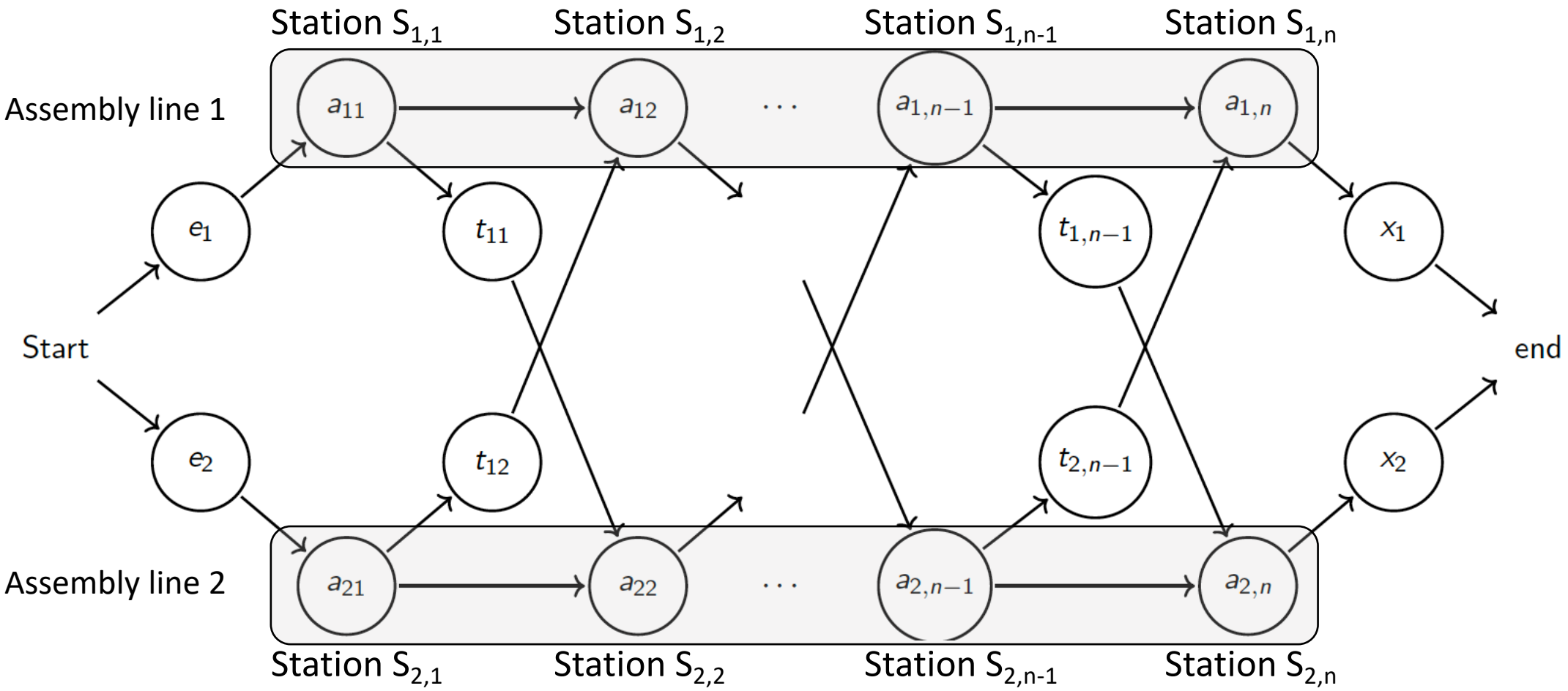
Assembly line scheduling

Assembly line scheduling solves a manufacturing problem in industry.

The Wolksvagen automotive company produces cars in a factory that has two assembly lines, denoted as $i = 1$ or 2.

A vehicle chassis enters each assembly line, and has parts added to it at n different stations. The finished vehicle exits at the end of the line.

Assembly line



e_i = Entry time for line *i*

$a_{i,j}$ = Assembly time for Station *j*, on line *i*

$t_{i,j}$ = Transfer time away from line i, after station $S_{i,j}$

x_i = Exit time for vehicle to leave line i

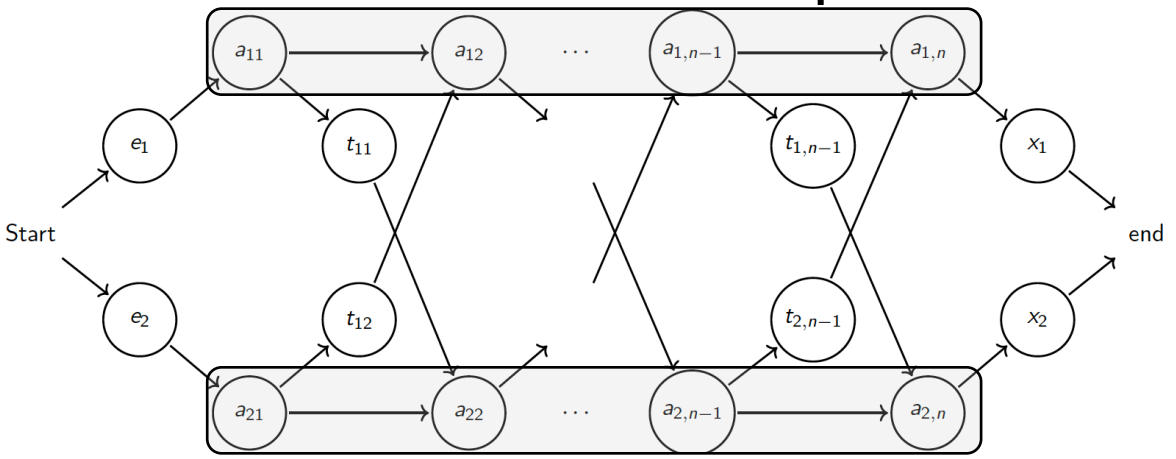
Assembly line scheduling

Each assembly line has n stations, numbered $j = 1, 2, \ldots, n$.

We denote the j^{th} station on line i as: $S_{i,j}$

The assembly time taken at station $S_{i,j}$ is $a_{i,j}$

Each line also has an entry time, e_i, the time taken for the chassis to enter line i, and an exit time, x_i, the time taken for the completed vehicle to leave line i.

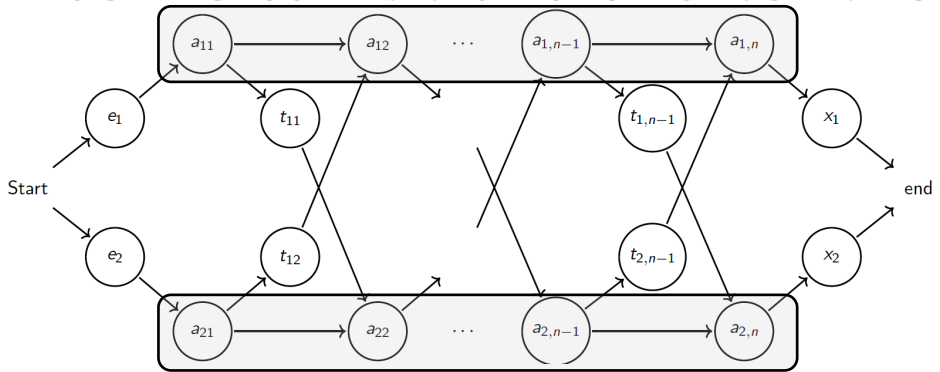


Assembly line scheduling

Ordinarily, once a chassis enters a line, it will stay on that line until completion. However, sometimes a rush order comes in, where we seek to complete a vehicle in the fastest possible time.

