Week 13: All-Pairs Shortest Paths

Algorithms & Data Structures

Thorben Klabunde

th-kl.ch

December 15, 2025

Agenda

Quiz

Assignment

Remarks on Assignment 12

We still have **quite some ground to cover today** and most exercises relate to the exchange argument and cut property, which we covered in detail in the last two sessions. Therefore, **we will skip a detailed in-class discussion** today.

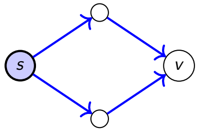
However, the exercises and the introduced notions and terminology (heavy and light edges) are **important**! **Please refer to the Master Solution** and **do not hesitate to reach out if anything should still be unclear!**

All-Pairs Shortest Paths

Recall: Single-Source Shortest Paths (SSSP)

The Problem

Given a source $s \in V$, find the shortest path distances $d(s, v)$ to **all** other vertices $v \in V$.



The Landscape of Algorithms:

Graph Type / Constraint	Algorithm	Runtime
Unweighted	**BFS**	$O(m+n)$ ✓
Weights $c(e) \geq 0$	**Dijkstra**	$O((m+n)\log n)$✓
General Weights	**Bellman-Ford**	$O(nm)$ **!**
DAG (Directed Acyclic)	**Topo Sort + DP**	$O(m+n)$ ✓

Problem Definition: All-Pairs Shortest Paths

Goal

Given directed graph $G = (V, E)$ with edge weights $c : E \to \mathbb{R}$.

Find the shortest path distance $\delta(u, v)$ for **every pair of vertices** $(u, v) \in V \times V$.

Output: An $n \times n$ matrix D where $D_{uv} = \delta(u, v)$.

Edge Cases:

- $\delta(u, v) = \infty$ if v is unreachable from u.
- $\delta(u, v) = -\infty$ if the path touches a negative cycle.

Why do we need this?

Motivation: Pre-computation! Instead of running Dijkstra every time a user asks for a route, we look up the distance in $O(1)$ from our matrix.

Naive Approach: Run SSSP n Times

Idea: Just iterate through every vertex $v \in V$ and run a Single-Source algorithm.

Runtime Analysis for APSP ($n\times$ SSSP):

Weights	**Algorithm**	**Sparse ($m \approx n$)**	**Dense ($m \approx n^2$)**
Unweighted	$n\times$ BFS	$O(n^2)$	$O(n^3)$
Non-negative	$n\times$ Dijkstra	$O(n^2 \log n)$	$O(n^3 \log n)$
General	$n\times$ **Bellman-Ford**	$O(n^2 m) \approx O(n^3)$	$O(n^4)$

The Bottleneck

For general graphs (negative weights allowed), the naive approach is **extremely slow** ($O(n^4)$) on dense graphs!

Question: Can we design a specialized algorithm to beat $O(n^4)$?

Today's Results

Good news: We can beat $n\times$ Bellman-Ford!

New Algorithms

Algorithm	**Runtime**	**Best for**
Floyd-Warshall	$O(n^3)$	Dense graphs
Johnson	$O(n(m+n)\log n)$	Sparse graphs

Comparison:

- Floyd-Warshall: Saves factor of n over Bellman-Ford in dense case!
- Johnson: Same as running Dijkstra n times (even with negative weights!)

Counterintuitive: Johnson achieves Dijkstra-like performance with negative weights! How? Because all-pairs is a harder problem, giving us more flexibility...

DP Attempt 1: Subproblem

Obvious subproblems: d_{uv} = shortest path from u to v

Problem: This leads to infinite recursion!

Solution: Add a parameter to make progress

Subproblem Definition (recall Bellman-Ford)

$d_{uv}^{(m)}$ = weight of shortest path from u to v using **at most** m **edges**

- Now we have a natural notion of "smaller": smaller m
- Eventually we solve $d_{uv}^{(n-1)}$ (at most $n-1$ edges)
- If no negative cycles: simple paths have $\leq n-1$ edges, so $d_{uv}^{(n-1)} = \delta(u, v)$

Bonus: Negative cycle detection!
If $d_{vv}^{(n-1)} < 0$ for some v, then negative cycle exists.

DP Method 1: Recurrence

Question: How to compute $d_{uv}^{(m)}$ from smaller subproblems?

Guess: What is the last edge in the shortest path?

