Dynamic Programming

Part 2

6.7 Sequence Alignment in Linear Space

Sequence Alignment: Linear Space

Q. Can we avoid using quadratic space?

Easy. Optimal value in $O(m + n)$ space and $O(mn)$ time.

- Compute $OPT(i, \cdot)$ from $OPT(i-1, \cdot)$.
- No longer a simple way to recover alignment itself.

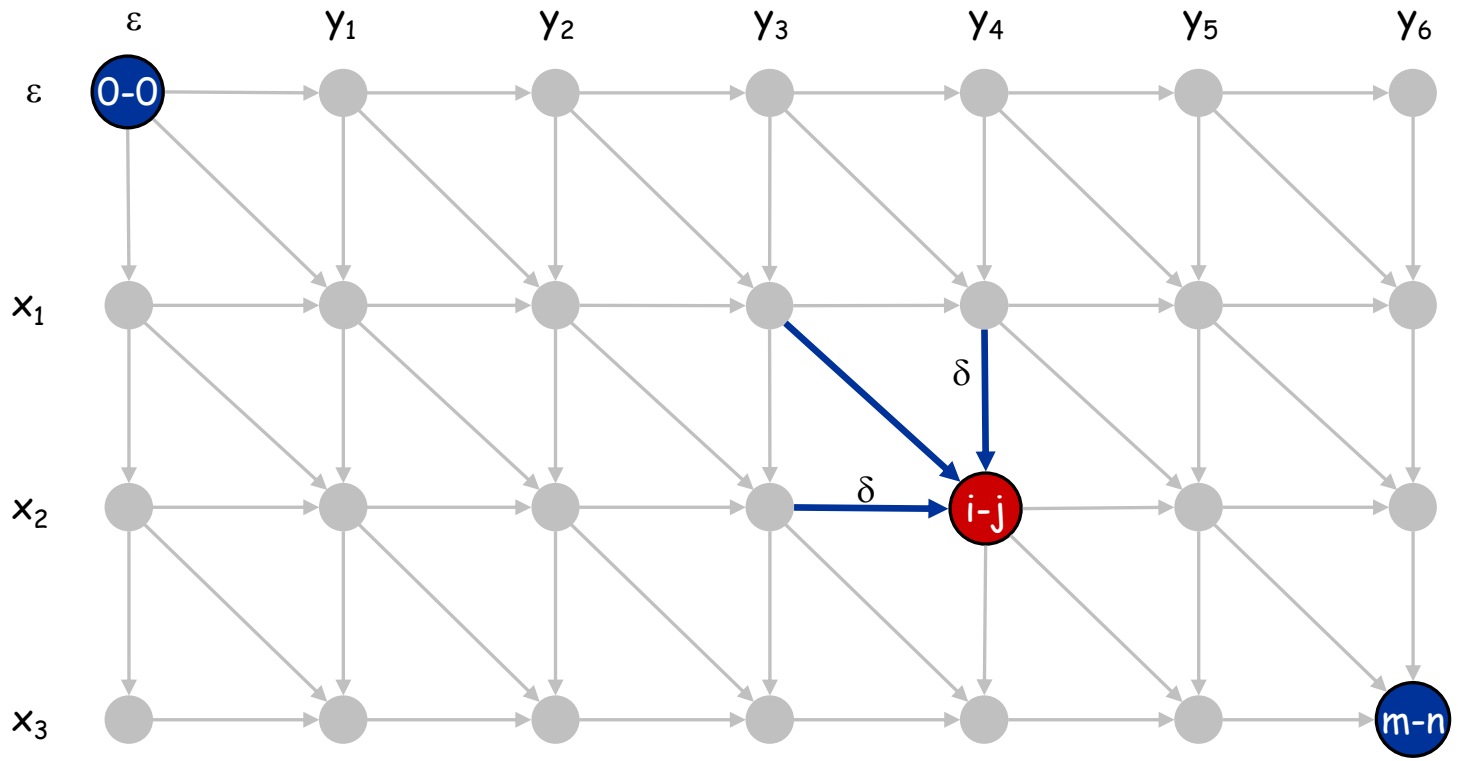
Theorem. [Hirschberg 1975] Optimal alignment in $O(m + n)$ space and $O(mn)$ time.

- Clever combination of divide-and-conquer and dynamic programming.
- Inspired by idea of Savitch from complexity theory.

Sequence Alignment: Linear Space

Edit distance graph.

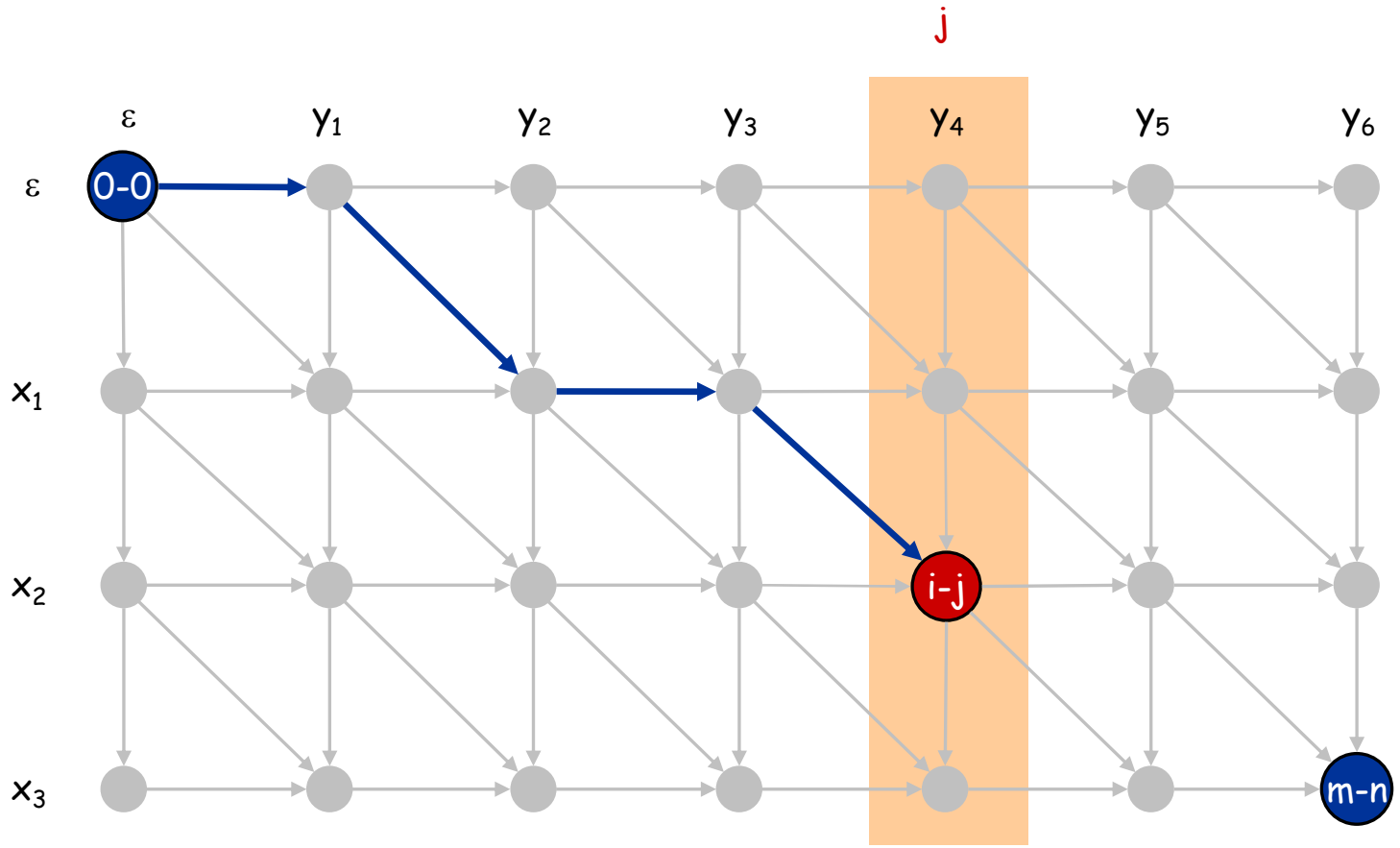
- Let f(i, j) be shortest path from (0,0) to (i, j).
- Observation: f(i, j) = OPT(i, j).



Sequence Alignment: Linear Space

Edit distance graph.

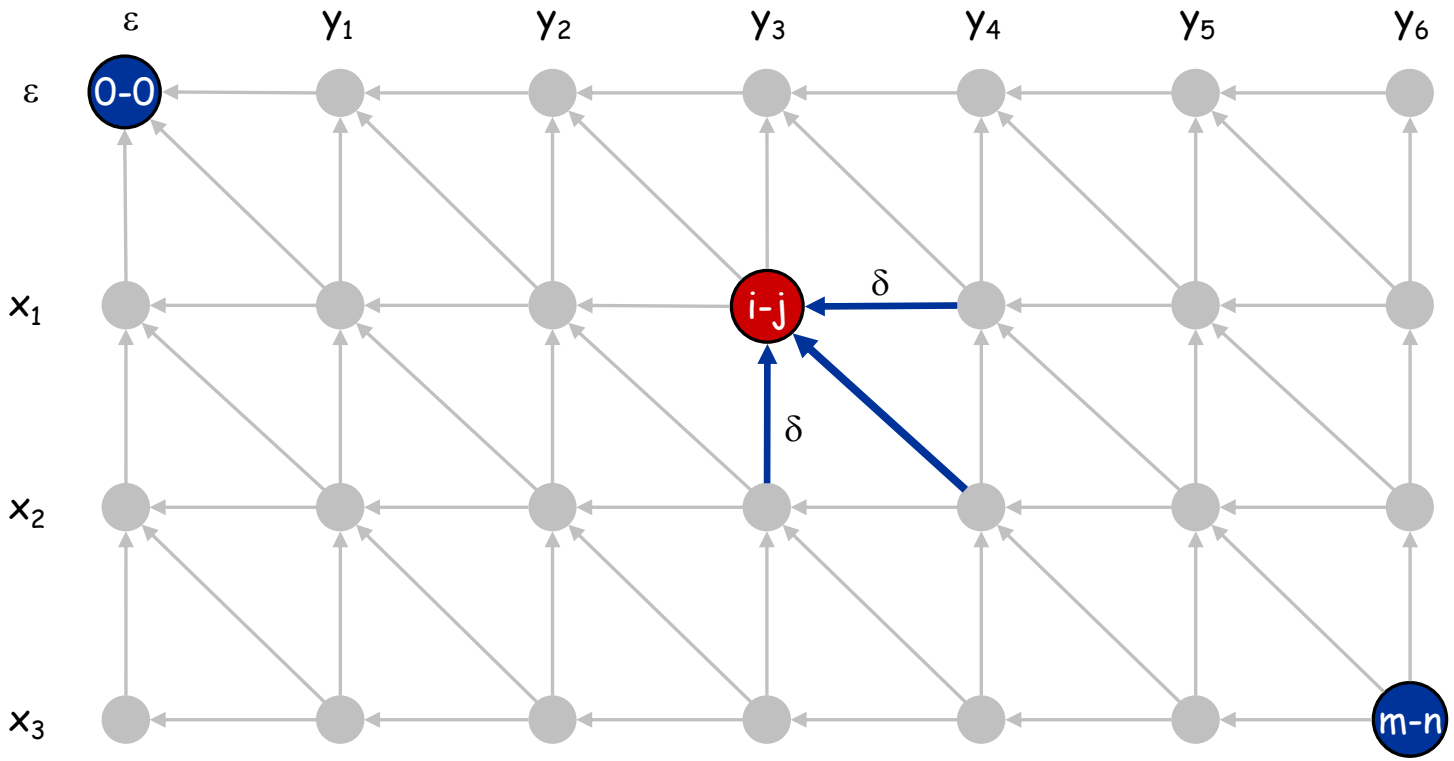
- Let f(i, j) be shortest path from (0,0) to (i, j).
- Can compute f (•, j) for any j in O(mn) time and O(m + n) space.



