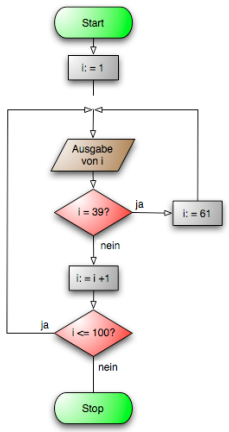
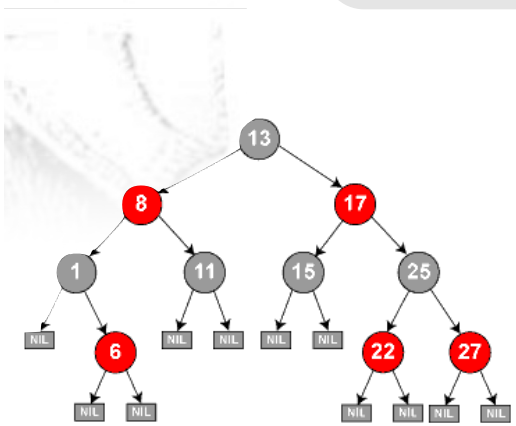
Algorithmen und Datenstrukturen





Version 1.31 vom 9. Juni 2024

FH AACHEN UNIVERSITY OF APPLIED SCIENCES

Fachhochschule Aachen

Lehrgebiet für Angewandte Informatik

Prof. Dr. rer. nat. Jörg Striegnitz

VII. Graphen

7. Graphen

7.1. Einführung und Begriffsbestimmungen
7.2. Traversierung
7.3. Topologisches Sortieren
7.4. Kürzeste Wege
  7.4.1. Einführung
  7.4.2. *Single-Source Best Path* - Algorithmen
  7.4.3. *All-Pairs Best Path* - Algorithmen
7.5. Maximaler Fluss
7.6. Minimaler Spannbaum

© 2006-2023 Jörg Striegnitz

VII. Graphen

7. Graphen

7.1. Einführung und Begriffsbestimmungen
7.2. Traversierung
7.3. Topologisches Sortieren
7.4. Kürzeste Wege
7.4.1. Einführung
7.4.2. *Single-Source Best Path* Algorithmen
7.4.3. *All-Pairs Best Path* Algorithmen
7.5. Maximaler Fluss
7.6. Minimaler Spannbaum

© 2006-2023 Jörg Striegnitz

Graph: Definition

D Gerichteter und ungerichteter Graph

Ein **Graph** $G = (V, E)$ besteht aus einer endlichen Knotenmenge V, sowie einer Relation $E \subseteq V \times V$ - der Kantenmenge. Betrachtet man unsortierte Paare (E ist symmetrisch: also $(v_1, v_2) \in E \Leftrightarrow (v_2, v_1) \in E$), so heißt G **ungerichteter Graph**, ansonsten **gerichteter Graph**.

© 2006-2024 Jörg Striegnitz

Graph: Definition

D **Gerichteter und ungerichteter Graph**

Ein **Graph** $G = (V, E)$ besteht aus einer endlichen Knotenmenge V, sowie einer Relation $E \subseteq V \times V$ - der Kantenmenge. Betrachtet man unsortierte Paare (E ist symmetrisch: also $(v_1, v_2) \in E \Leftrightarrow (v_2, v_1) \in E$), so heißt G **ungerichteter Graph**, ansonsten **gerichteter Graph**.

- Die Bezeichnungen der Mengen leiten sich aus dem Englischen ab
  - V - ***V****ertices* - die Knotenmenge (auch gebräuchlich: *node*)
  - E - ***E****dges* , die Kantenmenge

© 2006-2024 Jörg Striegnitz

Graph: Definition

D **Gerichteter und ungerichteter Graph**

Ein **Graph** $G = (V, E)$ besteht aus einer endlichen Knotenmenge V, sowie einer Relation $E \subseteq V \times V$ - der Kantenmenge. Betrachtet man unsortierte Paare (E ist symmetrisch: also $(v_1, v_2) \in E \Leftrightarrow (v_2, v_1) \in E$), so heißt G **ungerichteter Graph**, ansonsten **gerichteter Graph**.

- Die Bezeichnungen der Mengen leiten sich aus dem Englischen ab
  - V - ***V**ertices* - die Knotenmenge (auch gebräuchlich: *node*)
  - E - ***E**dges* , die Kantenmenge
- Wenn wir **nicht** zwischen gerichtetem und ungerichtetem Graph unterscheiden wollen, sprechen wir einfach von einem Graph

© 2006-2024 Jörg Striegnitz

Graph: Definition

D **Gerichteter und ungerichteter Graph**

Ein **Graph** $G = (V, E)$ besteht aus einer endlichen Knotenmenge V, sowie einer Relation $E \subseteq V \times V$ - der Kantenmenge. Betrachtet man unsortierte Paare (E ist symmetrisch: also $(v_1, v_2) \in E \Leftrightarrow (v_2, v_1) \in E$), so heißt G **ungerichteter Graph**, ansonsten **gerichteter Graph**.

- Die Bezeichnungen der Mengen leiten sich aus dem Englischen ab
  - V - ***V**ertices* - die Knotenmenge (auch gebräuchlich: *node*)
  - E - ***E**dges* , die Kantenmenge
- Wenn wir **nicht** zwischen gerichtetem und ungerichtetem Graph unterscheiden wollen, sprechen wir einfach von einem Graph
- Sei G=(V,E)
  - mit $|V|$ bezeichnen wir **Knotenzahl**
  - mit $|E|$ bezeichnen wir die **Kantenzahl**; offenbar gilt:
    - gerichteter Graph: $|E| \leq |V|^2$
    - ungerichteter Graph: $|E| \leq |V| \cdot (|V|-1) / 2$

© 2006-2024 Jörg Striegnitz

Bildliche Darstellung von Graphen

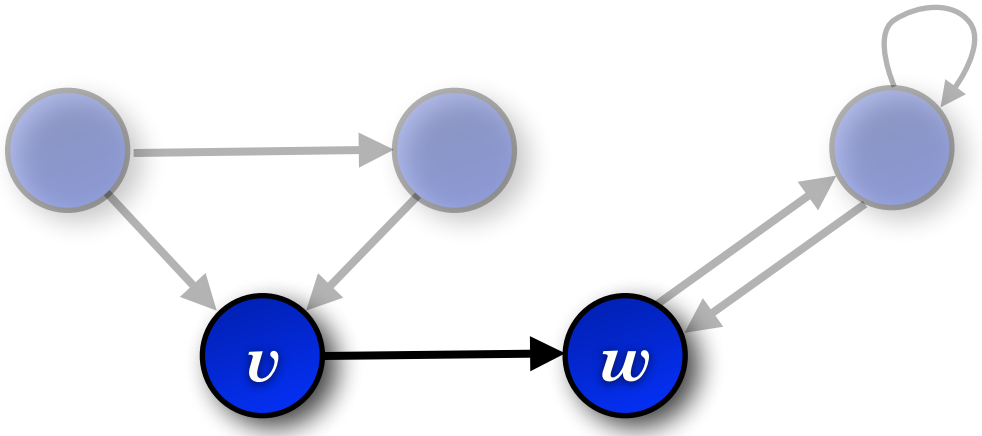
- **Graphen werden ähnlich wie Bäume dargestellt (Anmerkung:** Auch Bäume sind Graphen!**):**

© 2006-2024 Jörg Striegnitz

Bildliche Darstellung von Graphen

- **Graphen werden ähnlich wie Bäume dargestellt (Anmerkung:** Auch Bäume sind Graphen!**):**

