Datenstrukturen und Algorithmen

Vorlesung 18: All-Pairs Shortest Path

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

http://www-i2.informatik.rwth-aachen.de/i2/dsal12/

26. Juni 2012

RWTH AACHEN UNIVERSITY

Übersicht

Übersicht

Binäre Relationen

Binäre Relation

Eine binäre Relation über einer Menge S ist eine Teilmenge von $R \subseteq S \times S = S^2$ (daher binär).

Binäre Relationen

Binäre Relation

Eine binäre Relation über einer Menge S ist eine Teilmenge von $R \subseteq S \times S = S^2$ (daher binär). Für jedes Paar $(u, v) \in S^2$ gibt R an, ob es enthalten ist (z. B. weil es eine bestimmte Eigenschaft erfüllt).

Binäre Relationen

Binäre Relation

Eine binäre Relation über einer Menge S ist eine Teilmenge von $R \subseteq S \times S = S^2$ (daher binär). Für jedes Paar $(u, v) \in S^2$ gibt R an, ob es enthalten ist (z. B. weil es eine bestimmte Eigenschaft erfüllt).

- Schreibweisen: uRv, $uv \in R$, $R(u, v)$, `R[u,v] = true`, ...

Binäre Relationen

Binäre Relation

Eine binäre Relation über einer Menge S ist eine Teilmenge von $R \subseteq S \times S = S^2$ (daher binär). Für jedes Paar $(u, v) \in S^2$ gibt R an, ob es enthalten ist (z. B. weil es eine bestimmte Eigenschaft erfüllt).

- Schreibweisen: uRv, $uv \in R$, $R(u, v)$, `R[u,v] = true`, ...

Reflexivität, Transitivität

Eine Relation R ist reflexiv, wenn $uu \in R$ für alle $u \in S$.
Sie heißt transitiv, wenn mit $uv \in R$ und $vw \in R$ auch $uw \in R$ ist.

Binäre Relationen

Binäre Relation

Eine binäre Relation über einer Menge S ist eine Teilmenge von $R \subseteq S \times S = S^2$ (daher binär). Für jedes Paar $(u, v) \in S^2$ gibt R an, ob es enthalten ist (z. B. weil es eine bestimmte Eigenschaft erfüllt).

- Schreibweisen: uRv, $uv \in R$, $R(u, v)$, `R[u,v] = true`, ...

Reflexivität, Transitivität

Eine Relation R ist reflexiv, wenn $uu \in R$ für alle $u \in S$.
Sie heißt transitiv, wenn mit $uv \in R$ und $vw \in R$ auch $uw \in R$ ist.

Transitive Hülle

Die (Reflexiv-)Transitive Hülle (transitiver Abschluss) R^* der Relation R ist die kleinste Erweiterung (Obermenge) $R \subseteq R^* \subseteq S^2$, so dass R^* reflexiv und transitiv ist.

Transitive Hülle bei Graphen

Betrachte die folgenden Fragen für alle Paare von Knoten in einem Graph:

Transitive Hülle bei Graphen

Betrachte die folgenden Fragen für alle Paare von Knoten in einem Graph:

- Gibt es einen Pfad von Knoten u nach v?

Transitive Hülle bei Graphen

Betrachte die folgenden Fragen für alle Paare von Knoten in einem Graph:

- Gibt es einen Pfad von Knoten u nach v?
- Was ist der kürzeste Pfad von u nach v?

Transitive Hülle bei Graphen

Betrachte die folgenden Fragen für alle Paare von Knoten in einem Graph:

- Gibt es einen Pfad von Knoten u nach v?
- Was ist der kürzeste Pfad von u nach v?

Transitive Hülle eines Graphen

- Betrachte als Menge S die Menge der Knoten eines Graphen G, sowie
- als geordnete Paare in R die Kanten in G.

Transitive Hülle bei Graphen

Betrachte die folgenden Fragen für alle Paare von Knoten in einem Graph:

- Gibt es einen Pfad von Knoten u nach v?
- Was ist der kürzeste Pfad von u nach v?

Transitive Hülle eines Graphen

- Betrachte als Menge S die Menge der Knoten eines Graphen G, sowie
- als geordnete Paare in R die Kanten in G.

Für die transitive Hülle gilt dann: $uv \in R^*$ gdw. es gibt einen Pfad von u nach v.

Transitive Hülle bei Graphen

Betrachte die folgenden Fragen für alle Paare von Knoten in einem Graph:

- Gibt es einen Pfad von Knoten u nach v?
- Was ist der kürzeste Pfad von u nach v?

Transitive Hülle eines Graphen

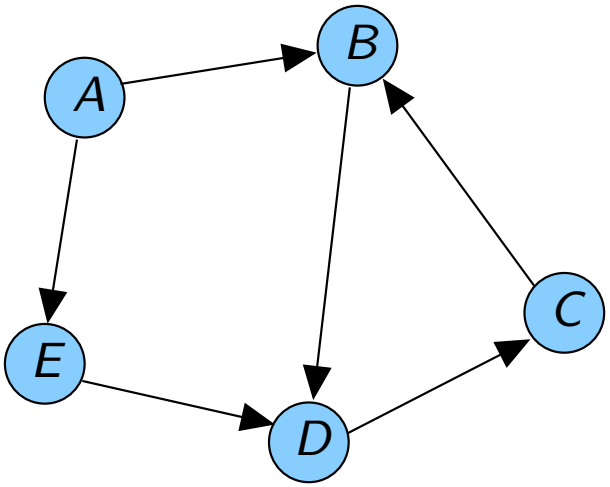
- Betrachte als Menge S die Menge der Knoten eines Graphen G, sowie
- als geordnete Paare in R die Kanten in G.

Für die transitive Hülle gilt dann: $uv \in R^*$ gdw. es gibt einen Pfad von u nach v.

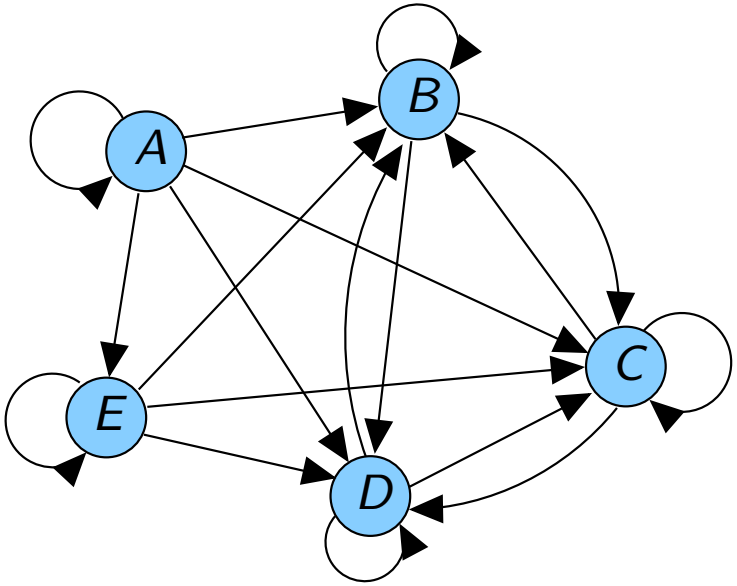
Wie berechnet man nun die transitive Hülle einer binären Relation?

Transitive Hülle – Beispiel

$$R = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$ und die transitive Hülle $R^* = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}$



Binäre Relation R



Transitive Hülle R^*

Finden der transitiven Hülle mittels Tiefensuche (I)

Idee: Teste von jedem Knoten `i` aus mit DFS die Erreichbarkeit.

Finden der transitiven Hülle mittels Tiefensuche (I)

Idee: Teste von jedem Knoten `i` aus mit DFS die Erreichbarkeit.

- Setze `R[i,j] = true`, wenn bei der Tiefensuche von Knoten s_i aus der Knoten s_j gefunden wird.

Finden der transitiven Hülle mittels Tiefensuche (I)

Idee: Teste von jedem Knoten `i` aus mit DFS die Erreichbarkeit.

- Setze `R[i,j] = true`, wenn bei der Tiefensuche von Knoten s_i aus der Knoten s_j gefunden wird.
  (Dadurch füllen wir R^* sozusagen zeilenweise).

Finden der transitiven Hülle mittels Tiefensuche (I)

Idee: Teste von jedem Knoten `i` aus mit DFS die Erreichbarkeit.

- Setze `R[i,j] = true`, wenn bei der Tiefensuche von Knoten s_i aus der Knoten s_j gefunden wird.
  (Dadurch füllen wir R^* sozusagen zeilenweise).

$\Rightarrow$ Bei Adjazenzlistendarstellung erhält man eine Zeitkomplexität von $\Theta(n \cdot (n + m)) \approx \Theta(n \cdot m)$ im Worst-Case.

Finden der transitiven Hülle mittels Tiefensuche (I)

Idee: Teste von jedem Knoten `i` aus mit DFS die Erreichbarkeit.

- Setze `R[i,j] = true`, wenn bei der Tiefensuche von Knoten s_i aus der Knoten s_j gefunden wird.
  (Dadurch füllen wir R^* sozusagen zeilenweise).

$\Rightarrow$ Bei Adjazenzlistendarstellung erhält man eine Zeitkomplexität von $\Theta(n \cdot (n + m)) \approx \Theta(n \cdot m)$ im Worst-Case.

- Man kann direkt `R[k,j] = true` für alle s_k auf dem Pfad von s_i nach s_j (die auf dem Stack liegen) setzen.

Finden der transitiven Hülle mittels Tiefensuche (I)

Idee: Teste von jedem Knoten `i` aus mit DFS die Erreichbarkeit.

- Setze `R[i,j] = true`, wenn bei der Tiefensuche von Knoten s_i aus der Knoten s_j gefunden wird.
  (Dadurch füllen wir R^* sozusagen zeilenweise).

$\Rightarrow$ Bei Adjazenzlistendarstellung erhält man eine Zeitkomplexität von $\Theta(n \cdot (n+m)) \approx \Theta(n \cdot m)$ im Worst-Case.

- Man kann direkt `R[k,j] = true` für alle s_k auf dem Pfad von s_i nach s_j (die auf dem Stack liegen) setzen.
  Das verbessert die Worst-Case-Laufzeit jedoch nicht.

Finden der transitiven Hülle mittels Tiefensuche (I)

Idee: Teste von jedem Knoten `i` aus mit DFS die Erreichbarkeit.

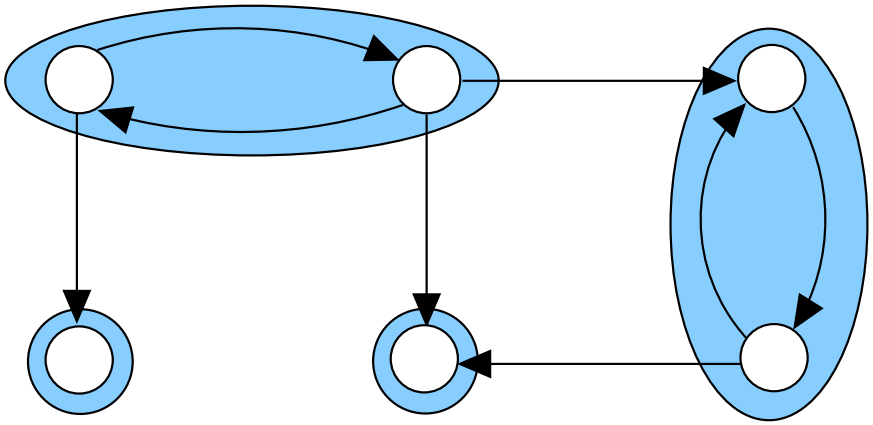
- Setze `R[i,j] = true`, wenn bei der Tiefensuche von Knoten s_i aus der Knoten s_j gefunden wird.
  (Dadurch füllen wir R^* sozusagen zeilenweise).

$\Rightarrow$ Bei Adjazenzlistendarstellung erhält man eine Zeitkomplexität von $\Theta(n \cdot (n + m)) \approx \Theta(n \cdot m)$ im Worst-Case.

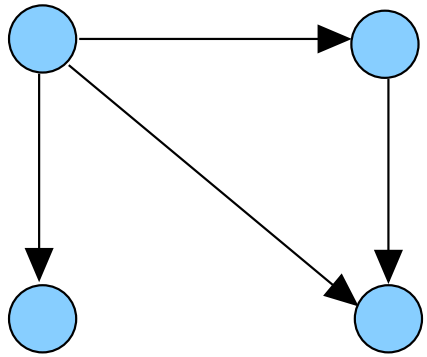
- Man kann direkt `R[k,j] = true` für alle s_k auf dem Pfad von s_i nach s_j (die auf dem Stack liegen) setzen.
  Das verbessert die Worst-Case-Laufzeit jedoch nicht.

Weitere Verbesserung: Verwende den Kondensationsgraph von G:

Erinnerung: Kondensationsgraph – Beispiel



starke Zusammenhangskomponenten



Ein nicht-zusammenhängender Digraph und seine Kondensation.

Finden der transitiven Hülle mittels Tiefensuche (II)

Finden der transitiven Hülle mittels Tiefensuche (II)

1. Bestimme die starken Komponenten von G.

Finden der transitiven Hülle mittels Tiefensuche (II)

1. Bestimme die starken Komponenten von G. $\Theta(n+m)$

Finden der transitiven Hülle mittels Tiefensuche (II)

1. Bestimme die starken Komponenten von G. $\Theta(n+m)$
2. Finde die Erreichbarkeitsrelation des Kondensationsgraphen $G{\downarrow}$.

Finden der transitiven Hülle mittels Tiefensuche (II)

1. Bestimme die starken Komponenten von G. $\Theta(n+m)$
2. Finde die Erreichbarkeitsrelation des Kondensationsgraphen $G{\downarrow}$. $\Theta(\hat{n} \cdot \hat{m})$

Finden der transitiven Hülle mittels Tiefensuche (II)

1. Bestimme die starken Komponenten von G. $\Theta(n+m)$
2. Finde die Erreichbarkeitsrelation des Kondensationsgraphen $G{\downarrow}$. $\Theta(\hat{n} \cdot \hat{m})$
3. Erweitere die Erreichbarkeitsrelation für $G{\downarrow}$ auf ganz G, indem alle Knoten in $G{\downarrow}$ durch die jeweiligen aus G, die auf diesen Knoten in $G{\downarrow}$ kollabiert wurden.

Finden der transitiven Hülle mittels Tiefensuche (II)

1. Bestimme die starken Komponenten von G. $\Theta(n+m)$
2. Finde die Erreichbarkeitsrelation des Kondensationsgraphen $G{\downarrow}$. $\Theta(\hat{n} \cdot \hat{m})$
3. Erweitere die Erreichbarkeitsrelation für $G{\downarrow}$ auf ganz G, indem alle Knoten in $G{\downarrow}$ durch die jeweiligen aus G, die auf diesen Knoten in $G{\downarrow}$ kollabiert wurden. $O(n^2)$

Finden der transitiven Hülle mittels Tiefensuche (II)

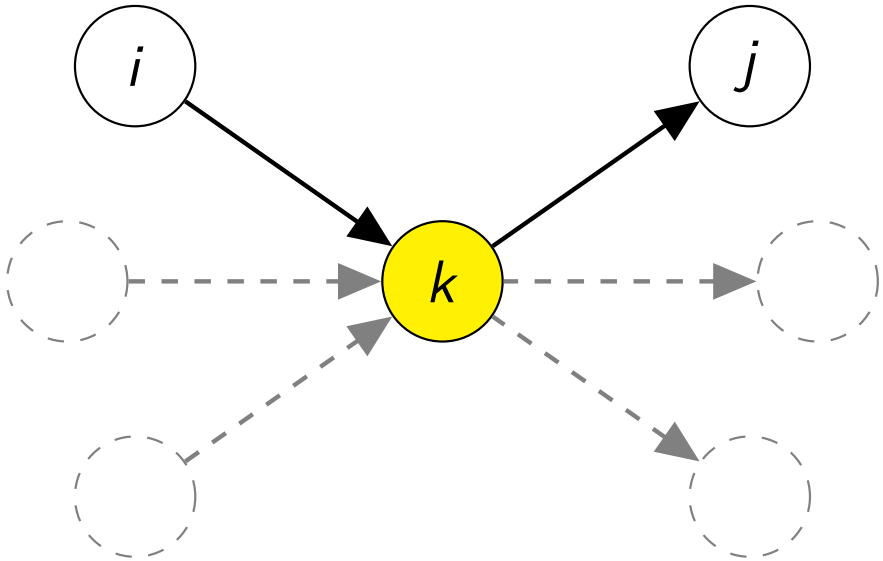
1. Bestimme die starken Komponenten von G. $\Theta(n+m)$
2. Finde die Erreichbarkeitsrelation des Kondensationsgraphen $G{\downarrow}$. $\Theta(\hat{n} \cdot \hat{m})$
3. Erweitere die Erreichbarkeitsrelation für $G{\downarrow}$ auf ganz G, indem alle Knoten in $G{\downarrow}$ durch die jeweiligen aus G, die auf diesen Knoten in $G{\downarrow}$ kollabiert wurden. $O(n^2)$

- Es gibt – insbesondere für Adjazenzmatrizen – eine andere Möglichkeit: Den Algorithmus von Warshall.

Algorithmus von Warshall – Idee (I)

Wir betrachten zunächst den einfachsten Fall:

- Aus `R[i,k]` und `R[k,j]` folgt die Erreichbarkeit `R[i,j] = true`.

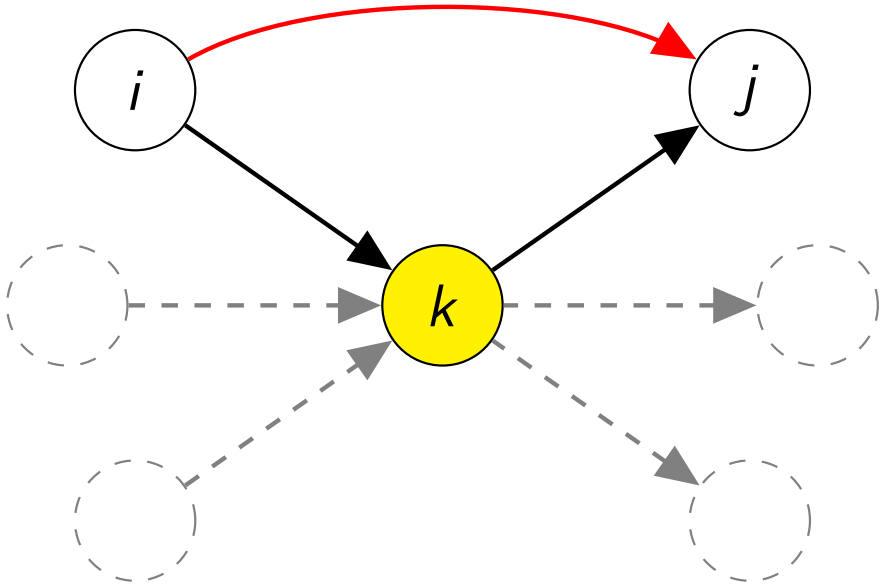



```
1 Wähle einen Knoten k ∈ V;
2   foreach (eingehende Kante (i, k) ∈ E)
3     foreach (ausgehende Kante (k, j) ∈ E)
4       Füge (i, j) zu E hinzu.
```

Algorithmus von Warshall – Idee (I)

Wir betrachten zunächst den einfachsten Fall:

- Aus `R[i,k]` und `R[k,j]` folgt die Erreichbarkeit `R[i,j] = true`.