Sequence Alignment: Linear Space

Edit distance graph.

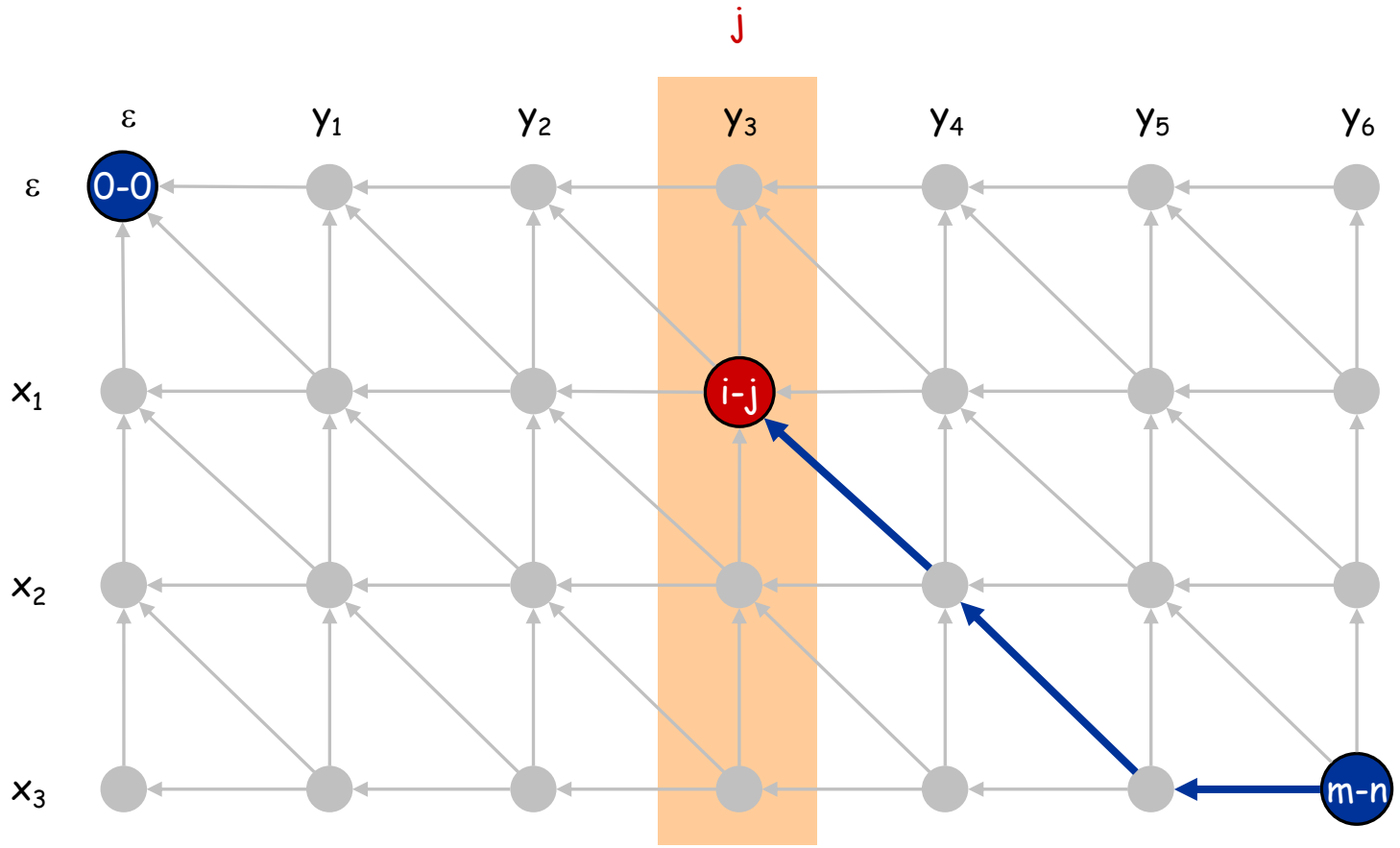
- Let g(i, j) be shortest path from (i, j) to (m, n).
- Can compute by reversing the edge orientations and inverting the roles of (0, 0) and (m, n)



Sequence Alignment: Linear Space

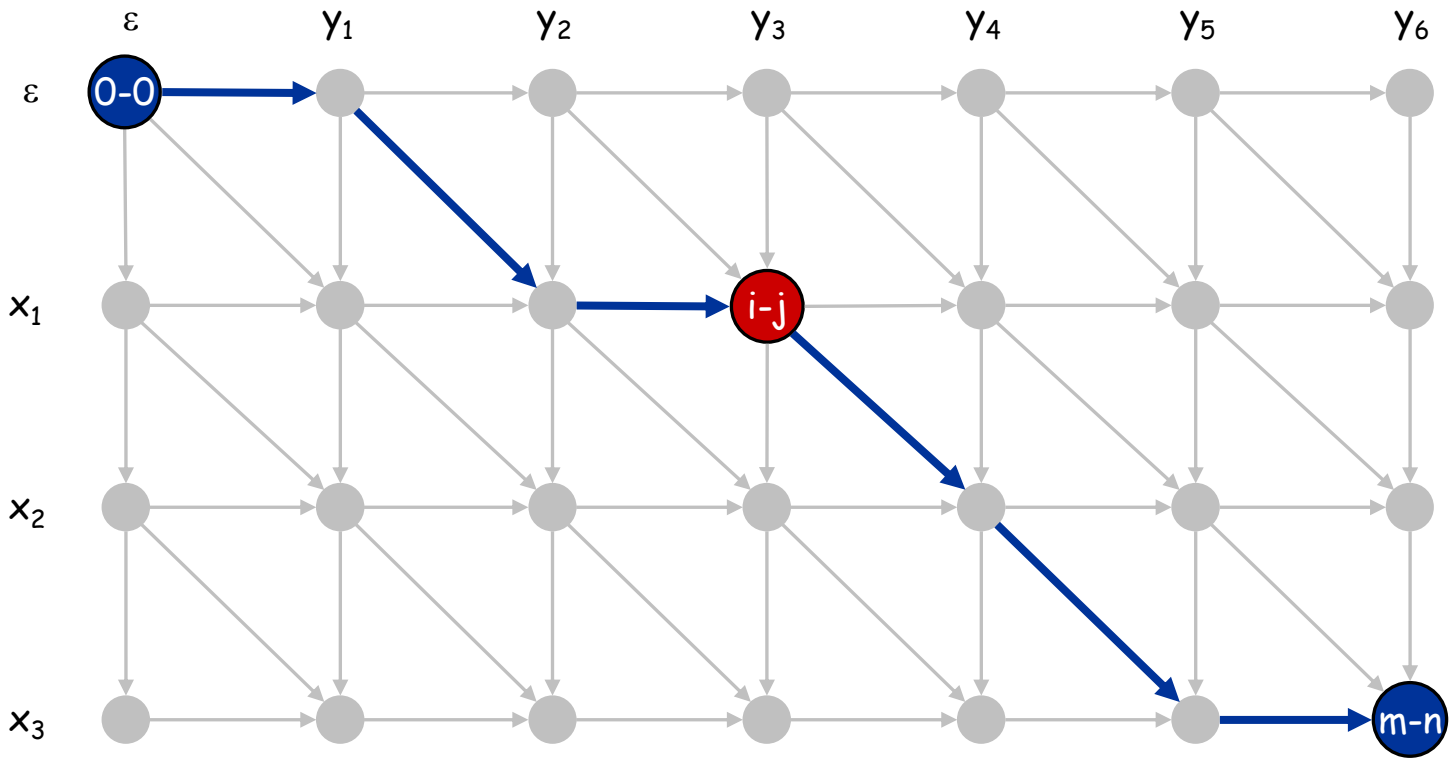
Edit distance graph.

- Let g(i, j) be shortest path from (i, j) to (m, n).
- Can compute g(•, j) for any j in O(mn) time and O(m + n) space.