For these orders, the vehicle must still pass through *n* stations, but it can transfer from one line to the other, after leaving any station.

The time to transfer a chassis between line *i* after leaving Station $S_{i,j}$ is $t_{i,j}$, where $i = 1, 2$ and $j = 1, 2, \ldots, n-1$ (since we can't transfer after the last station).



Brute force

Like the knapsack problem, the brute force approach yields exponential complexity time.

At each station, we can make two choices: stay on the line, or transfer. That is, our set of possible decisions doubles at each station. Since we have *n* stations, there are 2^n possible ways to choose stations.

That is, the complexity of the assembly line scheduling problem with a brute force method is $\Omega(2^n)$.

Today’s outline

1. Dynamic programming
2. Assembly line scheduling
3. Fastest path and a recursive solution
4. Example
5. Dynamic programming iterative solution

Fastest way through the factory

Lets think about the fastest way for a chassis to move from the start, to station $S_{1,j}$.

If $j = 1$, then it can only have come from one place, the vehicle entry point, and so it is trivial to determine how long it takes to get through station $S_{1,j}$.

Fastest way through the factory

But for $j = 2, 3, \ldots, n$, there are two choices:

1. The chassis came from station $S_{1,j-1}$ and then to $S_{1,j}$. Here, the time of moving between j-1 to j is zero, as they are on the same line.
2. The chassis came from $S_{2,j-1}$, then transferred to $S_{1,j}$. Here, the transfer time between lines was $t_{2,j-1}$.

Fastest way through the factory

We will first suppose that the fastest way through $S_{1,j}$ is from $S_{1,j-1}$.

A crucial observation is that the chassis must've taken the fastest time to station $S_{1,j-1}$. Why?

If there exists a faster way to $S_{1,j-1}$, our chassis would've taken it. We could then substitute this faster sub-route into our route to $S_{1,j}$.

But this would then lead to a faster time for $S_{1,j}$, which implies that $S_{1,j}$ was not the fastest way through the plant: a contradiction.

The symmetric argument applies to the fastest time through $S_{2,j}$

Optimal substructure

Thus, our optimal solution (fastest time through Station $S_{i,j}$) for assembly line scheduling contains within it, other optimal solutions to subproblems.

This property is known as **optimal substructure**, which we also saw in our knapsack problems.

As noted in L22, this problem property is an essential requirement for a dynamic programming solution.

Therefore, we can build an optimal solution to the fastest time problem, by building optimal solutions to subproblems.

Developing a recursive solution

Let $f_i[j]$ be the fastest time to get a chassis from the start through to station j on line i: $S_{i,j}$.

Let f^* be the fastest time for the chassis to get all the way through the factory, arriving at the exit as a finished vehicle.

For the station to reach the exit, it must get all the way to station n on either line 1 or 2, and then exit the factory. Since one of these must be the fastest ways, we can define our fastest time solution as:

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

Where *x1* and *x2* are exit times from lines 1 and 2, respectively

Developing a recursive solution

Let us next define the fastest times through station 1, on lines 1 and 2. This sub-problem is easy to solve, as it is simply the entry time e_i plus the first station assembly time $a_{i,1}$:

$$f_1[1] = e_1 + a_{1,1},$$
$$f_2[1] = e_2 + a_{2,1}$$

Developing a recursive solution

Lets now define the fastest time for $f_i[j]$ for stations $j = 2,\ldots, n$

We already established that the fastest time through $S_{1,j}$ is either:

- from $S_{1,j-1}$ then into $S_{1,j}$ OR
- From $S_{2,j-1}$, transfer from line 2 to line 1, then through $S_{1,j}$.

Therefore:

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

Developing a recursive solution

We can now define our final recursive equations:

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{If } j = 1, \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{If } j \geq 2. \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{If } j = 1, \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{If } j \geq 2. \end{cases}$$

Tracing our way through the factory

The last thing we need to do is keep track of the stations that we passed through, which are used in constructing our fastest way solution.

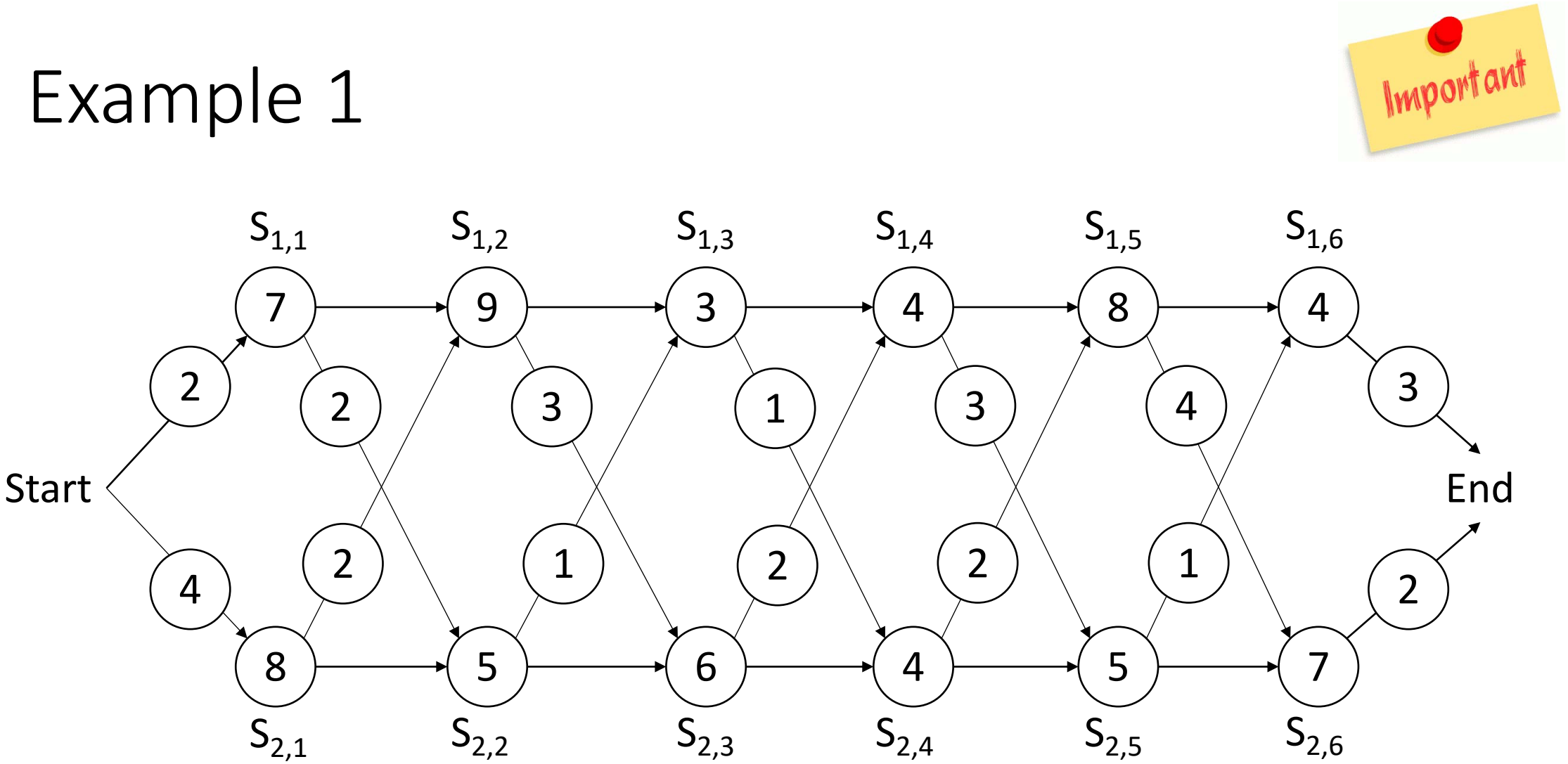
Let $L_i[j]$ be the line number, either 1 or 2, whose station $j - 1$ is used in a fastest way through station $S_{i,j}$.