Gerichteter Graph



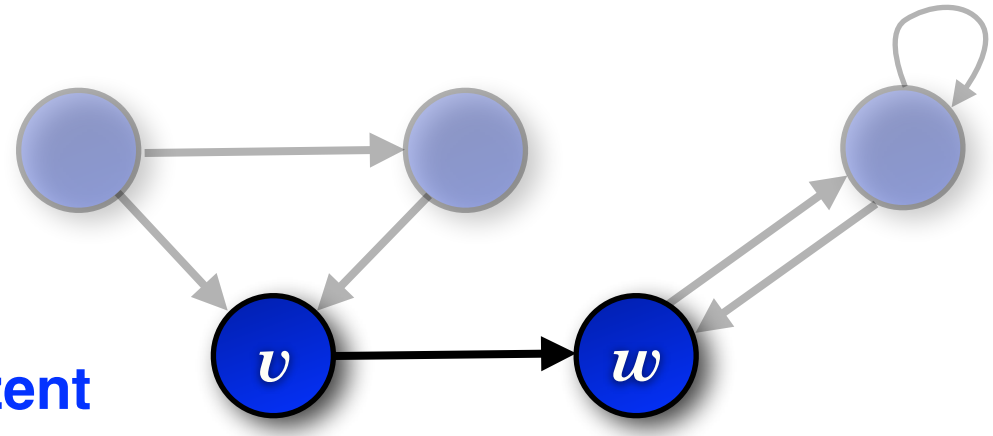
© 2006-2024 Jörg Striegnitz

Bildliche Darstellung von Graphen

- **Graphen werden ähnlich wie Bäume dargestellt (Anmerkung:** Auch Bäume sind Graphen!**):**

Gerichteter Graph

- v ist **Vorgänger** von w
- w ist **Nachfolger** von v
- v und w sind **Nachbarn** bzw. **adjazent**



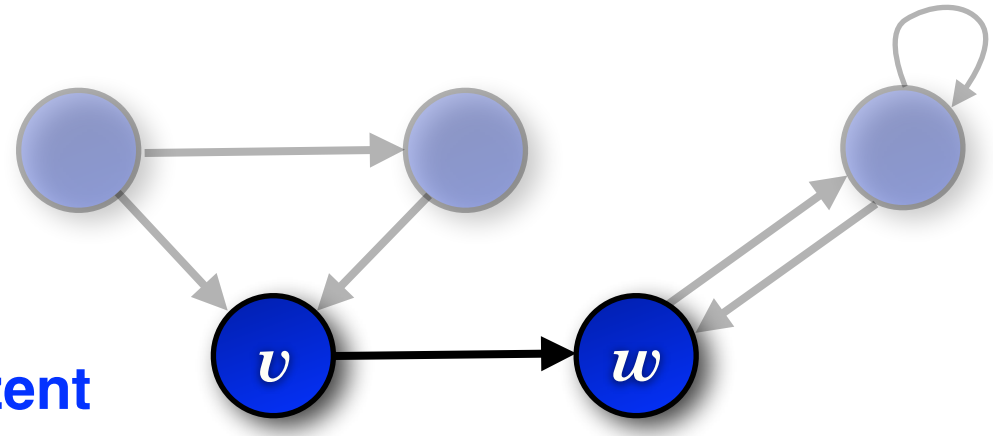
© 2006-2024 Jörg Striegnitz

Bildliche Darstellung von Graphen

- **Graphen werden ähnlich wie Bäume dargestellt (Anmerkung:** Auch Bäume sind Graphen!**):**

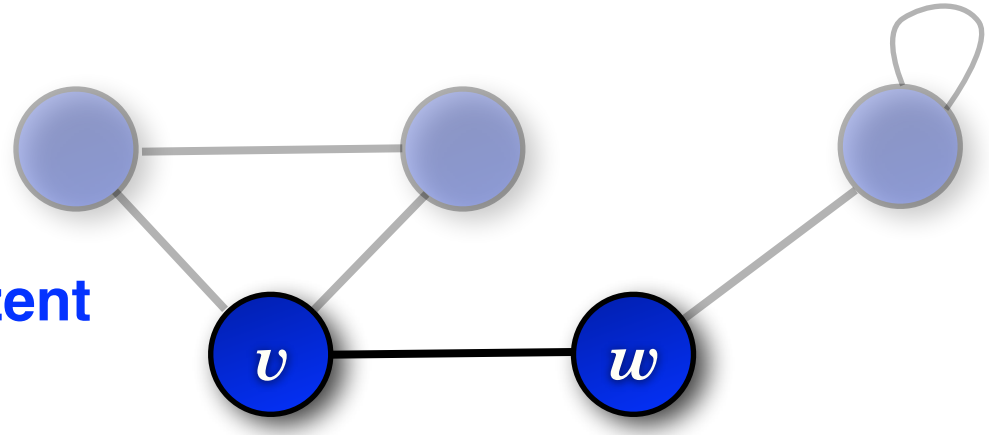
Gerichteter Graph

- v ist **Vorgänger** von w
- w ist **Nachfolger** von v
- v und w sind **Nachbarn** bzw. **adjazent**



Ungerichteter Graph

- v und w sind **Nachbarn** bzw. **adjazent**



© 2006-2024 Jörg Striegnitz

Gewichtete Graphen

- **Man kann Markierungen oder Beschriftungen für Kanten einführen**

D **Gewichteter Graph**

Ein **gewichteter Graph** ist ein Graph $G = (V, E)$ zusammen mit einer totalen Gewichtsfunktion weight : $E \rightarrow R$, wobei R eine beliebige Menge (z.B. die reellen Zahlen $\mathbb{R}$) ist.

© 2006-2024 Jörg Striegnitz

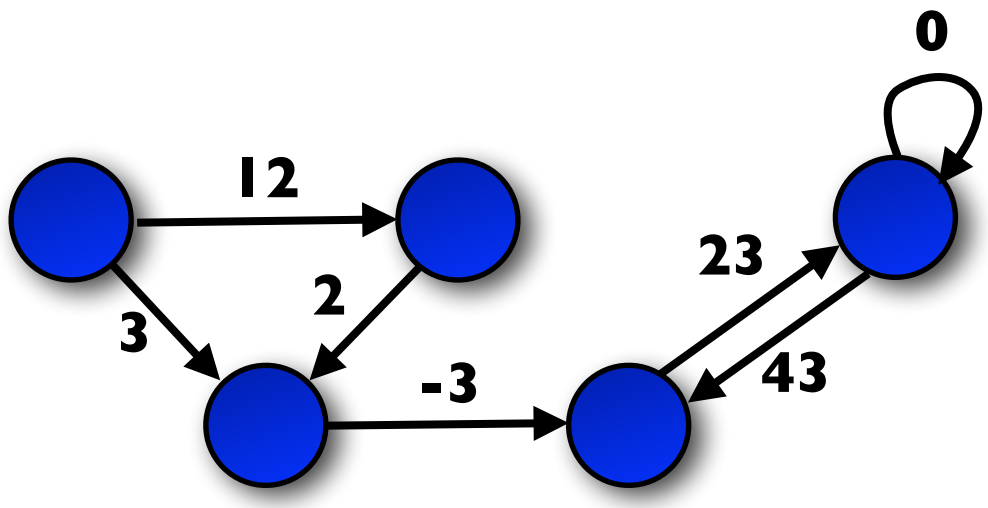
Gewichtete Graphen

- **Man kann Markierungen oder Beschriftungen für Kanten einführen**

D **Gewichteter Graph**

Ein **gewichteter Graph** ist ein Graph $G = (V, E)$ zusammen mit einer totalen Gewichtsfunktion weight : $E \rightarrow R$, wobei R eine beliebige Menge (z.B. die reellen Zahlen $\mathbb{R}$) ist.

- **In der bildlichen Darstellung:**



© 2006-2024 Jörg Striegnitz

Graphen - Wozu?

- **Zahlreiche praktische Fragestellungen lassen sich durch Graphen darstellen; z.B.**
  - Beziehungen zwischen Personen
    - Person A kennt Person B
    - Person A spielt gegen Person B
  - Verbindungen zwischen Punkten
    - Straßennetz
    - Eisenbahnnetz
    - Telefonnetz
    - Internet
    - elektronische Schaltkreise

© 2006-2024 Jörg Striegnitz

Graphen - Wozu?

- **Zahlreiche praktische Fragestellungen lassen sich durch Graphen darstellen; z.B.**
  - Beziehungen zwischen Personen
    - Person A kennt Person B
    - Person A spielt gegen Person B
  - Verbindungen zwischen Punkten
    - Straßennetz
    - Eisenbahnnetz
    - Telefonnetz
    - Internet
    - elektronische Schaltkreise
- **Typische Fragestellungen**
  - Existiert eine Verbindung von A nach B
  - Existiert eine zweite Verbindung, falls diese blockiert ist?
  - Was ist die kürzeste Verbindung von A nach B
  - Minimaler Spannbaum
  - Optimale Rundreise (*Traveling Salesman Problem*)

© 2006-2024 Jörg Striegnitz

Graphen: Weitere Definitionen

D Knotengrad / Pfad / Teilgraph

Der **Grad eines Knotens** entspricht der Summe der ein- und ausgehenden Kanten.