Let the last edge be (x, v) for some vertex x

Recurrence

$$d_{uv}^{(m)} = \min_{x \in V} \left\{ d_{ux}^{(m-1)} + c(x, v) \right\}$$

Interpretation: Try all possible last vertices x, take the minimum

Base case:

$$d_{uv}^{(0)} = \begin{cases} 0 & \text{if } u = v \\ \infty & \text{otherwise} \end{cases}$$

Order of computation: Process all $d^{(0)}$, then all $d^{(1)}$, then all $d^{(2)}$, etc.
(Within each m, can compute (u, v) pairs in any order)

DP Method 1: Pseudocode

```
function AllPairsDP1($G = (V, E, c)$)
    // Base case
    for $u \in V$ do
        $d_{uu} \leftarrow 0$
    for $u \in V, v \in V \setminus \{u\}$ do
        $d_{uv} \leftarrow \infty$

    // DP iteration
    for $m = 1$ to $n - 1$ do
        for $u \in V$ do
            for $v \in V$ do
                for $x \in V$ do
                    if $d_{uv} > d_{ux} + c(x, v)$ then          ▷ Relaxation step
                        $d_{uv} \leftarrow d_{ux} + c(x, v)$
    return $d$
```

Runtime: $O(n) \times O(n) \times O(n) \times O(n) = O(n^4)$, not that great!

Floyd-Warshall

Floyd-Warshall: Different Subproblems

Goal: Remove n factor to get $O(n^3)$ by choosing a different subproblem.

Assume vertices are numbered $1, 2, \ldots, n$

New Subproblem (Floyd-Warshall)

d_{uv}^k = weight of shortest path from u to v using only vertices from $\{1, 2, \ldots, k\}$ as **intermediate** vertices

(Note: u and v themselves can be $> k$; only intermediate vertices restricted)

Goal: Compute d_{uv}^n for all pairs (can use all vertices as intermediates)

Progress: Still n^3 subproblems, but...

Floyd-Warshall: The Key Insight

Question: How to compute d^k from d^{k-1}?

Guess: Is vertex k used in the shortest path from u to v?

Two cases:

1. **Vertex k NOT in path:** Then path uses only $\{1, \ldots, k-1\}$
   Cost: d_{uv}^{k-1}
2. **Vertex k IS in path:** Then path is $u \rightsquigarrow k \rightsquigarrow v$
   Cost: $d_{uk}^{k-1} + d_{kv}^{k-1}$

Recurrence:

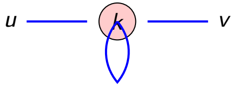
$$d_{uv}^{k} = \min\left\{d_{uv}^{k-1}, \quad d_{uk}^{k-1} + d_{kv}^{k-1}\right\}$$

Only 2 choices instead of n choices! Constant time per subproblem!

Floyd-Warshall: Why Only 2 Choices?

Before: Guessed which vertex comes last $\rightarrow$ n choices
Now: Guess whether vertex k appears at all $\rightarrow$ 2 choices

What if k appears multiple times?



Creates a cycle through k!

When is this useful?

- If cycle cost > 0: wasteful
- If cycle cost $= 0$: doesn't help
- If cycle cost < 0: infinite loop!

Assumption: No negative cycles
Then using k once is optimal!

Your Turn: Complete Floyd-Warshall

Fill in the blanks to complete the algorithm

```
1:  function FloydWarshall($G = (V, E, c)$)
2:      // Base case
3:      for $u, v \in V$ do
4:          if $u = v$ then
5:              $d_{uv} \leftarrow$ ____________
6:          else if $(u, v) \in E$ then
7:              $d_{uv} \leftarrow$ ____________
8:          else
9:              $d_{uv} \leftarrow$ ____________
10:
11:     // DP: gradually allow more intermediate vertices
12:     for $k =$ ______ to ______ do
13:         for $u =$ ______ to ______ do
14:             for $v =$ ______ to ______ do
15:                 if $d_{uv} >$ __________________ then
16:                     $d_{uv} \leftarrow$ __________________
17:     return $d$
```

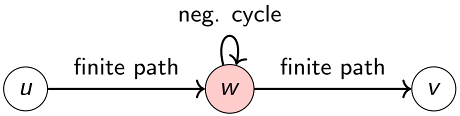
Floyd-Warshall: Handling Negative Cycles

Shortest Path Weight with Negative Cycles

$$\delta(u,v) = \begin{cases} -\infty & \text{if } \exists w : d^n_{uw} < \infty, d^n_{wv} < \infty, \text{ and } d^n_{ww} < 0 \\ d^n_{uv} & \text{otherwise} \end{cases}$$

Interpretation:

- If there exists vertex w on a negative cycle
- AND w is reachable from u (i.e., $d^n_{uw} < \infty$)
- AND v is reachable from w (i.e., $d^n_{wv} < \infty$)
- Then we can loop through the negative cycle infinitely $\rightarrow \delta(u,v) = -\infty$



Johnson

Can We Do Better for Sparse Graphs?

