INFDEV036A - Algorithms Lesson Unit 6

G. Costantini, F. Di Giacomo

costg@hr.nl, giacf@hr.nl – Office H4.206

Today

- ~~Why is my code slow?~~
  - ~~Empirical and complexity analysis~~
- ~~How do I order my data?~~
  - ~~Sorting algorithms~~
- ~~How do I structure my data?~~
  - ~~Linear, tabular, recursive data structures~~
- How do I represent relationship networks?
  - Graphs

More detailed agenda

- What is **dynamic programming**?
  - Fibonacci example; memoization and bottom-up approaches
- How can we find the shortest paths between **all pairs** of nodes in a graph?
  - Floyd Warshall algorithm

Dynamic Programming

Fibonacci sequence, Memoization, Bottom up, General idea

Fibonacci Sequence
Definition

- Sequence of integer numbers, built according to the following rules
  - Elements 0 and 1 are 1
  - Any other element is the result of the sum of the two preceding elements in the sequence
- Recursive formulation

$$F(0) = 1$$

$$F(1) = 1$$

$$F(n) = F(n-1) + F(n-2)$$

Fibonacci Sequence

Recursive function

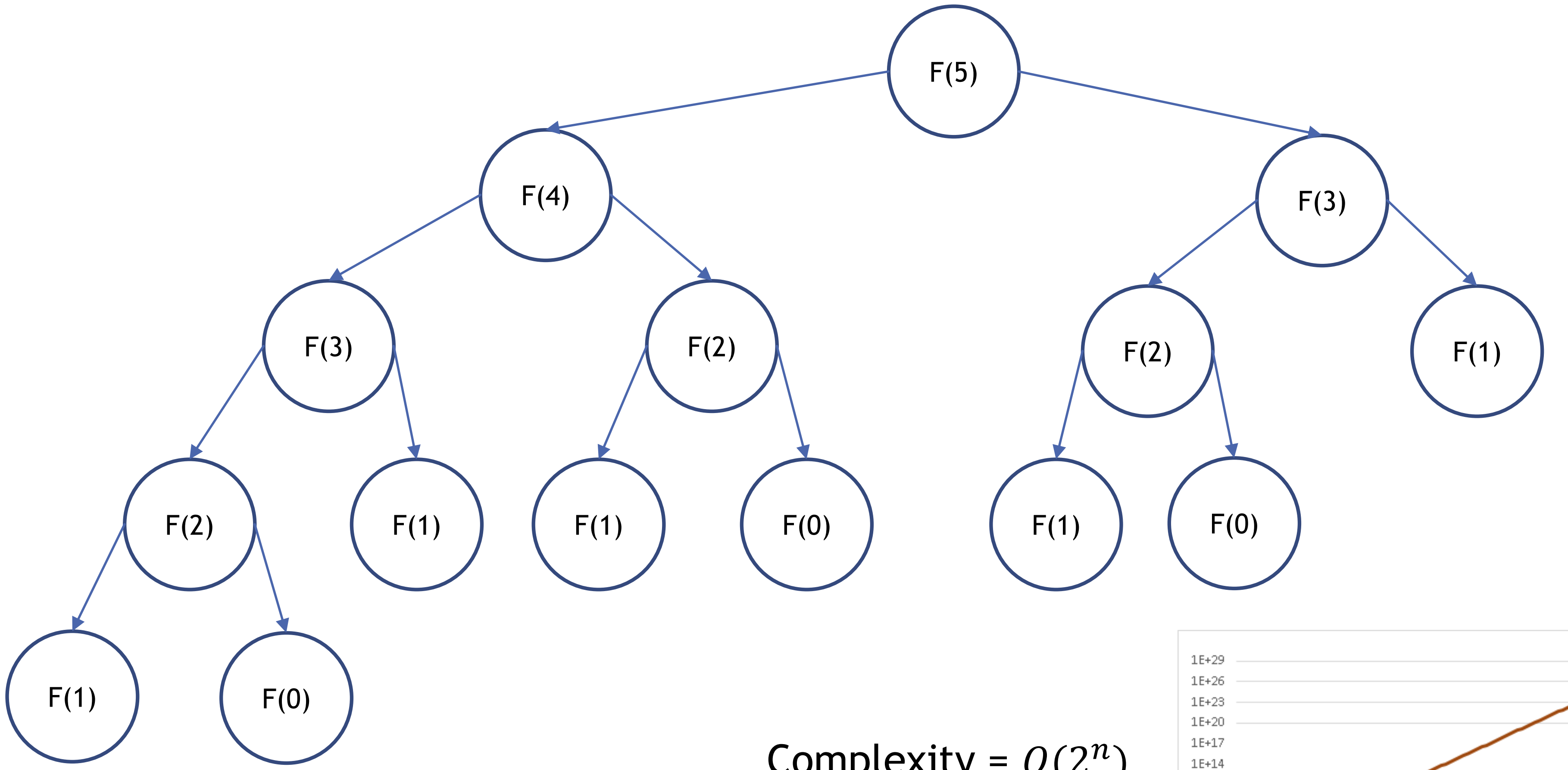
```
function fib(n)
  if n <= 1 return 1
  return fib(n – 1) + fib(n – 2)
```