© 2006-2024 Jörg Striegnitz

Graphen: Weitere Definitionen

D Knotengrad / Pfad / Teilgraph

Der **Grad eines Knotens** entspricht der Summe der ein- und ausgehenden Kanten.

Ein **Pfad** ist eine Folge von Knoten $\langle v_1, \ldots, v_n \rangle$ mit $(v_i, v_{i+1}) \in E$ für $1 \leq i < n$, also eine Folge zusammenhängender Kanten.

- Die **Pfadlänge** entspricht der Anzahl der Kanten auf dem Pfad.
- Ein Pfad heißt **einfacher Pfad**, wenn alle Knoten auf dem Pfad paarweise verschieden sind.
- Ein **Zyklus** ist ein Pfad $\langle v_1, \ldots v_n \rangle$ mit $v_1 = v_n$. Ist die Pfadlänge 1, so spricht man von einer **Schleife**.

© 2006-2024 Jörg Striegnitz

Graphen: Weitere Definitionen

D Knotengrad / Pfad / Teilgraph

Der **Grad eines Knotens** entspricht der Summe der ein- und ausgehenden Kanten.

Ein **Pfad** ist eine Folge von Knoten $\langle v_1, \ldots, v_n \rangle$ mit $(v_i, v_{i+1}) \in E$ für $1 \leq i < n$, also eine Folge zusammenhängender Kanten.

- Die **Pfadlänge** entspricht der Anzahl der Kanten auf dem Pfad.
- Ein Pfad heißt **einfacher Pfad**, wenn alle Knoten auf dem Pfad paarweise verschieden sind.
- Ein **Zyklus** ist ein Pfad $\langle v_1, \ldots v_n \rangle$ mit $v_1 = v_n$. Ist die Pfadlänge 1, so spricht man von einer **Schleife**.

Ein **Teilgraph** eines Graphen $G = (V, E)$ ist ein Graph $G' = (V', E')$ mit $V' \subseteq V$ und $E' \subseteq E \cap (V' \times V')$.

© 2006-2024 Jörg Striegnitz

D Knotengrad / Pfad / Teilgraph

Der **Grad eines Knotens** entspricht der Summe der ein- und ausgehenden Kanten.

Ein **Pfad** ist eine Folge von Knoten $\langle v_1, \ldots, v_n \rangle$ mit $(v_i, v_{i+1}) \in E$ für $1 \leq i < n$, also eine Folge zusammenhängender Kanten.

- Die **Pfadlänge** entspricht der Anzahl der Kanten auf dem Pfad.
- Ein Pfad heißt **einfacher Pfad**, wenn alle Knoten auf dem Pfad paarweise verschieden sind.
- Ein **Zyklus** ist ein Pfad $\langle v_1, \ldots v_n \rangle$ mit $v_1 = v_n$. Ist die Pfadlänge 1, so spricht man von einer **Schleife**.

Ein **Teilgraph** eines Graphen $G = (V, E)$ ist ein Graph $G' = (V', E')$ mit $V' \subseteq V$ und $E' \subseteq E \cap (V' \times V')$.

Sei $G = (V, E)$ ein Graph und $V' \subseteq V$. Der **von V' induzierte Teilgraph** $G[V'] = (V', E')$ von G, ist ein Teilgraph von G mit $E' = E \cap (V' \times V')$.

© 2006-2024 Jörg Striegnitz

Graph-Darstellungen

- **Die bildliche Darstellung ist anschaulich, aber ungeeignet, um Graphen in einem Computer darzustellen**
- **Je nach Zielsetzung**
  - **Knotenorientierte Speicherung**
    - **Adjazenzmatrix**
    - **Adjazenzliste**
  - **Kantenorientierte Speicherung**
    - **Inzidenzmatrix**
- **Vereinfachung:** identifiziere Knoten mit natürlichen Zahlen
- **Sei daher** $V=\{1,2,\ldots,|V|\}$

© 2006-2024 Jörg Striegnitz

JAVA-`interface` für Graphen (exemplarisch) I

```
// Skizze einer Schnittstelle zu einem Graphen.
// Sehr unvollständig, da z.B. keine Knoten hinzugefügt oder gelöscht
// werden können!
public interface Graph<V> {
  public int numNodes();            // Anzahl der Knoten im Graph

  public V getNode(int id);         // Knoten "id" holen
  public void setNode(int id,V n); // Knoten "id" setzen

  public void addEdge(int v1,int v2);    // Kante hinzufügen
  public void removeEdge(int v1,int v2); // Kante löschen

  public boolean connected(int v1,int v2); // Test: Knoten verbunden?
  // Intern werden Knoten als "NodeWithId" gespeichert. Wiederholte
  // Aufrufe dieser Methode sollten zu einer id immer dasselbe Objekt
  // liefern!
  public NodeWithId<V> getNodeWithId(int id); // Paar aus Knoten und
                                              // id holen
}
```


© 2006-2024 Jörg Striegnitz

JAVA-`interface` für Graphen (exemplarisch) II

- **Intern werden Knoten als Paar aus**
  - Knotenwert und
  - Knoten-Id

  gespeichert.

```
public class NodeWithId<V> {
  public V vertex; // Der eigentliche Knoten
  public int id;   // id des Knotens
  public NodeWithId(V v,int i) { vertex=v; id=i; }
}
```


© 2006-2024 Jörg Striegnitz

Adjazenzmatrix I

- **Sei $G=(V,E)$ ein Graph mit $|V|=n$**

© 2006-2024 Jörg Striegnitz

Adjazenzmatrix I

- **Sei G=(V,E) ein Graph mit $|V|=n$**
- **Die Adjazenzmatrix zu G ist eine $n\times n$-Matrix $A=(a_{ij})$ mit**

$$a_{ij} = \begin{cases} 1 & \text{falls } (v_i, v_j) \in E \\ 0 & \text{sonst} \end{cases}$$

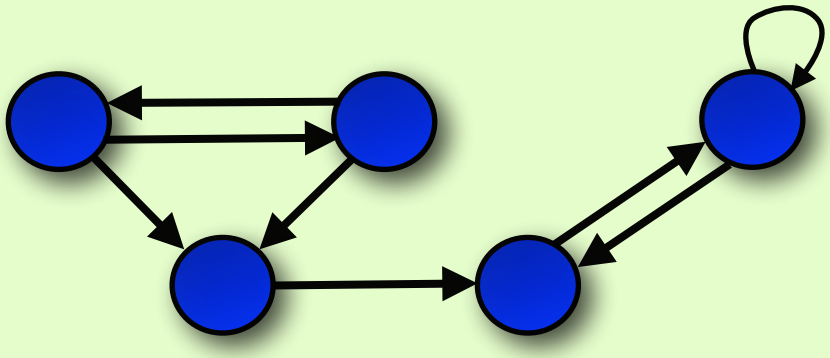
© 2006-2024 Jörg Striegnitz

Adjazenzmatrix I

- **Sei G=(V,E) ein Graph mit $|V|=n$**
- **Die Adjazenzmatrix zu G ist eine $n\times n$-Matrix $A=(a_{ij})$ mit**

$$a_{ij} = \begin{cases} 1 & \text{falls } (v_i, v_j) \in E \\ 0 & \text{sonst} \end{cases}$$

B Adjazenzmatrix



		nach				
		1	2	3	4	5
von	1	0	1	0	1	0
	2	1	0	0	1	0
	3	0	0	1	0	1
	4	0	0	0	0	1
	5	0	0	1	0	0

Beobachtung: Für ungerichtete Graphen ist die Adjazenzmatrix offenbar **symmetrisch**