Current best:

- Floyd-Warshall: $O(n^3)$
- For sparse graphs ($m = O(n)$): still $O(n^3)$

Question: Running Dijkstra n times gives $O(n(m+n)\log n)$
For sparse graphs: $O(n^2 \log n)$ — much better than $O(n^3)$!

But... Dijkstra requires non-negative weights.

Crazy idea: What if we could make all edge weights non-negative?
Then we could use Dijkstra even with originally negative weights!

Game-Plan

Johnson's algorithm does exactly this by performing **three steps**:

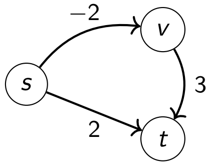
1. **Find a function** $h : V \to \mathbb{R}$ such that $w_h(u, v) = w(u, v) + h(u) - h(v) \geq 0$ for all $u, v \in V$ or determine that a negative-weight cycle exists
2. **Run Dijkstra from every vertex** in the newly weighted graph $G = (V, E, w_h)$ to find $\delta_h(u, v)$
3. **Compute** $\delta(u, v)$ from $\delta_h(u, v)$

We now want to show:

1. *why* this set-up works,
2. *when* we can hope to find such a function h, and
3. *how* we find h

The Naive Attempt at Enforcing Positive Weights Fails

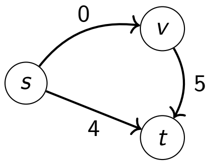
Naive idea: Add constant D to all edges where $D = |\min_{e \in E} c(e)|$



Original costs:

- Via v: $-2 + 3 = 1$ (shortest)
- Direct: 2

Add $D = 2$ to all edges:



New costs:

- Via v: $0 + 5 = 5$
- Direct: 4 (now shortest!)

FAILS! Different paths shifted by different amounts!
2-edge path: $+4$. 1-edge path: $+2$.

Johnson's Insight: Vertex Potentials

Key observation: We need all *s*-*t* paths to shift by the **same** amount!

Johnson's reweighting:
Assign each vertex v a "potential" (or "height") $h(v) \in \mathbb{R}$
Define new edge weights: $\hat{c}(u,v) = c(u,v) + h(u) - h(v)$

Claim: For any path $P = (v_0, v_1, \ldots, v_k)$: $\hat{c}(P) = c(P) + h(s) - h(t)$

Proof: Let $u, v \in V$ be arbitrary and let P be an arbitrary path from u to v:

$$\begin{aligned}
\hat{c}(P) &= \sum_{i=0}^{k-1} \hat{c}(v_i, v_{i+1}) \overset{\text{Def. of } \hat{c}}{=} \sum_{i=0}^{k-1} (c(v_i, v_{i+1}) + h(v_i) - h(v_{i+1})) \\
&= \sum_{i=0}^{k-1} c(v_i, v_{i+1}) + \underbrace{(h(v_0) - h(v_1)) + (h(v_1) - h(v_2)) + \cdots + (h(v_{k-1}) - h(v_k))}_{\text{telescopes to } h(v_0) - h(v_k)} \\
&= c(P) + h(s) - h(t) \qquad \square
\end{aligned}$$

Johnson's Reweighting: The Magic

Consequence of telescoping:

All paths from s to t get shifted by **exactly** $h(s) - h(t)$

$\implies$ Shortest paths are preserved!

Therefore:

- If $\hat{c}(u, v) \geq 0$ for all edges, we can run Dijkstra
- Dijkstra finds shortest paths in $\hat{c}$ graph
- These are also shortest paths in original c graph!
- Just need to convert back: $\delta(u, v) = \hat{\delta}(u, v) - h(u) + h(v)$

Remaining question: How do we find h such that $\hat{c}(u, v) \geq 0$?

Finding h: System of Difference Constraints

Requirement: $\hat{c}(u,v) = c(u,v) + h(u) - h(v) \geq 0$ for all edges (u,v)

Rearrange: $h(v) - h(u) \leq c(u,v)$ for all $(u,v) \in E$

System of Difference Constraints: Find $h : V \to \mathbb{R}$ satisfying:

$$h(v) \leq h(u) + c(u,v) \quad \forall (u,v) \in E$$

We now show that if there are no negative cycles, we can easily find a solution to the system using methods we already know!