- Execution time

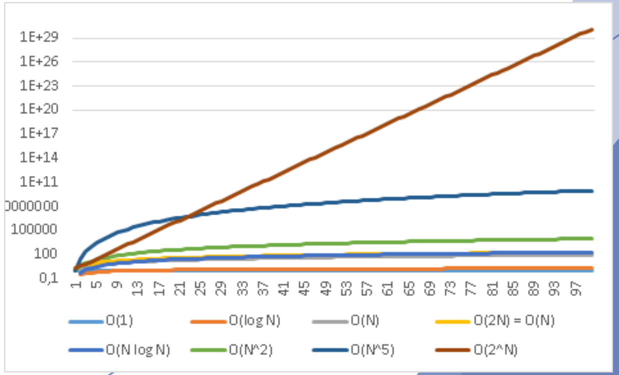
n	Time (ms)
30	4
35	50
40	550
45	5817
>50	Happy watching!

Fibonacci Sequence

Why so slow?



Complexity = $O(2^n)$



Fibonacci Sequence

Why so slow?

“*Did I ever tell you what the definition of insanity is? Insanity is doing the exact... same *** thing... over and over again expecting... things to change...*”

- We are doing the same thing over and over again!!!
  - F(3) computed twice, F(2) computed three times...
- Ideas on how to speed up the process?
  - We could save the result of sub-problems (= *memoization*)!

Fibonacci Sequence

Memoization

- Idea
  - Save the result of sub-problems into a data structure (called *lookup table*)
- New approach
  - Before making a recursive call, we check the lookup table…
    - **Sub-problem never solved yet?** → Make the recursive call
    - **Sub-problem already solved?** → Read the result stored in the table
  - Add the result of the current recursive call to the lookup table

Fibonacci Sequence

Memoized algorithm (recursive)

- Suppose `m` is a map object which maps each value of `fib` that has already been calculated to its result

```
var m := map(0 → 1, 1 → 1)
function fib(n)
  if key n is not in map m
    m[n] := fib(n – 1) + fib(n – 2)
  return m[n]
```

Fibonacci Sequence
Memoization

```
var m := map(0 → 0, 1 → 1)
function fib(n)
  if key n is not in map m
    m[n] := fib(n - 1) + fib(n - 2)
  return m[n]
```

- Time complexity?
  - Accessing the lookup table requires $O(1)$
  - Sub-problems are n
  - Time complexity then is $\boldsymbol{O(n)}$: we compute n times sub-problems that require $O(1)$ time
- Space complexity?
  - $\boldsymbol{O(n)}$ memory space to save the sub-problems results

Fibonacci Sequence

Bottom up algorithm (iterative)

- Other possible approach
  - Build the result of the computation starting from the base case of the recursion
  - At each iteration save the intermediate results to use at the next step
- Same time complexity $O(n)$ as the recursive version, but only $O(1)$ of space complexity

```
function fib(n)
  if n = 0
    return 1
  else
    var previousFib := 1, currentFib := 1
    repeat n – 1 times // loop is skipped if n = 1
      var newFib := previousFib + currentFib
      previousFib := currentFib
      currentFib  := newFib
    return currentFib
```