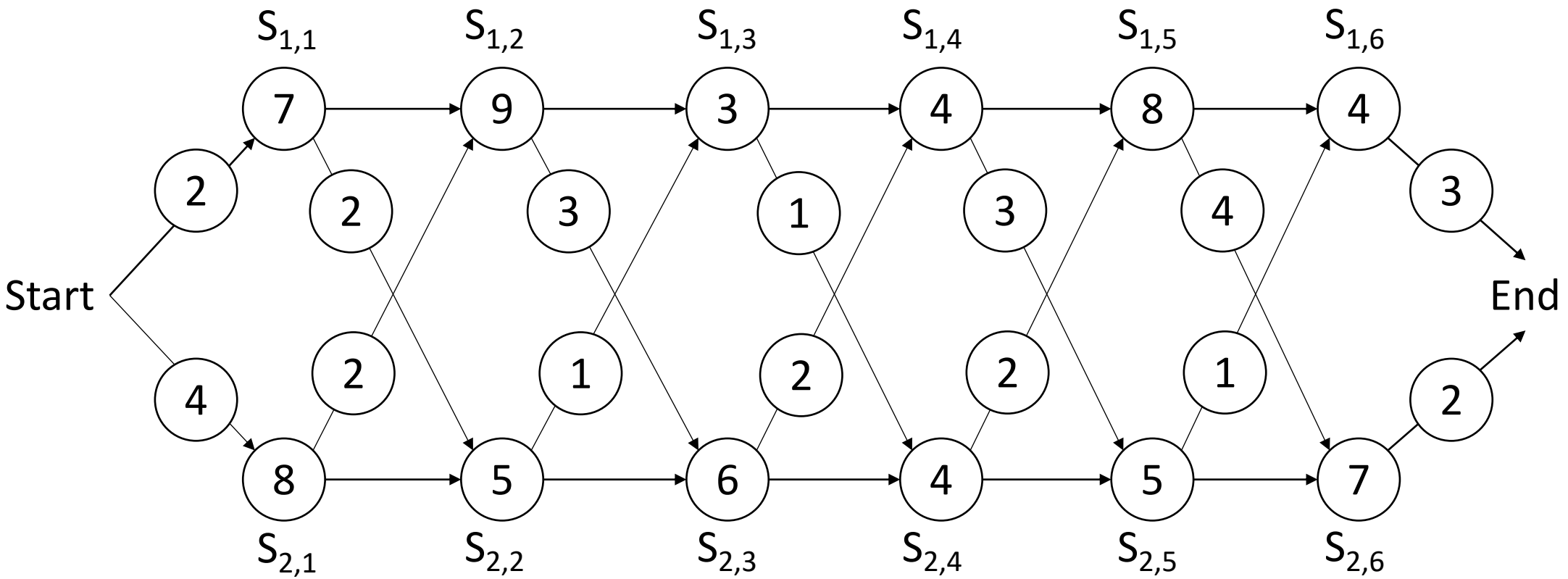
Let L^* be the line whose station n is used in a fastest way through the entire factory.

Today’s outline

1. Dynamic programming
2. Assembly line scheduling
3. Fastest path and a recursive solution
4. Example
5. Dynamic programming iterative solution

Example 1


Important
$S_{1,1}$
$S_{1,2}$
$S_{1,3}$
$S_{1,4}$
$S_{1,5}$
$S_{1,6}$
Start
End
7
9
3
4
8
4
2
2
3
1
3
4
3
4
2
1
2
2
1
2
8
5
6
4
5
7
$S_{2,1}$
$S_{2,2}$
$S_{2,3}$
$S_{2,4}$
$S_{2,5}$
$S_{2,6}$

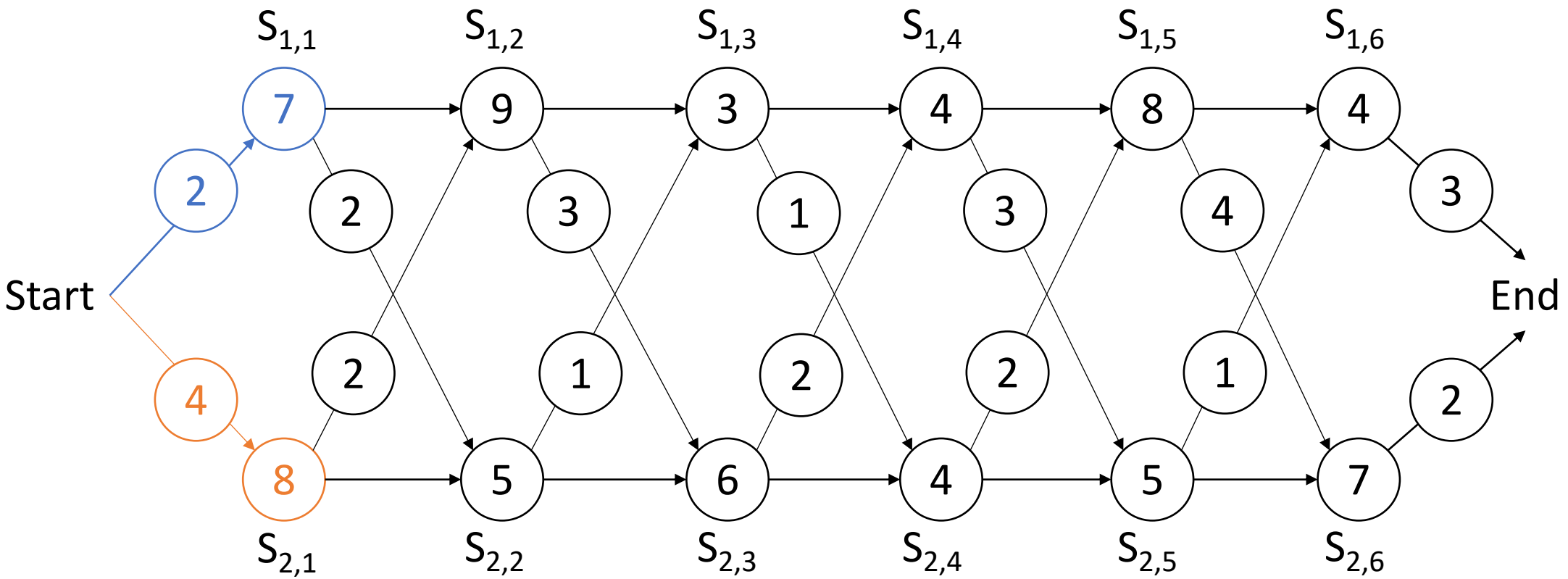


$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{If } j = 1, \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{If } j \geq 2. \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{If } j = 1, \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{If } j \geq 2. \end{cases}$$