Theorem

The system $h(v) - h(u) \leq c(u,v)$ has a solution $\iff$ there exist no negative cycles

Johnson: When does a solution exist?

Theorem: The system $h(v) - h(u) \leq c(u, v)$ has a solution $\iff$ no negative cycles exist

Proof ($\Rightarrow$): By contraposition. Let $C = (v_0, v_1, \ldots, v_k, v_0)$ be a negative cycle and suppose for contradiction that the system was solvable:
We get the following system of constraints:

$$\begin{gathered} h(v_1) - h(v_0) \leq c(v_0, v_1) \\ h(v_2) - h(v_1) \leq c(v_1, v_2) \\ \vdots \\ h(v_0) - h(v_k) \leq c(v_k, v_0) \end{gathered}$$

Adding them up notice that the LHS sums to 0 since for every $v \in C$, $h(v)$ appears as the first (positive) and second (negative) term.

But then: $0 \leq c(C) < 0$, a contradiction, as required. $\square$

Finding h: System of Difference Constraints

($\Rightarrow$) Assume that no negative cycles exist and recall the system we want to solve:

System of Difference Constraints: Find $h : V \to \mathbb{R}$ satisfying:

$$h(v) \leq h(u) + c(u, v) \quad \forall (u, v) \in E$$

Notice that this looks like the **triangle inequality**!

If $h(v) = \delta(s, v)$ for some source s, then triangle inequality guarantees:

$$\delta(s, v) \leq \delta(s, u) + c(u, v)$$

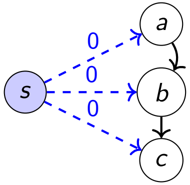
If no negative cycles exist, **shortest path distances satisfy our constraints**!

Johnson's Construction

Problem: Which source s to use? Need to reach all vertices!

Johnson's Trick

1. Add new vertex s to graph
2. Add edges (s, v) with $c(s, v) = 0$ for all $v \in V$
3. Run Bellman-Ford from s
4. Set $h(v) = \delta(s, v)$ for all v



Why it works:

- s can reach all vertices
- No new cycles created
- Triangle inequality holds
- All $h(v) \leq 0$ (paths from s have cost ≤ 0)

Johnson's Algorithm: Putting It Together

1. **Find h:** Add source s, run Bellman-Ford
   - If negative cycle detected: STOP (no solution)
   - Otherwise: $h(v) = \delta(s, v)$ for all v
   - Runtime: $O(nm)$
2. **Reweight edges:** Compute $\hat{c}(u, v) = c(u, v) + h(u) - h(v)$
   - Now all $\hat{c}(u, v) \geq 0$ (triangle inequality!)
   - Runtime: $O(m)$
3. **Run Dijkstra n times:** Once from each vertex
   - Computes $\hat{\delta}(u, v)$ for all pairs
   - Runtime: $n \times O((m + n) \log n) = O(n(m + n) \log n)$
4. **Convert back:** $\delta(u, v) = \hat{\delta}(u, v) - h(u) + h(v)$
   - Runtime: $O(n^2)$

Total runtime: $O(nm + n(m + n) \log n) = O(n(m + n) \log n)$

Matrix Multiplication

Matrix Multiplication Connection

Crazy idea: Shortest paths ≈ matrix multiplication.

Standard Matrix Multiplication ($C = A \times B$):

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{kj}$$

Recall our initial Shortest Path Recurrence: *(attempt 1 before Floyd-Warshall)*

$$d_{ij}^{(m)} = \min_{k=1}^{n} \left\{ d_{ik}^{(m-1)} + w_{kj} \right\}$$

Looking at the indices, these **expressions resemble each other**. But the operations don't match... Or do they?

Choosing the Algebra: Min-Plus

Note: The operations of (naive) matrix multiplication can be performed in **any semiring**.

In particular, we can **define our own semiring algebra** with a **"new" multiplication and addition** operation and apply matrix multiplication.

If the result is meaningful is a different question but we can do so!