Floyd Warshall algorithm

All pairs shortest paths problem

Shortest path problem

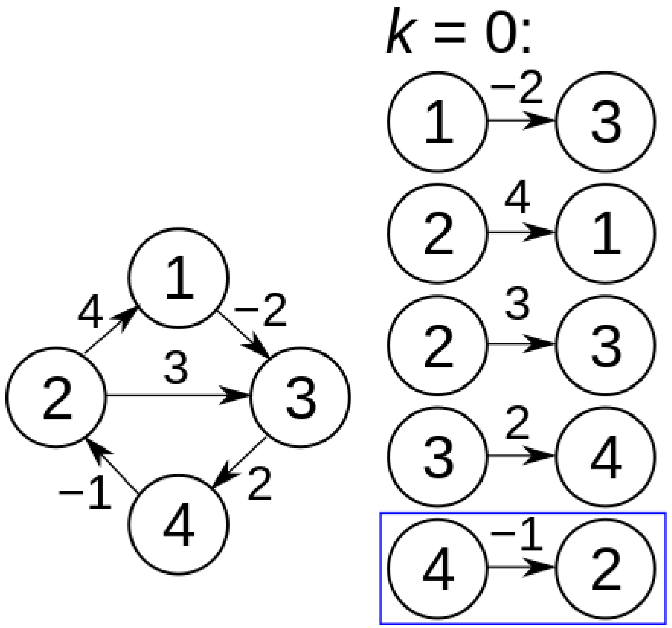
- Remember Dijkstra?
  - Shortest paths starting from ONE node to ALL other nodes
- What if we need shortest paths from multiple source nodes?
  - Floyd-Warshall: shortest paths starting from ALL nodes to ALL nodes
  - Example of dynamic programming

Floyd-Warshall algorithm

- Compares all possible paths through the graph between each pair of vertices
- Every combination of edges is tested
- It works by incrementally improving an estimate on the shortest path between two vertices, until the estimate is optimal
- Based on a recursive idea