j	1	2	3	4	5	6
$f_1[j]$						
$f_2[j]$						

j	2	3	4	5	6
$L_1[j]$					
$L_2[j]$					

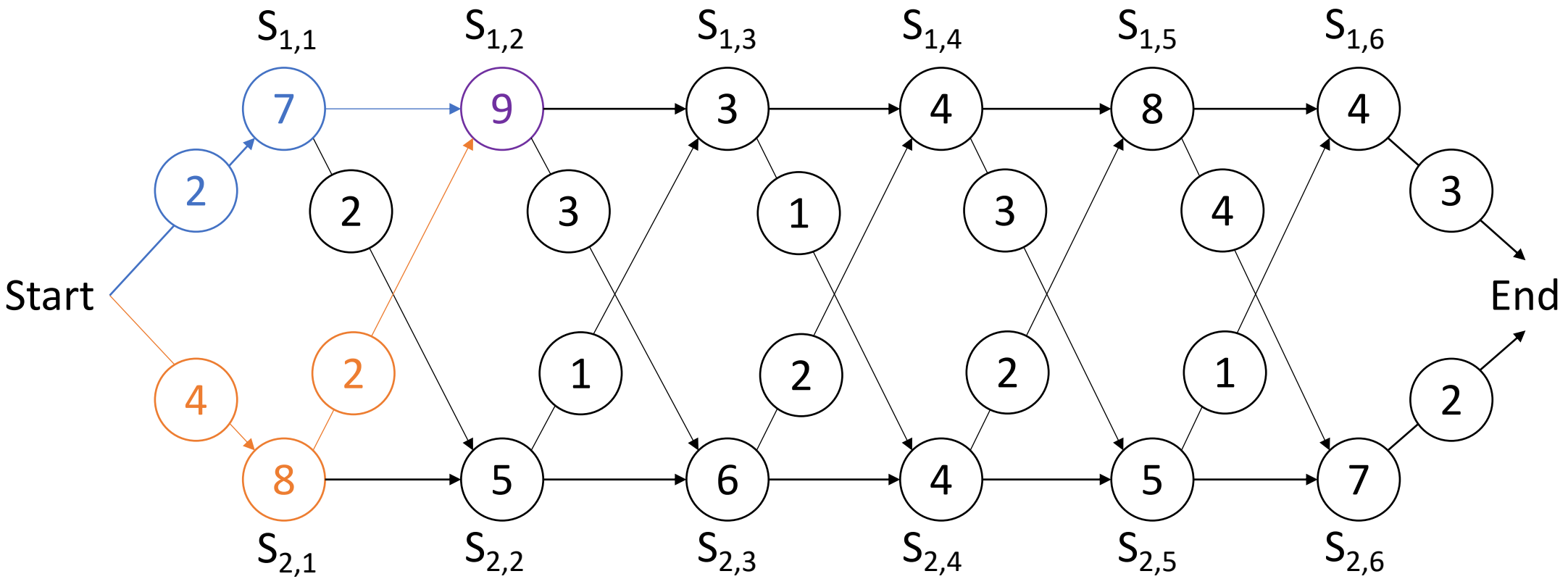


$$f_1[1] = 2 + 7$$

$$f_2[1] = 4 + 8$$

j	1	2	3	4	5	6
$f_1[j]$	9					
$f_2[j]$	12					

j	2	3	4	5	6
$L_1[j]$					
$L_2[j]$					

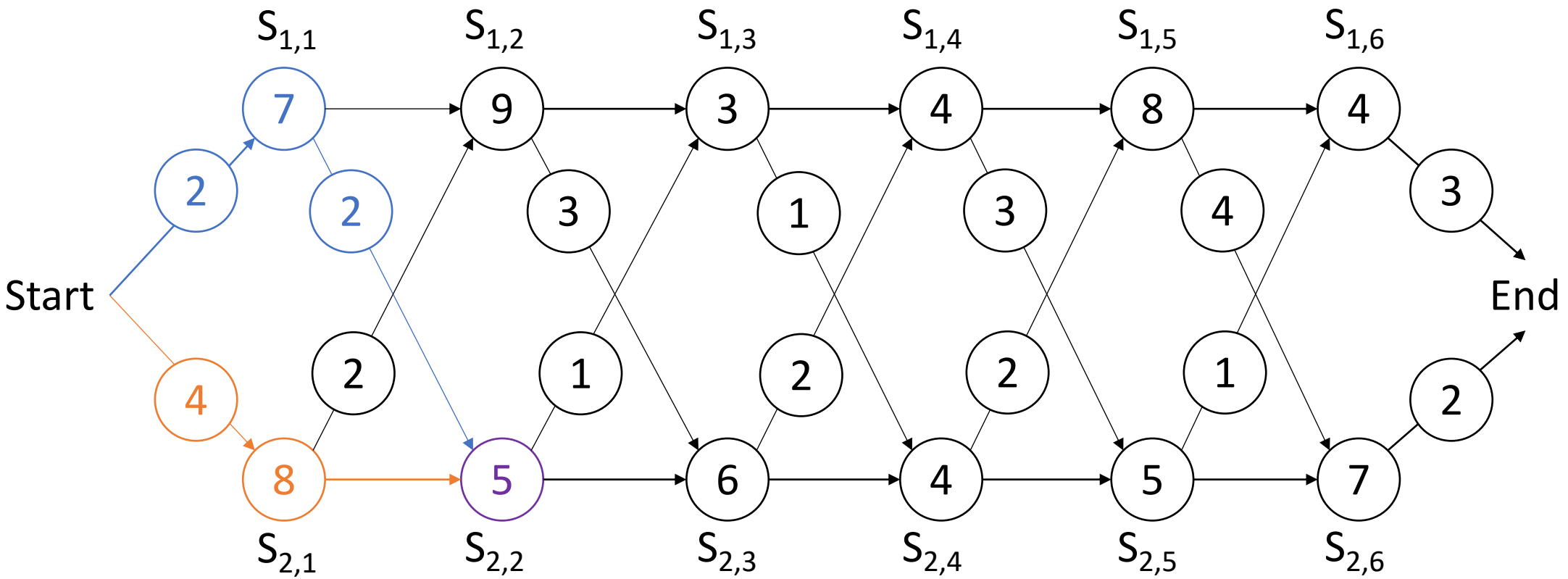


$$f_1[2] = \min(9 + 9, 12 + 2 + 9) = 18$$

$$f_2[2] =$$

j	1	2	3	4	5	6
$f_1[j]$	9	18				
$f_2[j]$	12	16				

j	2	3	4	5	6
$L_1[j]$					
$L_2[j]$					

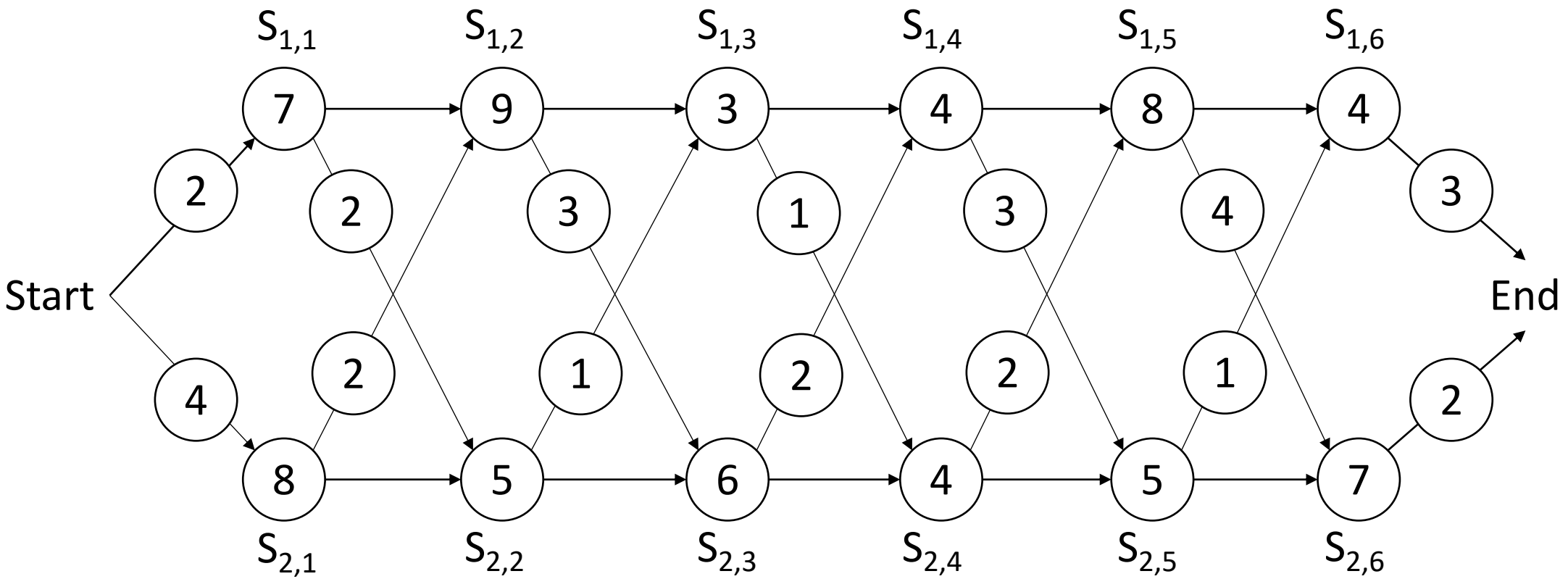