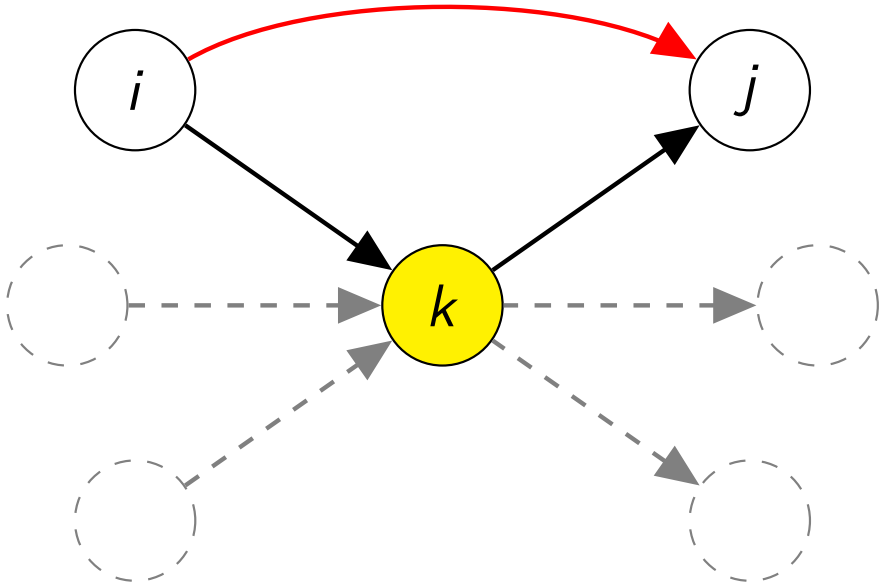



```
1 Wähle einen Knoten k ∈ V;
2   foreach (eingehende Kante (i, k) ∈ E)
3     foreach (ausgehende Kante (k, j) ∈ E)
4       Füge (i, j) zu E hinzu.
```

Algorithmus von Warshall – Idee (I)

Wir betrachten zunächst den einfachsten Fall:

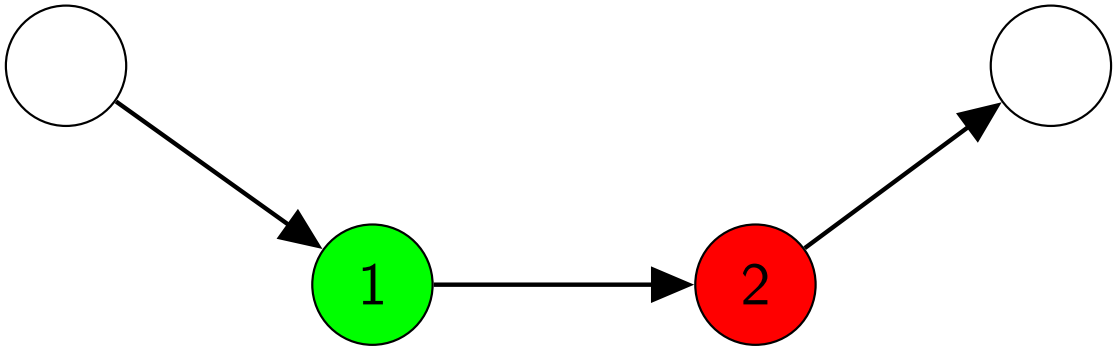
- Aus `R[i,k]` und `R[k,j]` folgt die Erreichbarkeit `R[i,j] = true`.




```
1 foreach (k ∈ V)
2   foreach (eingehende Kante (i, k) ∈ E)
3     foreach (ausgehende Kante (k, j) ∈ E)
4       Füge (i, j) zu E hinzu.
```

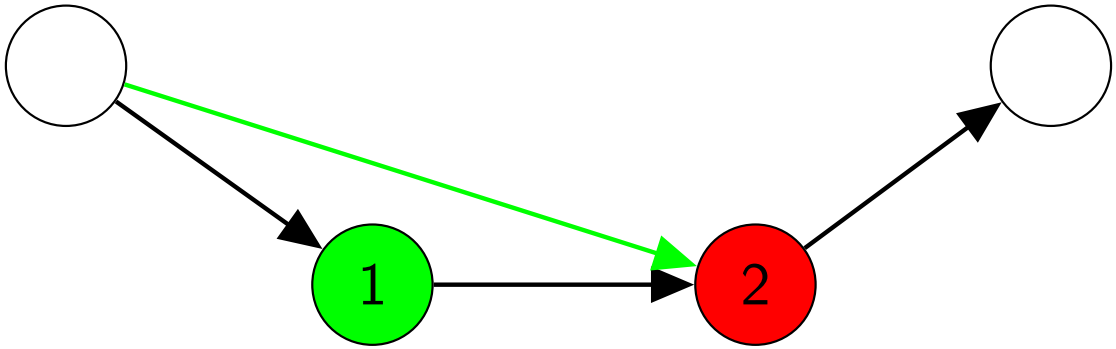
Algorithmus von Warshall – Idee (II)

Das reicht bereits aus, um längere Pfade zu berücksichtigen:



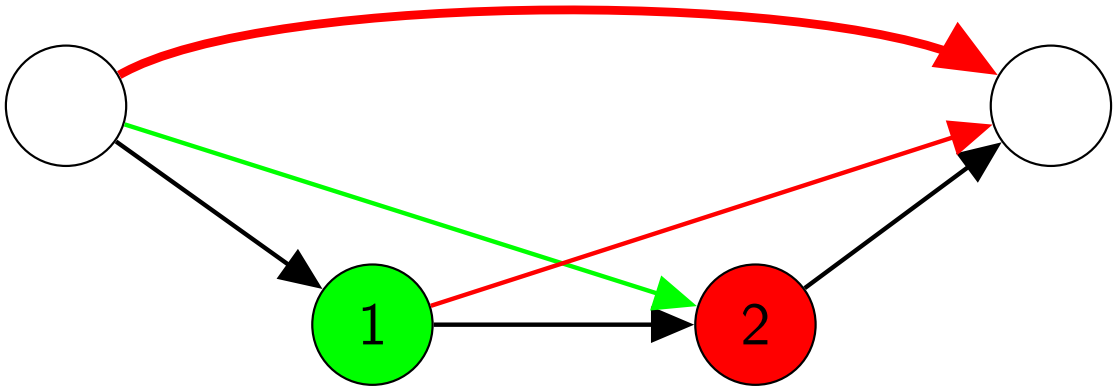
Algorithmus von Warshall – Idee (II)

Das reicht bereits aus, um längere Pfade zu berücksichtigen:



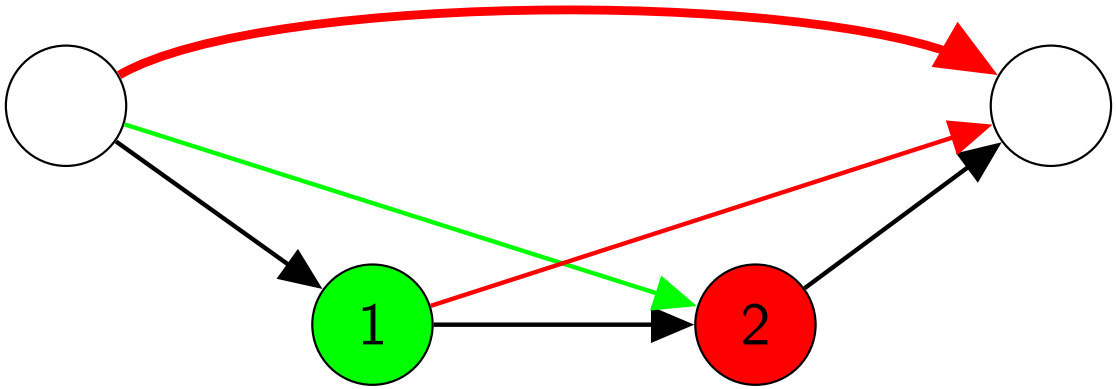
Algorithmus von Warshall – Idee (II)

Das reicht bereits aus, um längere Pfade zu berücksichtigen:

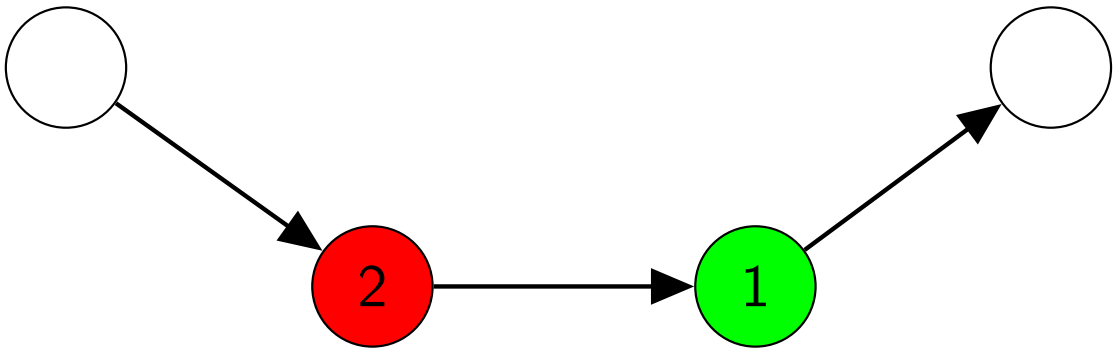


Algorithmus von Warshall – Idee (II)

Das reicht bereits aus, um längere Pfade zu berücksichtigen:

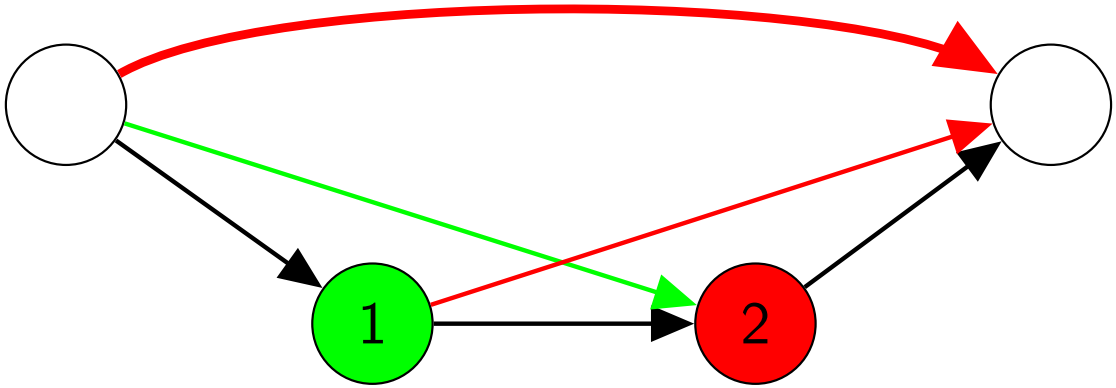


- Die Reihenfolge spielt dabei keine Rolle:

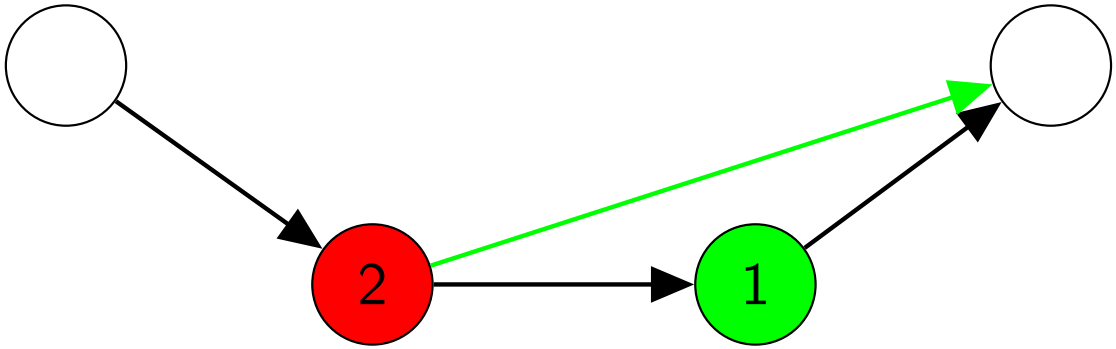


Algorithmus von Warshall – Idee (II)

Das reicht bereits aus, um längere Pfade zu berücksichtigen:

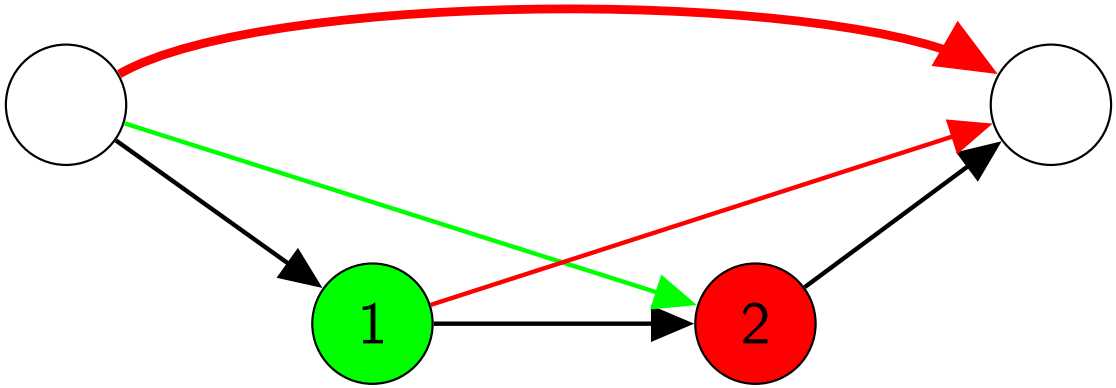


- Die Reihenfolge spielt dabei keine Rolle:

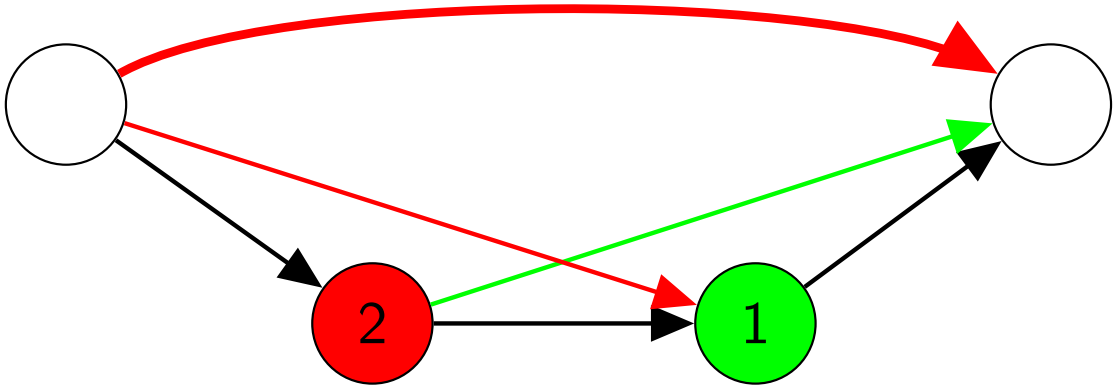


Algorithmus von Warshall – Idee (II)

Das reicht bereits aus, um längere Pfade zu berücksichtigen:

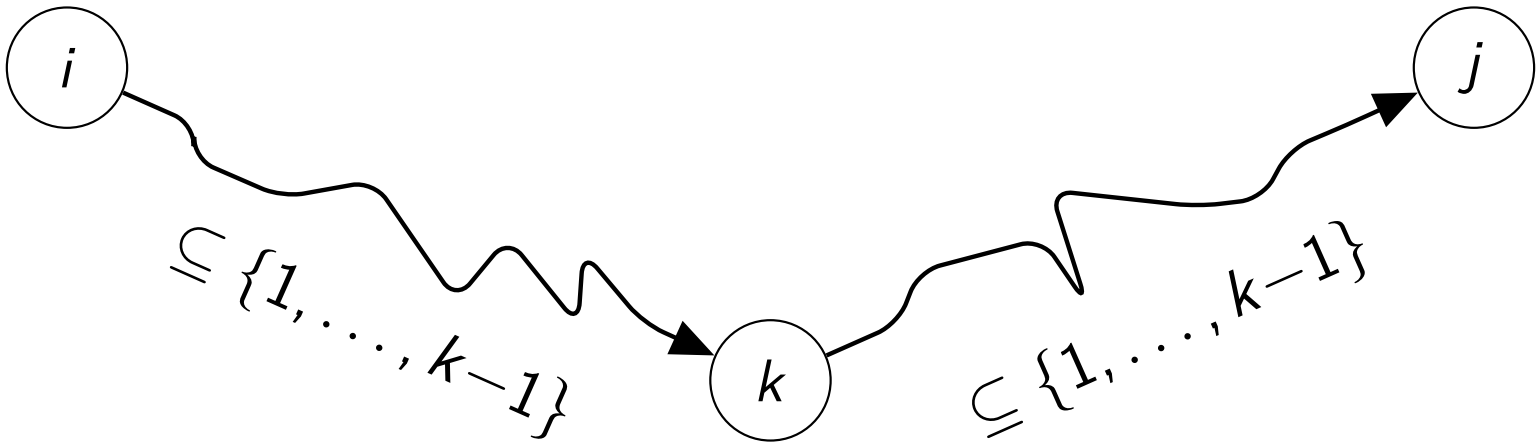


- Die Reihenfolge spielt dabei keine Rolle:



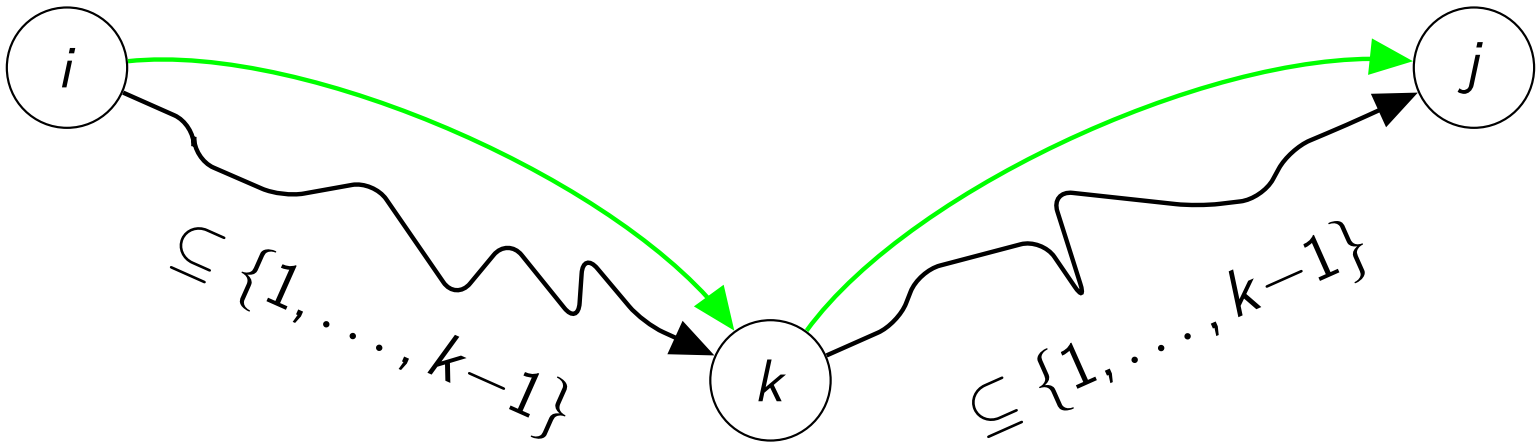
Algorithmus von Warshall – Idee (III)

Allgemeiner Fall:



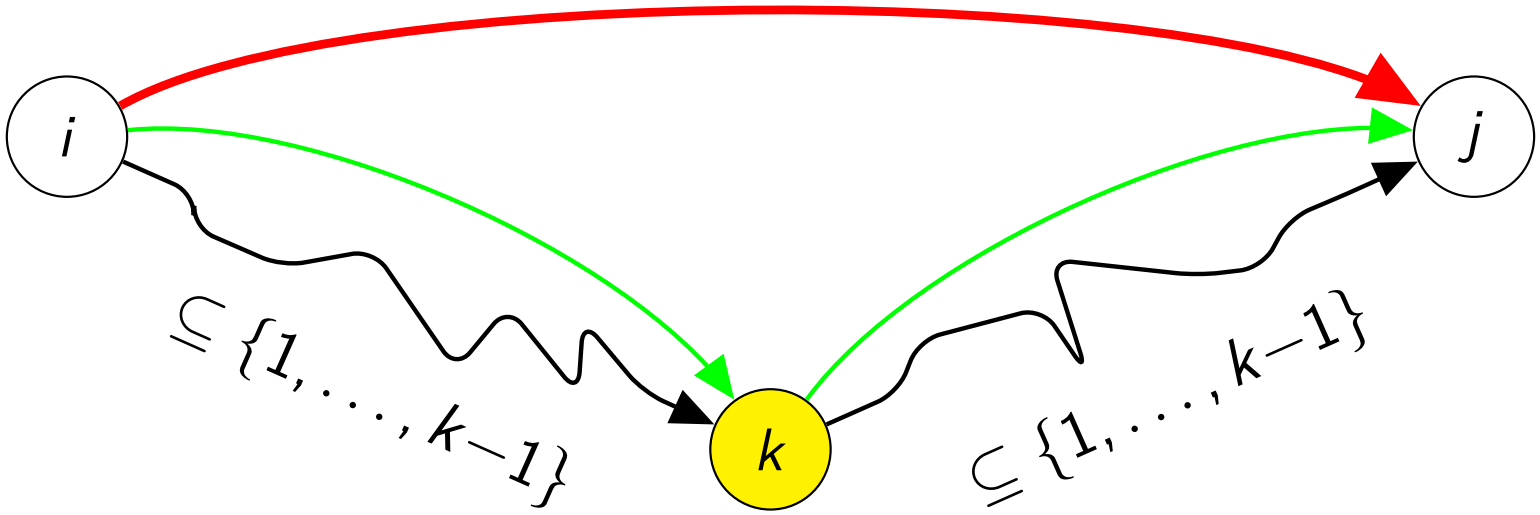
Algorithmus von Warshall – Idee (III)