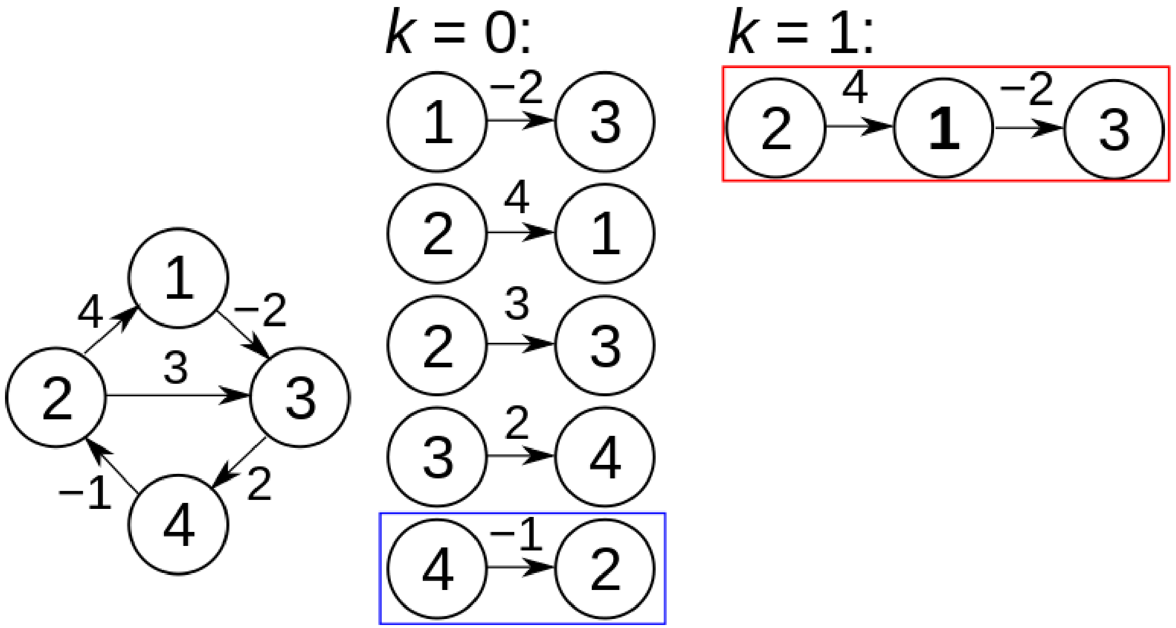
Floyd-Warshall algorithm

- Example



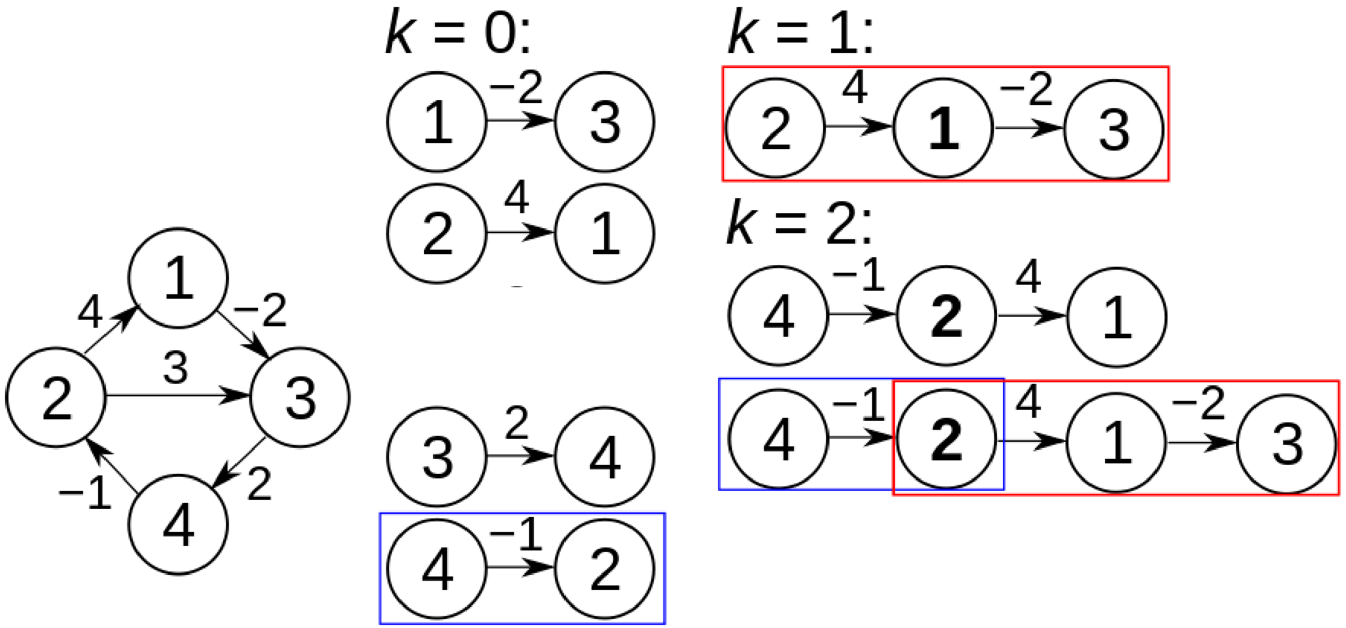
Floyd-Warshall algorithm

- Example



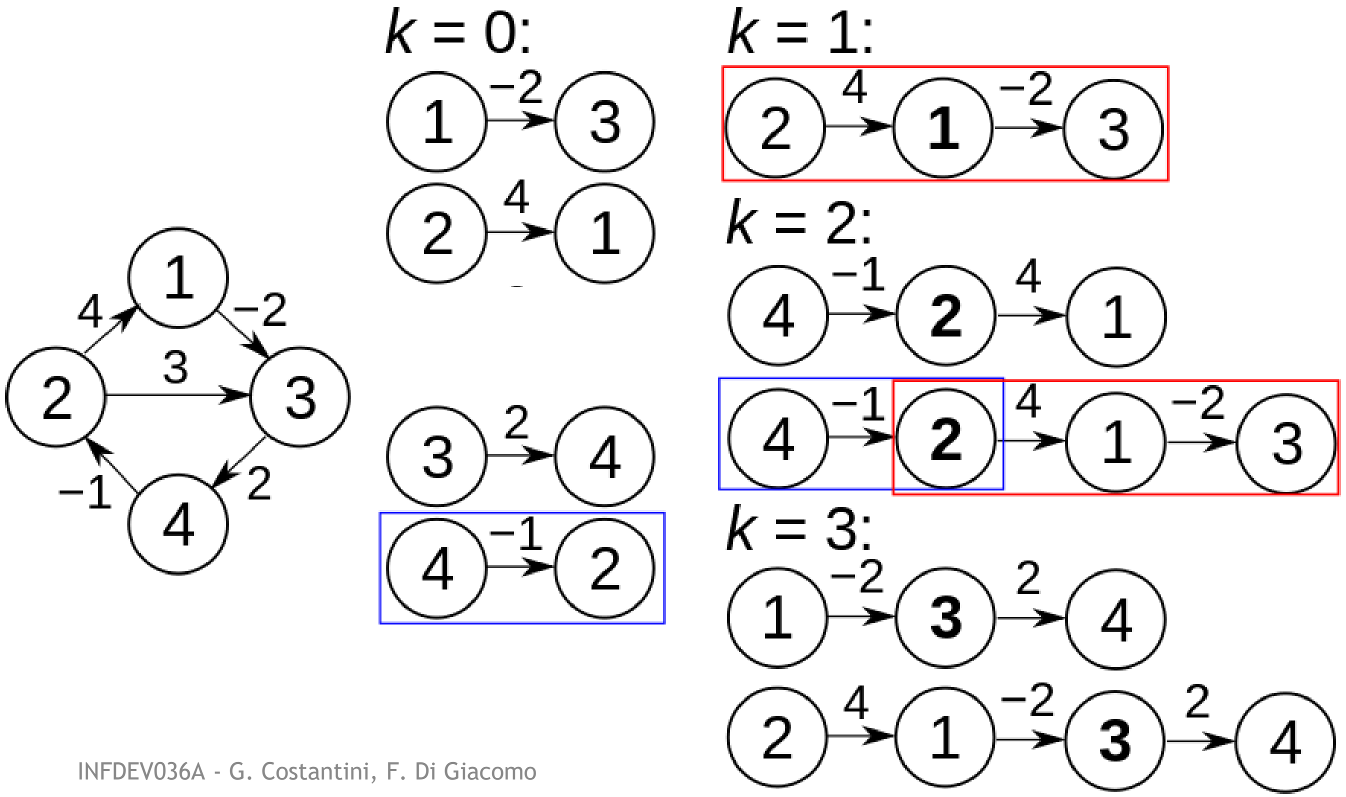
Floyd-Warshall algorithm

- Example



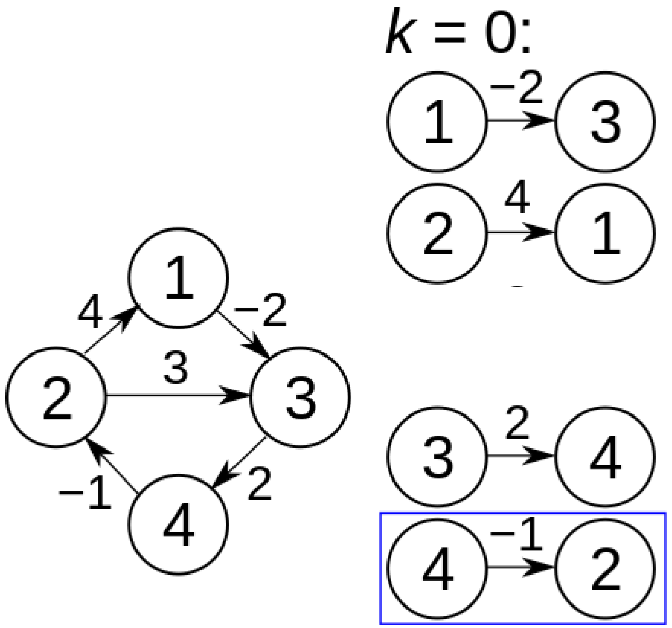