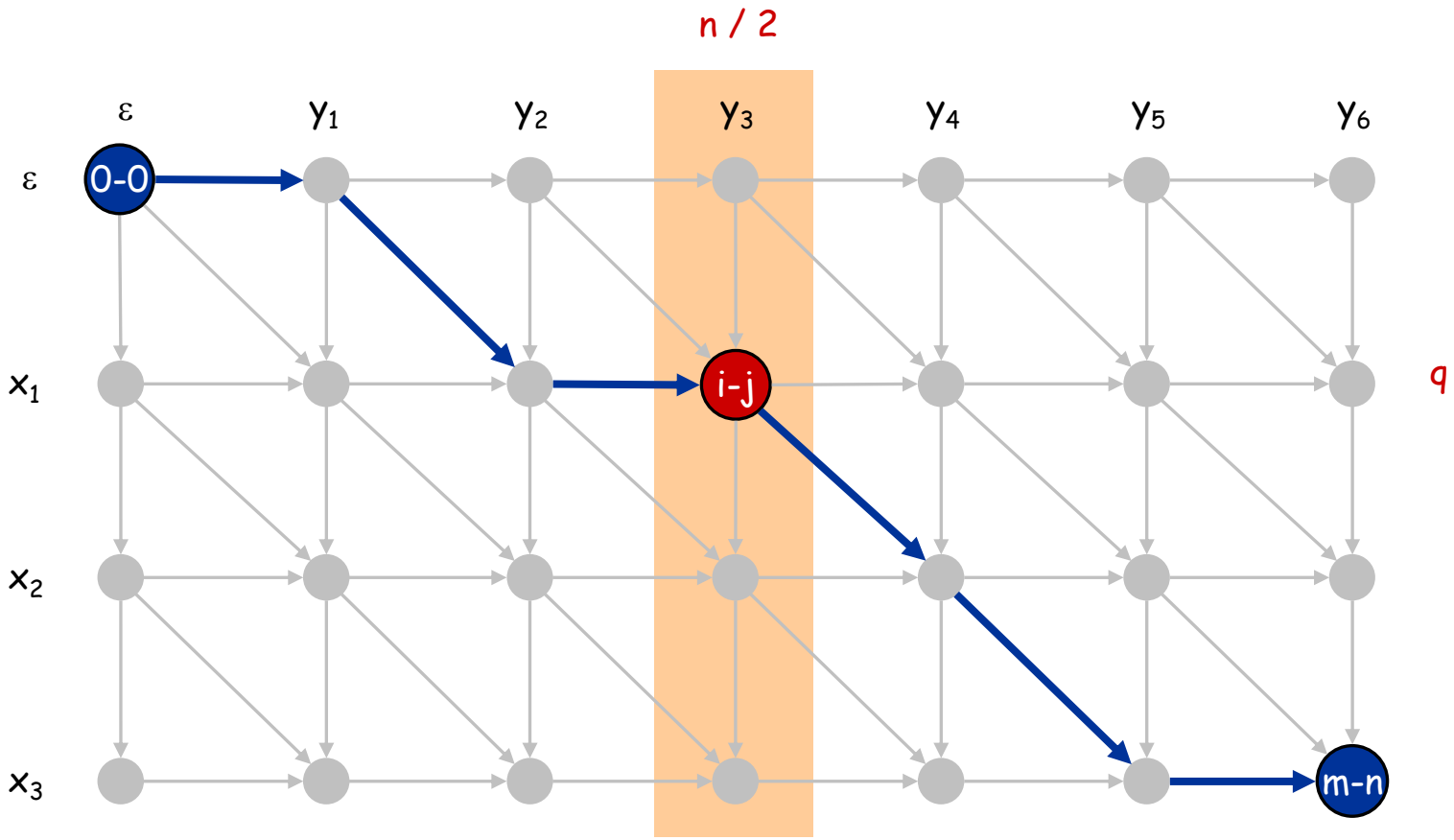
Sequence Alignment: Linear Space

Observation 1. The cost of the shortest path that uses (i, j) is f(i, j) + g(i, j).



Sequence Alignment: Linear Space

Observation 2. let q be an index that minimizes f(q, n/2) + g(q, n/2). Then, the shortest path from (0, 0) to (m, n) uses (q, n/2).

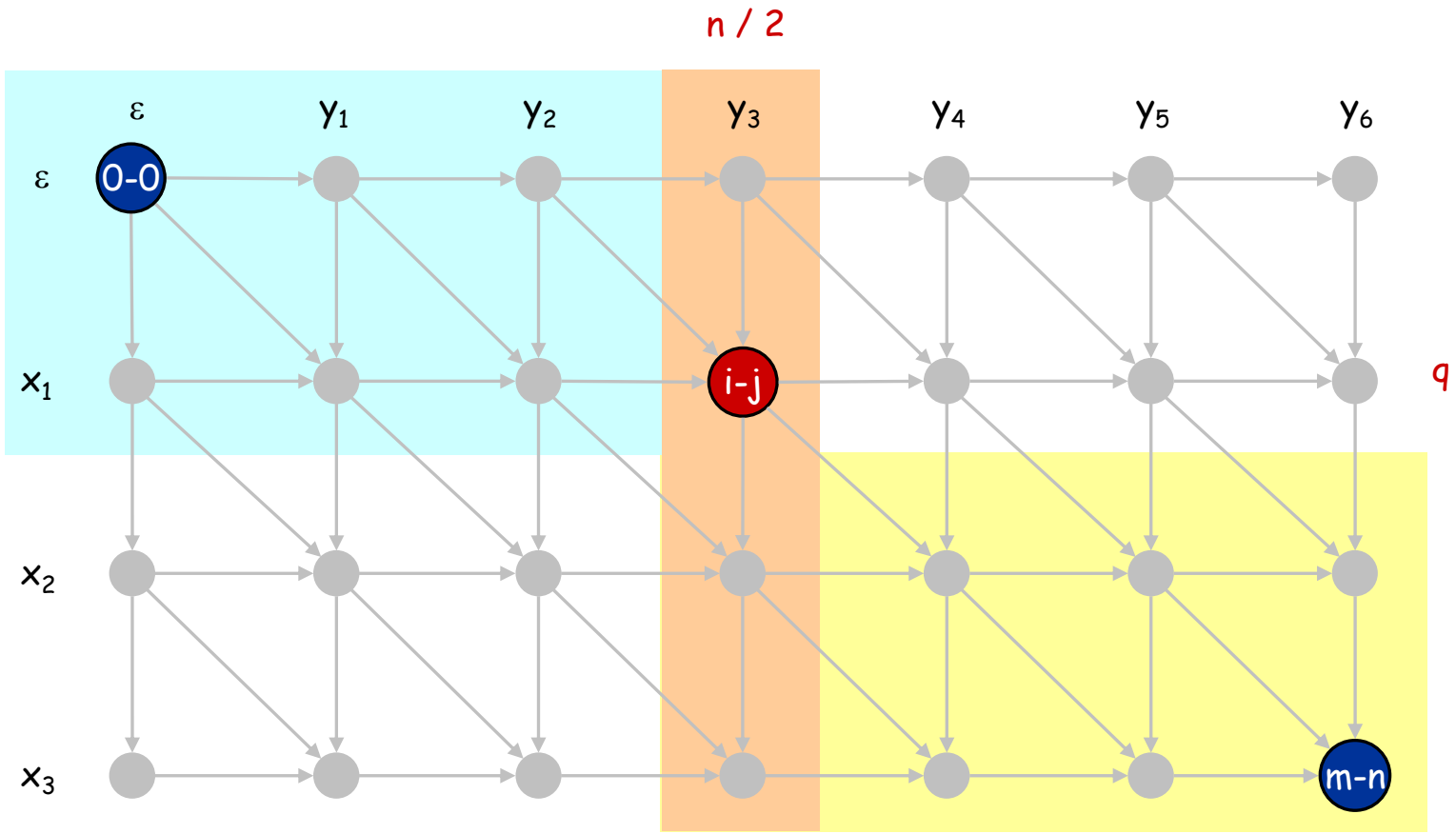


Sequence Alignment: Linear Space

Divide: find index q that minimizes f(q, n/2) + g(q, n/2) using DP.

- Align x_q and $y_{n/2}$.

Conquer: recursively compute optimal alignment in each piece.



Sequence Alignment: Running Time Analysis Warmup

Theorem. Let $T(m, n)$ = max running time of algorithm on strings of length at most m and n. $T(m, n) = O(mn \log n)$.

Pf. $T(m,n)$ is monotone nondecreasing in both m and n.

$$T(m,n) \leq 2\ T(m, n/2) + O(mn)$$

which solves to $T(m,n) = O(mn \log n)$

Remark. Analysis is not tight because two sub-problems are of size $(q, n/2)$ and $(m - q, n/2)$. In next slide, we save log n factor.

Sequence Alignment: Running Time Analysis

Theorem. Let T(m, n) = max running time of algorithm on strings of length m and n. T(m, n) = O(mn).

Pf. (by induction on n)

- O(mn) time to compute f(•, n/2) and g (•, n/2) and find index q.
- T(q, n/2) + T(m - q, n/2) time for two recursive calls.
- Choose constant c so that:

$$\begin{aligned} T(m, 2) &\le c\,m \\ T(2, n) &\le c\,n \\ T(m, n) &\le c\,m\,n + T(q, n/2) + T(m - q, n/2) \end{aligned}$$

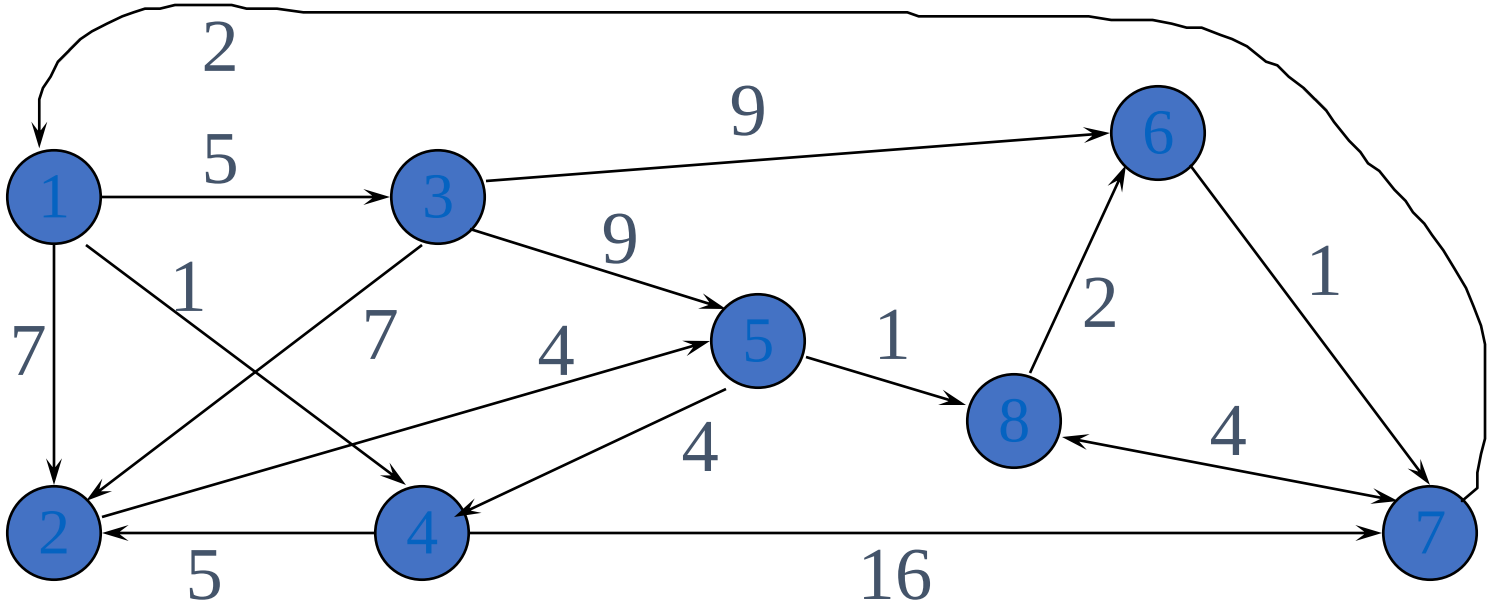
- Base cases: m = 2 or n = 2.
- Inductive hypothesis: T(m, n) ≤ 2cmn.