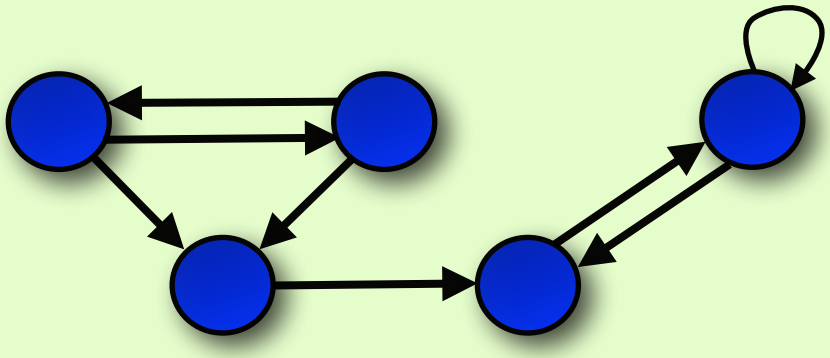
© 2006-2024 Jörg Striegnitz

Adjazenzmatrix I

- **Sei G=(V,E) ein Graph mit $|V|=n$**
- **Die Adjazenzmatrix zu G ist eine $n\times n$-Matrix $A=(a_{ij})$ mit**

$$a_{ij} = \begin{cases} 1 & \text{falls } (v_i, v_j) \in E \\ 0 & \text{sonst} \end{cases}$$

B Adjazenzmatrix



		nach				
		1	2	3	4	5
von	1	0	1	0	1	0
	2	1	0	0	1	0
	3	0	0	1	0	1
	4	0	0	0	0	1
	5	0	0	1	0	0

Beobachtung: Für ungerichtete Graphen ist die Adjazenzmatrix offenbar **symmetrisch**

- **Eine Matrix lässt sich leicht durch ein zweidimensionales Feld darstellen**

© 2006-2024 Jörg Striegnitz

Adjazenzmatrix II

- **Vorteile**

© 2006-2024 Jörg Striegnitz

Adjazenzmatrix II

- **Vorteile**
  - ✓ Entscheidung, ob $(v_i, v_j) \in E$ in Zeit $O(1)$
  - ✓ Ansatz erweiterbar auf gewichtete Graphen
    Zusätzliche Matrix, in die Wert der Kostenfunktion eingetragen wird

© 2006-2024 Jörg Striegnitz

Adjazenzmatrix II

- **Vorteile**
  - ✓ Entscheidung, ob $(v_i, v_j) \in E$ in Zeit $O(1)$
  - ✓ Ansatz erweiterbar auf gewichtete Graphen
    Zusätzliche Matrix, in die Wert der Kostenfunktion eingetragen wird
- **Nachteile**

© 2006-2024 Jörg Striegnitz

Adjazenzmatrix II

- **Vorteile**
  - ✓ Entscheidung, ob $(v_i, v_j) \in E$ in Zeit $O(1)$
  - ✓ Ansatz erweiterbar auf gewichtete Graphen
    Zusätzliche Matrix, in die Wert der Kostenfunktion eingetragen wird
- **Nachteile**
  - ✗ Platzbedarf ist stets $O(|V|^2)$ - ineffizient falls $|E| \ll |V|^2$
  - ✗ Initialisierung benötigt Zeit $O(|V|^2)$

© 2006-2024 Jörg Striegnitz

Adjazenzmatrix: Implementierung in JAVA I

```
import java.util.ArrayList;

public class AMGraph<V> implements Graph<V> {
  boolean AdjMatrix[][]; // Adjazenzmatrix
  ArrayList<NodeWithId<V> > nodes; // Liste der Knoten
  int nodeCount;                   // Anzahl der Knoten

  // Nicht gut: Anzahl der Knoten muss zu Beginn feststehen
  public AMGraph(int n) { nodeCount=n ;
                          AdjMatrix = new boolean[n][n];
                          nodes = new ArrayList<V>(n);
                          for (int i=0 ; i<n ; ++i)
                            nodes.add(new NodeWithId<V>(null,i) );
                        }

  // Manipulation von Knoten
  public void setNode(int id,V n) { nodes.get(id).vertex = n; }
  public V getNode(int id)        { return nodes.get(id).vertex; }
```

... Fortsetzung auf nächster Folie ...

© 2006-2024 Jörg Striegnitz

Adjazenzmatrix: Implementierung in JAVA II

... siehe Vorgängerfolie

```
  // Manipulation von Kanten
  public void addEdge(int v1, int v2) {
    AdjMatrix[v1][v2]=true;
  }
  public void removeEdge(int v1, int v2) {
    AdjMatrix[v1][v2]=false;
  }

  // Test, ob zwei Knoten verbunden sind
  public boolean connected(int v1, int v2) {
    return AdjMatrix[v1][v2];
  }

  // Anzahl der Knoten im Graph
  public int getNodeCount() { return nodeCount;  }

}
```


© 2006-2024 Jörg Striegnitz

Adjazenzliste

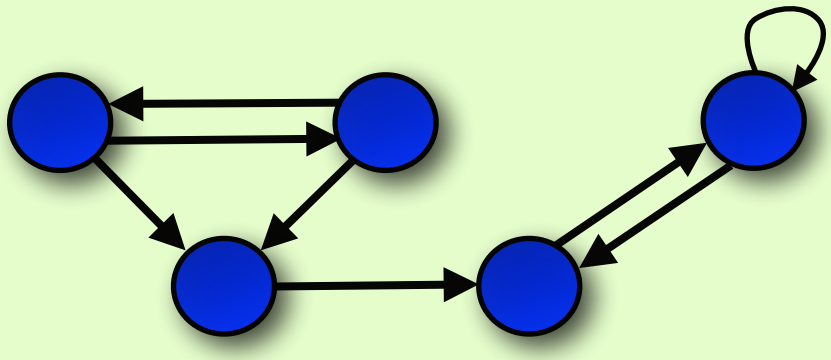
- **In einer Adjazenzliste wird für jeden Knoten eine Liste seiner Nachbarknoten angelegt:**

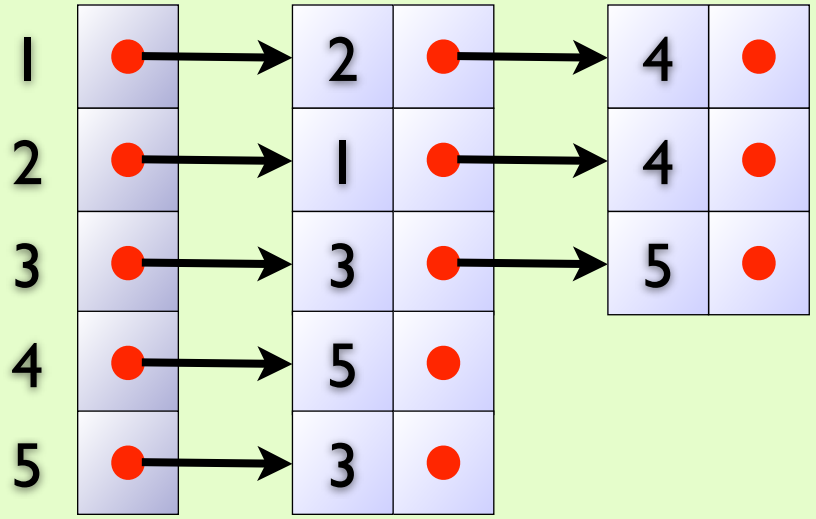
© 2006-2024 Jörg Striegnitz

Adjazenzliste

- **In einer Adjazenzliste wird für jeden Knoten eine Liste seiner Nachbarknoten angelegt:**