Floyd-Warshall algorithm

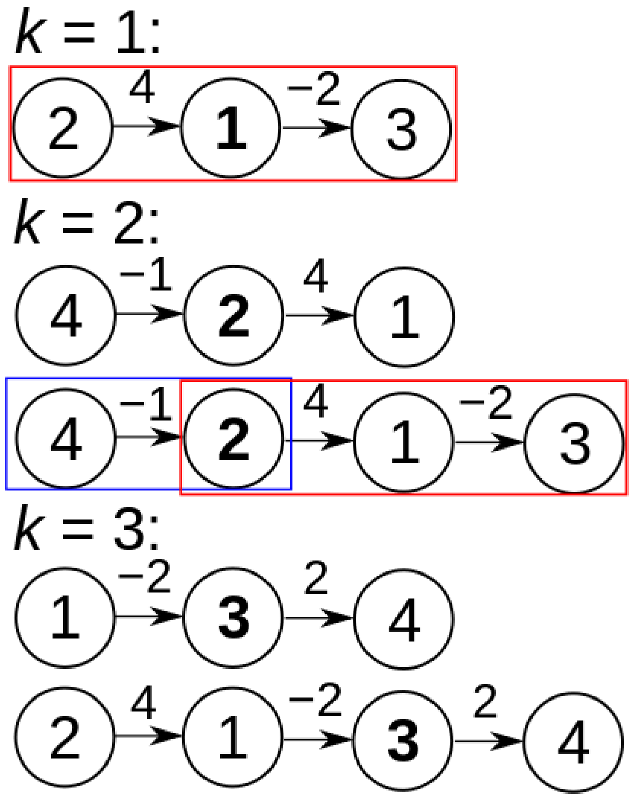
- Example

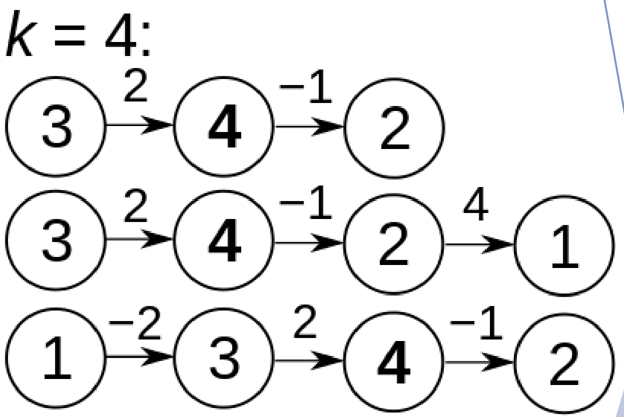


Floyd-Warshall algorithm

- Example







Floyd-Warshall algorithm

- Consider
  - A graph G with vertices V numbered from 1 to N
  - A function $shortestPath(i, j, k)$ which returns the shortest possible path from i to j using vertices only from the set $\{1,2, \ldots, k\}$ as intermediate points along the way

- Goal
  - Find the shortest path from each i to each j using only vertices from 1 to $k + 1$
  - What could this shortest path be?