Allgemeiner Fall:



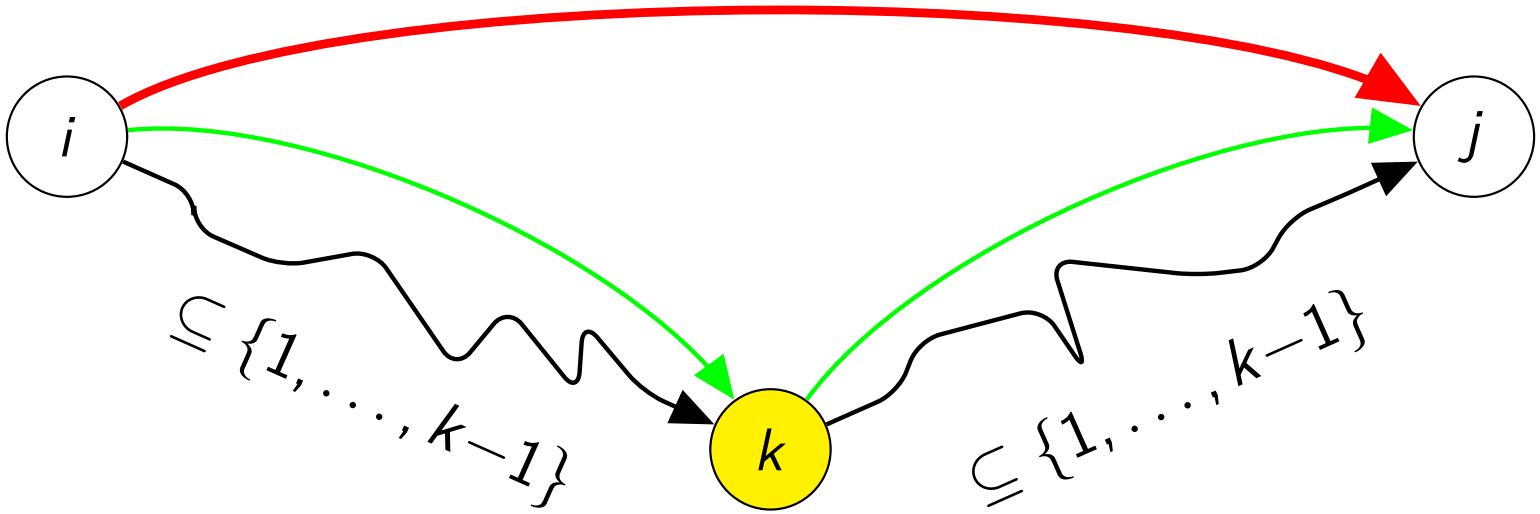
Algorithmus von Warshall – Idee (III)

Allgemeiner Fall:



Algorithmus von Warshall – Idee (III)

Allgemeiner Fall:



- Das lässt sich als Rekursionsgleichung schreiben, wobei $t_{ij}^{(k)} = \texttt{true}$ besagt, dass nach Berücksichtigung der Zwischenknoten $\{1,\ldots,k\}$ der Knoten j von i aus erreichbar ist:

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)} \right)$$

Algorithmus von Warshall – Idee (IV)

$$t_{ij}^{(k)} = \begin{cases} \texttt{false} & \text{für } k = 0, \text{ falls } (i,j) \notin E \\ \texttt{true} & \text{für } k = 0, \text{ falls } (i,j) \in E \\ t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)} \right) & \text{für } k > 0 \end{cases}$$

Algorithmus von Warshall – Idee (IV)

$$t_{ij}^{(k)} = \begin{cases} \texttt{false} & \text{für } k = 0, \text{ falls } (i,j) \notin E \\ \texttt{true} & \text{für } k = 0, \text{ falls } (i,j) \in E \\ t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)} \right) & \text{für } k > 0 \end{cases}$$

- Da zur Berechnung von $t_{ij}^{(k)}$ nur $t_{ij}^{(k-1)}$ – und keine ältere Werte $t_{ij}^{(n)}$ mit $n < k-1$ – gebraucht wird, kann die Berechnung direkt im Ausgabearray (in-place) erfolgen: `R[i,j]` = $t_{ij}^{(\cdot)}$.

Algorithmus von Warshall – Implementierung

```
1 foreach (k ∈ V)
2   foreach (eingehende Kante (i, k) ∈ E)
3     foreach (ausgehende Kante (k, j) ∈ E)
4       Füge (i, j) zu E hinzu.
```

Algorithmus von Warshall – Implementierung

```
1 foreach (k ∈ V)
2   foreach (eingehende Kante (i, k) ∈ E)
3     foreach (ausgehende Kante (k, j) ∈ E)
4       Füge (i, j) zu E hinzu.
```

```
1  void transClos(bool A[][], int n, bool &R[][]) {
2    for (int i = 0; i < n; i++)
3      for (int j = 0; j < n; j++)
4        R[i,j] = A[i,j]; // Kopiere A nach R
5
6    for (int i = 0; i < n; i++)
7      R[i,i] = true; // reflexive Hülle / reflexiver Abschluss
8
9    for (int k = 0; k < n; k++)
10     for (int i = 0; i < n; i++)
11       for (int j = 0; j < n; j++)
12         R[i,j] = R[i,j] || (R[i,k] && R[k,j]);
13 }
```

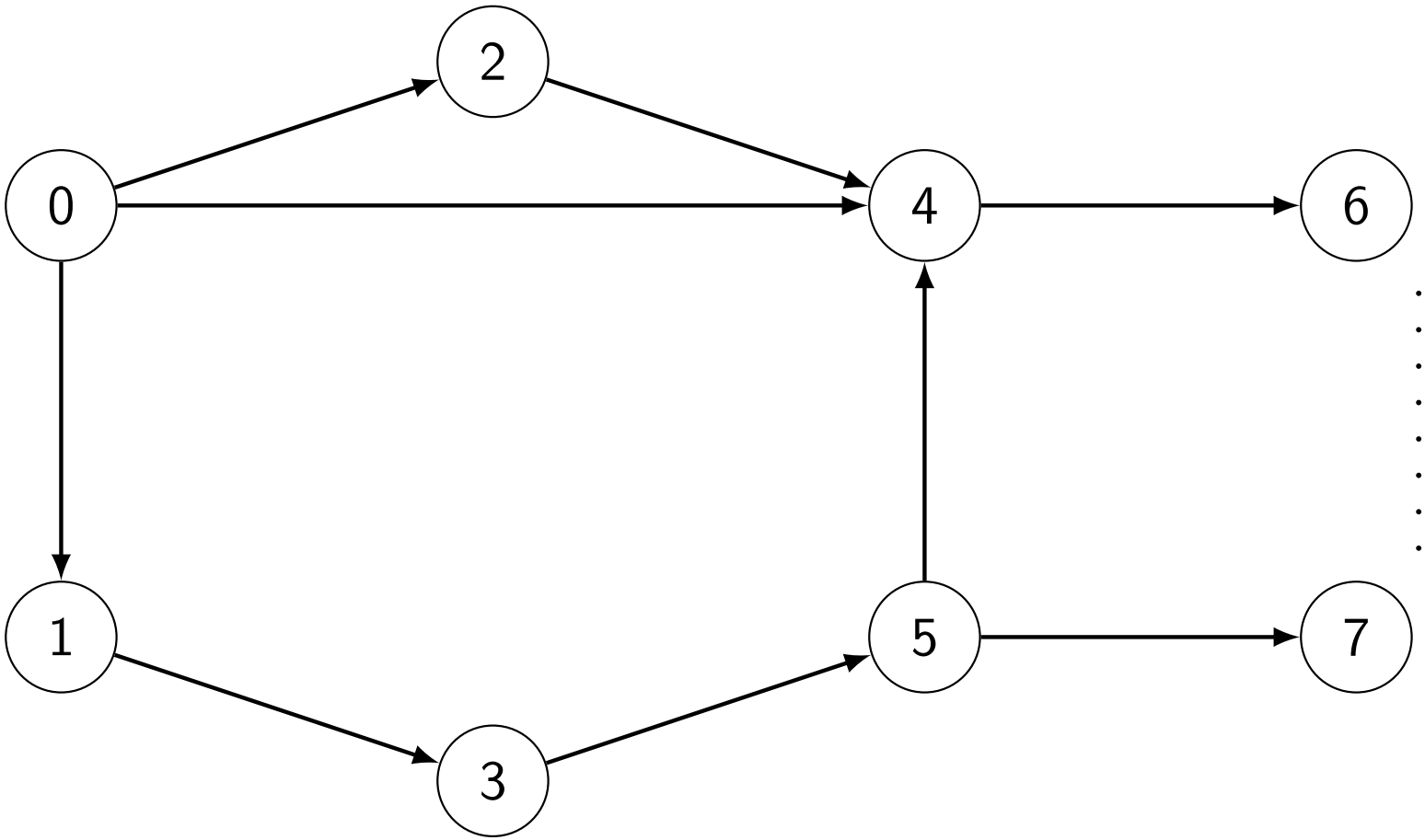
Algorithmus von Warshall – Implementierung

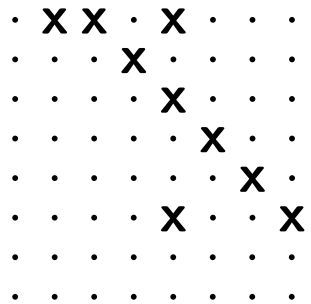
```
1 foreach (k ∈ V)
2   foreach (eingehende Kante (i, k) ∈ E)
3     foreach (ausgehende Kante (k, j) ∈ E)
4       Füge (i, j) zu E hinzu.
```

```
1  void transClos(bool A[][], int n, bool &R[][]) {
2    for (int i = 0; i < n; i++)
3      for (int j = 0; j < n; j++)
4        R[i,j] = A[i,j]; // Kopiere A nach R
5
6    for (int i = 0; i < n; i++)
7      R[i,i] = true; // reflexive Hülle / reflexiver Abschluss
8
9    for (int k = 0; k < n; k++)
10     for (int i = 0; i < n; i++)
11       for (int j = 0; j < n; j++)
12         R[i,j] = R[i,j] || (R[i,k] && R[k,j]);
13 }
```

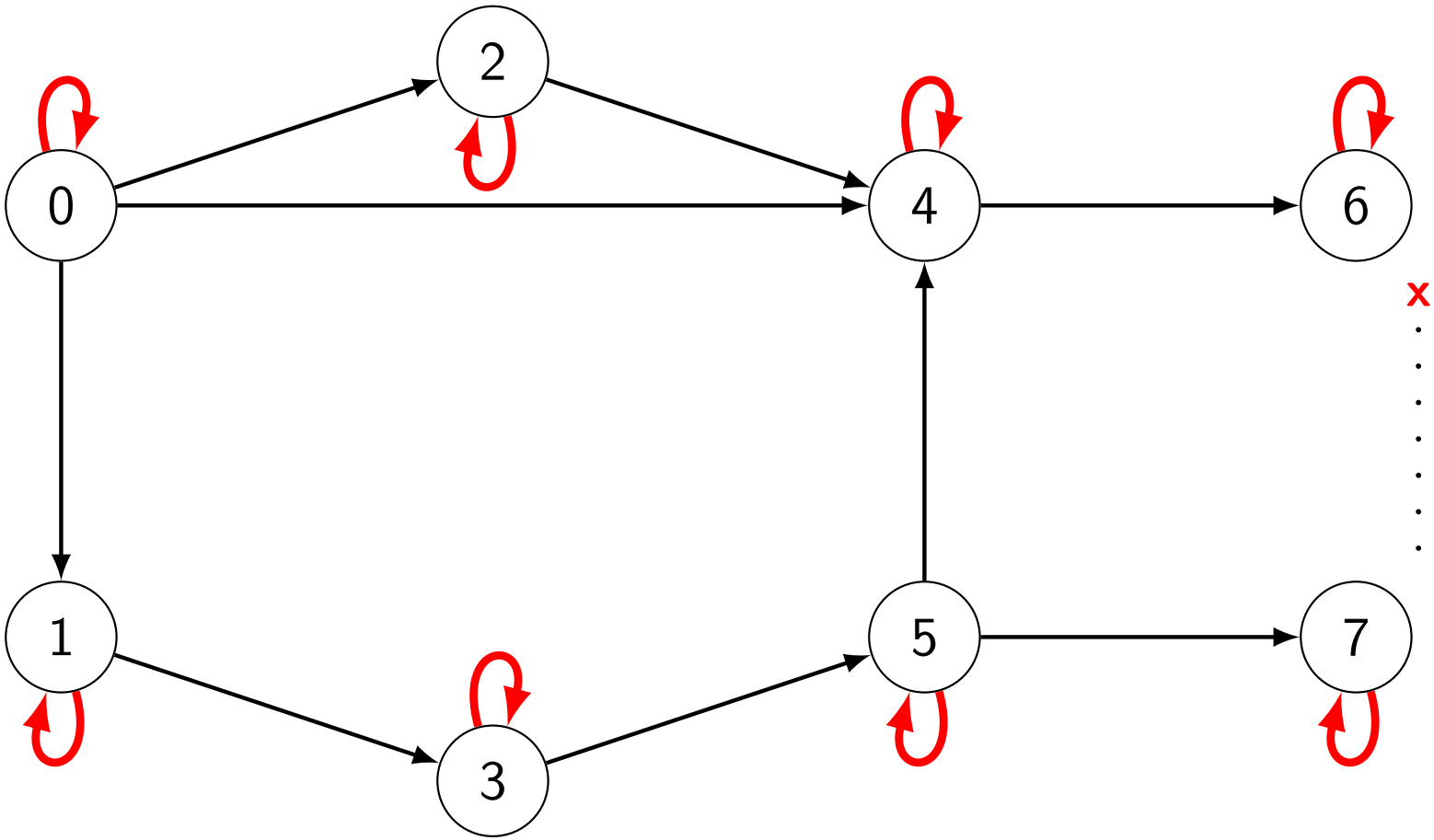
- Zeitkomplexität: $\Theta(n^3)$, Platzkomplexität: $\Theta(n^2)$.

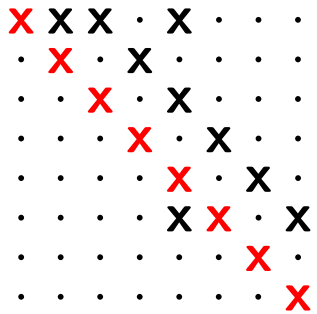
Algorithmus von Warshall – Beispiel (I)



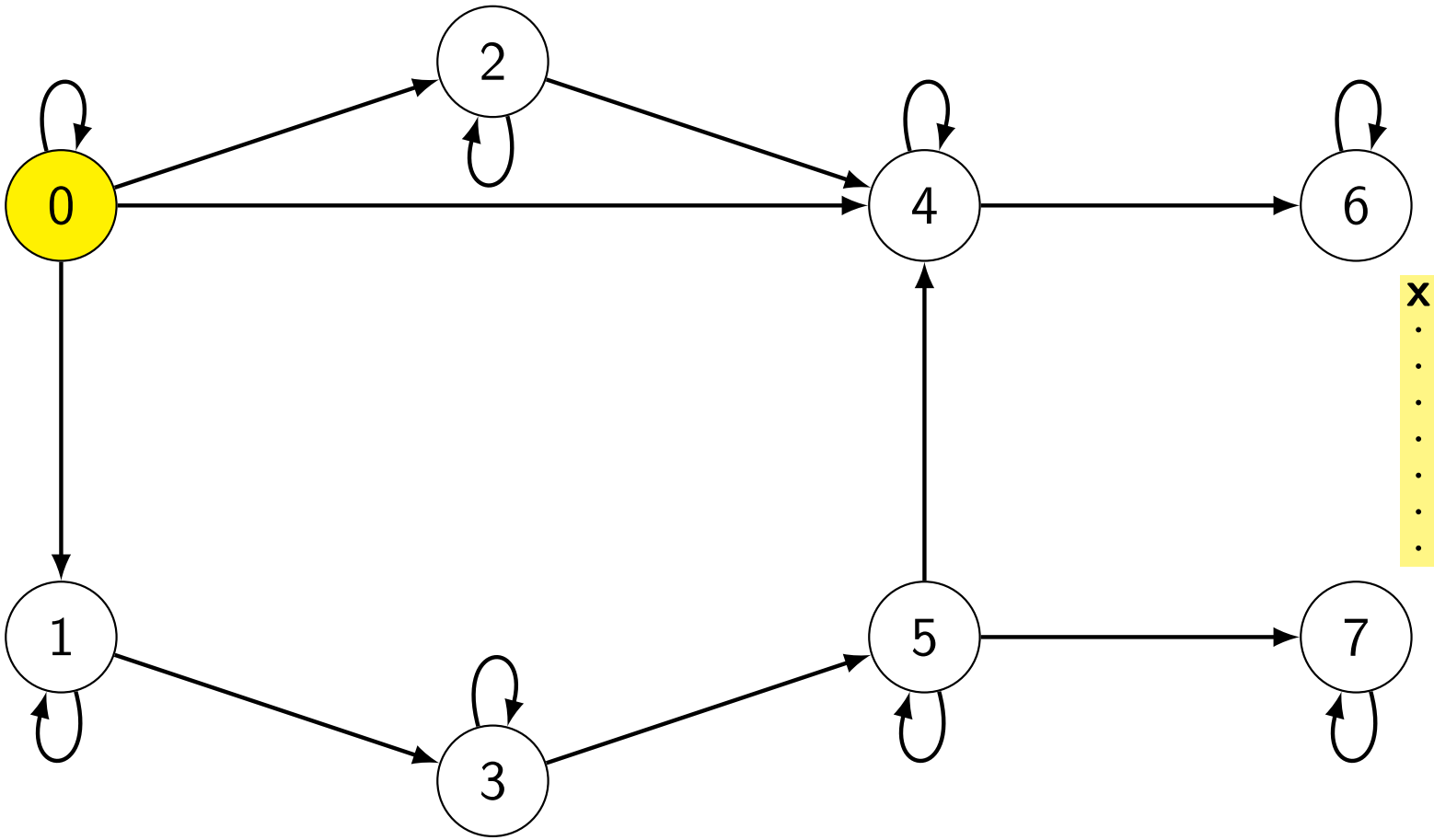


Algorithmus von Warshall – Beispiel (I)