$$f_1[2] = \min(\mathbf{9} + \mathbf{9}, 12 + 2 + 9) = 18$$

$$f_2[2] = \min(12 + 5, \mathbf{9} + \mathbf{2} + \mathbf{5}) = 16$$

j	1	2	3	4	5	6
$f_1[j]$	9	18				
$f_2[j]$	12	16				

j	2	3	4	5	6
$L_1[j]$					
$L_2[j]$					



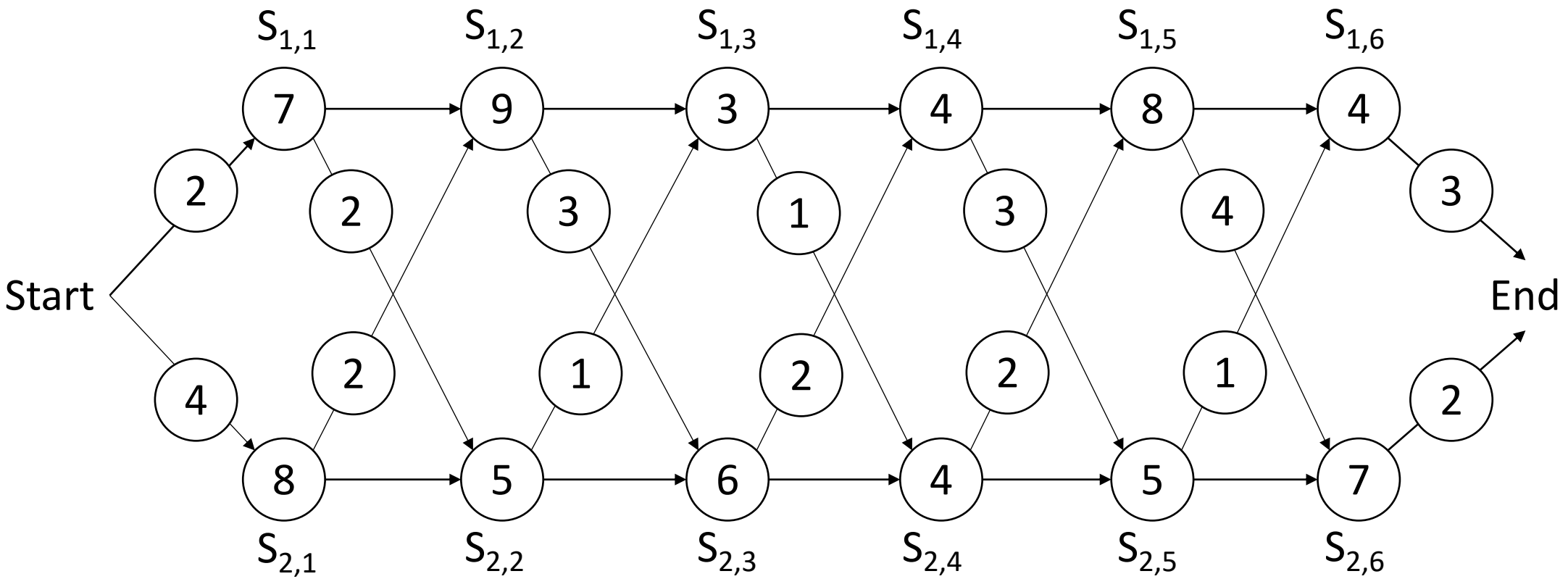
$$f_1[2] = \min(\mathbf{9} + 9, 12 + 2 + 9) = 18$$

$$f_2[2] = \min(12 + 5, \mathbf{9} + 2 + 5) = 16$$

For $L_i[2]$, we now record the station $S_{i,j}$ we came in *through* for our fastest (min) time