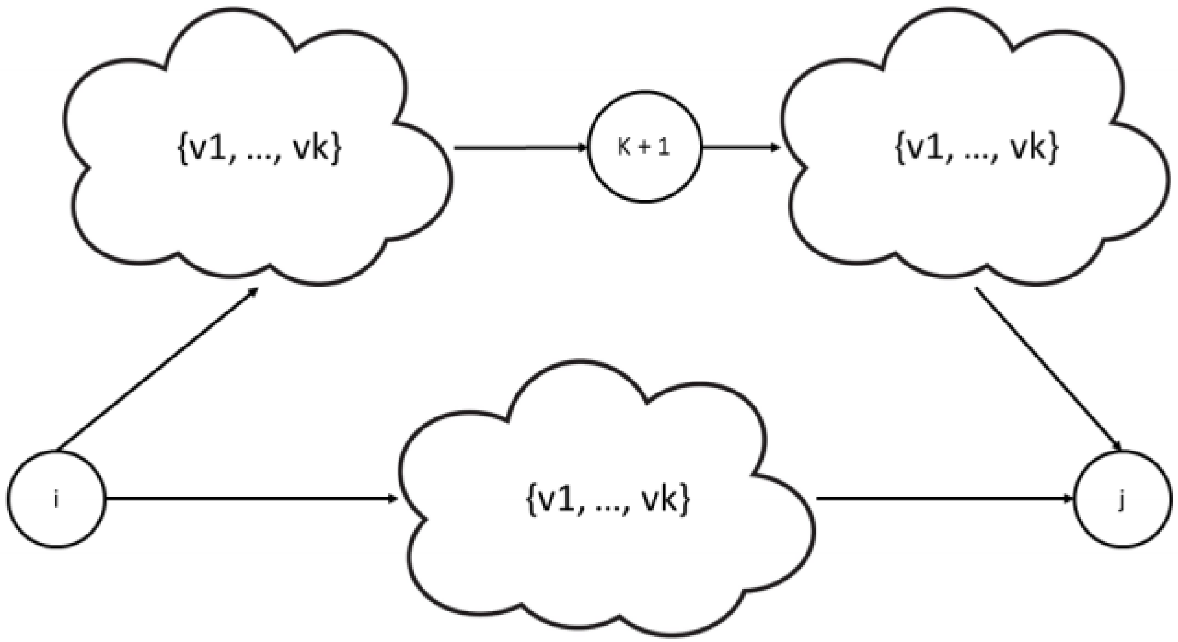
Floyd-Warshall algorithm

- Goal
  - Find the shortest path from each i to each j using only vertices from 1 to $k + 1$
  - What could this shortest path be?
    1. a path that only uses vertices in the set $\{1, \dots, k\}$
    2. a path that goes from i to $k + 1$ and then from $k + 1$ to j

  - In other words… can we improve the shortest path between i and j if we pass through the vertex $k + 1$?

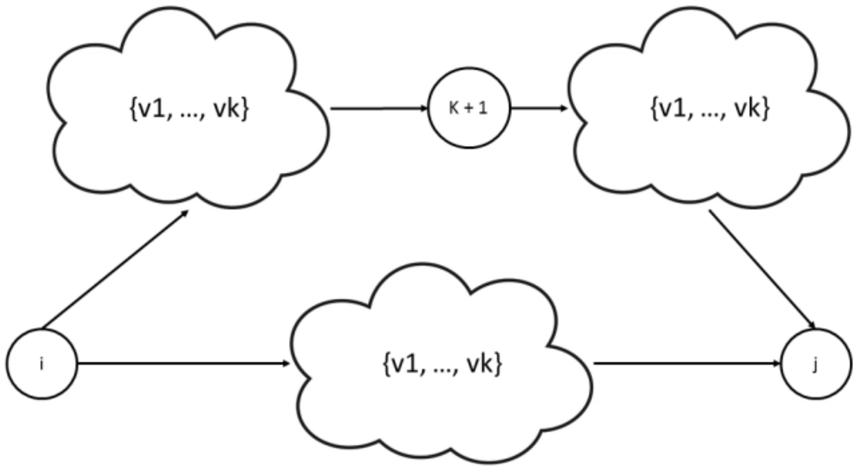
Floyd-Warshall algorithm

- Can we improve the shortest path between i and j if we pass through the vertex $k + 1$?



Floyd-Warshall algorithm

- Can we improve the shortest path between i and j if we pass through the vertex $k+1$?



$$\boldsymbol{shortestPath(i, j, k+1)}$$
$$\boldsymbol{= min(shortestPath(i, j, k), shortestPath(i, k+1, k) + shortestPath(k+1, j, k))}$$

Floyd-Warshall algorithm

- What is the base (easiest) case of $shortestPath(i, j, k)$?
  - $k = 0 \rightarrow$ the path between i and j does not involve any other vertex
  - Length of the path = weight of the edge $w(i, j)$
    - Remember that if two vertices are not directly connected, then $w(i, j) = \infty$

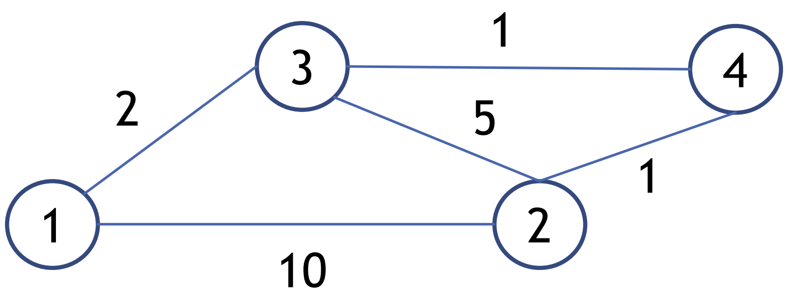
Floyd-Warshall algorithm

- What is the base (easiest) case of $shortestPath(i, j, k)$?
  - $k = 0$ → the path between i and j does not involve any other vertex
  - Length of the path = weight of the edge $w(i, j)$
    - Remember that if two vertices are not directly connected, then $w(i, j) = \infty$