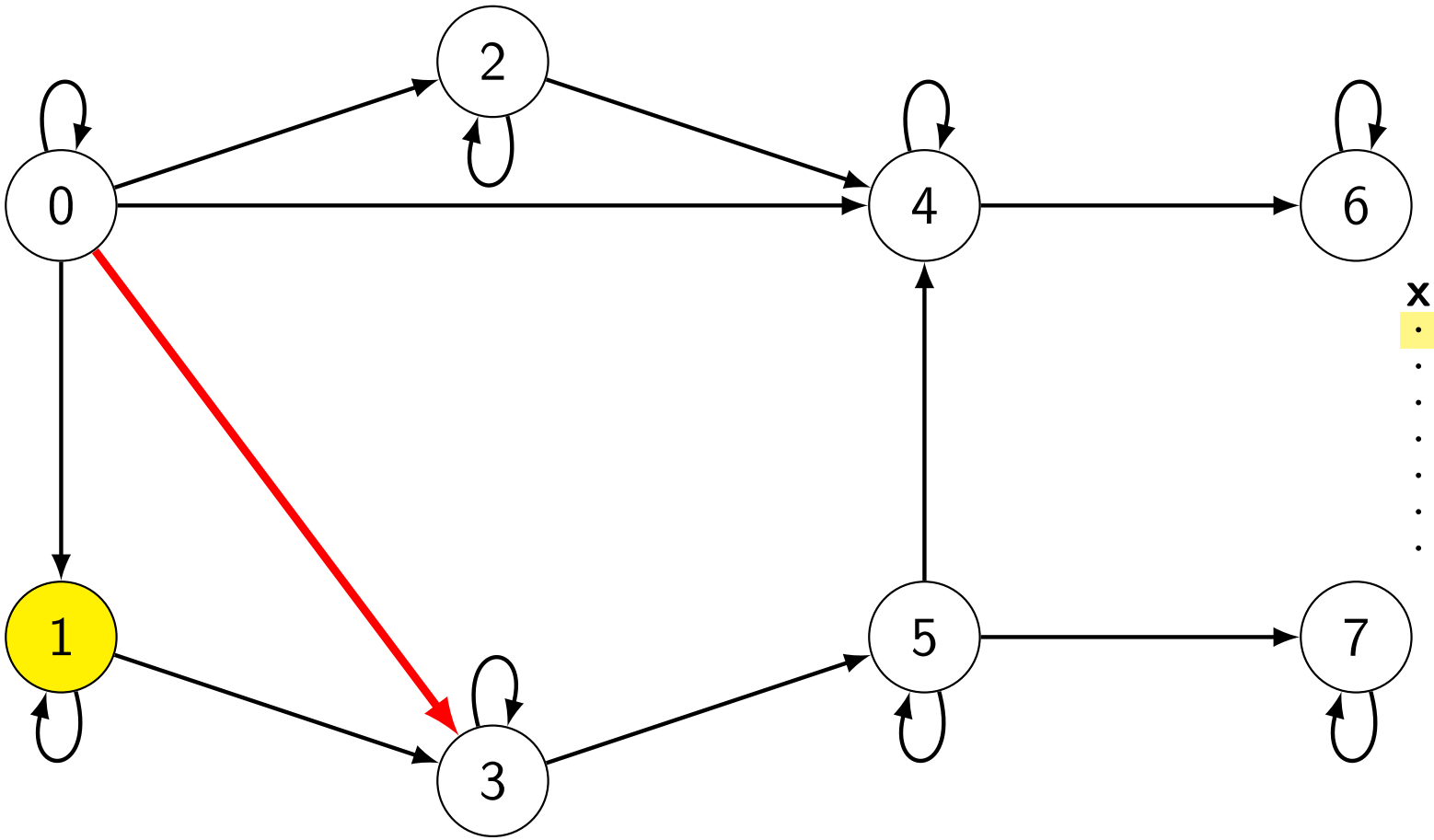


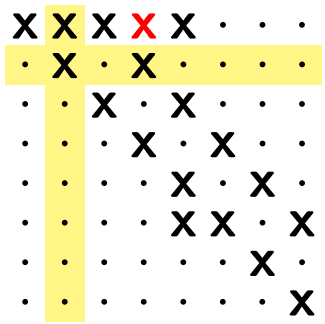
Algorithmus von Warshall – Beispiel (I)



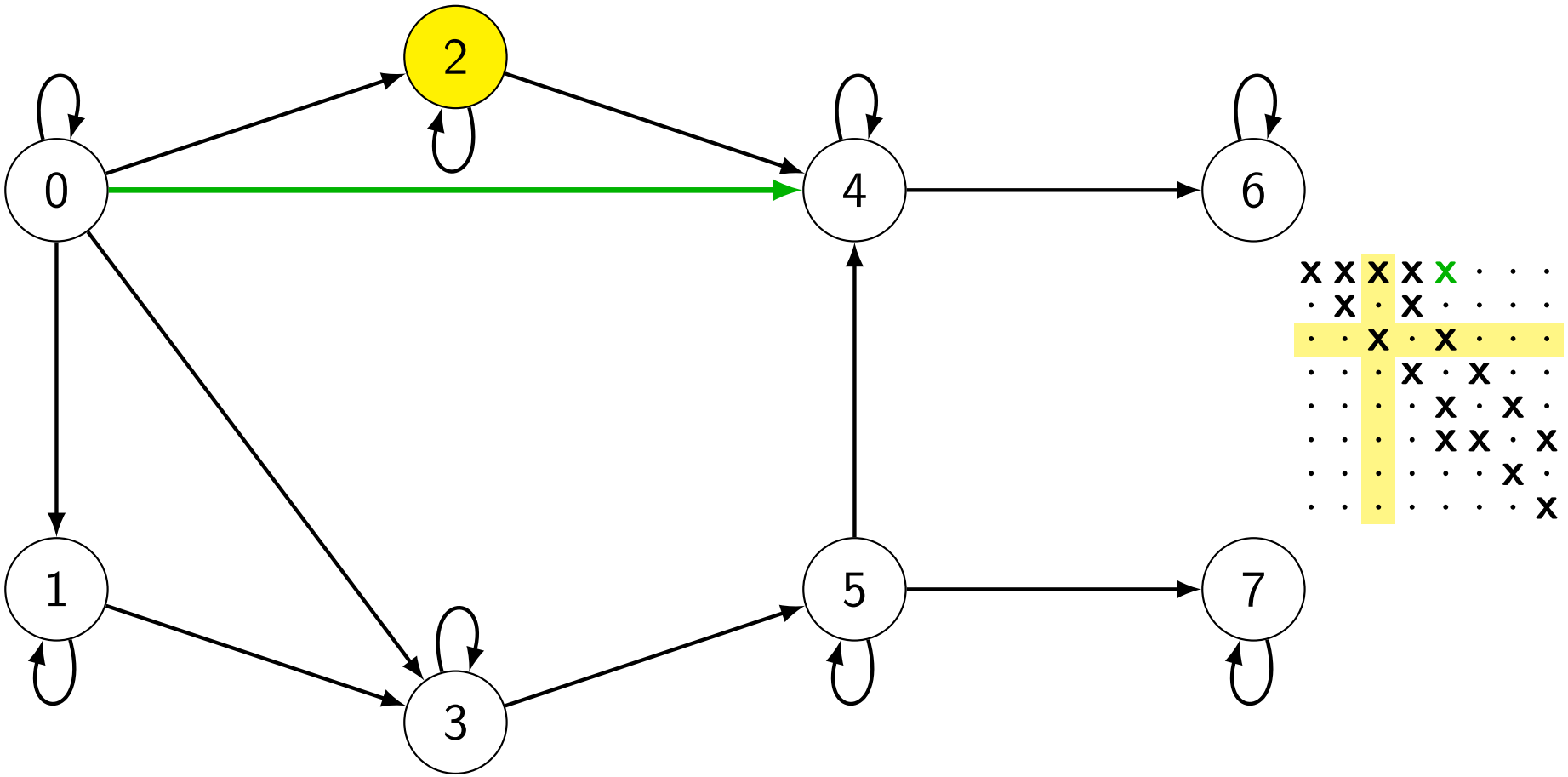
X	X	X	·	X	·	·	·
·	X	·	X	·	·	·	·
·	·	X	·	X	·	·	·
·	·	·	X	·	X	·	·
·	·	·	·	X	·	X	·
·	·	·	·	X	X	·	X
·	·	·	·	·	·	X	·
·	·	·	·	·	·	·	X

Algorithmus von Warshall – Beispiel (I)

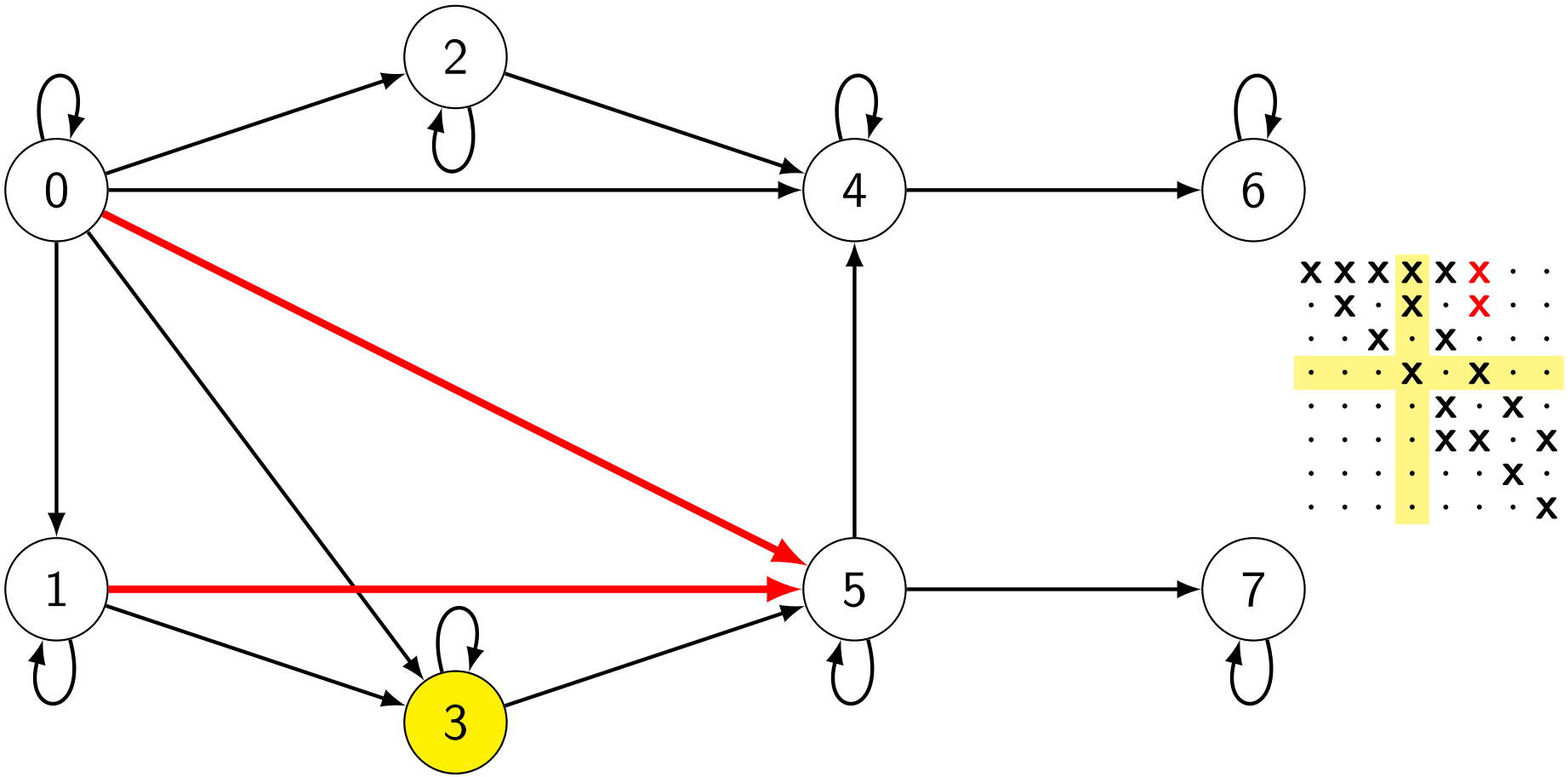




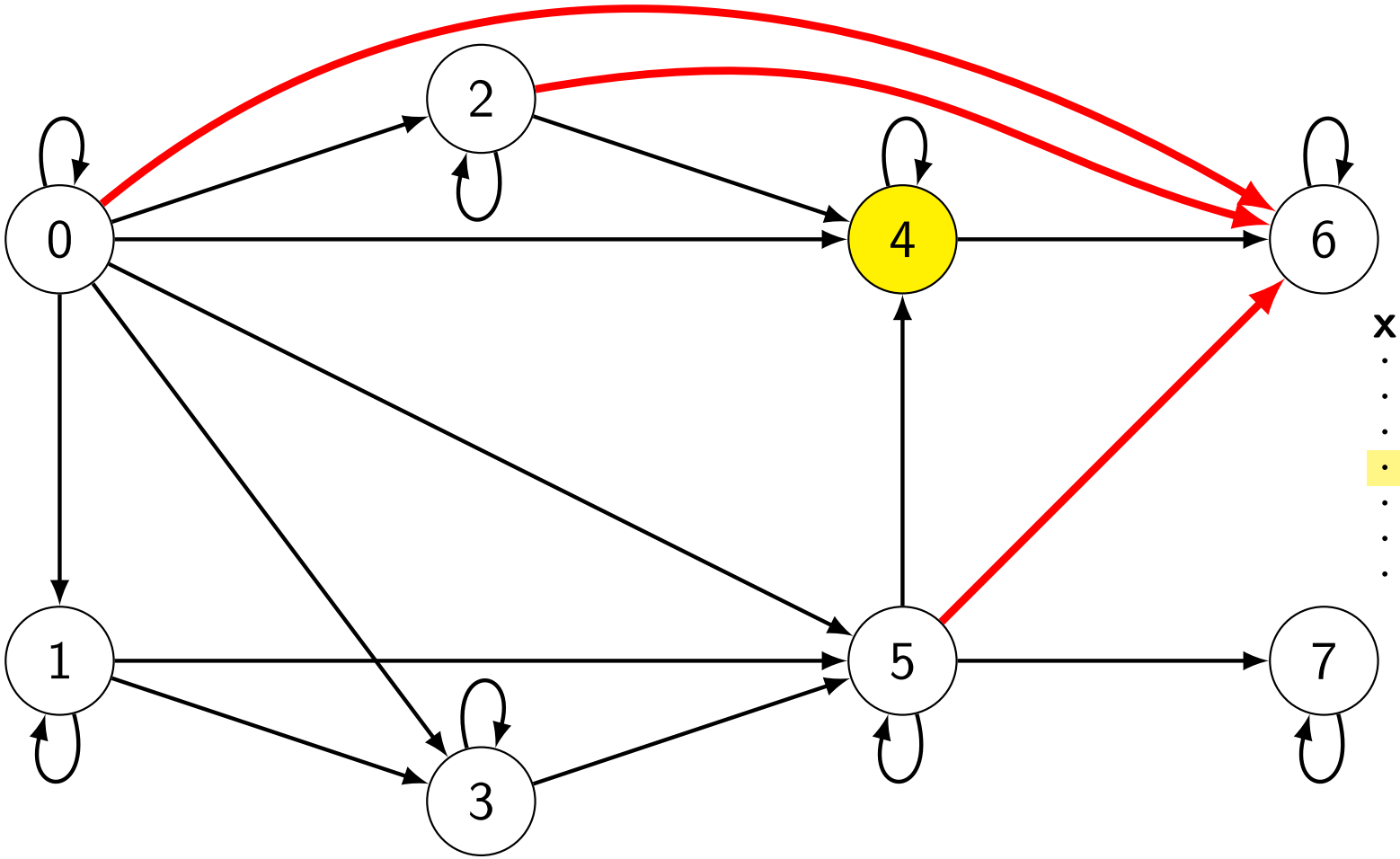
Algorithmus von Warshall – Beispiel (I)

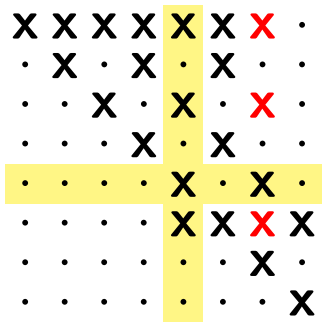


Algorithmus von Warshall – Beispiel (I)

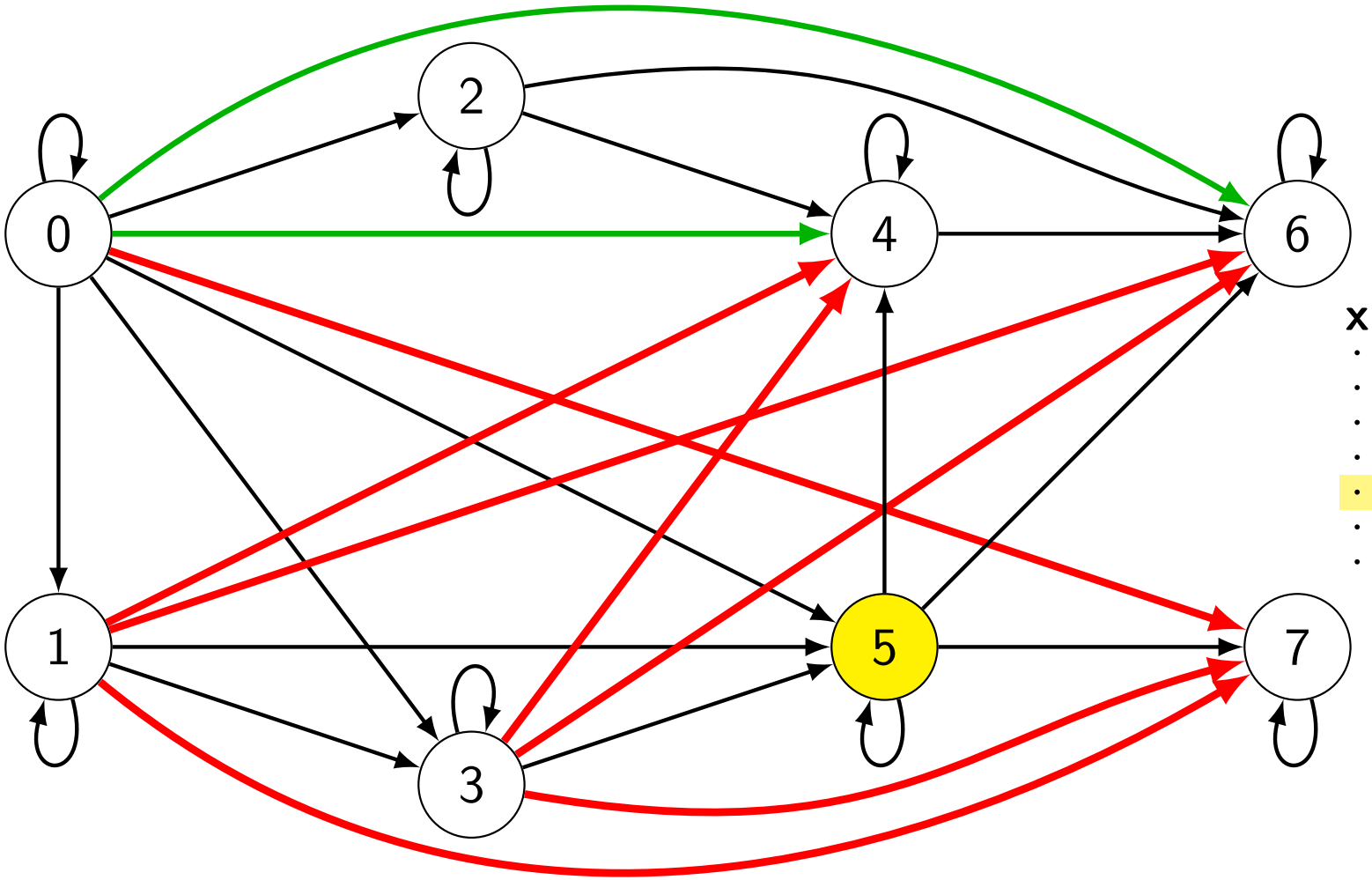


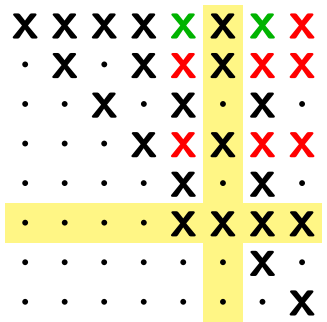
Algorithmus von Warshall – Beispiel (I)