B Adjazenzliste

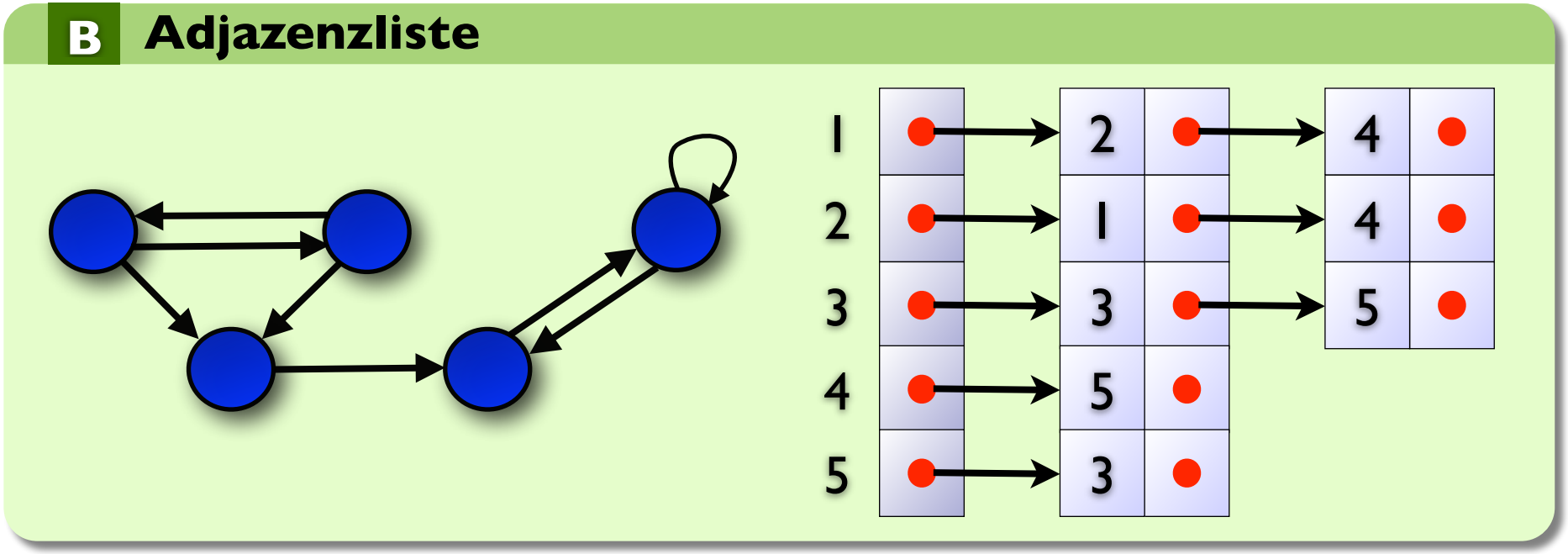




© 2006-2024 Jörg Striegnitz

Adjazenzliste

- **In einer Adjazenzliste wird für jeden Knoten eine Liste seiner Nachbarknoten angelegt:**

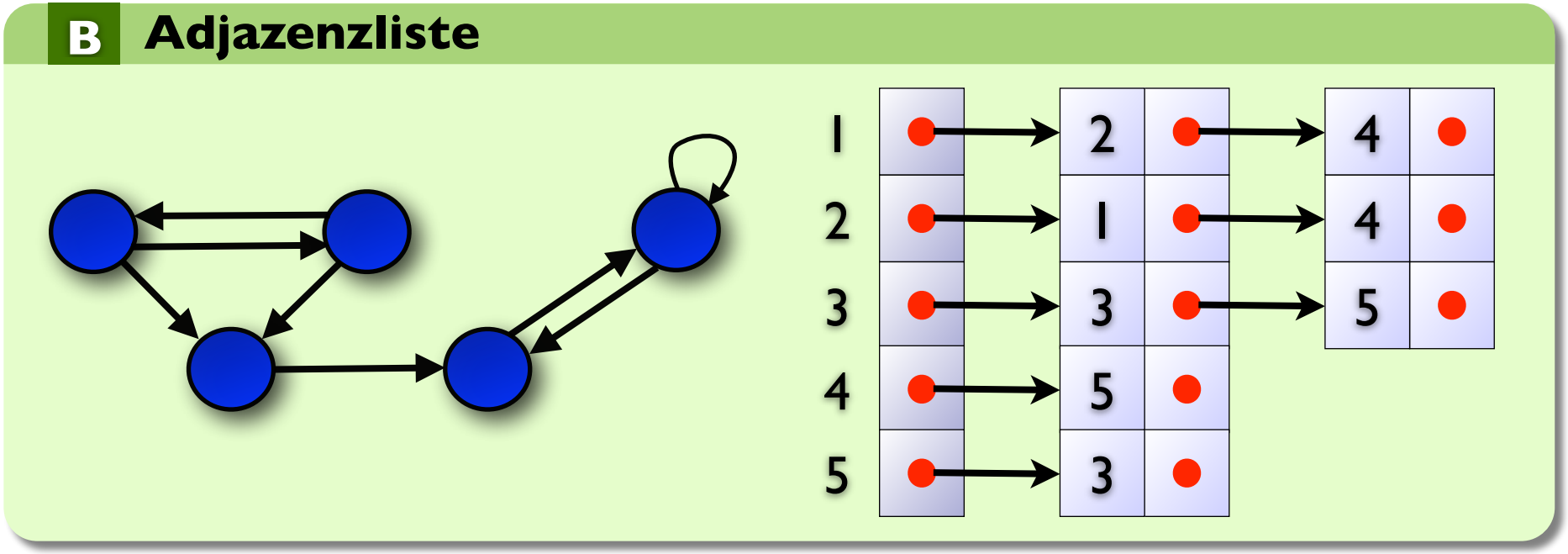


- **Vorteile**

© 2006-2024 Jörg Striegnitz

Adjazenzliste

- **In einer Adjazenzliste wird für jeden Knoten eine Liste seiner Nachbarknoten angelegt:**



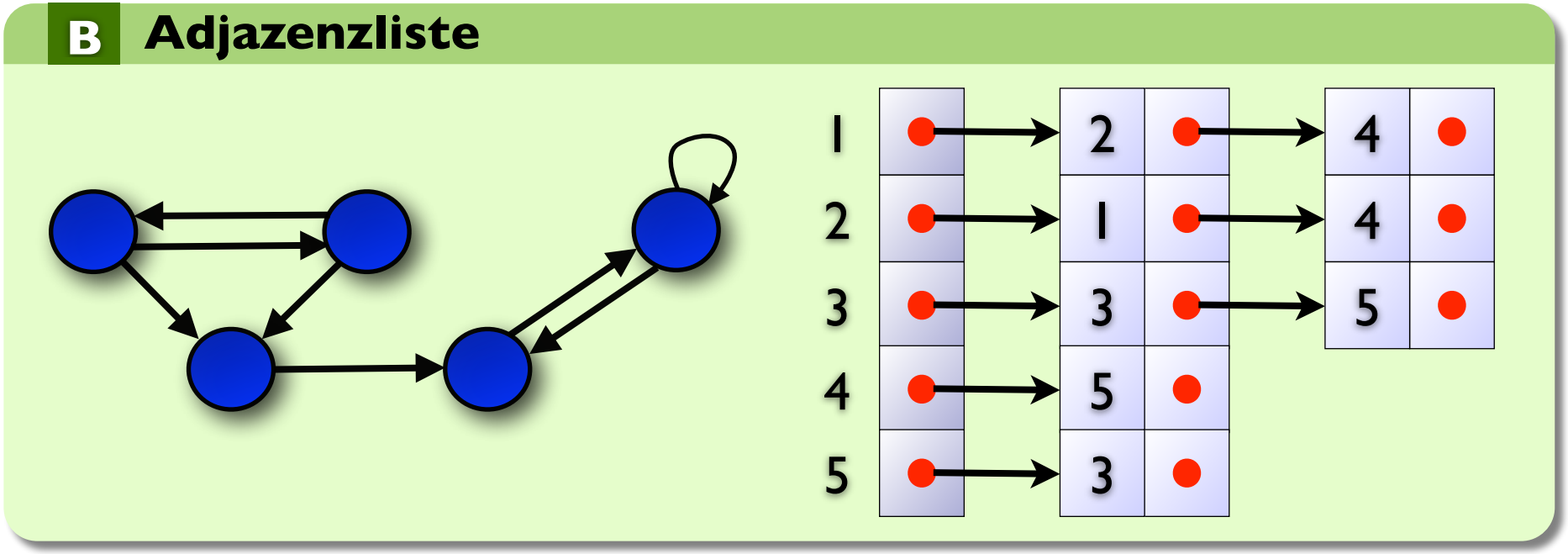
- **Vorteile**

✓ geringer Platzbedarf von $O(|V|+|E|)$

© 2006-2024 Jörg Striegnitz

Adjazenzliste

- **In einer Adjazenzliste wird für jeden Knoten eine Liste seiner Nachbarknoten angelegt:**

