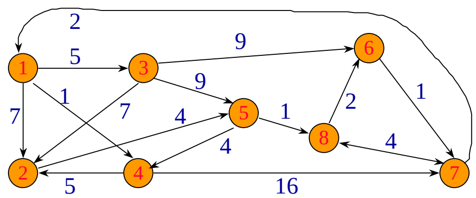
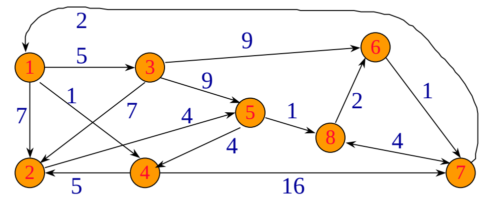
All-Pairs Shortest Paths

- 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).

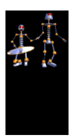


Dijkstra's Single Source Algorithm

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



Performance

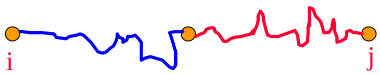


- Time complexity is $O(n^3)$ time.
- Works only when no edge has a cost < 0.

Dynamic Programming Solution

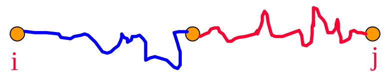
- Time complexity is $\Theta(n^3)$ time.
- Works so long as there is no cycle whose length is < 0.
- When there is a cycle whose length is < 0, some shortest paths aren't finite.
  - If vertex 1 is on a cycle whose length is -2, each time you go around this cycle once you get a 1 to 1 path that is 2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

Decision Sequence



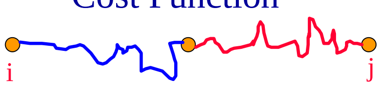
- First decide the highest intermediate vertex (i.e., largest vertex number) on the shortest path from i to j.
- If the shortest path is i, 2, 6, 3, 8, 5, 7, j the first decision is that vertex 8 is an intermediate vertex on the shortest path and no intermediate vertex is larger than 8.
- Then decide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

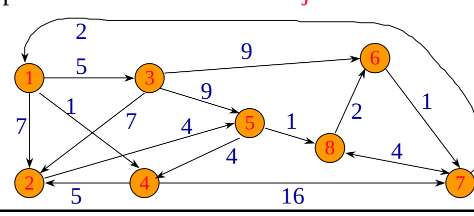
Cost Function



- 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.

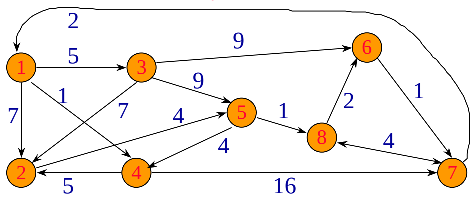
c(i,j,n)

- 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.



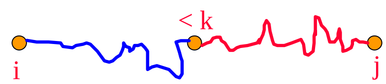
c(i,j,0)

- 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.



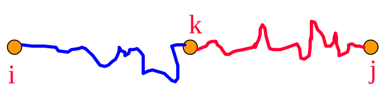
Recurrence For c(i,j,k), k > 0

- 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).



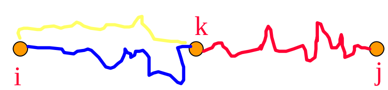
Recurrence For c(i,j,k)), k > 0

- Shortest path goes through vertex k.



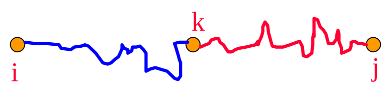
- We may assume that vertex k is not repeated because no cycle has negative length.
- Largest permissible intermediate vertex on i to k and k to j paths is k-1.

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



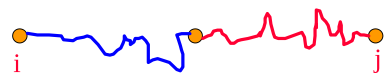
- i to k path must be a shortest i to k path that goes through no vertex larger than k-1.
- If not, replace current i to k path with a shorter i to k path to get an even shorter i to j path.

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.

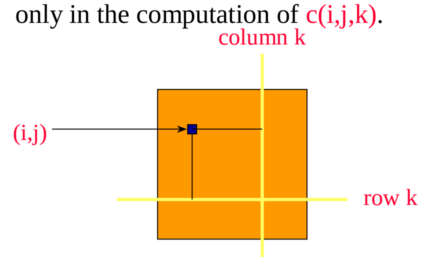
Floyd's Shortest Paths Algorithm

```
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).

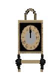
Space Reduction

- 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 Theta(n^3).
- Theta(n^2) space is needed for c(*,*).

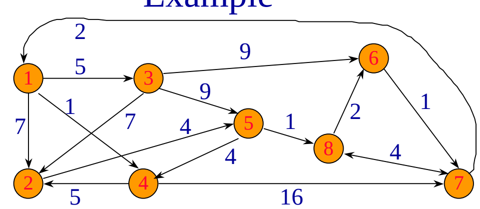


Building The Shortest Paths

- 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)

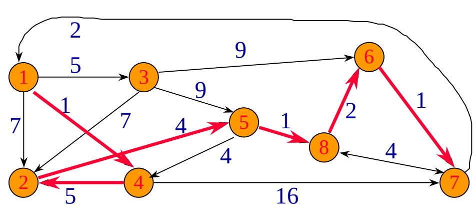
Final Cost Matrix c(*,*) = c(*,*,n)

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

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

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

  1 → 8 → 7
- kay(1,8) = 5

  1 → 5 → 8 → 7
- kay(1,5) = 4

  1 → 4 → 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 → 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)