	Standard Arithmetic (Field $\mathbb{R}$)	**Min-Plus Semiring** (Tropical Semiring)
Set S	$\mathbb{R}$	$\mathbb{R} \cup \{\infty\}$
Addition ($\oplus$)	$+$	min
Multiplication ($\otimes$)	$\times$	$+$
Add. Identity (0)	0	∞ (since $\min(x, \infty) = x$)
Mult. Identity (1)	1	0 (since $x + 0 = x$)

Matrix Multiplication $=$ APSP

1. The Weight Matrix (W):

$$W_{kj} = \text{weight of edge } (k \to j)$$

2. Distance Matrix ($D^{(m-1)}$):

$$D_{ik}^{(m-1)} = \text{dist } (i \to k) \text{ using } \leq m-1 \text{ edges}$$

3. The Product Calculation: Let's compute the entry (i,j) of $D^{(m-1)} \otimes W$ (*here* $\otimes$ *as matr. mult*):

$$(D^{(m-1)} \otimes W)_{ij} = \bigoplus_{k=1}^{n} \left(D_{ik}^{(m-1)} \otimes W_{kj} \right)$$

Substitute our Semiring definitions ($\oplus \to \min$, $\otimes \to +$):

$$= \min_{k=1}^{n} \left(\underbrace{D_{ik}^{(m-1)}}_{\text{Path } i \to k} + \underbrace{W_{kj}}_{\text{Edge } k \to j} \right)$$

The Connection

This formula is exactly our DP recurrence: $d_{ij}^{(m)} = \min_k (d_{ik}^{(m-1)} + w_{kj})$

Therefore, one step of Shortest Path is one Matrix Multiplication: $\mathbf{D}^{(\mathbf{m})} = \mathbf{D}^{(\mathbf{m-1})} \otimes \mathbf{W}$

Concrete Example: Computing One Cell

Let's compute entry $[\mathbf{2}, \mathbf{3}]$ of the product $D \otimes W$.

We need **Row 2** of Left Matrix and **Col 3** of Right Matrix.

$$\begin{pmatrix} \cdot & \cdot & \cdot \\ \infty & \mathbf{0} & \mathbf{2} \\ \cdot & \cdot & \cdot \end{pmatrix} \otimes \begin{pmatrix} \cdot & \cdot & \mathbf{8} \\ \cdot & \cdot & \mathbf{1} \\ \cdot & \cdot & \mathbf{0} \end{pmatrix}$$

Abstract (Semiring)

Formula: $\bigoplus(Row_k \otimes Col_k)$

$k = 1$: $\infty \otimes 8$

$k = 2$: $\oplus \quad (0 \otimes 1)$

$k = 3$: $\oplus \quad (2 \otimes 0)$

Result: $(\infty \otimes 8) \oplus (0 \otimes 1) \oplus (2 \otimes 0)$

Concrete (Intuition)

Meaning: $\min(Row_k + Col_k)$

$k = 1$: $\infty + 8 = \infty$

$k = 2$: $\min(0 + 1) = 1$

$k = 3$: $\min(2 + 0) = 2$

Result: $\min(\infty, 1, 2) = \mathbf{1}$

The shortest path from node 2 to 3 going through an intermediate node k is length 1.

APSP via Iterative Squaring

Notice: To compute APSP, we simply need to compute the n-th power! We can do so efficiently using iterative squaring:

$$
\begin{aligned}
W^{\otimes 1} &= W \\
W^{\otimes 2} &= W^{\otimes 1} \otimes W^{\otimes 1} \\
W^{\otimes 4} &= W^{\otimes 2} \otimes W^{\otimes 2} \\
W^{\otimes 8} &= W^{\otimes 4} \otimes W^{\otimes 4} \\
&\vdots
\end{aligned}
$$

Only $\log n$ **multiplications** needed to reach $W^{\otimes n}$!

Runtime: $O(\log n) \times O(n^3) = O(n^3 \log n)$. Not yet better than Floyd-Warshall...

Can we use one of the more efficient matrix multiplication algorithms to beat Floyd-Warshall? For instance, **Strassen's algorithm**?

Why Not Faster Matrix Multiplication?

Bad news: Strassen ($O(n^{2.807})$) requires subtraction but min has no inverse:

- Given $\min(a, b) = c$, cannot recover a and b
- The min-plus structure is a **semiring**, not a ring
- No "minus" operation exists

Conclusion: For all-pair shortest paths, we do not know a better way to perform matrix multiplication than in $O(n^3)$ yielding $O(n^3 \log n)$ through iterative squaring.

However, there IS a related problem where fast matrix multiplication helps!