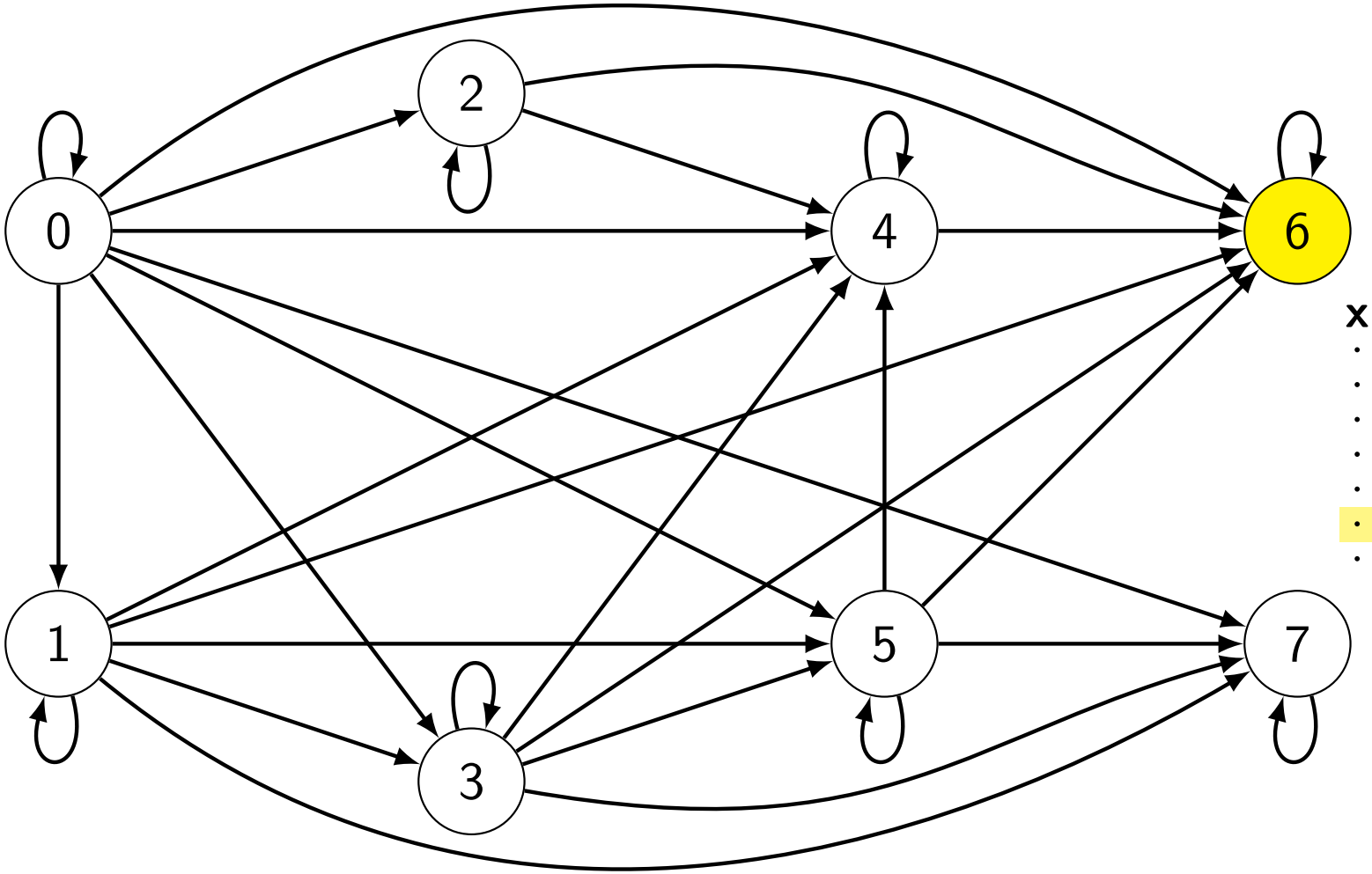


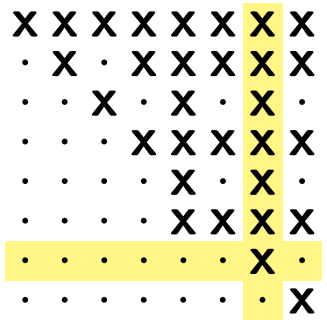
Algorithmus von Warshall – Beispiel (I)



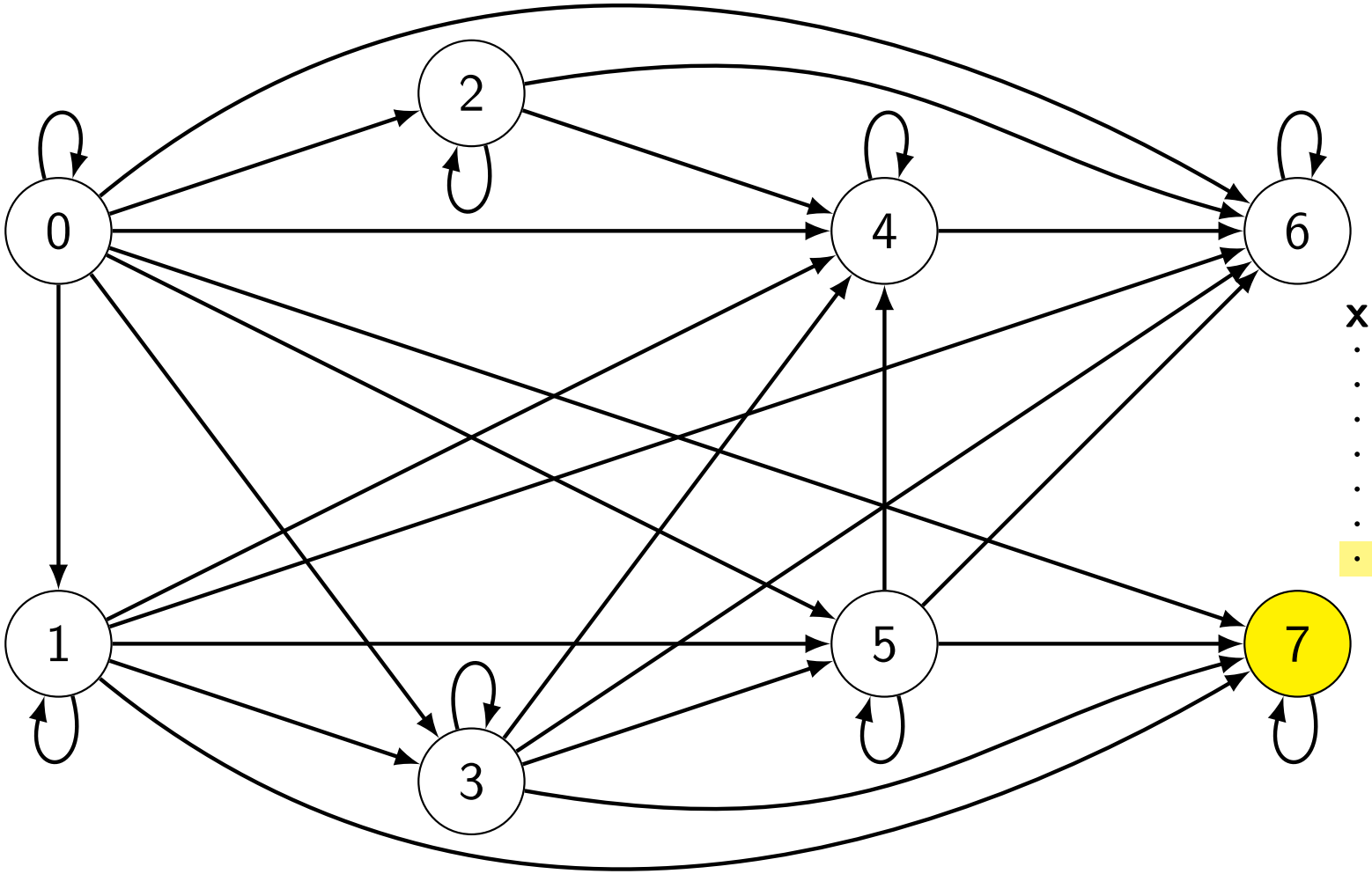


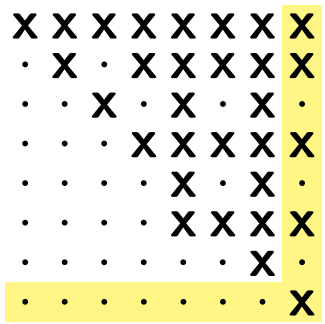
Algorithmus von Warshall – Beispiel (I)





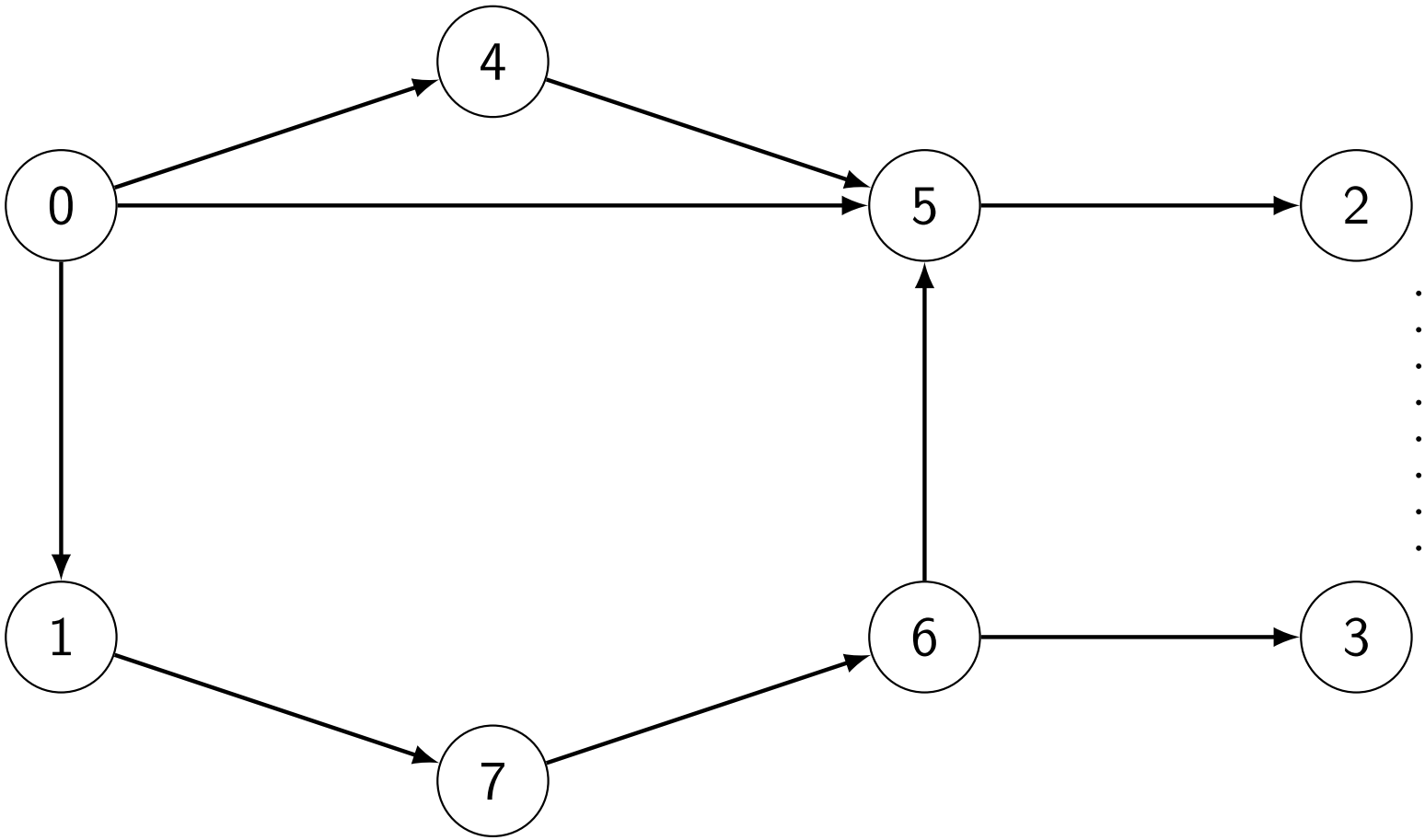
Algorithmus von Warshall – Beispiel (I)

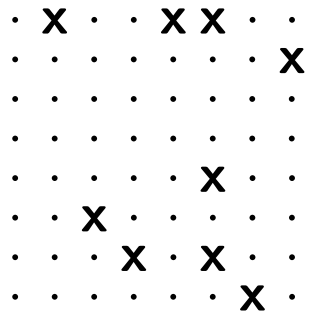




Algorithmus von Warshall – Beispiel (II)

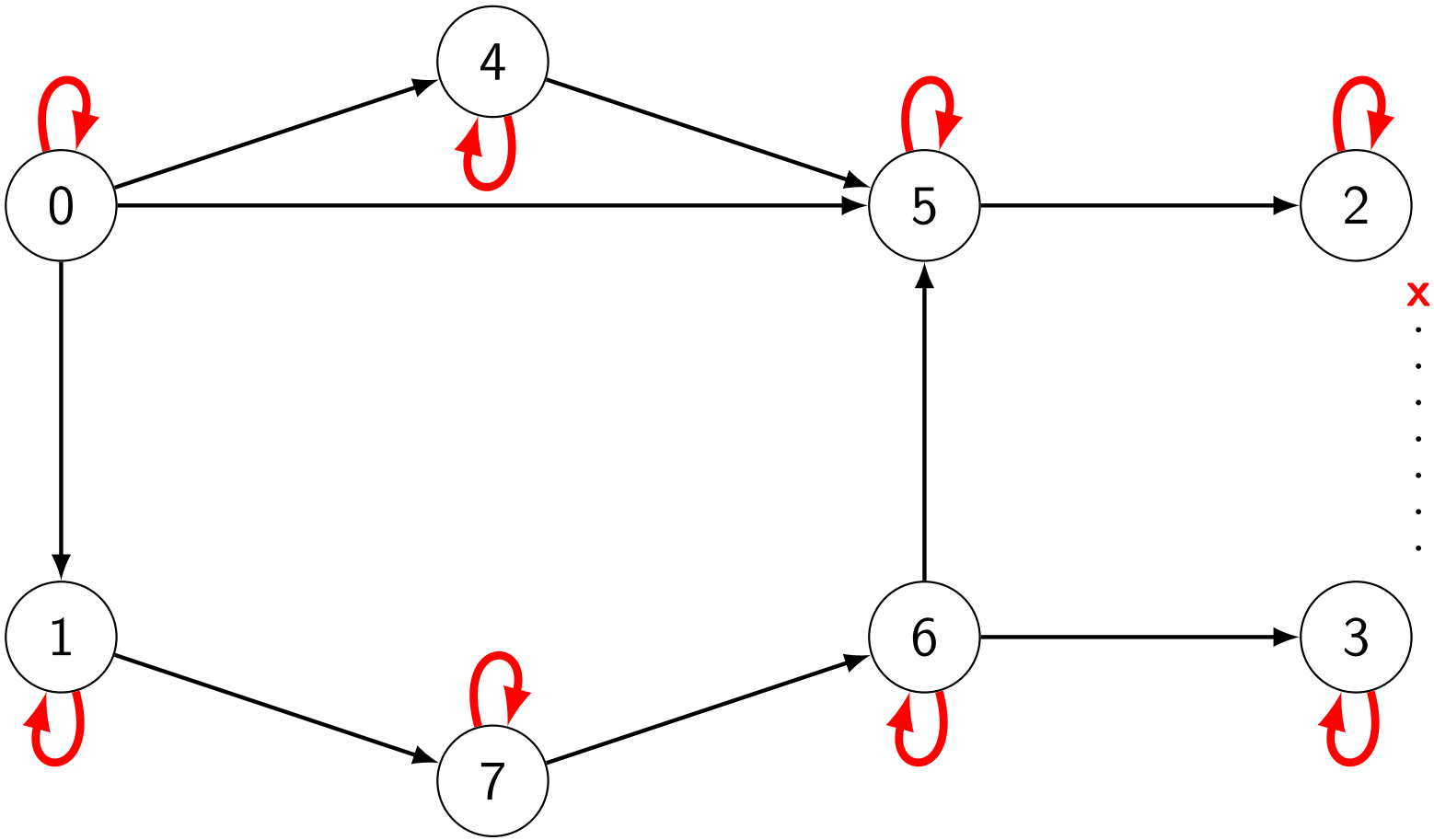
Andere Reihenfolge der Knoten:

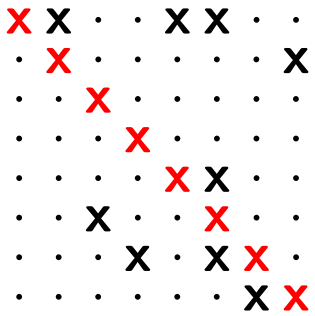




Algorithmus von Warshall – Beispiel (II)

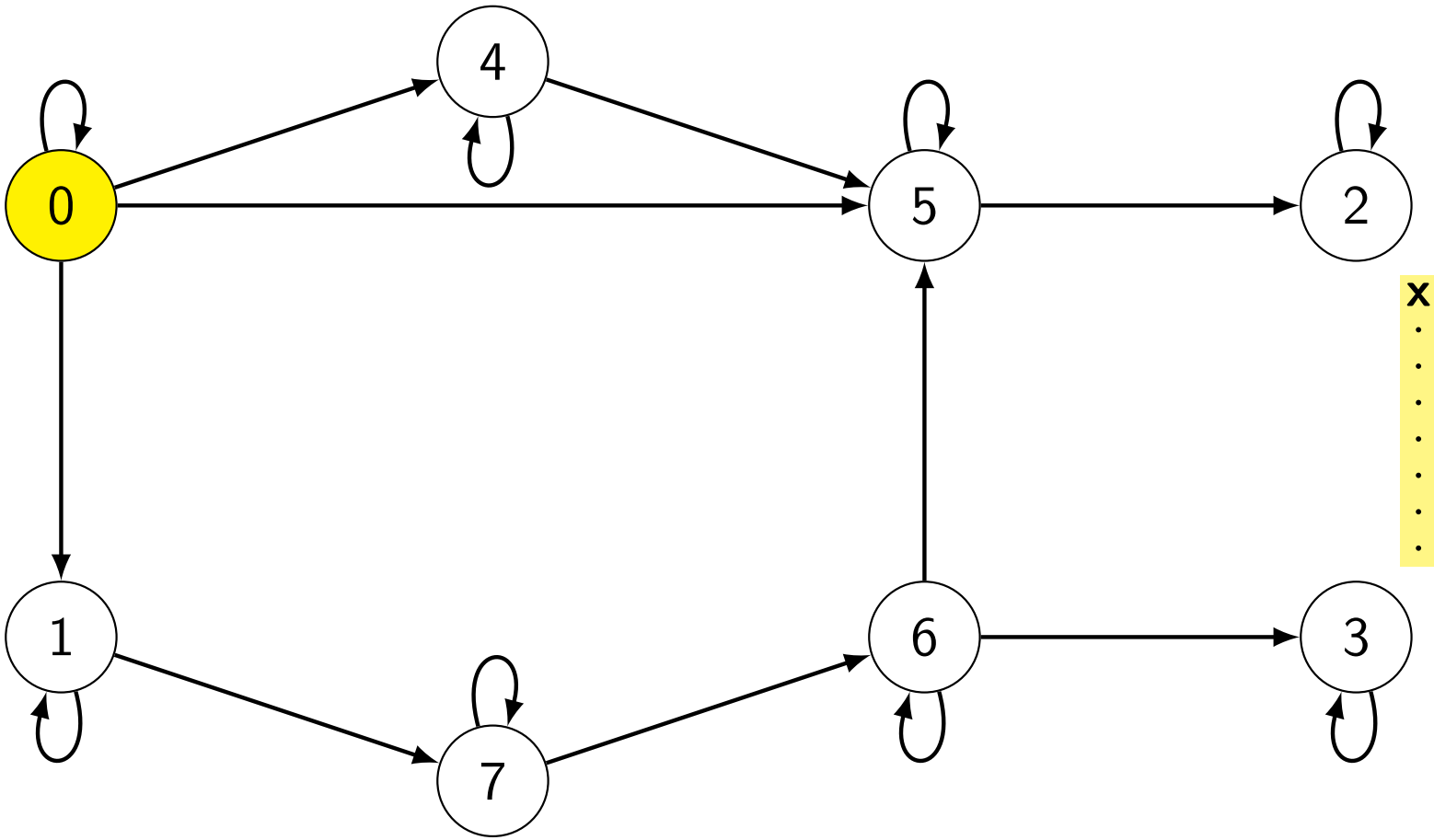
Andere Reihenfolge der Knoten:





Algorithmus von Warshall – Beispiel (II)