$$\begin{aligned} T(m,n) &\le T(q,n/2) + T(m-q,n/2) + cmn \\ &\le 2cqn/2 + 2c(m-q)n/2 + cmn \\ &= cqn + cmn - cqn + cmn \\ &= 2cmn \end{aligned}$$

All-Pairs Shortest Paths Problem

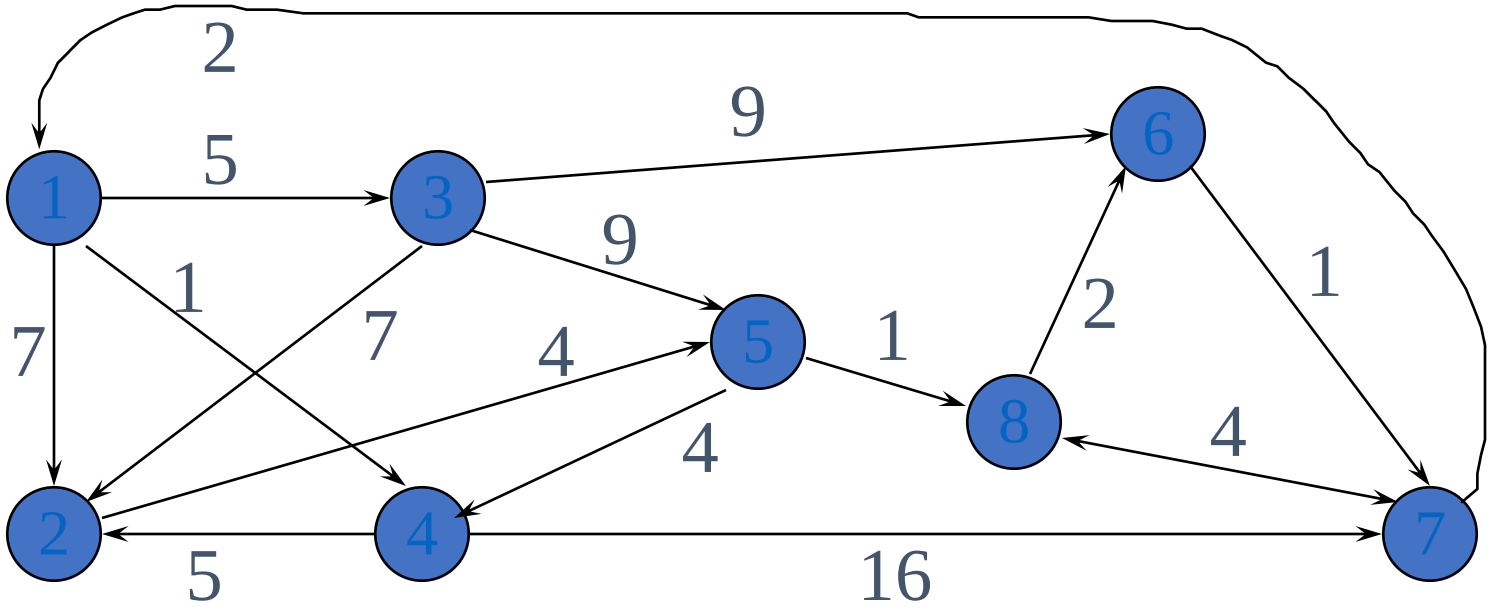
All-Pairs Shortest Paths

- Given an n-vertex directed weighted graph, find a shortest path from vertex i to vertex j for each of the n^2 vertex pairs (i,j).

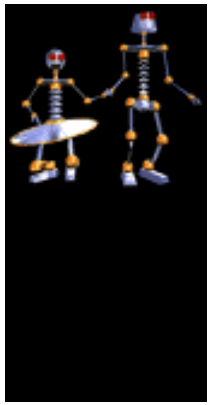


Dijkstra's Single Source Algorithm

- Use Dijkstra's algorithm n times, once with each of the n vertices as the source vertex.



Performance

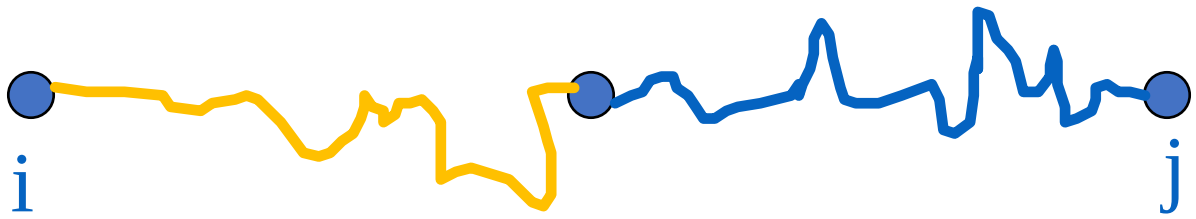


- Time complexity is $O(n^3)$ time.
- Works only when no edge has a cost < 0.

Dynamic Programming Solution

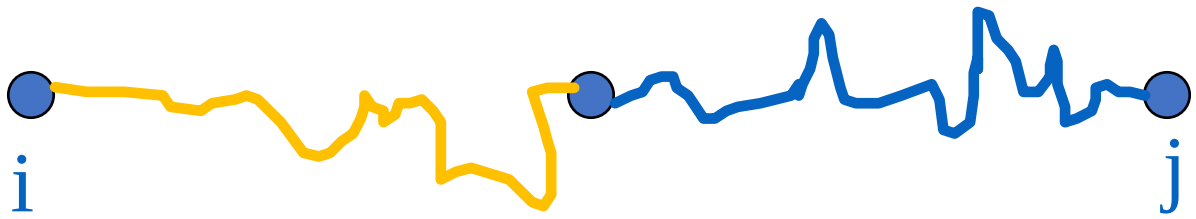
- Time complexity is Theta(n^3) time.
- Works so long as there is no cycle whose length is < 0.
- When there is a cycle whose length is < 0, some shortest paths aren't finite.
  - If vertex 1 is on a cycle whose length is -2, each time you go around this cycle once you get a 1 to 1 path that is 2 units shorter than the previous one.
- Simpler to code, smaller overheads.
- Known as Floyd's shortest paths algorithm.

Decision Sequence



- First decide the highest intermediate vertex (i.e., largest vertex number) on the shortest path from i to j.
- If the shortest path is i, 2, 6, 3, 8, 5, 7, j the first decision is that vertex 8 is an intermediate vertex on the shortest path and no intermediate vertex is larger than 8.
- Then decide the highest intermediate vertex on the path from i to 8, and so on.

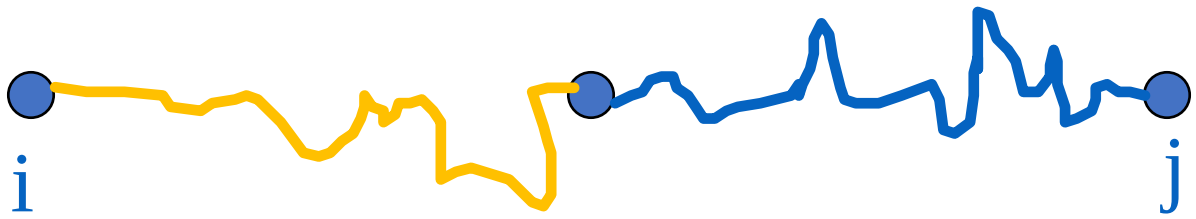
Problem State



- (i,j,k) denotes the problem of finding the shortest path from vertex i to vertex j that has no intermediate vertex larger than k.

- (i,j,n) denotes the problem of finding the shortest path from vertex i to vertex j (with no restrictions on intermediate vertices).

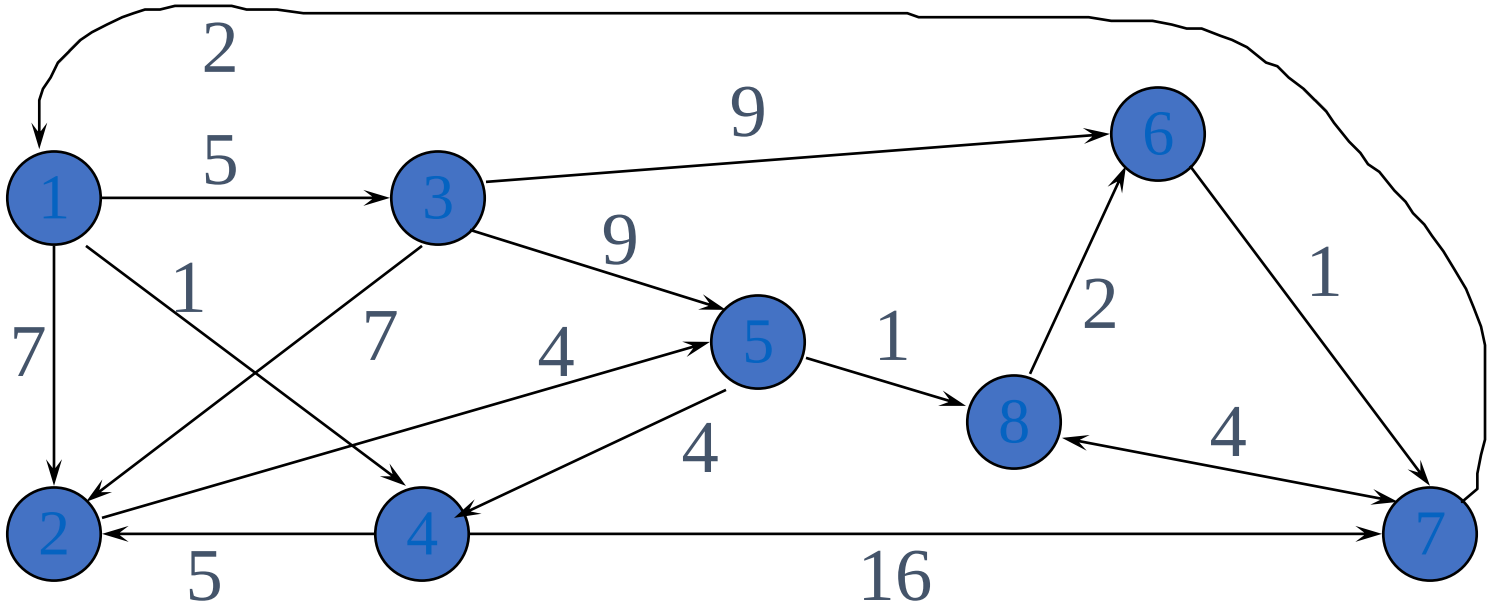
Cost Function



- Let c(i,j,k) be the length of a shortest path from vertex i to vertex j that has no intermediate vertex larger than k.