All-Pair Reachability (Transitive Closure)

Transitive Closure Problem

Input: Directed graph $G = (V, E)$ with adjacency matrix A. Output: For all pairs (i, j), does there exist **any** path from i to j?

$$T[i, j] = 1 \iff \exists \text{ path } i \rightsquigarrow j$$

Algebra: Boolean Semiring

- **Set:** $\{0, 1\}$
- **Addition ($\oplus$):** $\vee$ (OR) $\rightarrow$ "Is there a path via neighbor 1 OR 2?"
- **Multiplication ($\otimes$):** $\wedge$ (AND) $\rightarrow$ "Step to k AND then $k \rightarrow j$?"

Note: Self-Loop Trick

Add Self-Loops to Allow Waiting

Idea: Add a self-loop to every node ($A' = A + I$).

- Meaning: You can "wait" at a node for a step.
- A path of length 1 ($u \to v$) becomes a path of length 2 ($u \to v \to v$).

Result: $(A + I)^{n-1}$ captures all paths of length $\leq n - 1$.

Example: Graph $1 \to 2 \to 3$. Can 1 reach 2 in ≤ 2 steps?

Without Self-Loops (A^2)

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \to A^2 = \begin{pmatrix} 0 & \mathbf{0} & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Missed $1 \to 2$! It found only length exactly 2 ($1 \to 3$).

With Loops ($(A + I)^2$)

$$A' = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \to (A')^2 = \begin{pmatrix} 1 & \mathbf{1} & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Found $1 \to 2$! Logic: $1 \to 2 \to \mathbf{2}$ (Wait).

Can we use Fast Matrix Multiplication?

Standard algorithms like Strassen ($O(n^{2.81})$) require a **Ring** (subtraction).

The Workaround:

1. Treat Boolean matrices as **Integer** matrices ($0, 1 \in \mathbb{Z}$).
2. Compute product using Fast MatMul over Integers.
3. Map any result > 0 to 1 (Boolean TRUE).

Result: Transitive Closure is solvable in roughly $O(n^{2.37})$ using the fastest known matrix multiplication algorithm. This is the fastest algorithm we know to solve the transitive closure!

Path Counting (Standard Arithmetic)

The Question

How many distinct walks of **exactly length** k exist from node i to node j?

The Algebra: Use standard arithmetic!

- $\oplus \to +$ (Addition sums up the options)
- $\otimes \to \times$ (Multiplication combines steps in a sequence)

Theorem

If A is the adjacency matrix (0 or 1), then:

$$(A^k)_{ij} = \text{Number of walks from } i \text{ to } j \text{ with length } k$$

Application (Triangle Counting): The number of triangles in a graph is Trace$(A^3)/3$.

- A^3_{ii} counts paths $i \to \cdots \to \cdots \to i$ (cycles of length 3).
- Divide by 3 because each triangle is counted once for each vertex.

Questions?

Additional Practice

Theory Task T4.

/ 12 P

Assume that there are n towns $T_1, \dots, T_n$ in the country Examistan. For each pair of distinct towns T_i and T_j, there is exactly one road from T_i to T_j. All of the roads in Examistan are one-way. This implies that there is always a road from T_i to T_j and another road from T_j to T_i. Each road has a nonnegative integer cost that you need to pay to use this road.

For simplicity you can assume that each town T_i is represented by its index i.

FS20

In the following subtasks b) and c), you can assume that the directed graph in a) is represented by a data structure that allows you to traverse the direct successors and direct predecessors of a vertex u in time $\mathcal{O}(\deg_+(u))$ and $\mathcal{O}(\deg_-(u))$ respectively, where $\deg_-(u)$ is the in-degree of vertex u and $\deg_+(u)$ is the out-degree of vertex u.

/ 6 P

b) Due to the epidemiological situation in Examistan, the authorities decided to reduce the number of trips between different towns. Now the only way to get from one town to another is to use the roads. Moreover, if you want to travel from town T_i to the other town T_j, you must visit a test center during your trip (in T_i or T_j or elsewhere with a detour). Since test centers are expensive, there are only $k < n$ of them, and they are located only in the first k towns $T_1, \ldots, T_k$ (i.e., one test center in each of these towns).

Assume that you need to fill the table of minimal costs required to travel between all pairs of towns, which takes into account the new rules of travelling. Provide an as efficient as possible algorithm that takes as input a graph G from task a) and a number k, and outputs a table C such that $C[i][j]$ is the minimal total cost of roads that one can use to get from T_i to T_j while also visiting a test center. You can assume that for all $1 \leq i \leq n$, $C[i][i] = 0$.

What is the running time of your algorithm in concise Θ-notation in terms of n and k? Justify your answer.