j	1	2	3	4	5	6
$f_1[j]$	9	18				
$f_2[j]$	12	16				

j	2	3	4	5	6
$L_1[j]$	1				
$L_2[j]$	1				

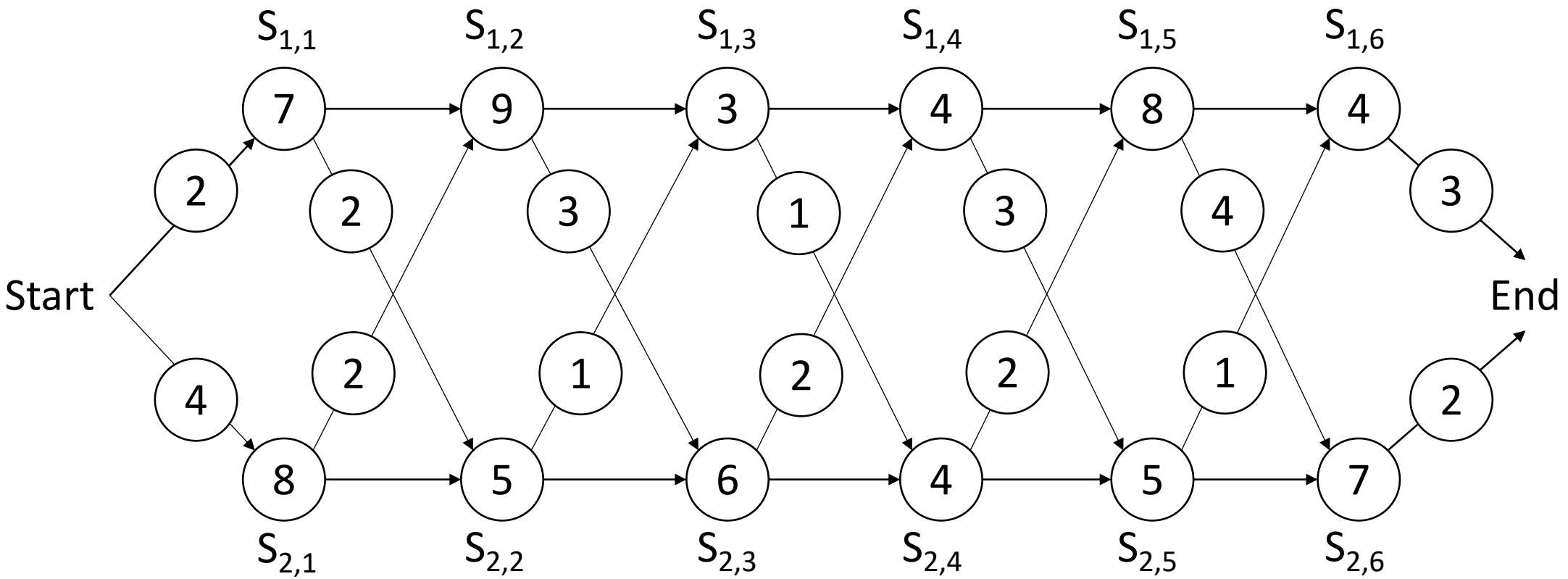


$$f_1[3] = \min(18 + 3, \mathbf{16} + \mathbf{1} + \mathbf{3}) = 20$$

$$f_2[3] = \min(\mathbf{16} + \mathbf{6}, 18 + 3 + 6) = 22$$

j	1	2	3	4	5	6
$f_1[j]$	9	18	20			
$f_2[j]$	12	16	22			

j	2	3	4	5	6
$L_1[j]$	1	2			
$L_2[j]$	1	2			

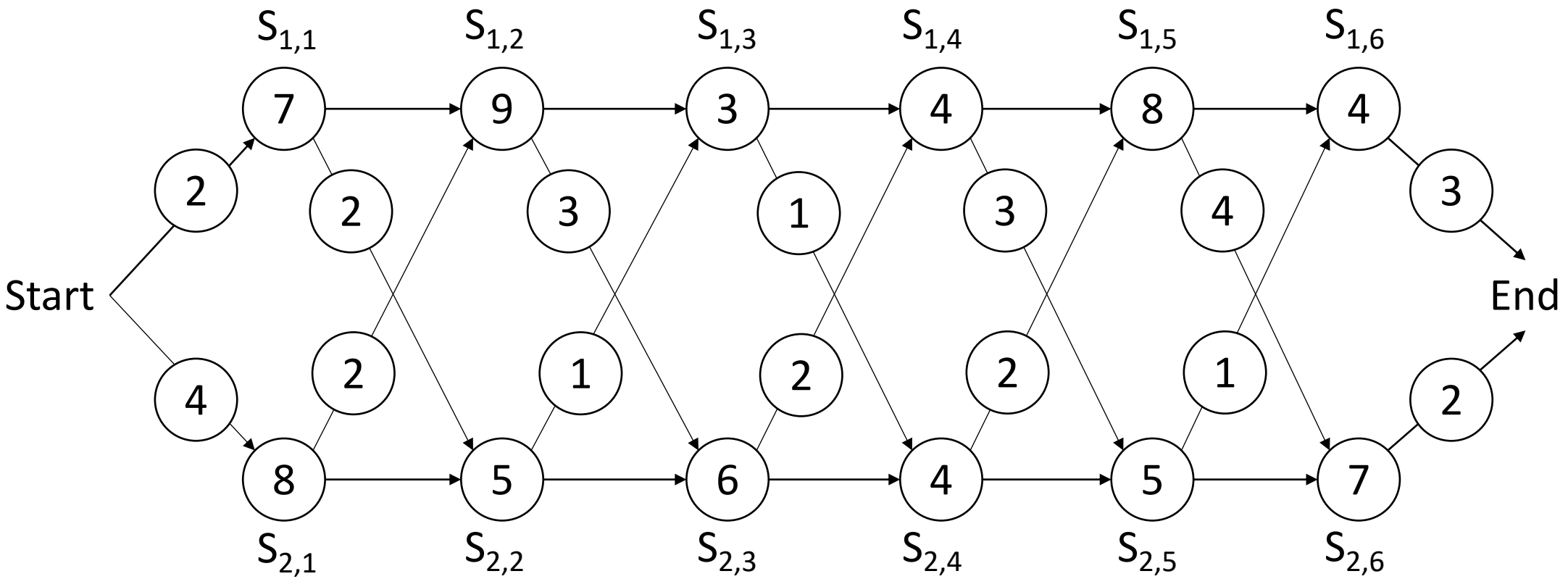


$$f_1[4] = \min(\mathbf{20} + \mathbf{4}, 22 + 2 + 4) = 24$$

$$f_2[4] = \min(22 + 4, \mathbf{20} + \mathbf{1} + \mathbf{4}) = 25$$

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24		
$f_2[j]$	12	16	22	25		

j	2	3	4	5	6
$L_1[j]$	1	2	1		
$L_2[j]$	1	2	1		

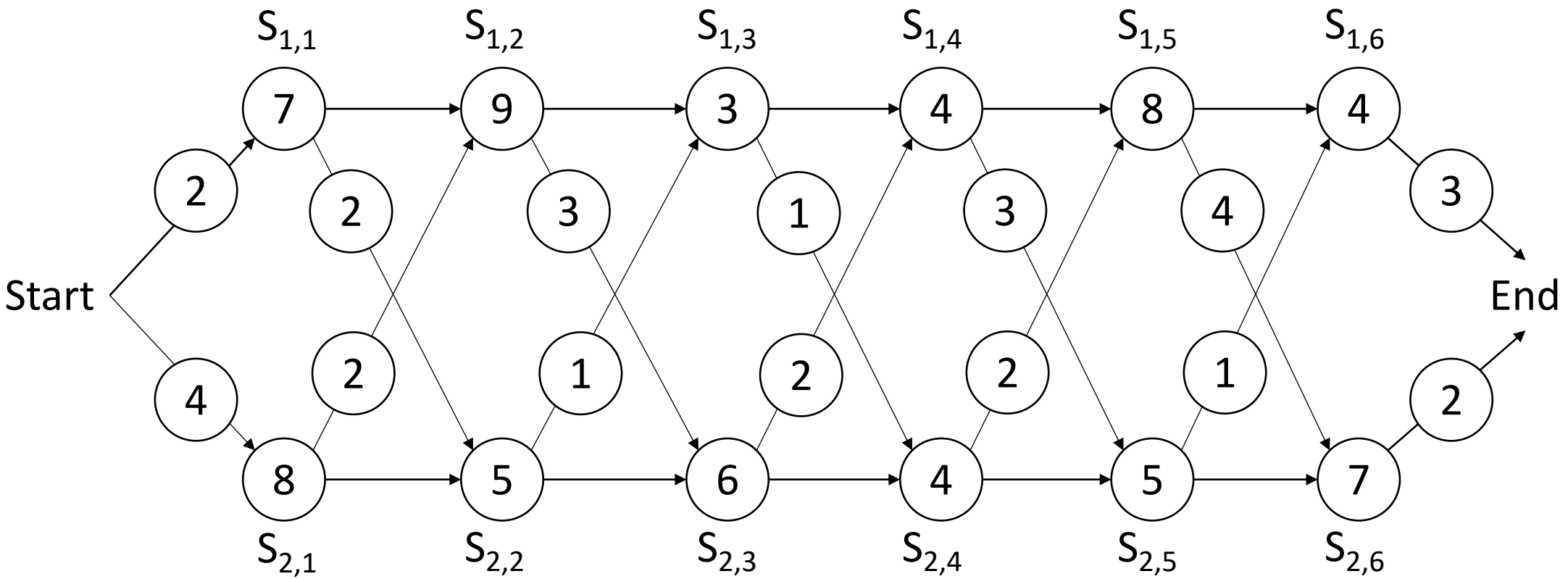


$$f_1[5] = \min(\mathbf{24} + \mathbf{8}, 22 + 2 + 4) = 32$$

$$f_2[5] = \min(\mathbf{25} + \mathbf{5}, 24 + 3 + 5) = 30$$

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	
$f_2[j]$	12	16	22	25	30	

j	2	3	4	5	6
$L_1[j]$	1	2	1	1	
$L_2[j]$	1	2	1	2	

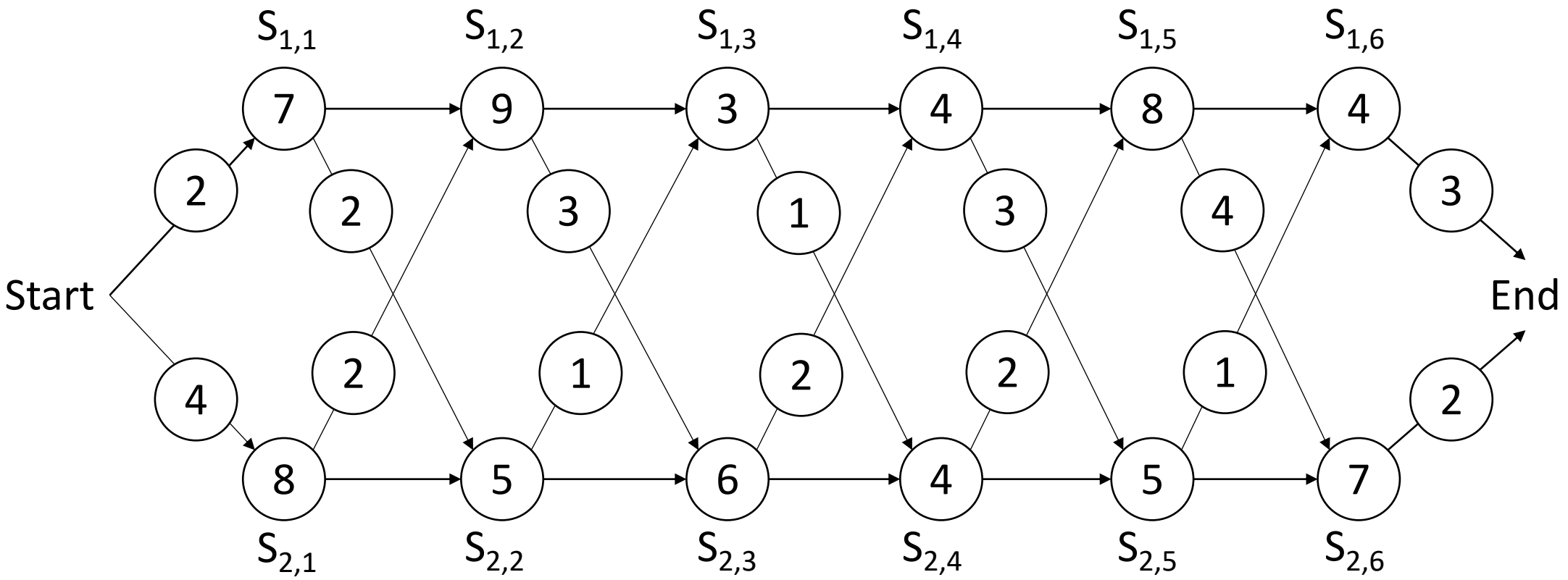


$$f_1[6] = \min(32 + 4, \mathbf{30} + \mathbf{1} + \mathbf{4}) = 35$$