- Combination of the two formulas → core of the Floyd Warshall algorithm

$$\boldsymbol{shortestPath(i, j, 0) = w(i, j)}$$
$$\boldsymbol{shortestPath(i, j, k+1)}$$
$$\boldsymbol{= min(shortestPath(i, j, k), shortestPath(i, k+1, k) + shortestPath(k+1, j, k))}$$

Floyd-Warshall and dynamic programming



We are recomputing the same paths multiple times, for example:

- sp(1,2,4) = min(sp(1,2,3),sp(1,4,3) + sp(4,2,3))
- sp(1,2,3) = min(sp(1,2,2),sp(1,3,2) + sp(3,2,2))
- sp(1,4,3) = min(sp(1,4,2),sp(1,3,2) + sp(3,4,2))

Floyd-Warshall memoization

- How do we avoid to compute the same thing multiple times?
  - Saving the intermediate results into a matrix!
  - This matrix contains the *length of the path from vertex i to vertex j*

- The algorithm uses an iterative bottom up approach
  - Initialization: filling the matrix with the *base case solutions*
    - 0 is the distance between a vertex and itself
    - w(i,j) is the distance between adjacent vertices
    - ∞ is the distance between non-adjacent vertices
  - Iterative part: filling in the matrix by iteratively expanding the set of intermediate vertices to use in the path

Floyd-Warshall algorithm

- Initialization ($k = 0$)
  - $shortestPath(i, j, 0) = w(i, j)$
- Loop (on k)
  - compute $shortestPath(i, j, k)$ for all (i, j) pairs when $k = 1$
    - See if passing for vertex 1 improves some shortest paths
  - compute $shortestPath(i, j, k)$ for all (i, j) pairs when $k = 2$
    - See if passing for vertex 2 improves some shortest paths
  - …
  - compute $shortestPath(i, j, k)$ for all (i, j) pairs when $k = N$
    - See if passing for vertex N improves some shortest paths
    - After this, we have found the shortest path for all (i, j) pairs using **any** intermediate vertices

Floyd-Warshall algorithm

```
dist ← |V| × |V| matrix of minimum distances initialized to ∞
for each vertex v
  dist[v][v] ← 0
for each edge (u,v)
  dist[u][v] ← w(u,v)
for k from 1 to |V|
  for i from 1 to |V|
    for j from 1 to |V|
      if dist[i][j] > dist[i][k] + dist[k][j]
        dist[i][j] ← dist[i][k] + dist[k][j]
      end if
```

Floyd-Warshall algorithm

- Previous code only computed the ***minimum distance*** between pairs of vertices, but not the ***actual minimum path***
- Small changes to the algorithm make it possible to save also the information on the actual path

```
dist ← |V| × |V| matrix of minimum distances initialized to ∞
next ← |V| × |V| matrix of vertex indices initialized to null
for each edge (u,v)
      dist[u][v] ← w(u,v)
      next[u][v] ← v
   for k from 1 to |V|
      for i from 1 to |V|
         for j from 1 to |V|
            if dist[i][k] + dist[k][j] < dist[i][j] then
               dist[i][j] ← dist[i][k] + dist[k][j]
               next[i][j] ← next[i][k]
```

Floyd-Warshall algorithm

- Complexity?
  - Three nested loops
  - Each loop performs $|V|$ iterations
  - $|V| \times |V| \times |V|$ iterations in total → $\boldsymbol{O(|V|^3)}$
    - Or, if we call n the number of nodes ($|V|$), $O(n^3)$

Homework

- Study the slides
- Answer the MC questions on GrandeOmega
- Implement Floyd-Warshall algorithm
- ***Do the sample exams***
  - *Practical assessment* from N@tschool
  - *Written exam* in GO