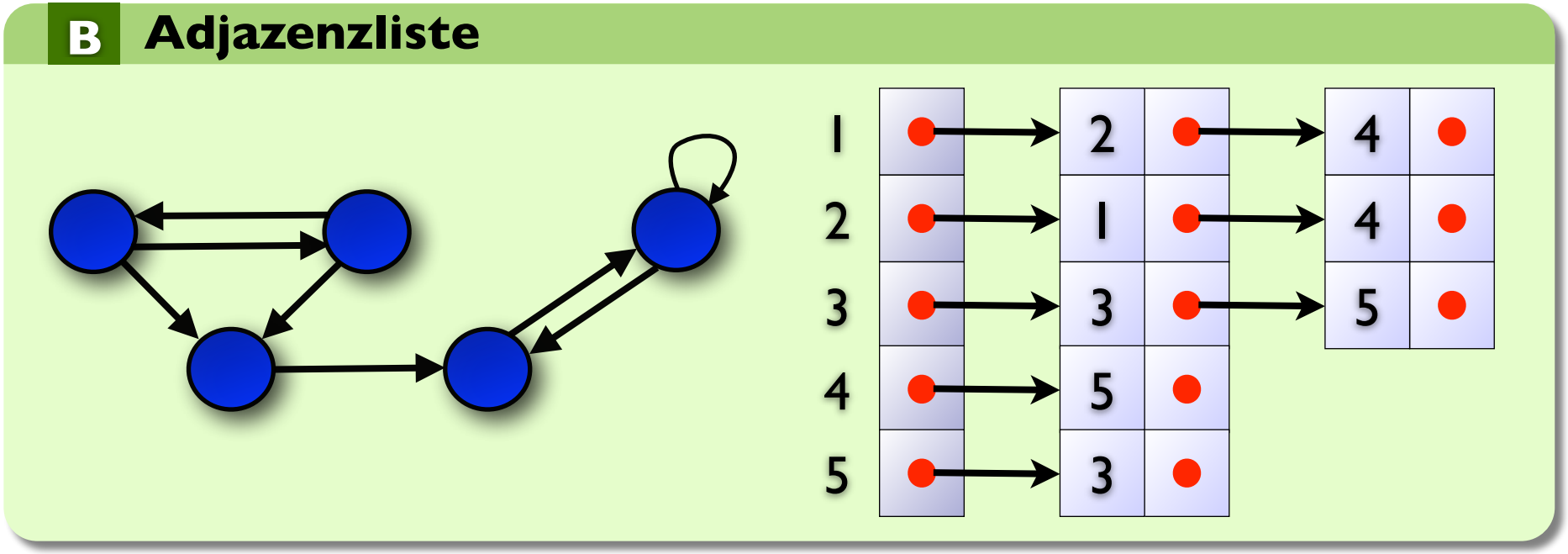


- **Vorteile**
  - ✓ geringer Platzbedarf von $O(|V|+|E|)$
  - ✓ Hinzufügen von Knoten einfacher

© 2006-2024 Jörg Striegnitz

Adjazenzliste

- **In einer Adjazenzliste wird für jeden Knoten eine Liste seiner Nachbarknoten angelegt:**



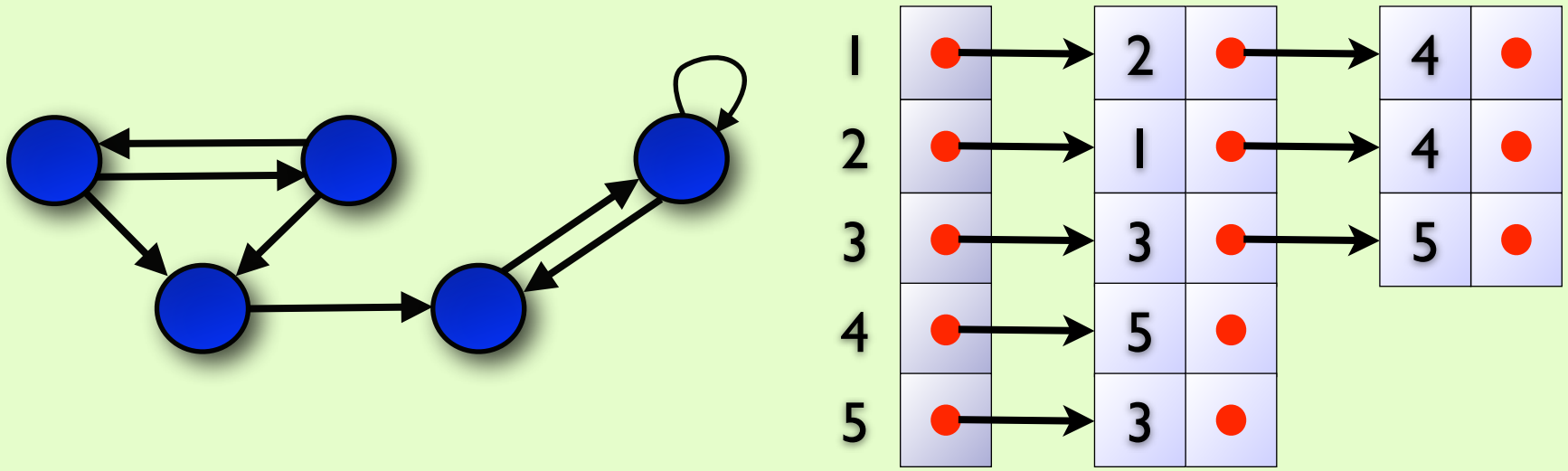
- **Vorteile**
  - ✓ geringer Platzbedarf von $O(|V|+|E|)$
  - ✓ Hinzufügen von Knoten einfacher
- **Nachteile**

© 2006-2024 Jörg Striegnitz

Adjazenzliste

- **In einer Adjazenzliste wird für jeden Knoten eine Liste seiner Nachbarknoten angelegt:**

B Adjazenzliste



- **Vorteile**

  ✓ geringer Platzbedarf von $O(|V|+|E|)$

  ✓ Hinzufügen von Knoten einfacher

- **Nachteile**

  ✗ Entscheidung, ob $(v_i,v_j)\in E$ in Zeit $O(|E|/|V|)$ im *average case*!

© 2006-2024 Jörg Striegnitz

Inzidenzmatrix

- **Sei G=(V,E) ein Graph mit $|V|$=n und $|E|$=m**
- **Die Inzidenzmatrix zu G ist eine n×m-Matrix B=(b_{ij}) mit**

$$b_{ij} = \begin{cases} 1 & \text{falls } v_i = v_a \\ 0 & \text{falls } v_i \notin \{b_a, v_b r\} \\ -1 & \text{falls } v_i = v_b \quad \wedge \quad v_i \neq v_a \end{cases}$$

Wobei $e_j = (v_a, v_b)$ die j-te Kante ist. Für ungerichtete Graphen gilt

$$b_{ij} = \begin{cases} 1 & \text{falls } v_i = v_a \ \vee \ v_i = v_b \\ 0 & \text{sonst} \end{cases}$$

© 2006-2024 Jörg Striegnitz

Inzidenzmatrix

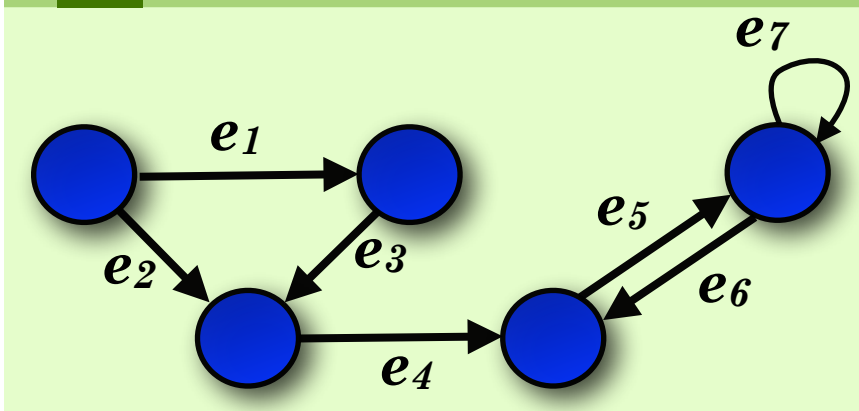
- **Sei G=(V,E) ein Graph mit $|V|=n$ und $|E|=m$**
- **Die Inzidenzmatrix zu G ist eine $n\times$m-Matrix B=(b_{ij}) mit**

$$b_{ij} = \begin{cases} 1 & \text{falls } v_i = v_a \\ 0 & \text{falls } v_i \notin \{b_a, v_b r\} \\ -1 & \text{falls } v_i = v_b \quad \wedge \quad v_i \neq v_a \end{cases}$$

Wobei $e_j = (v_a, v_b)$ die j-te Kante ist. Für ungerichtete Graphen gilt

$$b_{ij} = \begin{cases} 1 & \text{falls } v_i = v_a \ \vee \ v_i = v_b \\ 0 & \text{sonst} \end{cases}$$