$$f_2[6] = \min(\mathbf{30} + \mathbf{7}, 32 + 4 + 7) = 37$$

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

j	2	3	4	5	6
$L_1[j]$	1	2	1	1	2
$L_2[j]$	1	2	1	2	2

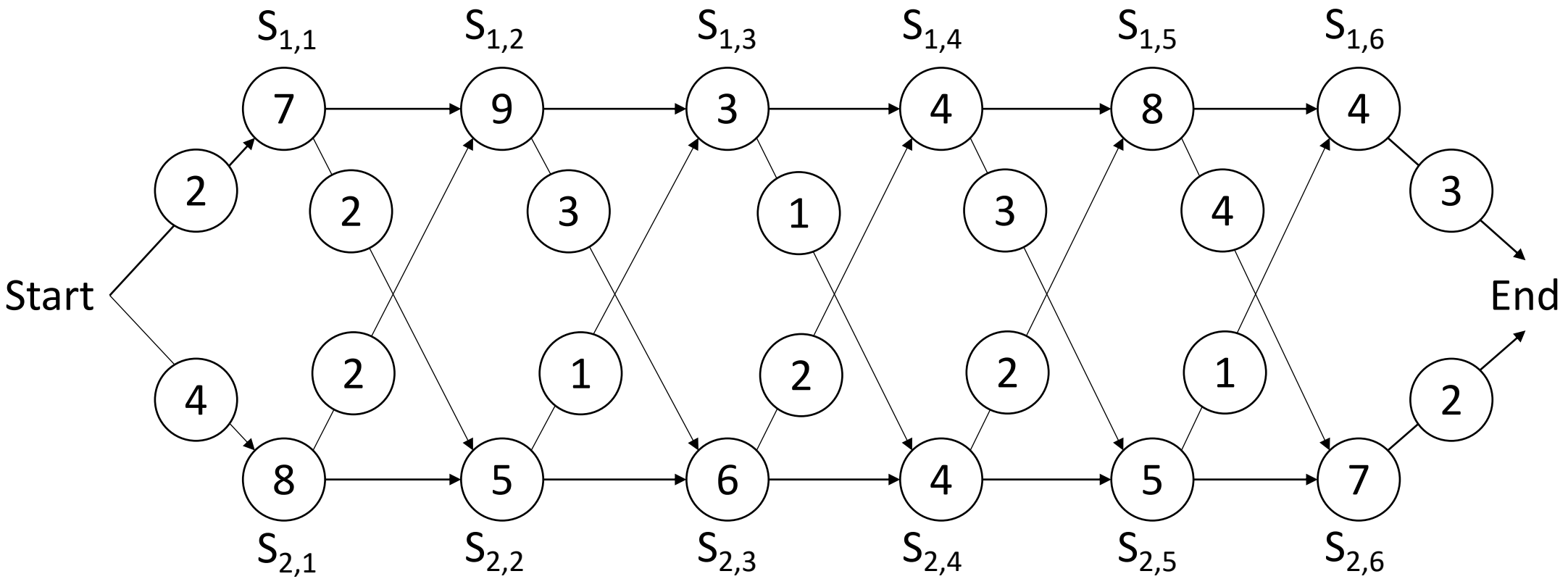


Finally, we add our exit times and identify solve for f^*

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

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

j	2	3	4	5	6
$L_1[j]$	1	2	1	1	2
$L_2[j]$	1	2	1	2	2



$$f^* = \min(\mathbf{35} + \mathbf{3}, 37 + 2) = 38$$

We set L* to 1, our fastest exit station

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

j	2	3	4	5	6
$L_1[j]$	1	2	1	1	2
$L_2[j]$	1	2	1	2	2

$L^* = 1$

Example: Tracing our path

Now that we have our path, the final step is to work our way **backwards** with our L-table, tracing our fastest path.

We start at L* = 1, so we would use station $S_{1,6}$.

If we look at $L_1[6]$, which is 2, so use station $S_{2,5}$.

Then $L_2[5] = 2$ (use $S_{2,4}$)

j	2	3	4	5	6
$L_1[j]$	1	2	1	1	**2**
$L_2[j]$	1	2	1	**2**	2

L* = 1

Example: Tracing our path

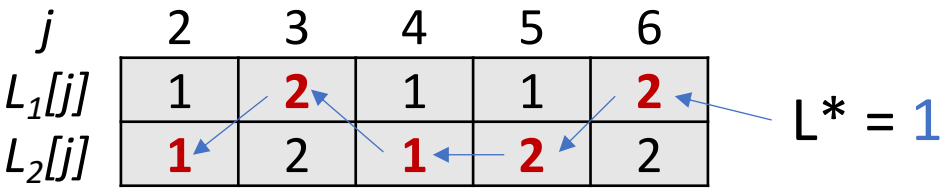
With $L_2[4] = 1$ (use $S_{1,3}$).