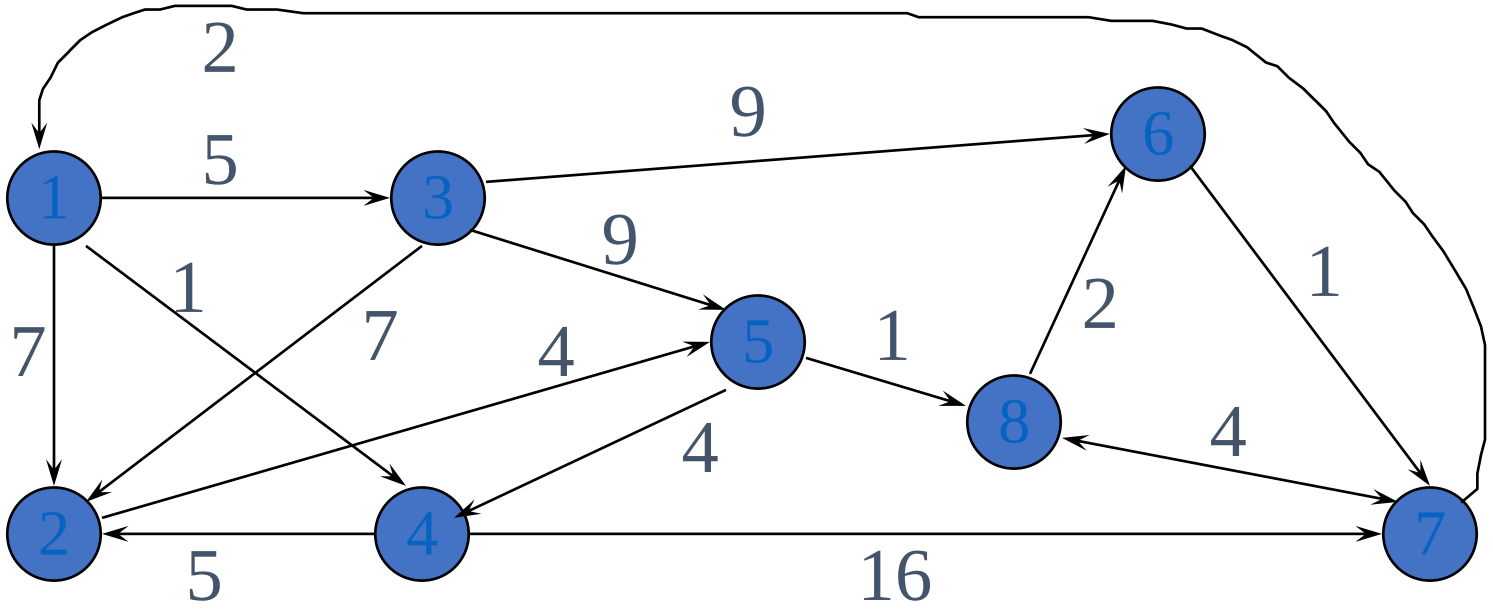
c(i,j,n)

- c(i,j,n) is the length of a shortest path from vertex i to vertex j that has no intermediate vertex larger than n.
- No vertex is larger than n.
- Therefore, c(i,j,n) is the length of a shortest path from vertex i to vertex j.



c(i,j,0)

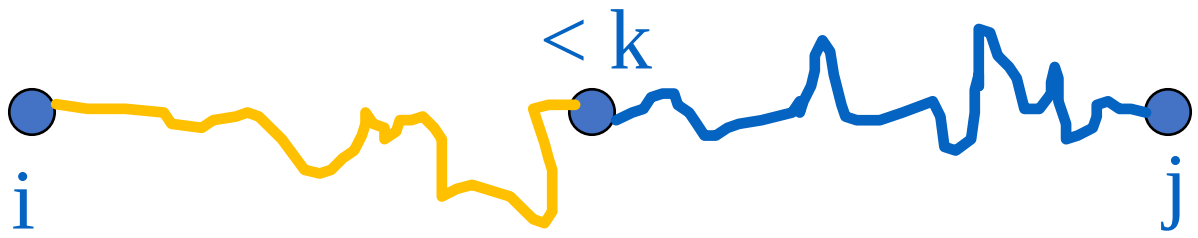
- c(i,j,0) is the length of a shortest path from vertex i to vertex j that has no intermediate vertex larger than 0.
  - Every vertex is larger than 0.
  - Therefore, c(i,j,0) is the length of a single-edge path from vertex i to vertex j.



Recurrence For c(i,j,k), k > 0

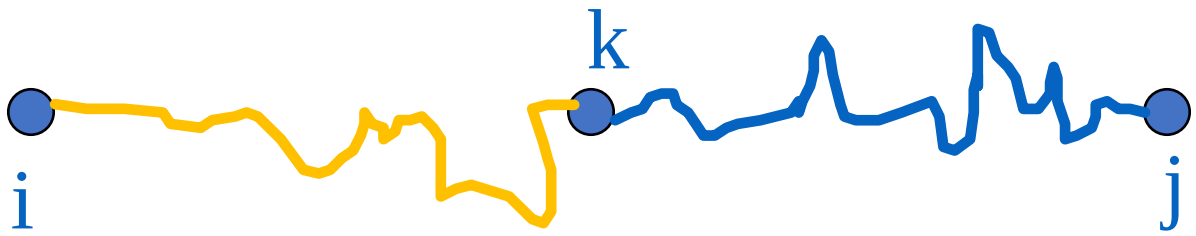
- The shortest path from vertex i to vertex j that has no intermediate vertex larger than k may or may not go through vertex k.

- If this shortest path does not go through vertex k, the largest permissible intermediate vertex is k-1. So the path length is c(i,j,k-1).



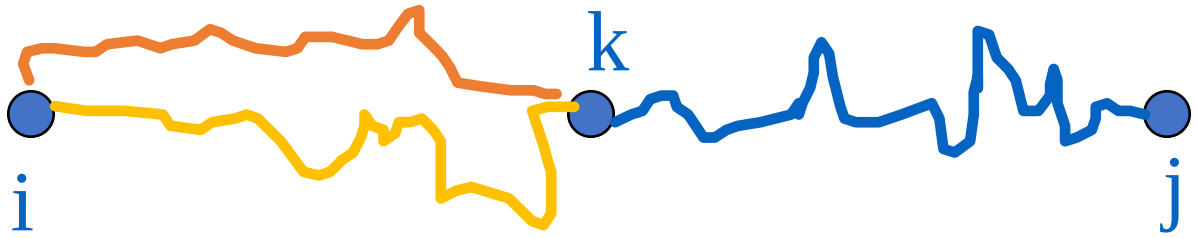
Recurrence For c(i,j,k)), k > 0

- Shortest path goes through vertex k.



- We may assume that vertex k is not repeated because no cycle has negative length.
- Largest permissible intermediate vertex on i to k and k to j paths is k-1.

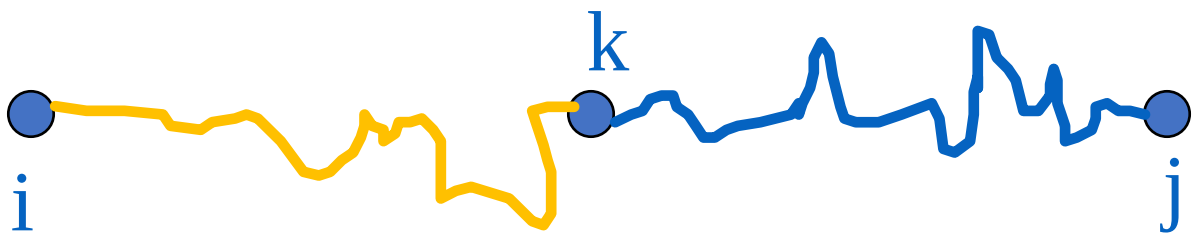
Recurrence For c(i,j,k)), k > 0



- i to k path must be a shortest i to k path that goes through no vertex larger than k-1.

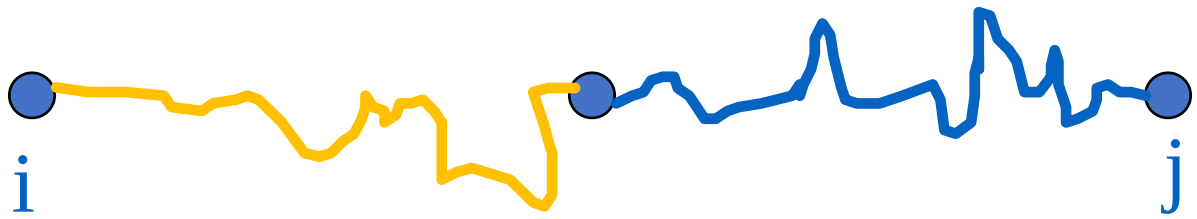
- If not, replace current i to k path with a shorter i to k path to get an even shorter i to j path.

Recurrence For c(i,j,k)), k > 0



- Similarly, k to j path must be a shortest k to j path that goes through no vertex larger than k-1.
- Therefore, length of i to k path is c(i,k,k-1), and length of k to j path is c(k,j,k-1).
- So, c(i,j,k) = c(i,k,k-1) + c(k,j,k-1).

Recurrence For c(i,j,k)), k > 0



- Combining the two equations for c(i,j,k), we get c(i,j,k) = min{c(i,j,k-1), c(i,k,k-1) + c(k,j,k-1)}.

- We may compute the c(i,j,k)s in the order k = 1, 2, 3, …, n.