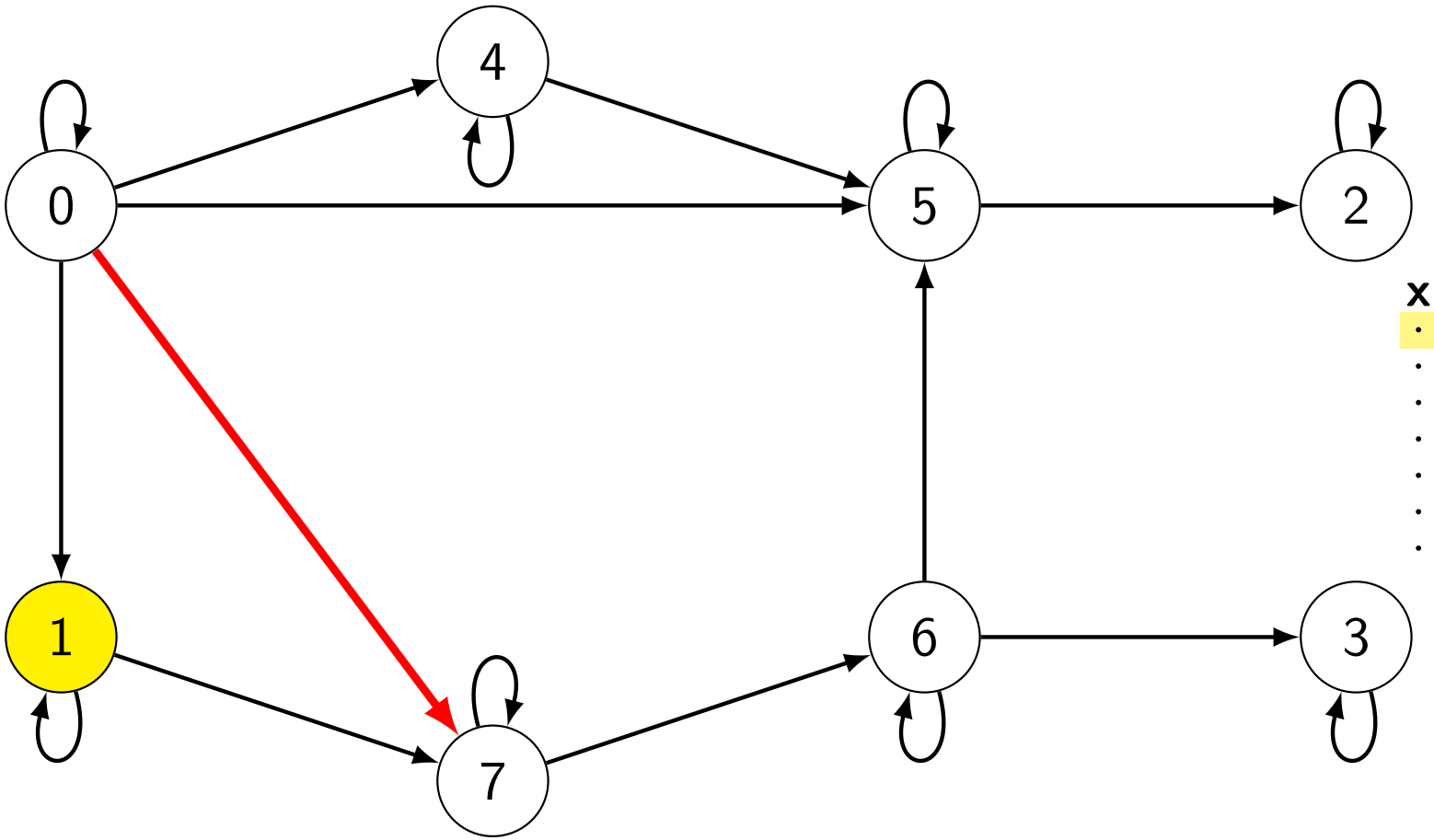
Andere Reihenfolge der Knoten:

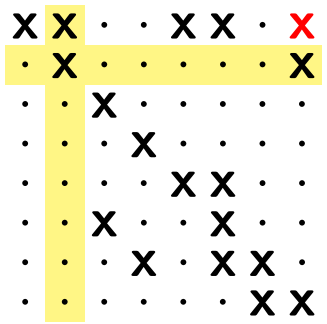


X	X	·	·	X	X	·	·
·	X	·	·	·	·	·	X
·	·	X	·	·	·	·	·
·	·	·	X	·	·	·	·
·	·	·	·	X	X	·	·
·	·	X	·	·	X	·	·
·	·	·	X	·	X	X	·
·	·	·	·	·	·	X	X

Algorithmus von Warshall – Beispiel (II)

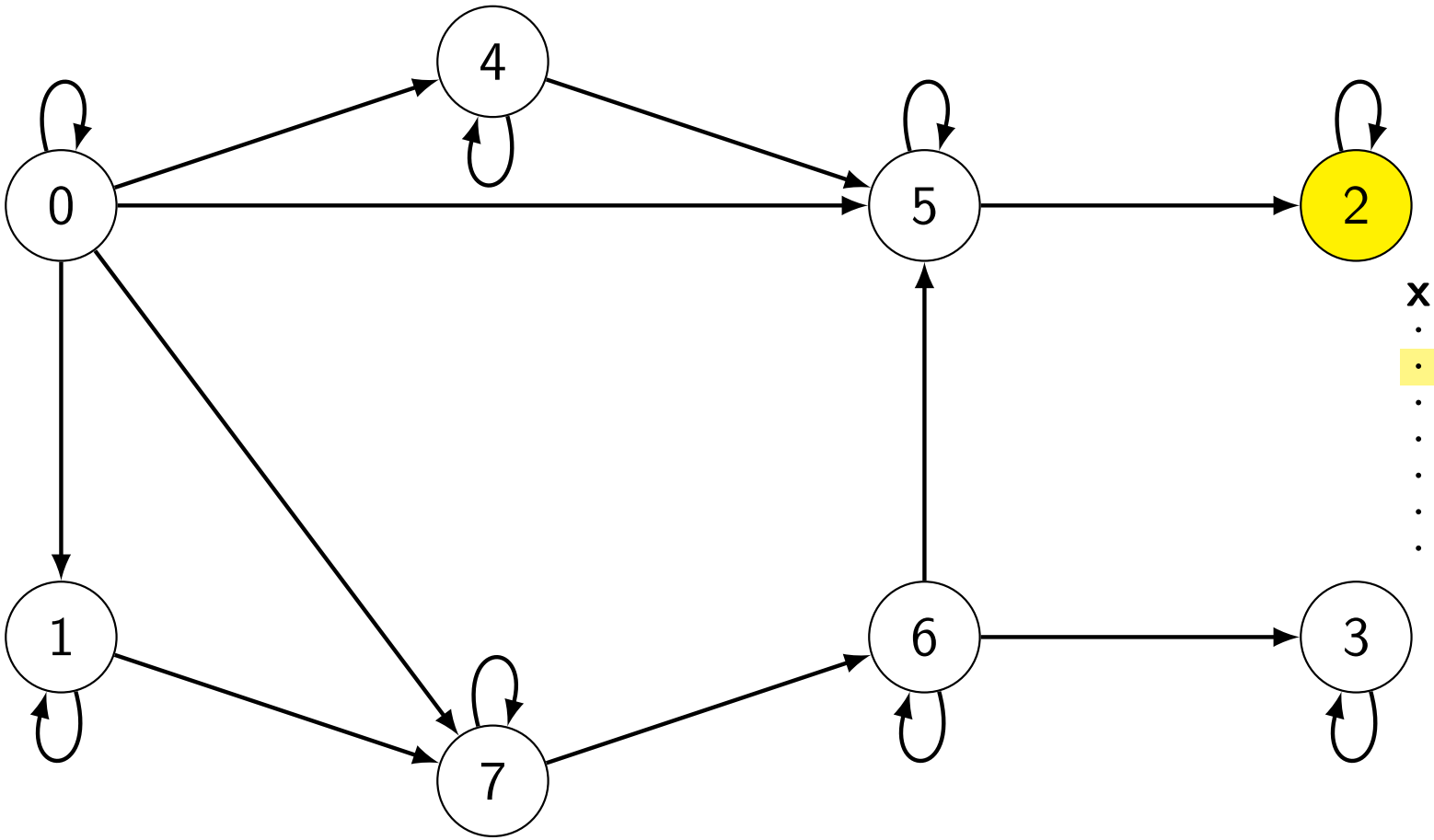
Andere Reihenfolge der Knoten:

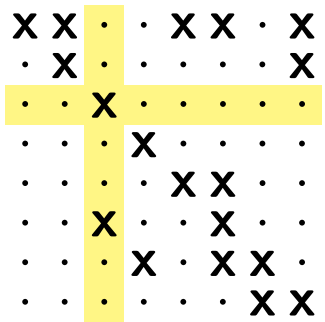




Algorithmus von Warshall – Beispiel (II)

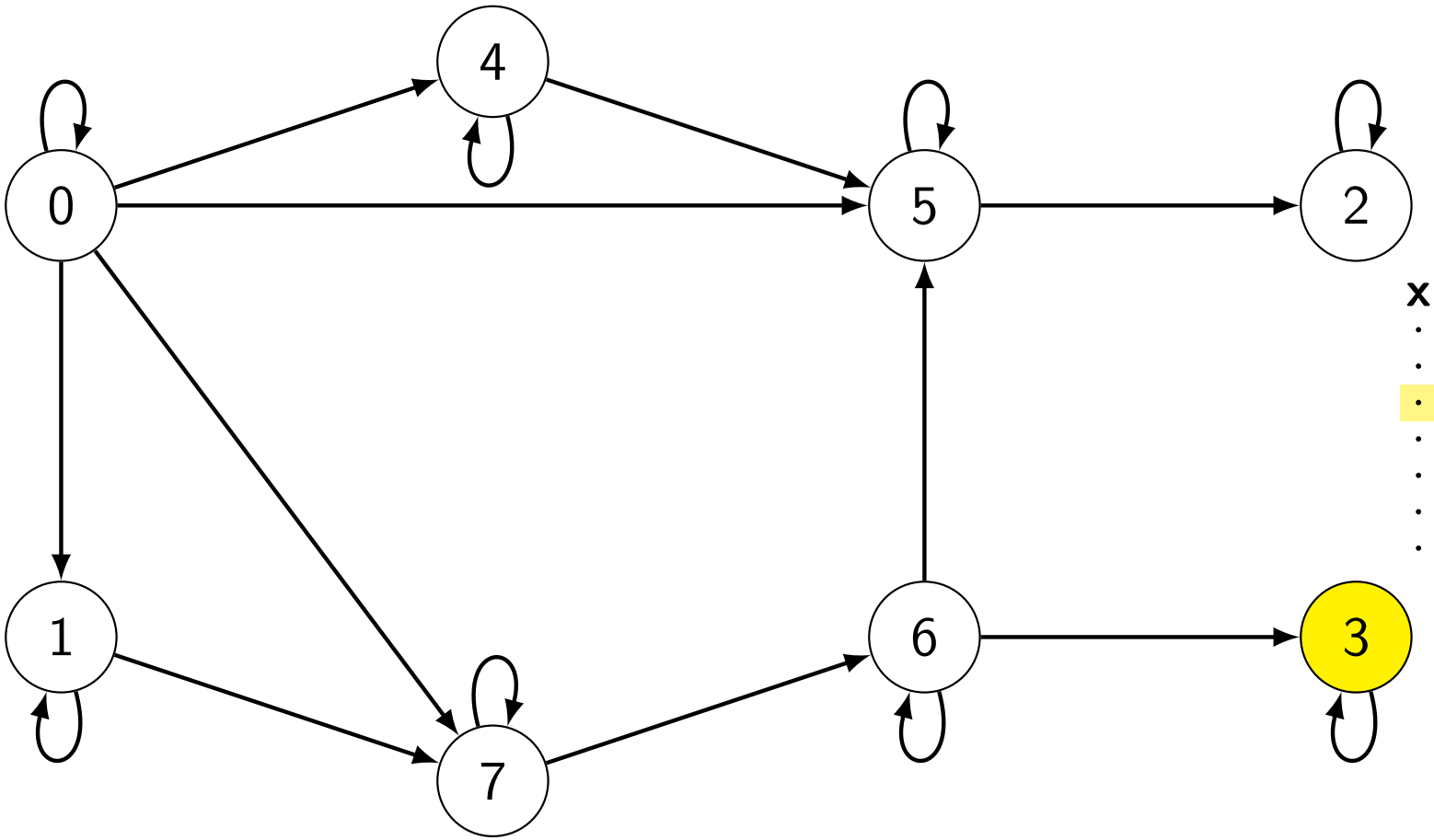
Andere Reihenfolge der Knoten:

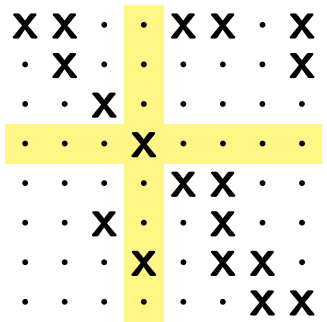




Algorithmus von Warshall – Beispiel (II)

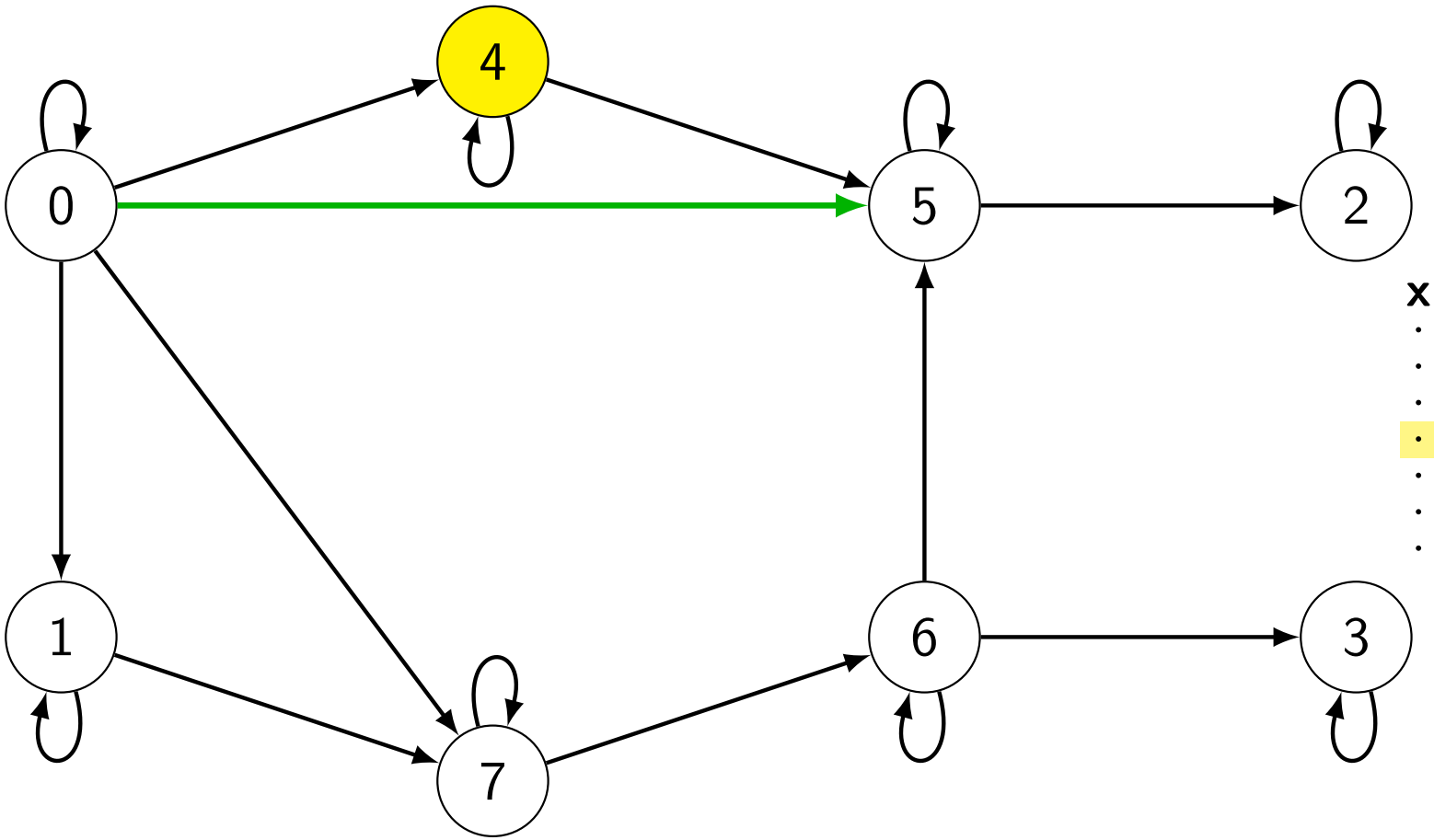
Andere Reihenfolge der Knoten:

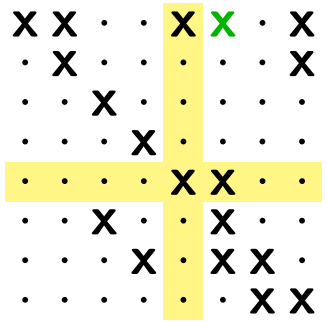




Algorithmus von Warshall – Beispiel (II)

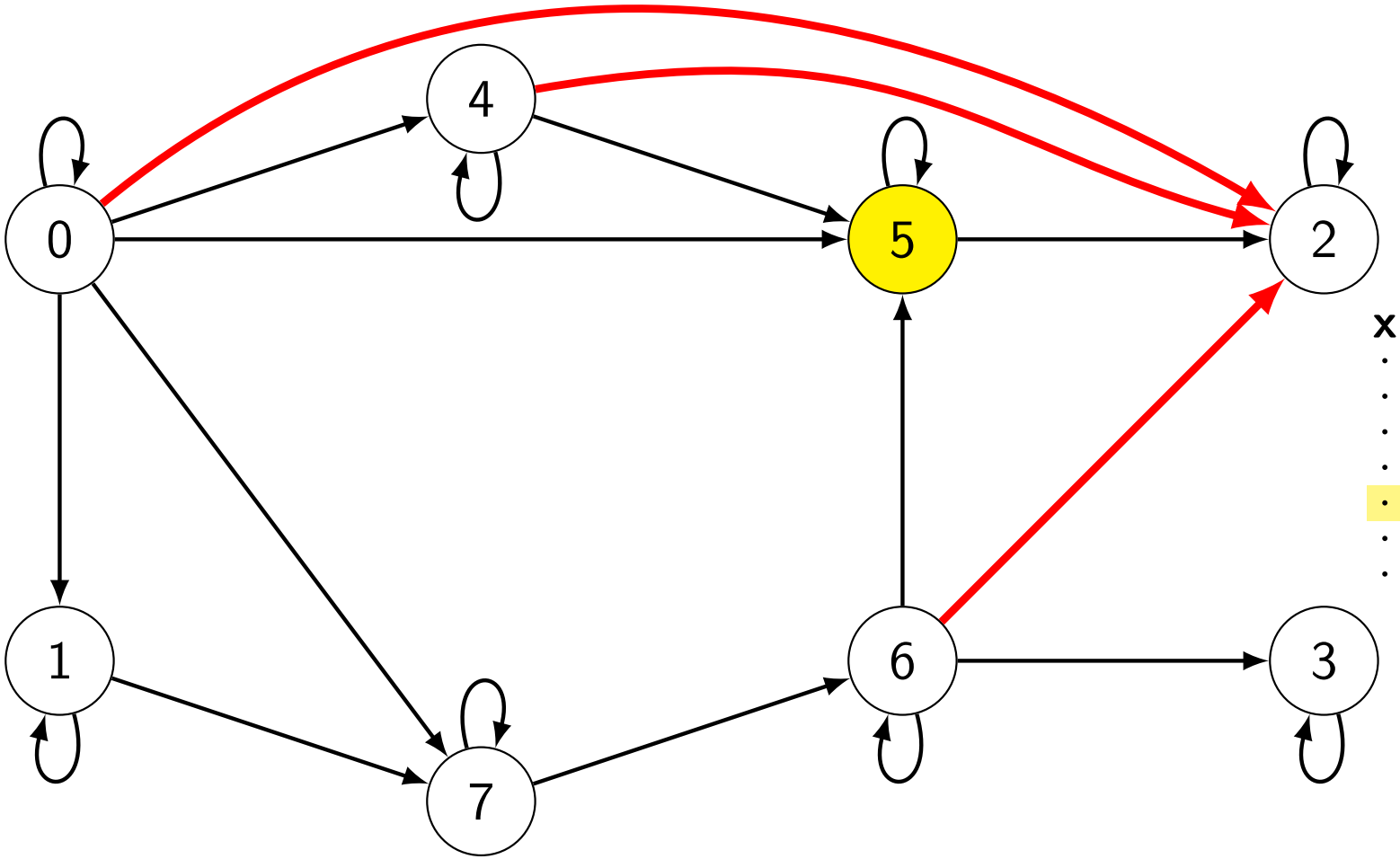
Andere Reihenfolge der Knoten:

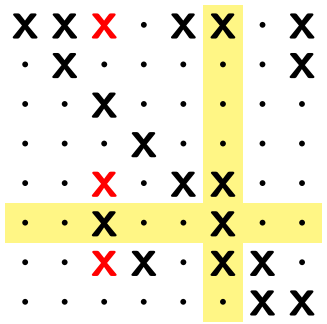




Algorithmus von Warshall – Beispiel (II)

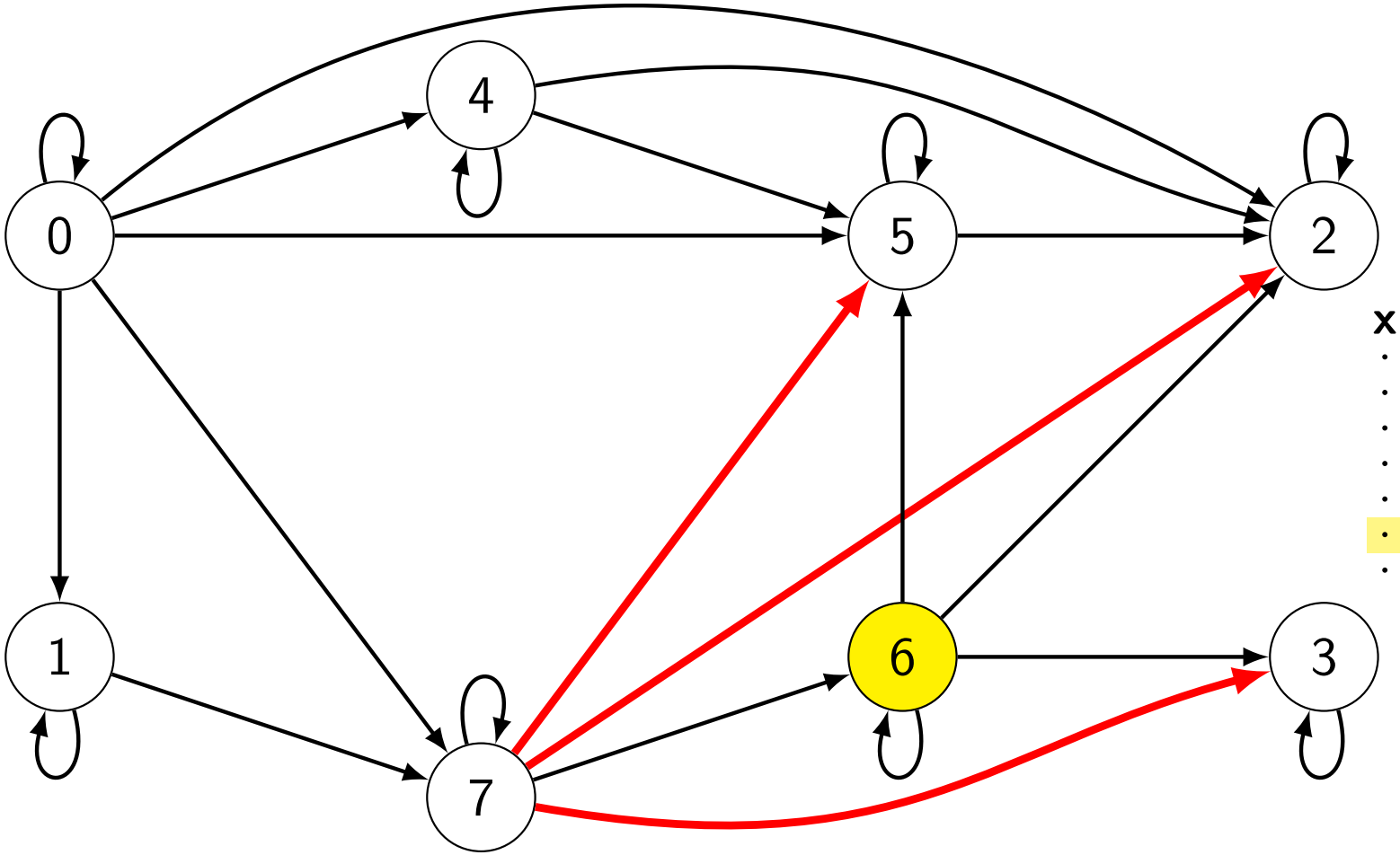
Andere Reihenfolge der Knoten:

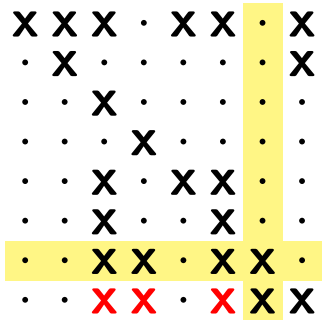




Algorithmus von Warshall – Beispiel (II)

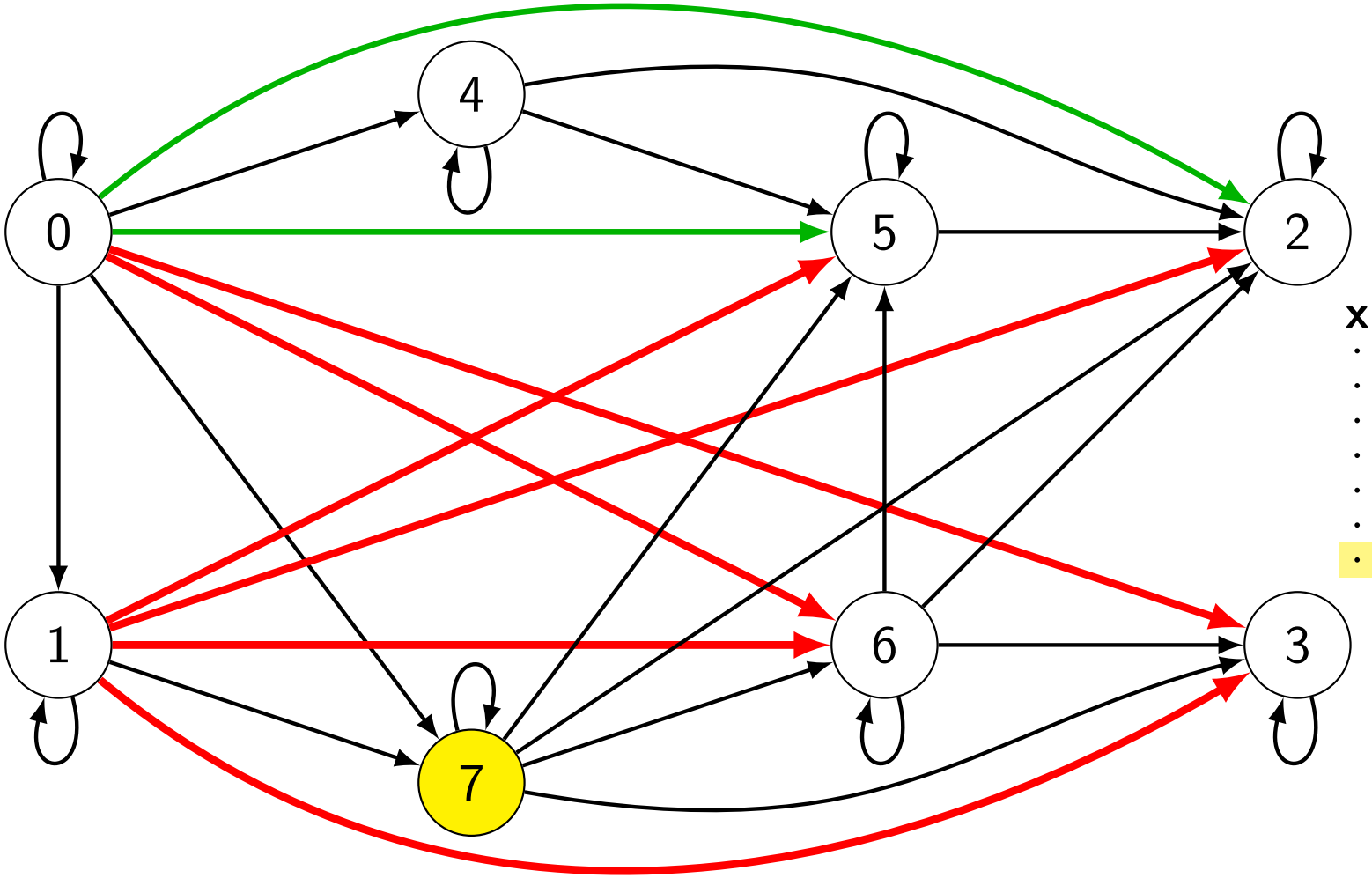
Andere Reihenfolge der Knoten:

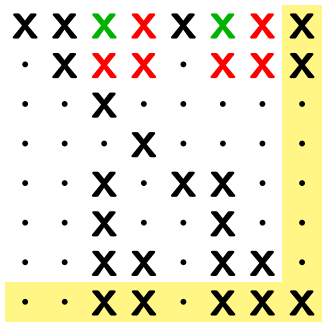




Algorithmus von Warshall – Beispiel (II)

Andere Reihenfolge der Knoten:





Übersicht

All-Pairs Shortest Paths

Wir betrachten gewichtete Digraphen $G = (V, E, W)$.

All-Pairs Shortest Paths

Wir betrachten gewichtete Digraphen $G = (V, E, W)$.

- Die Funktion W ordnet Kanten ein Gewicht zu.

All-Pairs Shortest Paths

Wir betrachten gewichtete Digraphen $G = (V, E, W)$.

- Die Funktion W ordnet Kanten ein Gewicht zu.
- Negative Gewichte sind zugelassen, aber keine Zyklen mit negativem Gewicht.

All-Pairs Shortest Paths

Wir betrachten gewichtete Digraphen $G = (V, E, W)$.

- Die Funktion W ordnet Kanten ein Gewicht zu.
- Negative Gewichte sind zugelassen, aber keine Zyklen mit negativem Gewicht.
- Nicht vorhandene Kanten haben Gewicht $W(\cdot, \cdot) = \infty$.

All-Pairs Shortest Paths

Wir betrachten gewichtete Digraphen $G = (V, E, W)$.

- Die Funktion W ordnet Kanten ein Gewicht zu.
- Negative Gewichte sind zugelassen, aber keine Zyklen mit negativem Gewicht.
- Nicht vorhandene Kanten haben Gewicht $W(\cdot, \cdot) = \infty$.

Problem (All-Pairs Shortest Path)

Berechne für jedes Paar `i, j` *die Länge* `D[i,j]` *des kürzesten Pfades.*

All-Pairs Shortest Paths

Wir betrachten gewichtete Digraphen $G = (V, E, W)$.

- Die Funktion W ordnet Kanten ein Gewicht zu.
- Negative Gewichte sind zugelassen, aber keine Zyklen mit negativem Gewicht.
- Nicht vorhandene Kanten haben Gewicht $W(\cdot, \cdot) = \infty$.

Problem (All-Pairs Shortest Path)

Berechne für jedes Paar `i, j` *die Länge* `D[i,j]` *des kürzesten Pfades.*

Naive Lösung: lasse ein SSSP-Algorithmus (z. B. Bellman-Ford) $|V|$ mal laufen.

All-Pairs Shortest Paths

Wir betrachten gewichtete Digraphen $G = (V, E, W)$.

- Die Funktion W ordnet Kanten ein Gewicht zu.
- Negative Gewichte sind zugelassen, aber keine Zyklen mit negativem Gewicht.
- Nicht vorhandene Kanten haben Gewicht $W(\cdot, \cdot) = \infty$.

Problem (All-Pairs Shortest Path)

Berechne für jedes Paar `i, j` *die Länge* `D[i,j]` *des kürzesten Pfades.*

Naive Lösung: lasse ein SSSP-Algorithmus (z. B. Bellman-Ford) $|V|$ mal laufen.

All-Pairs Shortest Paths

Wir betrachten gewichtete Digraphen $G = (V, E, W)$.

- Die Funktion W ordnet Kanten ein Gewicht zu.
- Negative Gewichte sind zugelassen, aber keine Zyklen mit negativem Gewicht.
- Nicht vorhandene Kanten haben Gewicht $W(\cdot, \cdot) = \infty$.

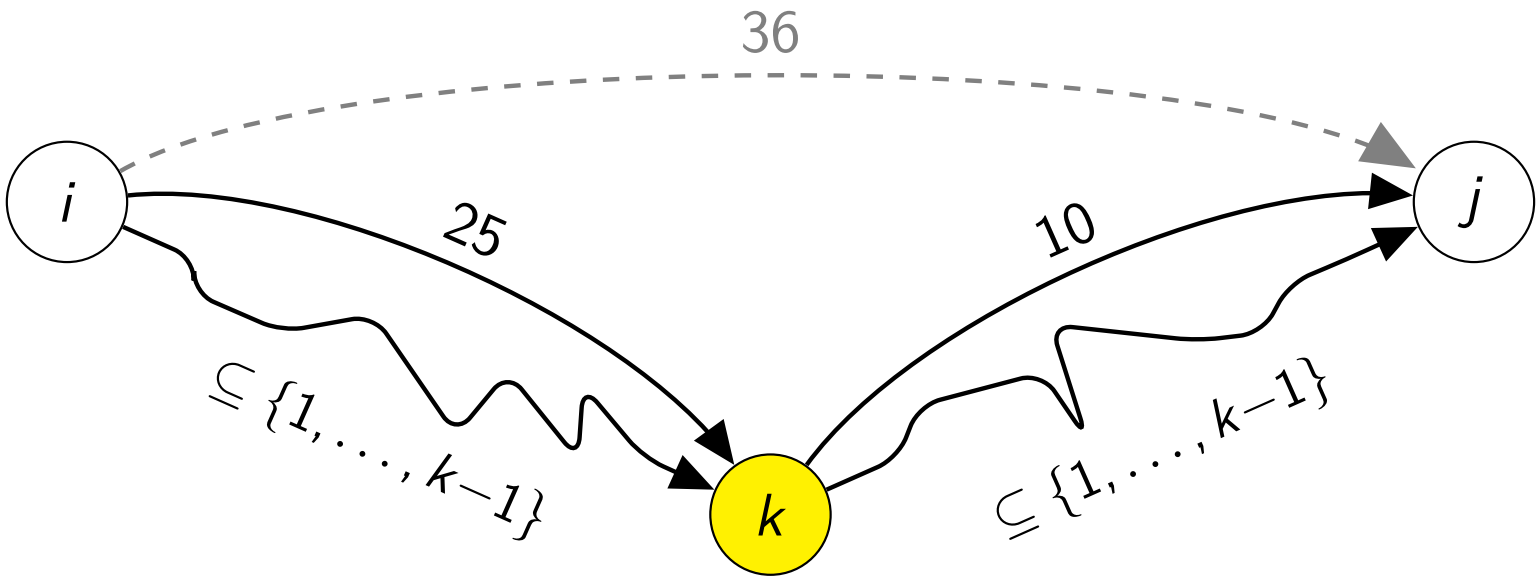
Problem (All-Pairs Shortest Path)

Berechne für jedes Paar `i, j` *die Länge* `D[i,j]` *des kürzesten Pfades.*

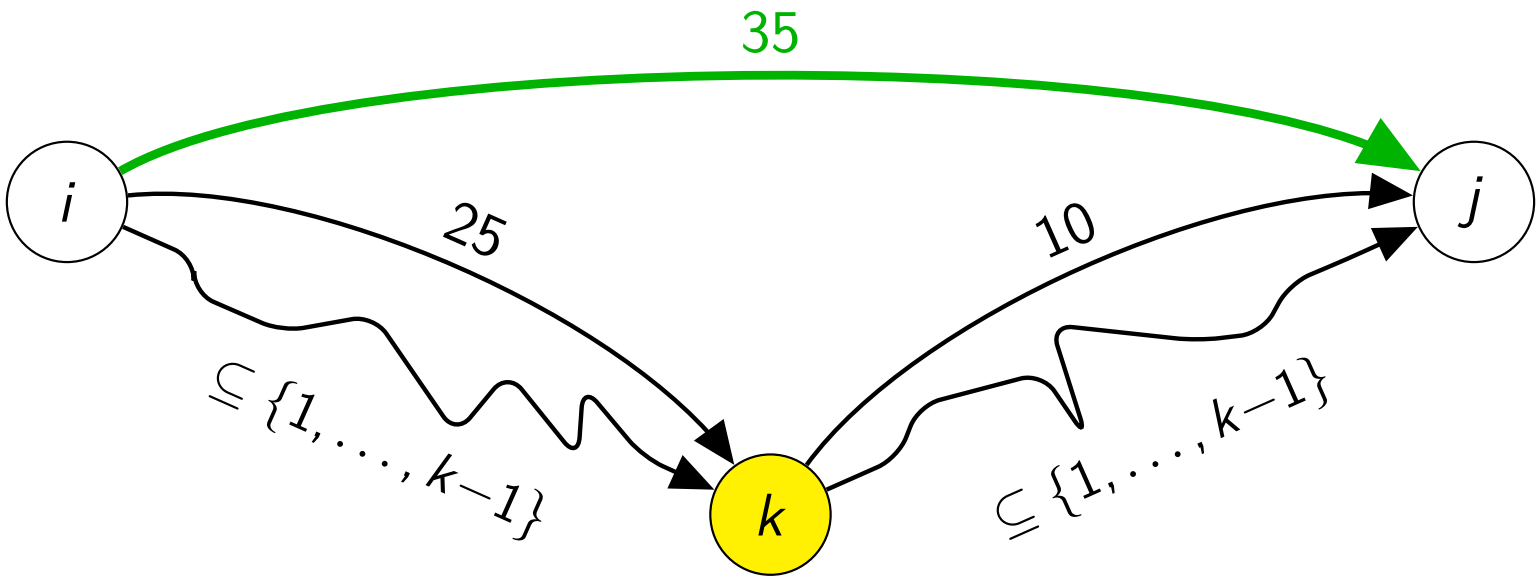
Naive Lösung: lasse ein SSSP-Algorithmus (z. B. Bellman-Ford) $|V|$ mal laufen. Dies führt zu einer Worst-Case Zeitkomplexität $O(|V|^4)$.

Effizientere Version: Floyd's Algorithmus.