B Inzidenzmatrix



		Kanten						
		e_1	e_2	e_3	e_4	e_5	e_6	e_7
Knoten	1	1	1	0	0	0	0	0
	2	-1	0	1	0	0	0	0
	3	0	0	0	0	-1	1	1
	4	0	-1	-1	1	0	0	0
	5	0	0	0	-1	1	-1	0

© 2006-2024 Jörg Striegnitz

Gewichtete Graphen: Implementierung in JAVA I

- **Um den Typ des Kantengewichts variabel halten zu können, formulieren Anforderungen in Form eines `interface`:**

```
// Die Gewichte der Kanten eines Graphen müssen
// bestimmte Eigenschaften aufweisen:
public interface Weight extends Comparable<Weight> {
  // Gewichte können miteinander vergleichen werden
  public int compareTo(Weight w);
  // Ein Gewicht darf unendlich groß sein
  public boolean isInfty();
  // Gewichte können einander zugewiesen werden
  public Weight assign(Weight rhs);
  // Gewichte können addiert und subtrahiert werden
  public Weight add(Weight w);
  public Weight subtract(Weight w);
}
```


© 2006-2024 Jörg Striegnitz

Gewichtete Graphen: Implementierung in JAVA II

- **Ein gewichteter Graph ist ein Graph:**

```
// Gewichtete Graphen verhalten sich wie Graphen ...
public interface WeightedGraph<V,R extends Weight>
                 extends Graph<V> {
  // ... zusätzlich kann das Gewicht einer Kante gesetzt
  // oder gelesen werden.
  public R getWeight(int v1,int v2);
  public void setWeight(int v1,int v2,Weight w);
}
```


© 2006-2024 Jörg Striegnitz

Typen von Graphen I

- **Man unterscheidet verschiedene Typen von Graphen:**

© 2006-2024 Jörg Striegnitz

Typen von Graphen I

- **Man unterscheidet verschiedene Typen von Graphen:**
  - **vollständiger Graph** (*complete Graph*)
    - Der Graph enthält jede mögliche Kante; also
    - $|E| = |V|^2$ für gerichtete Graphen
    - $|E| = |V| \cdot (|V| - 1)/2$ für ungerichtete Graphen

© 2006-2024 Jörg Striegnitz

Typen von Graphen I

- **Man unterscheidet verschiedene Typen von Graphen:**
  - **vollständiger Graph** (*complete Graph*)
    - Der Graph enthält jede mögliche Kante; also
    - $|E| = |V|^2$ für gerichtete Graphen
    - $|E| = |V| \cdot (|V| - 1)/2$ für ungerichtete Graphen
  - **dichter Graph** (*dense Graph*)
    - $|E| \approx |V|^2$ für gerichtete Graphen
    - $|E| \approx |V| \cdot (|V| - 1)/2$ für ungerichtete Graphen

© 2006-2024 Jörg Striegnitz

Typen von Graphen I

- **Man unterscheidet verschiedene Typen von Graphen:**
  - **vollständiger Graph** (*complete Graph*)
    - Der Graph enthält jede mögliche Kante; also
    - $|E| = |V|^2$ für gerichtete Graphen
    - $|E| = |V| \cdot (|V| - 1)/2$ für ungerichtete Graphen
  - **dichter Graph** (*dense Graph*)
    - $|E| \approx |V|^2$ für gerichtete Graphen
    - $|E| \approx |V| \cdot (|V| - 1)/2$ für ungerichtete Graphen
  - **dünner Graph** (*sparse Graph*)
    - $|E| \ll |V|^2$ für gerichtete Graphen
    - $|E| \ll |V| \cdot (|V| - 1)/2$ für ungerichtete Graphen

© 2006-2024 Jörg Striegnitz

Typen von Graphen I

- **Man unterscheidet verschiedene Typen von Graphen:**
  - **vollständiger Graph** (*complete Graph*)
    - Der Graph enthält jede mögliche Kante; also
    - $|E| = |V|^2$ für gerichtete Graphen
    - $|E| = |V| \cdot (|V| - 1)/2$ für ungerichtete Graphen
  - **dichter Graph** (*dense Graph*)
    - $|E| \approx |V|^2$ für gerichtete Graphen
    - $|E| \approx |V| \cdot (|V| - 1)/2$ für ungerichtete Graphen
  - **dünner Graph** (*sparse Graph*)
    - $|E| \ll |V|^2$ für gerichtete Graphen
    - $|E| \ll |V| \cdot (|V| - 1)/2$ für ungerichtete Graphen

- **Die Dichte eines Graphen ist Hauptkriterium bei der Wahl geeigneter Darstellungsformen**
  Ziel: optimale Speicherplatz und Zugriffseffizienz

© 2006-2024 Jörg Striegnitz

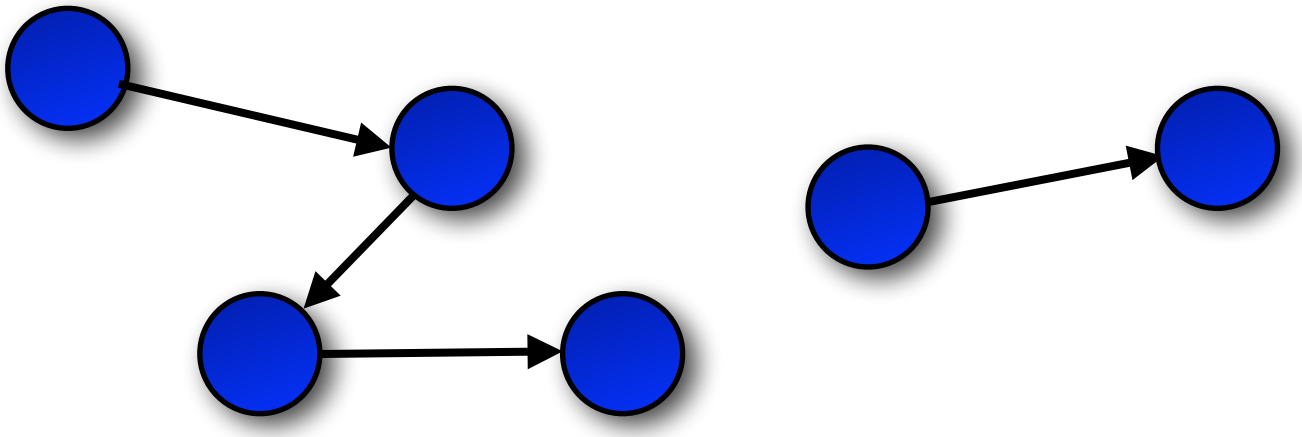
Typen von Graphen II

- **Die Definition eines Graphen lässt zu, dass nicht alle Knoten miteinander verbunden sein müssen:**

© 2006-2024 Jörg Striegnitz

Typen von Graphen II

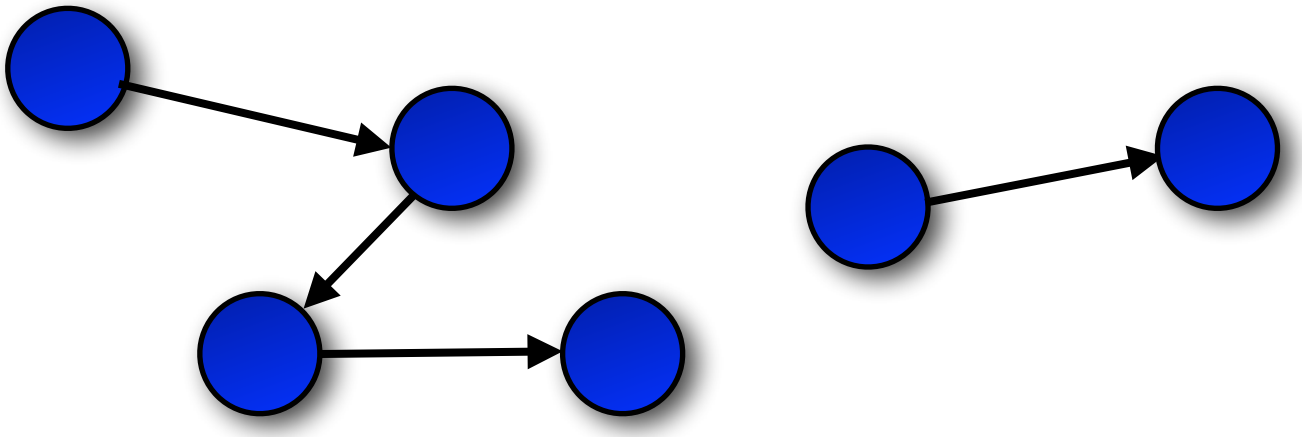
- **Die Definition eines Graphen lässt zu, dass nicht alle Knoten miteinander verbunden sein müssen:**



© 2006-2024 Jörg Striegnitz

Typen von Graphen II

- **Die Definition eines Graphen lässt zu, dass nicht alle Knoten miteinander verbunden sein müssen:**



D Zusammenhängender Graph

Sei $G = (V, E)$ ein Graph. Ist G ungerichtet, so heisst G **zusammenhängender Graph**, falls es für alle Paare $(v_i, v_j) \subseteq V \times V$ einen Pfad von v_i nach v_j gibt. Sei E_S der symmetrische Abschluss von E; d.h.

$$(v_1, v_2) \in E \Leftrightarrow (v_1, v_2) \in E_S \wedge (v_2, v_1) \in E_S$$

Ist G ein gerichteter Graph, so heisst er zusammenhängend, wenn der ungerichtete Graph $G = (V, E_S)$ zusammenhängend ist.