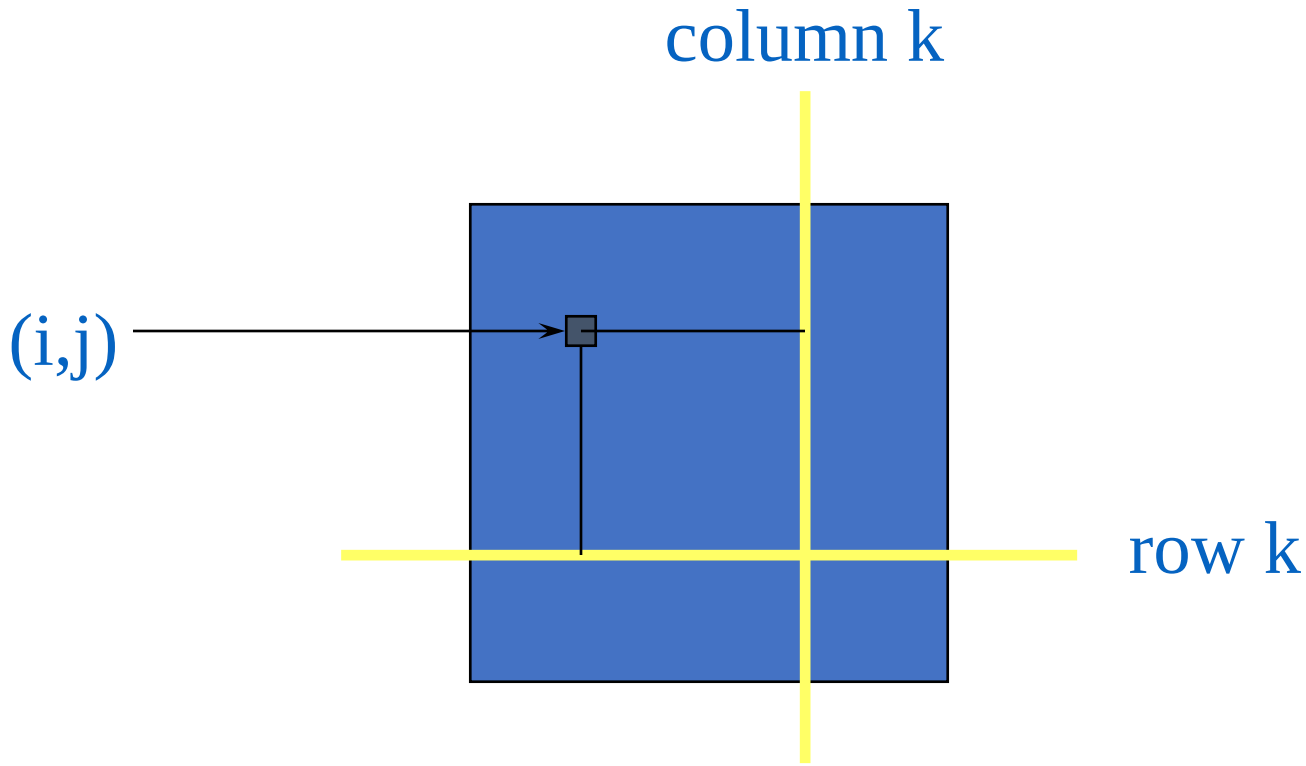
Floyd's Shortest Paths Algorithm

```
for (int k = 1; k <= n; k++)
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
      c(i,j,k) = min{c(i,j,k-1),
                     c(i,k,k-1) + c(k,j,k-1)};
```

- Time complexity is $O(n^3)$.
- More precisely $Theta(n^3)$.
- $Theta(n^3)$ space is needed for c(*,*,*).

Space Reduction

- c(i,j,k) = min{c(i,j,k-1), c(i,k,k-1) + c(k,j,k-1)}
- When neither i nor j equals k, c(i,j,k-1) is used only in the computation of c(i,j,k).



- So, c(i,j,k) can overwrite c(i,j,k-1).

Space Reduction

- $c(i,j,k) = \min\{c(i,j,k-1), c(i,k,k-1) + c(k,j,k-1)\}$

- When i equals k, $c(i,j,k-1)$ equals $c(i,j,k)$.
  - $c(k,j,k) = \min\{c(k,j,k-1), c(k,k,k-1) + c(k,j,k-1)\}$
    $= \min\{c(k,j,k-1), 0 + c(k,j,k-1)\}$
    $= c(k,j,k-1)$

- So, when i equals k, $c(i,j,k)$ can overwrite $c(i,j,k-1)$.
- Similarly, when j equals k, $c(i,j,k)$ can overwrite $c(i,j,k-1)$.
- So, in all cases $c(i,j,k)$ can overwrite $c(i,j,k-1)$.

Floyd's Shortest Paths Algorithm

```
for (int k = 1; k <= n; k++)
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
      c(i,j) = min{c(i,j), c(i,k) + c(k,j)};
```

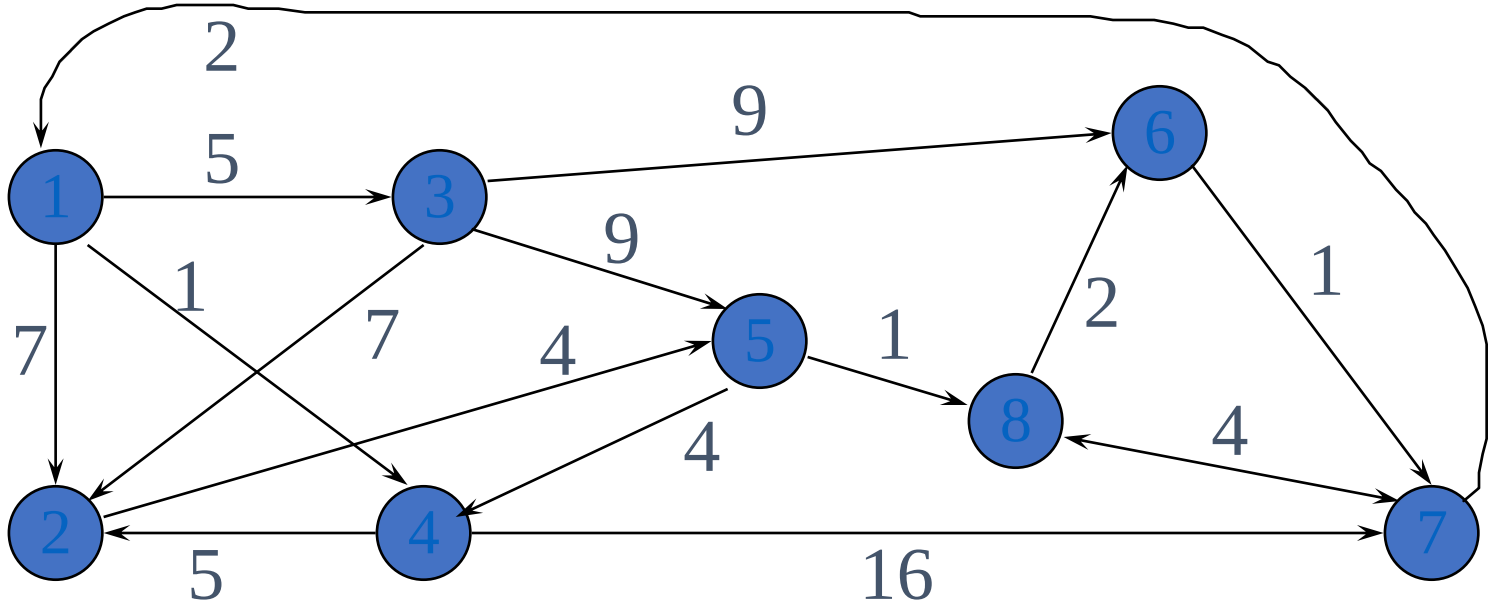
- Initially, $c(i,j) = c(i,j,0)$.
- Upon termination, $c(i,j) = c(i,j,n)$.
- Time complexity is $\text{Theta}(n^3)$.
- $\text{Theta}(n^2)$ space is needed for $c(*,*)$.

Building The Shortest Paths

- Let kay(i,j) be the largest vertex on the shortest path from i to j.
- Initially, kay(i,j) = 0 (shortest path has no intermediate vertex).

```
for (int k = 1; k <= n; k++)
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
      if (c(i,j) > c(i,k) + c(k,j))
      {kay(i,j) = k; c(i,j) = c(i,k) + c(k,j);}
```

Example



-	7	5	1	-	-	-	-
-	-	-	-	4	-	-	-
-	7	-	-	9	9	-	-
-	5	-	-	-	-	16	-
-	-	-	4	-	-	-	1
-	-	-	-	-	-	1	-
2	-	-	-	-	-	-	4
-	-	-	-	-	2	4	-

Initial Cost Matrix

c(*,*) = c(*,*,0)

Final Cost Matrix c(*,*) = c(*,*,n)

0	6	5	1	10	13	14	11
10	0	15	8	4	7	8	5
12	7	0	13	9	9	10	10
15	5	20	0	9	12	13	10
6	9	11	4	0	3	4	1
3	9	8	4	13	0	1	5
2	8	7	3	12	6	0	4
5	11	10	6	15	2	3	0

kay Matrix

7 0 0 5 0 0 6 5

8 0 8 0 2 8 8 5

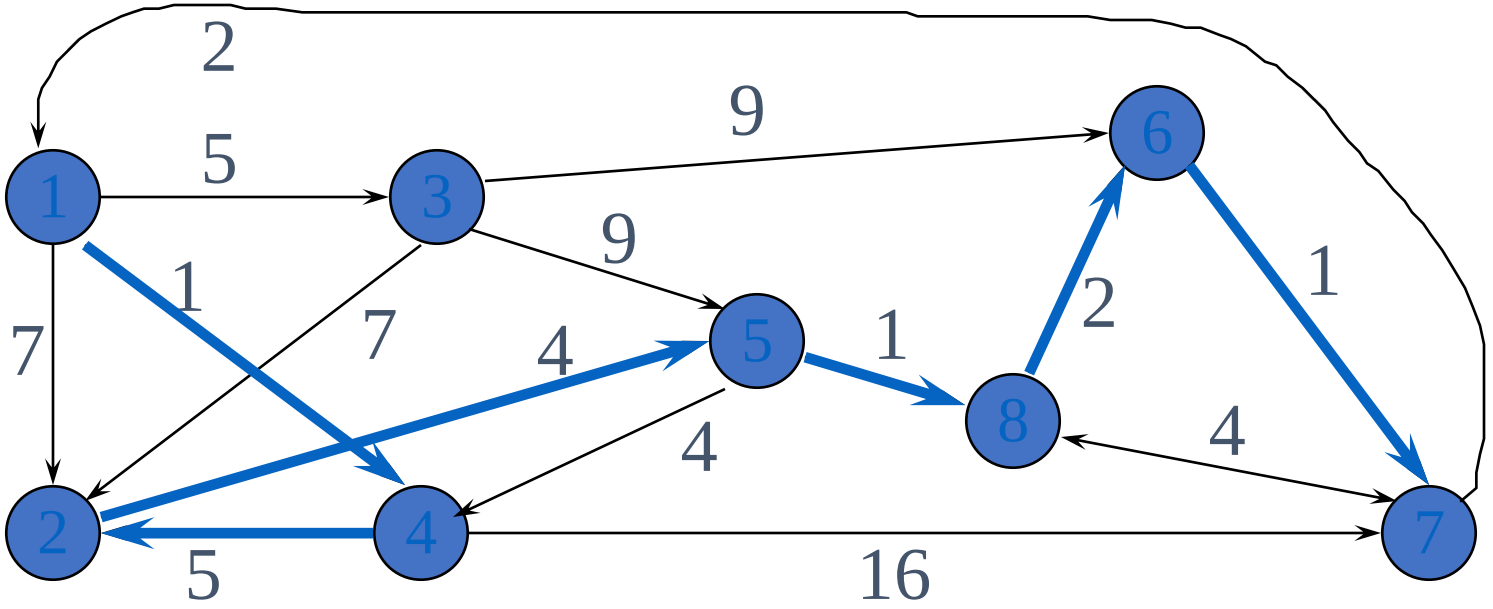
8 4 8 0 0 8 8 0

7 7 7 7 7 0 0 7

0 4 1 1 4 8 0 0

7 7 7 7 7 0 6 0

Shortest Path



Shortest path from 1 to 7.