With $L_1[3] = 2$ (use $S_{2,2}$).

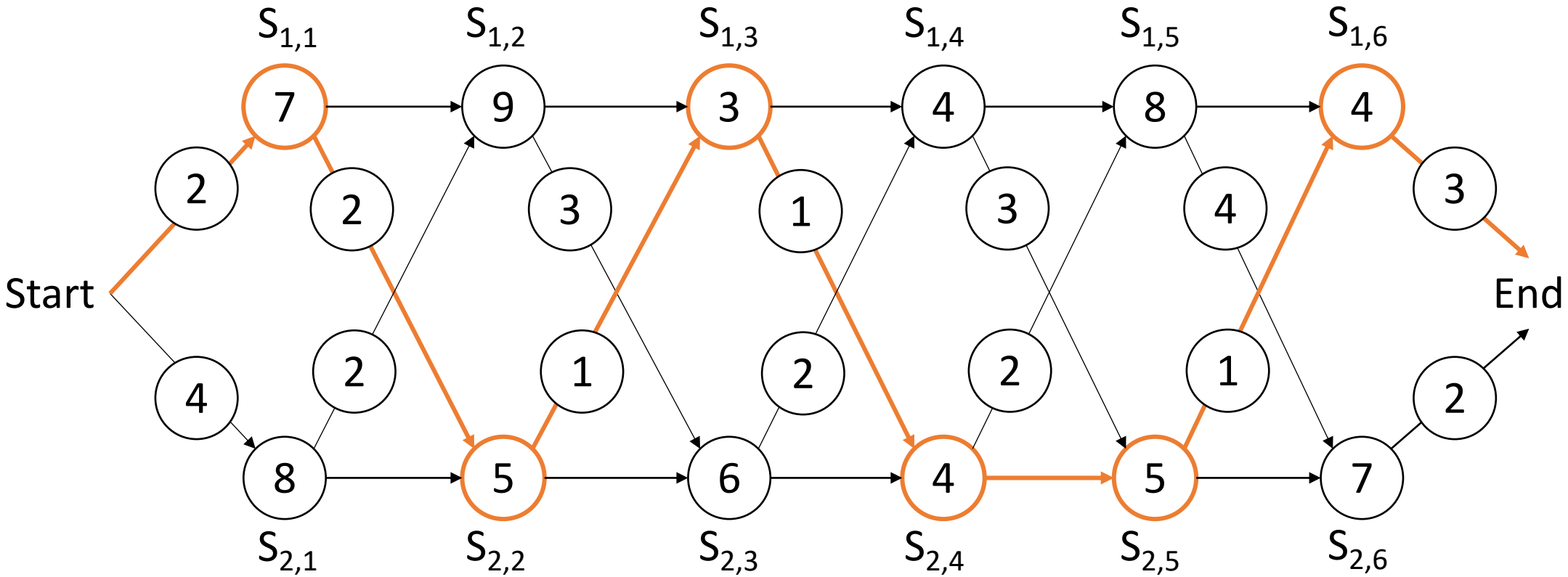
With $L_2[2] = 1$ (use $S_{1,1}$).

Therefore, our fastest path is:

$S_{1,1}, S_{2,2}, S_{1,3}, S_{2,4}, S_{2,5}, S_{1,6}$



Fastest path: $S_{1,1}$, $S_{2,2}$, $S_{1,3}$, $S_{2,4}$, $S_{2,5}$, $S_{1,6}$



Today’s outline

1. Dynamic programming
2. Assembly line scheduling
3. Fastest path and a recursive solution
4. Example
5. Dynamic programming iterative solution

An iterative solution

We will construct our iterative solution using the same logic that guided our in-class example.

We can avoid exponential running time by referencing previously calculated values, which we will store in our *f*-table. This saves us wasted cycles recomputing the same values.

As noted in L22, dynamic programming uses additional memory to save computation time. The cached results always take the form a table.

Recall for $j \geq 2$, $f_i[j]$ depends only on $f_1[j\text{-}1]$ and $f_2[j\text{-}1]$. By computing fi[j] by increasing station number, j moves left to right, we can compute our fastest way in Θ(n) time.

```
procedure Fastest-Way(a, t, e, x, n) // Assembly, trans, entry, exit times, # of stations
1:    f1[1] = e1 + a1,1                  // Initialise time after station 1
2:    f2[1] = e2 + a2,1
3:    for j = 2 to n
4:        if f1[j-1] + a1,j <= f2[j-1] + t2,j-1 + a1,j    // Line 1 finish times
5:            f1[j] = f1[j-1] + a1,j;  L1[j] = 1           // Stay on line 1
6:        else
7:            f1[j] = f2[j-1] + t2,j-1 + a1,j;  L1[j] = 2 // Move to line 2
8:        if f2[j-1] + a2,j <= f1[j-1] + t1,j-1 + a2,j    // Line 2 finish times
9:            f2[j] = f2[j-1] + a2;  L2[j] = 2            // Stay on line 2
10:       else
11:           f2[j] = f1[j-1] + t1,j-1 + a2,j;  L2[j] = 1 // Move to line 1
11:       if f1[n] + x1 ≤ f2[n] + x2
12:           f* = f1[n] + x1          // Fastest time is exit from Line 1
13:           L* = 1                   // Line 1 was most recent fastest way
14:       else
15:           f* = f2[n] + x2          // Fastest time is exit from Line 2
16:           L* = 2                   // Line 2 was most recent fastest way
17:   return fi, f*, Li, L*
end procedure
```

Printing our path

Having computed our fastest way solution, we need a final helper method to convert our L* and L-table into a path of stations.

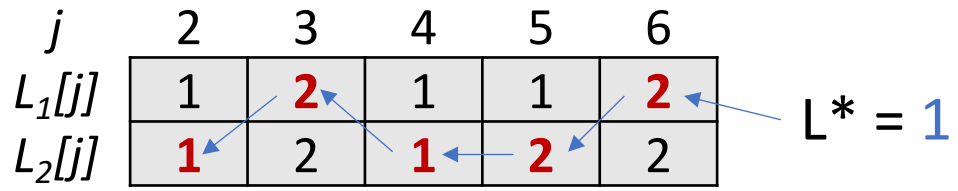
```
procedure Print-Stations(L*, Li, n) // Final line, array of lines, number of stations
1:    i = L*                                  // Initialise final exit line
2:    print “line ” + i + “, station ” + n // Print last station we exit from
3:    for j = n downto 2
4:        i = Li[j]                            // Set line number to print
5:        print “line ” + i + “, station ” + j-1 // Prints line, stations j-1 to 1
end procedure
```

Print output

This procedure outputs stations in reverse order, beginning with our exit station & line.

Print-Stations output:
line 1, station 6
line 2, station 5
line 2, station 4
line 1, station 3
line 2, station 2
line 1, station 1

Tables from our example



It's relatively easy to rework our algorithm to print stations in increasing order, to do so we would use recursion (left as an exercise).

Suggested reading

Dynamic programming is discussed in Section 15.

Section 15.3 is particularly relevant to understanding dynamic programming in general.

Assembly line scheduling is no longer covered in the 3rd edition of the textbook. It was replaced by rod-cutting in the 3rd edition, a more straightforward introduction to dynamic programming.

However, assembly line programming is covered in Section 15.1 of the 2nd edition.