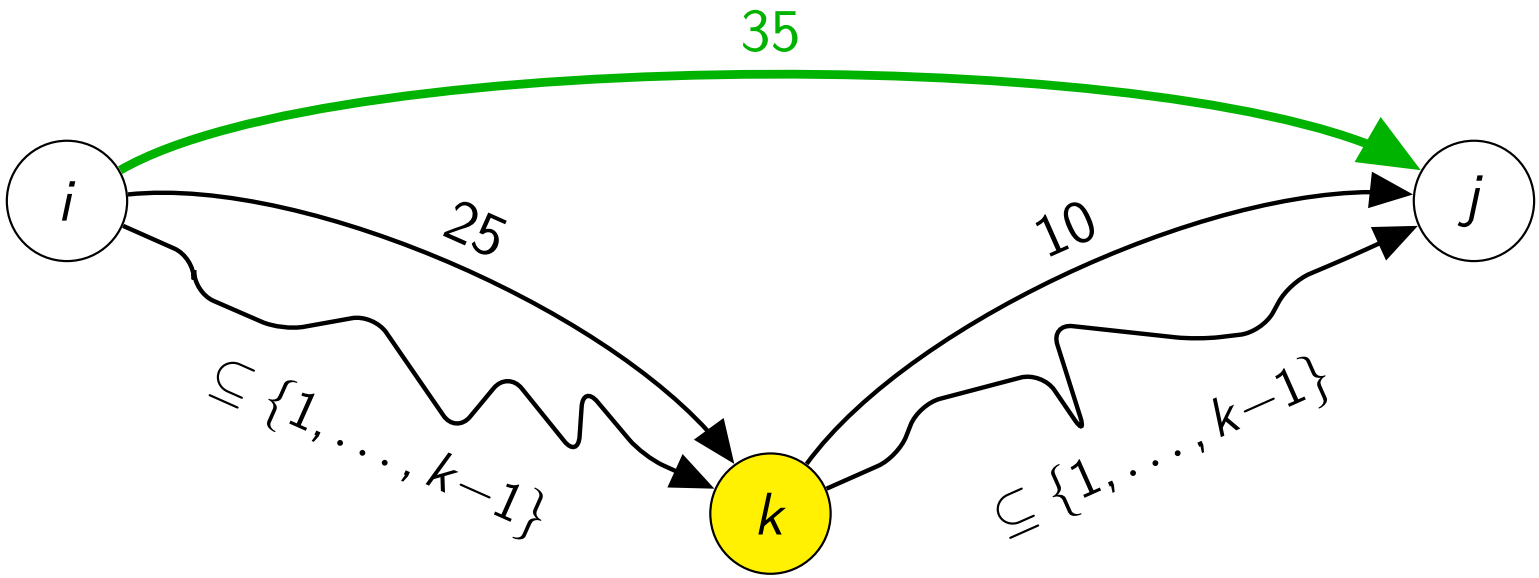
Der Algorithmus von Floyd – Idee



Der Algorithmus von Floyd – Idee

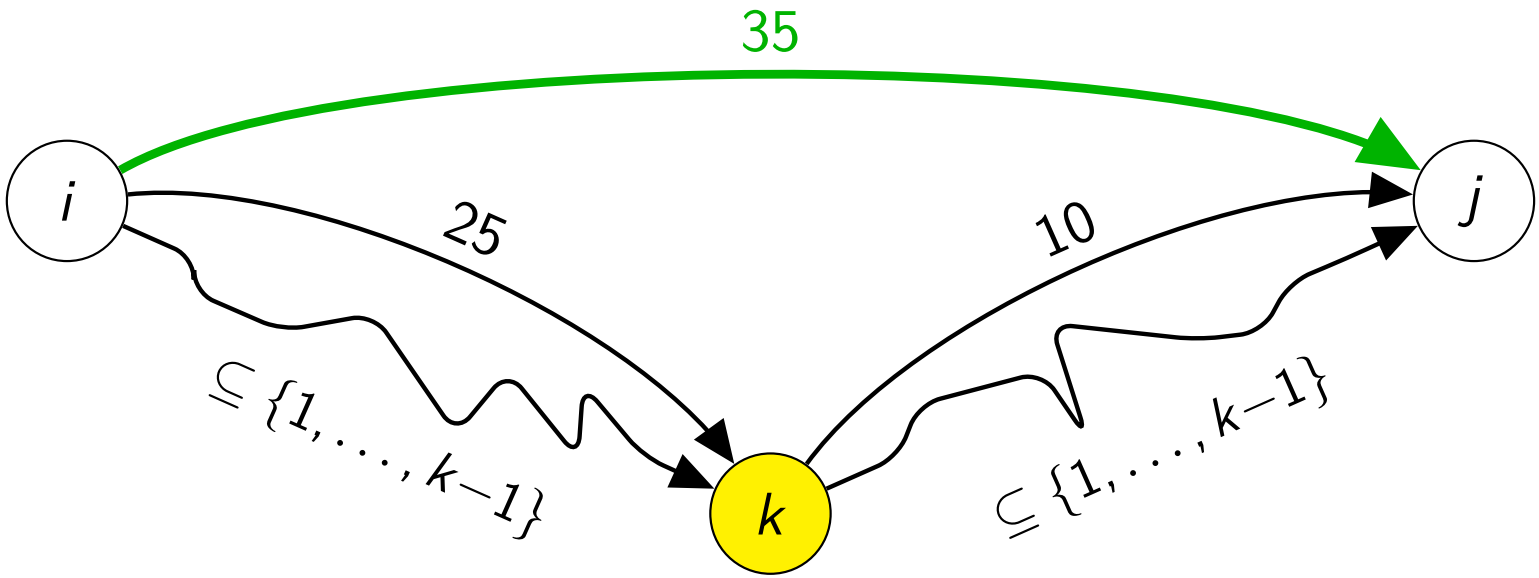


Der Algorithmus von Floyd – Idee



- Vorgehen wie bei Warshall, jedoch mit folgender Rekursionsgleichung:

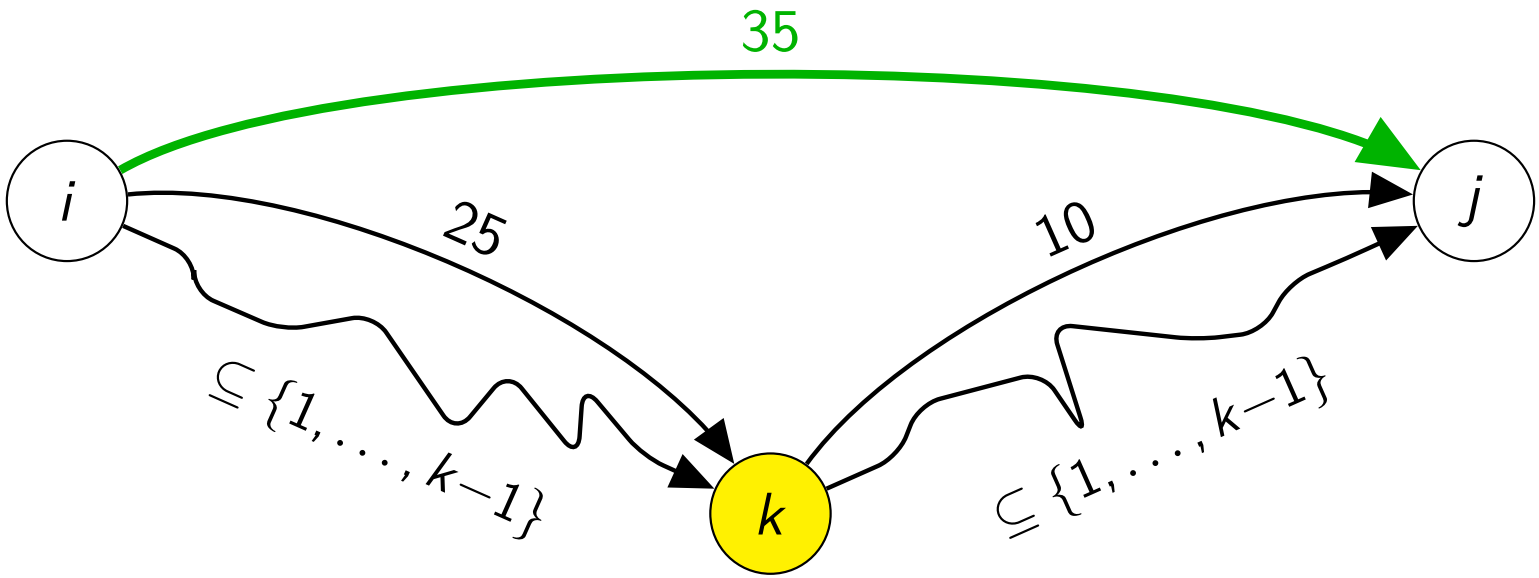
Der Algorithmus von Floyd – Idee



- Vorgehen wie bei Warshall, jedoch mit folgender Rekursionsgleichung:

$$d_{ij}^{(k)} = \begin{cases} W(i,j) & \text{für } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{für } k > 0 \end{cases}$$

Der Algorithmus von Floyd – Idee

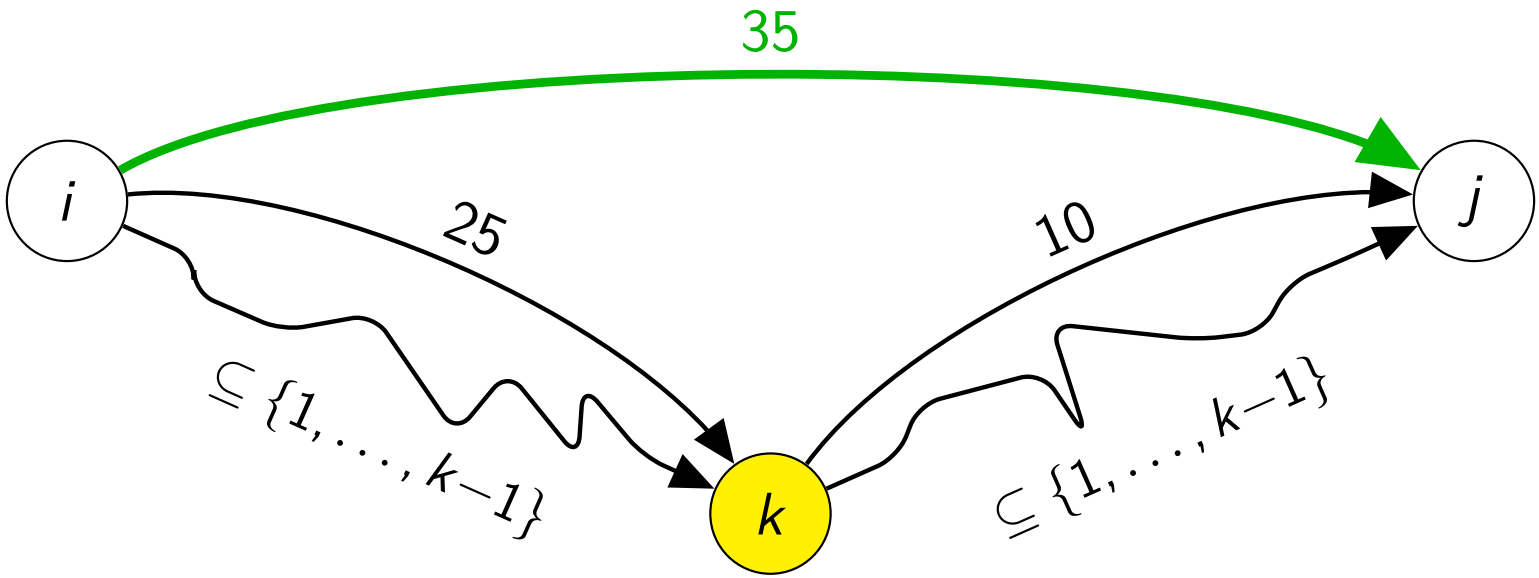


- Vorgehen wie bei Warshall, jedoch mit folgender Rekursionsgleichung:

$$d_{ij}^{(k)} = \begin{cases} W(i,j) & \text{für } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{für } k > 0 \end{cases}$$

$$\text{(statt: } t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)\text{)}$$

Der Algorithmus von Floyd – Idee



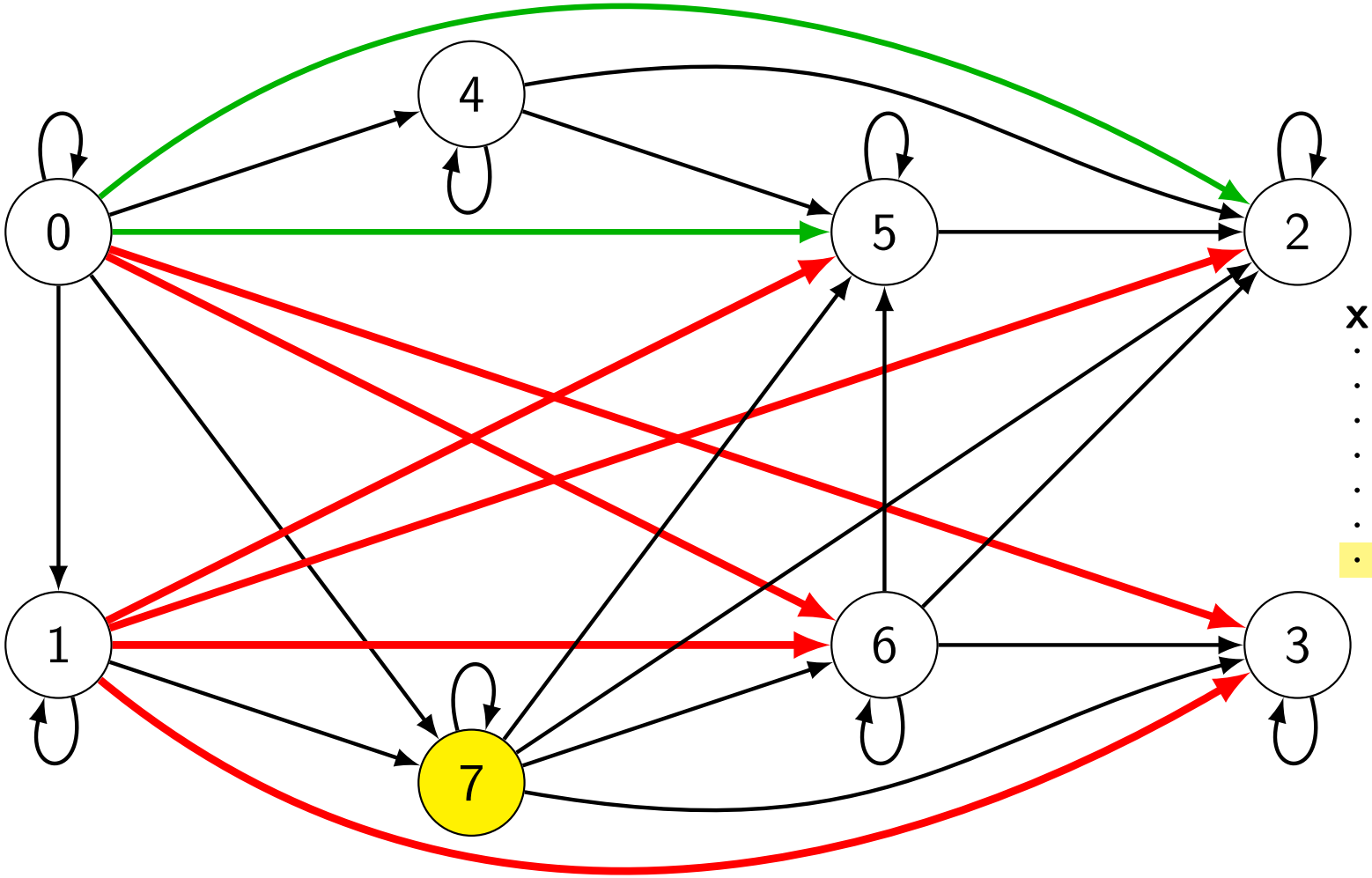
- Vorgehen wie bei Warshall, jedoch mit folgender Rekursionsgleichung:

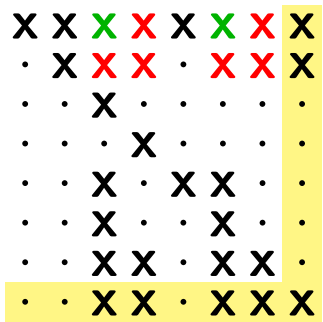
$$d_{ij}^{(k)} = \begin{cases} W(i,j) & \text{für } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{für } k > 0 \end{cases}$$

$$\text{(statt: } t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)\text{)}$$

- Auch hier arbeiten wir direkt im Ausgabearray: `D[i,j]` = $d_{ij}^{(\cdot)}$.

Der Algorithmus von Floyd – Beispiel





Der Algorithmus von Floyd – Implementierung

```
1  void floydSP(double W[][], int n, double &D[][]) {
2    for (int i = 0; i < n; i++)
3      for (int j = 0; j < n; j++)
4        D[i,j] = W[i,j]; // Kopiere W nach D
5
6    for (int i = 0; i < n; i++)
7      D[i,i] = 0;
8
9    for (int k = 0; k < n; k++)
10     for (int i = 0; i < n; i++)
11       for (int j = 0; j < n; j++)
12         D[i,j] = min(D[i,j], D[i,k] + D[k,j]);
13 }
```

Der Algorithmus von Floyd – Implementierung

```
1  void floydSP(double W[][], int n, double &D[][]) {
2    for (int i = 0; i < n; i++)
3      for (int j = 0; j < n; j++)
4        D[i,j] = W[i,j]; // Kopiere W nach D
5
6    for (int i = 0; i < n; i++)
7      D[i,i] = 0;
8
9    for (int k = 0; k < n; k++)
10     for (int i = 0; i < n; i++)
11       for (int j = 0; j < n; j++)
12         D[i,j] = min(D[i,j], D[i,k] + D[k,j]);
13 }
```

- Zeitkomplexität: $\Theta(n^3)$, Platzkomplexität: $\Theta(n^2)$.

Der Algorithmus von Floyd – Implementierung

```
1  void floydSP(double W[][], int n, double &D[][]) {
2    for (int i = 0; i < n; i++)
3      for (int j = 0; j < n; j++)
4        D[i,j] = W[i,j]; // Kopiere W nach D
5
6    for (int i = 0; i < n; i++)
7      D[i,i] = 0;
8
9    for (int k = 0; k < n; k++)
10     for (int i = 0; i < n; i++)
11       for (int j = 0; j < n; j++)
12         D[i,j] = min(D[i,j], D[i,k] + D[k,j]);
13 }
```

- Zeitkomplexität: $\Theta(n^3)$, Platzkomplexität: $\Theta(n^2)$.
- Hier nicht behandelt: Der Algorithmus kann auch mit negativen Zyklen umgehen.

Der Algorithmus von Floyd – Erweiterung

- Der Algorithmus lässt sich einfach auf die Rückgabe von Pfaden erweitern (z. B. Routingtabellen).

Der Algorithmus von Floyd – Erweiterung

- Der Algorithmus lässt sich einfach auf die Rückgabe von Pfaden erweitern (z. B. Routingtabellen).
- Dazu speichere zu jedem Paar i, j jeweils den letzten Zwischenknoten des kürzesten Pfads in π_{ij} (den Vorgänger von j).

Der Algorithmus von Floyd – Erweiterung

- Der Algorithmus lässt sich einfach auf die Rückgabe von Pfaden erweitern (z. B. Routingtabellen).
- Dazu speichere zu jedem Paar i, j jeweils den letzten Zwischenknoten des kürzesten Pfads in π_{ij} (den Vorgänger von j).

$$\pi_{ij}^{(k)} = \begin{cases} i & \text{für } k = 0, i \neq j, \text{ falls } W(i,j) \neq \infty \end{cases}$$

Der Algorithmus von Floyd – Erweiterung

- Der Algorithmus lässt sich einfach auf die Rückgabe von Pfaden erweitern (z. B. Routingtabellen).
- Dazu speichere zu jedem Paar i, j jeweils den letzten Zwischenknoten des kürzesten Pfads in π_{ij} (den Vorgänger von j).

$$\pi_{ij}^{(k)} = \begin{cases} i & \text{für } k = 0, i \neq j, \text{ falls } W(i,j) \neq \infty \\ \texttt{null} & \text{für } k = 0, \text{ sonst} \end{cases}$$

Der Algorithmus von Floyd – Erweiterung

- Der Algorithmus lässt sich einfach auf die Rückgabe von Pfaden erweitern (z. B. Routingtabellen).
- Dazu speichere zu jedem Paar i, j jeweils den letzten Zwischenknoten des kürzesten Pfads in π_{ij} (den Vorgänger von j).

$$\pi_{ij}^{(k)} = \begin{cases} i & \text{für } k = 0, i \neq j, \text{ falls } W(i,j) \neq \infty \\ \texttt{null} & \text{für } k = 0, \text{ sonst} \\ \pi_{kj}^{(k-1)} & \text{für } k > 0, \text{ falls } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Der Algorithmus von Floyd – Erweiterung

- Der Algorithmus lässt sich einfach auf die Rückgabe von Pfaden erweitern (z. B. Routingtabellen).
- Dazu speichere zu jedem Paar i, j jeweils den letzten Zwischenknoten des kürzesten Pfads in π_{ij} (den Vorgänger von j).

$$\pi_{ij}^{(k)} = \begin{cases} i & \text{für } k = 0, i \neq j, \text{ falls } W(i,j) \neq \infty \\ \texttt{null} & \text{für } k = 0, \text{ sonst} \\ \pi_{kj}^{(k-1)} & \text{für } k > 0, \text{ falls } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{ij}^{(k-1)} & \text{sonst} \end{cases}$$