Path length is 14.

Build A Shortest Path

0 4 0 0 4 8 8 5

8 0 8 5 0 8 8 5

7 0 0 5 0 0 6 5

8 0 8 0 2 8 8 5

8 4 8 0 0 8 8 0

7 7 7 7 7 0 0 7

0 4 1 1 4 8 0 0

7 7 7 7 7 0 6 0

- The path is 1 4 2 5 8 6 7.
- kay(1,7) = 8

- kay(1,8) = 5

- kay(1,5) = 4

Build A Shortest Path

0 4 0 0 4 8 8 5

8 0 8 5 0 8 8 5

7 0 0 5 0 0 6 5

8 0 8 0 2 8 8 5

8 4 8 0 0 8 8 0

7 7 7 7 7 0 0 7

0 4 1 1 4 8 0 0

7 7 7 7 7 0 6 0

- The path is 1 4 2 5 8 6 7.

1 → 4 → 5 → 8 → 7

- kay(1,4) = 0

1 4 → 5 → 8 → 7

- kay(4,5) = 2

1 4 → 2 → 5 → 8 → 7

- kay(4,2) = 0

1 4 2 → 5 → 8 → 7

Build A Shortest Path

0 4 0 0 4 8 8 5

8 0 8 5 0 8 8 5

7 0 0 5 0 0 6 5

8 0 8 0 2 8 8 5

8 4 8 0 0 8 8 0

7 7 7 7 7 0 0 7

0 4 1 1 4 8 0 0

7 7 7 7 7 0 6 0

- The path is 1 4 2 5 8 6 7.

  1 4 2 → 5 → 8 → 7

- kay(2,5) = 0

  1 4 2 5 → 8 → 7

- kay(5,8) = 0

  1 4 2 5 8 → 7

- kay(8,7) = 6

  1 4 2 5 8 → 6 → 7

Build A Shortest Path

0 4 0 0 4 8 8 5

8 0 8 5 0 8 8 5

7 0 0 5 0 0 6 5

8 0 8 0 2 8 8 5

8 4 8 0 0 8 8 0

7 7 7 7 7 0 0 7

0 4 1 1 4 8 0 0

7 7 7 7 7 0 6 0

- The path is 1 4 2 5 8 6 7.

  1 4 2 5 8 → 6 → 7

- kay(8,6) = 0

  1 4 2 5 8 6 → 7

- kay(6,7) = 0

  1 4 2 5 8 6 7

Output A Shortest Path

```
public static void outputPath(int i, int j)
{// does not output first vertex (i) on path
   if (i == j) return;
   if (kay[i][j] == 0)  // no intermediate vertices on path
      System.out.print(j + " ");
   else {// kay[i][j] is an intermediate vertex on the path
            outputPath(i, kay[i][j]);
            outputPath(kay[i][j], j);
         }
}
```

Time Complexity Of outputPath

O(number of vertices on shortest path)

Single Source Shortest Paths with Negative Weights

Single-Source All-Destinations Shortest Paths With Negative Costs

- Directed weighted graph.
- Edges may have negative cost.
- No cycle whose cost is < 0.
- Find a shortest path from a given source vertex s to each of the n vertices of the digraph.

Single-Source All-Destinations Shortest Paths With Negative Costs

- Dijkstra's $O(n^2)$ single-source greedy algorithm doesn't work when there are negative-cost edges.
- Floyd's $\Theta(n^3)$ all-pairs dynamic-programming algorithm does work in this case.

Bellman-Ford Algorithm

- Single-source all-destinations shortest paths in digraphs with negative-cost edges.
- Uses dynamic programming.
- Runs in $O(n^3)$ time when adjacency matrices are used.
- Runs in $O(ne)$ time when adjacency lists are used.

Decision Sequence

- To construct a shortest path from the source to vertex v, decide on the max number of edges on the path and on the vertex that comes just before v.

- Since the digraph has no cycle whose length is < 0, we may limit ourselves to the discovery of cycle-free (acyclic) shortest paths.
- A path that has no cycle has at most $n-1$ edges.

Problem State

- Problem state is given by (u,k), where u is the destination vertex and k is the max number of edges.
- (v,n-1) is the state in which we want the shortest path to v that has at most n-1 edges.

Cost Function

- Let d(v,k) be the length of a shortest path from the source vertex to vertex v under the constraint that the path has at most k edges.
- d(v,n-1) is the length of a shortest unconstrained path from the source vertex to vertex v.
- We want to determine d(v,n-1) for every vertex v.

Value Of d(*,0)

- d(v,0) is the length of a shortest path from the source vertex to vertex v under the constraint that the path has at most 0 edges.

- d(s,0) = 0.
- d(v,0) = infinity for v != s.

Recurrence For d(*,k), k > 0

- d(v,k) is the length of a shortest path from the source vertex to vertex v under the constraint that the path has at most k edges.
- If this constrained shortest path goes through no edge, then d(v,k) = d(v,0).

Recurrence For d(*,k), k > 0

- If this constrained shortest path goes through at least one edge, then let w be the vertex just before v on this shortest path (note that w may be s).

- We see that the path from the source to w must be a shortest path from the source vertex to vertex w under the constraint that this path has at most k-1 edges.
- d(v,k) = d(w,k-1) + length of edge (w,v).

Recurrence For d(*,k), k > 0

- d(v,k) = d(w,k-1) + length of edge (w,v).

- We do not know what w is.
- We can assert
  - d(v,k) = min{d(w,k-1) + length of edge (w,v)}, where the min is taken over all w such that (w,v) is an edge of the digraph.
- Combining the two cases considered yields:
  - d(v,k) = min{d(v,0),
    min{d(w,k-1) + length of edge (w,v)}}

Pseudocode To Compute d(*,*)

```
// initialize d(*,0)
d(s,0) = 0;
d(v,0) = infinity, v != s;
// compute d(*,k), 0 < k < n
for (int k = 1; k < n; k++)
{
    d(v,k) = d(v,0), 1 <= v <= n;
     for (each edge (u,v))
         d(v,k) = min{d(v,k), d(u,k-1) + cost(u,v)}
}
```

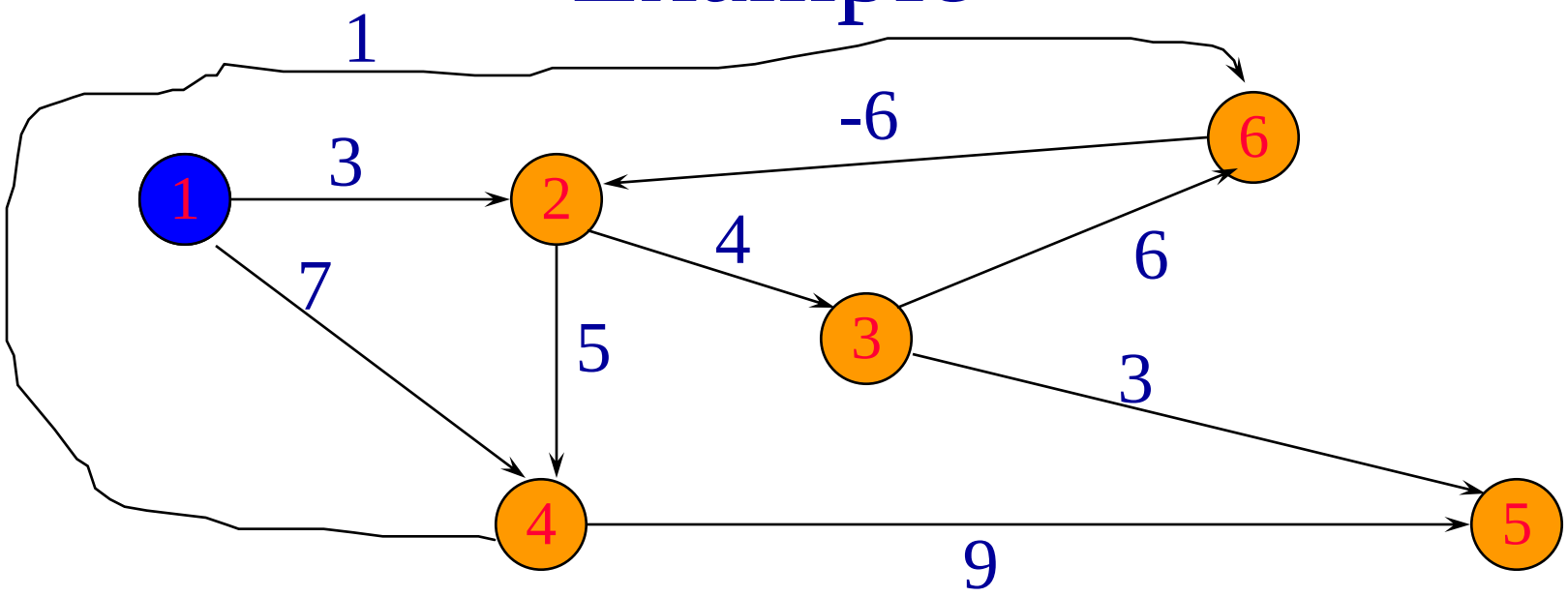
Complexity

- Theta(n) to initialize d(*,0).
- Theta(n^2) to compute d(*,k) for each k > 0 when adjacency matrix is used.
- Theta(e) to compute d(*,k) for each k > 0 when adjacency lists are used.
- Overall time is Theta(n^3) when adjacency matrix is used.
- Overall time is Theta(ne) when adjacency lists are used.
- Theta(n^2) space needed for d(*,*).

p(*,*)

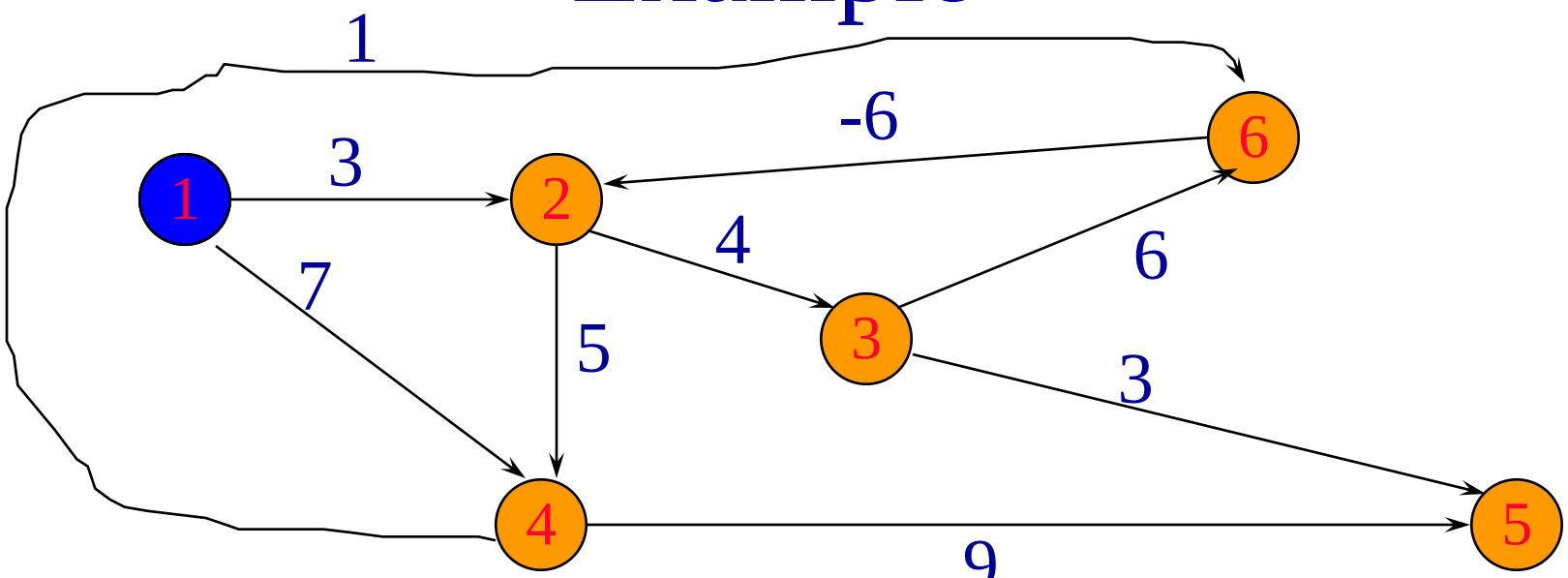
- Let p(v,k) be the vertex just before vertex v on the shortest path for d(v,k).
- p(v,0) is undefined.
- Used to construct shortest paths.

Example



Source vertex is 1.

Example



	1	2	3	4	5	6
0	0	-	-	-	-	-
1	0	3	-	7	-	-
2	0	3	7	7	16	8
3	0	2	7	7	10	8
4	0	2	6	7	10	8

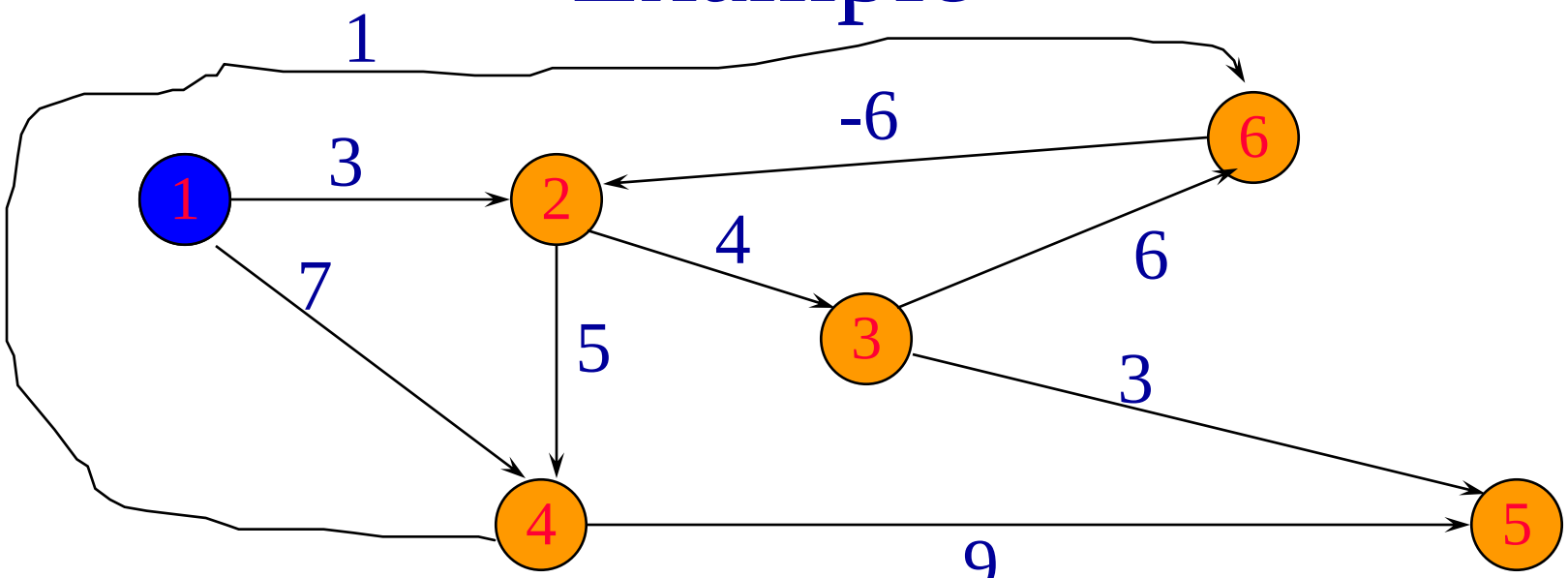
d(v,k)

v →

-	-	-	-	-	-	k
-	1	-	1	-	-	↓
-	1	2	1	4	4	
-	6	2	1	3	4	
-	6	2	1	3	4	

p(v,k)

Example



	1	2	3	4	5	6
4	0	2	6	7	10	8
5	0	2	6	7	9	8

d(v,k)

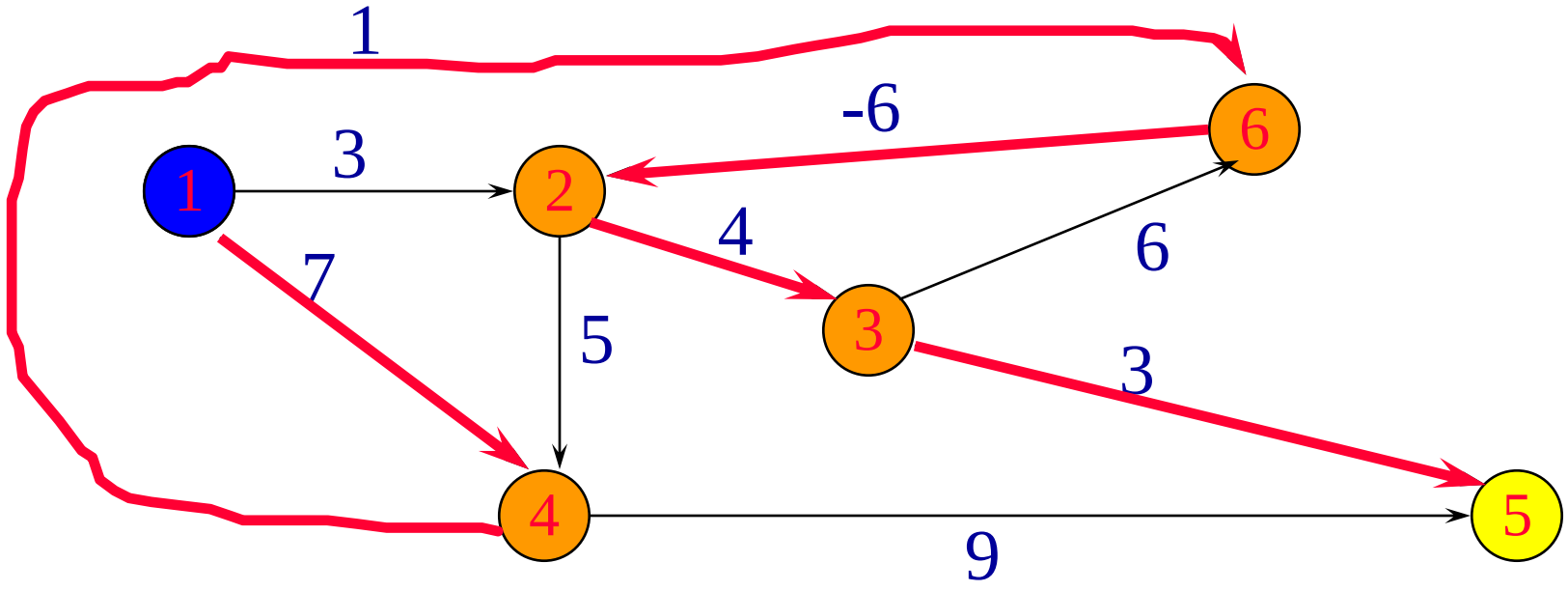
v →

	6	2	1	3	4
-	6	2	1	3	4
-	6	2	1	3	4

k ↓

p(v.k)

Shortest Path From 1 To 5



	1	2	3	4	5	6
5	0	2	6	7	9	8

d(v,5)

1	2	3	4	5	6
-	6	2	1	3	4

p(v,5)

Observations

- d(v,k) = min{d(v,0),
  min{d(w,k-1) + length of edge (w,v)}}
- d(s,k) = 0 for all k.
- If d(v,k) = d(v,k-1) for all v, then d(v,j) = d(v,k-1), for all j >= k-1 and all v.
- If we stop computing as soon as we have a d(*,k) that is identical to d(*,k-1) the run time becomes
  - $O(n^3)$ when adjacency matrix is used.
  - O(ne) when adjacency lists are used.

Observations

- The computation may be done in-place.

  d(v) = min{d(v), min{d(w) + length of edge (w,v)}}

  instead of

  d(v,k) = min{d(v,0),

  min{d(w,k-1) + length of edge (w,v)}}

- Following iteration k, d(v,k+1) <= d(v) <= d(v,k)
- On termination d(v) = d(v,n-1).
- Space requirement becomes O(n) for d(*) and p(*).