© 2006-2024 Jörg Striegnitz

VII. Graphen

7. Graphen

7.1. Einführung und Begriffsbestimmungen
7.2. Traversierung
7.3. Topologisches Sortieren
7.4. Kürzeste Wege
7.4.1. Einführung
7.4.2. *Single-Source Best Path* Algorithmen
7.4.3. *All-Pairs Best Path* Algorithmen
7.5. Maximaler Fluss
7.6. Minimaler Spannbaum

© 2006-2023 Jörg Striegnitz

Expansion eines Graphen

D Expansion eines Graphen

Sei $G = (V, E)$ ein Graph und $v \in V$. Die **Expansion** $X_G(v)$ von G in v ist ein Baum, der wie folgt definiert:

1. Falls v keine Nachfolger hat, besteht $X_G(v)$ nur aus dem Knoten v.
2. Falls $v_1, \ldots, v_k$ Nachfolger von v sind, ist $X_G(v)$ der Baum mit der Wurzel v und den Teilbäumen $X_G(v_1), \ldots, X_G(v_k)$.

© 2006-2024 Jörg Striegnitz

Expansion eines Graphen

D Expansion eines Graphen

Sei $G = (V, E)$ ein Graph und $v \in V$. Die **Expansion** $X_G(v)$ von G in v ist ein Baum, der wie folgt definiert:

1. Falls v keine Nachfolger hat, besteht $X_G(v)$ nur aus dem Knoten v.
2. Falls $v_1, \ldots, v_k$ Nachfolger von v sind, ist $X_G(v)$ der Baum mit der Wurzel v und den Teilbäumen $X_G(v_1), \ldots, X_G(v_k)$.

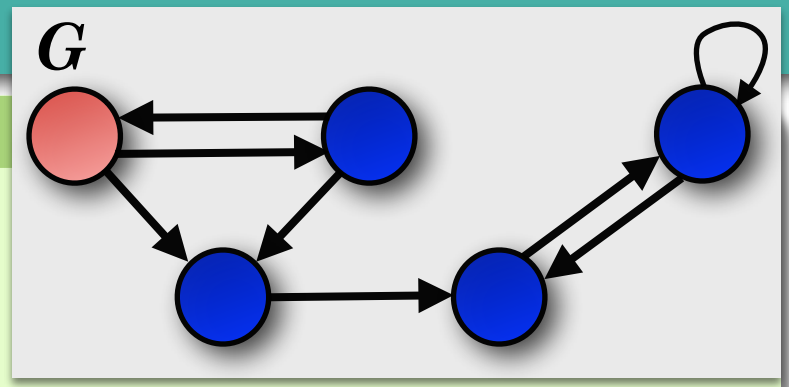
- **Beachte:**
  - Die Knoten eines Graphen kommen in der Regel mehrfach im Baum vor
  - Der Baum ist unendlich, wenn der Graph Zyklen enthält
  - Der Baum $X_G(v)$ stellt die Menge aller Pfade dar, die von v ausgehen

© 2006-2024 Jörg Striegnitz

Beispiel: Expansion

B Expansion eines Knotens

Expansion von Knoten 1

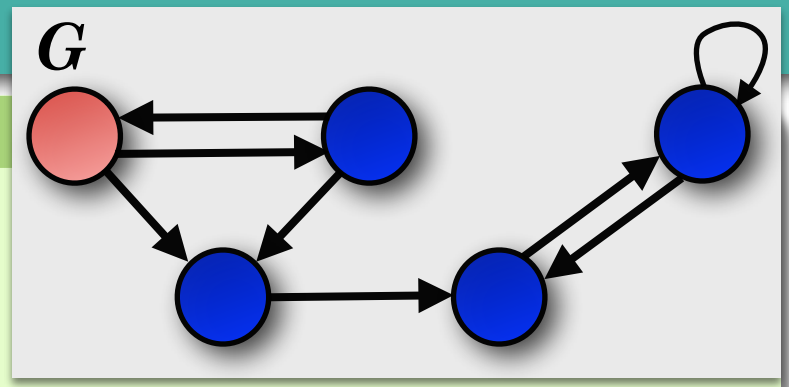


© 2006-2024 Jörg Striegnitz

Beispiel: Expansion

B Expansion eines Knotens

Expansion von Knoten 1



Der Baum ist aufgrund des vorhandenen Zyklus
(z.B. bei Knoten 3) **unendlich hoch**

© 2006-2024 Jörg Striegnitz

Strategien zum Traversieren

- **Wie bei Bäumen unterscheidet man:**
  - **Tiefensuche (*depth first*)**
    - *preorder* oder *postorder traversal* möglich (in der Literatur meist *preorder*)
    - mittels **Stack** (oder rekursiv)
  - **Breitensuche (*breadth first*)**
    - auch *level order traversal* genannt
    - mittels **Queue**

© 2006-2024 Jörg Striegnitz

Strategien zum Traversieren

- **Wie bei Bäumen unterscheidet man:**
  - **Tiefensuche** (***depth first***)
    - *preorder* oder *postorder traversal* möglich (in der Literatur meist *preorder*)
    - mittels **Stack** (oder rekursiv)
  - **Breitensuche** (***breadth first***)
    - auch *level order traversal* genannt
    - mittels **Queue**
- **Wichtige Modifikationen:**
  - Knoten können in Expansion mehrfach vorkommen!
  - Markieren von besuchten Knoten
  - Abbruch bei Besuch von markierten Knoten

© 2006-2024 Jörg Striegnitz

Strategien zum Traversieren

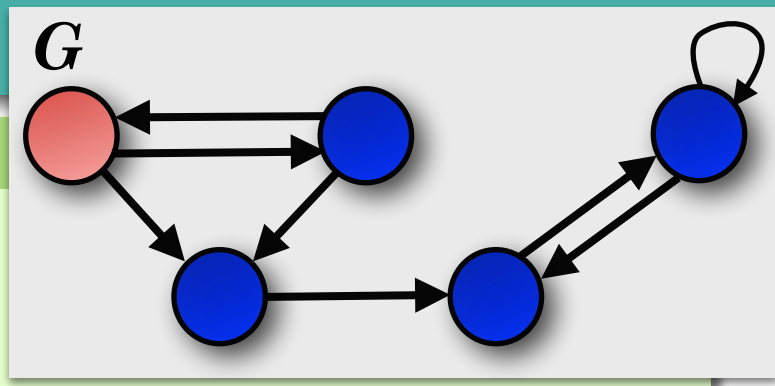
- **Wie bei Bäumen unterscheidet man:**
  - **Tiefensuche (*depth first*)**
    - *preorder* oder *postorder traversal* möglich (in der Literatur meist *preorder*)
    - mittels **Stack** (oder rekursiv)
  - **Breitensuche (*breadth first*)**
    - auch *level order traversal* genannt
    - mittels **Queue**
- **Wichtige Modifikationen:**
  - Knoten können in Expansion mehrfach vorkommen!
  - Markieren von besuchten Knoten
  - Abbruch bei Besuch von markierten Knoten
- **Programm-Ansatz:**
  - Markiere alle Knoten als „*not visited*“
  - Abarbeiten der Knoten:

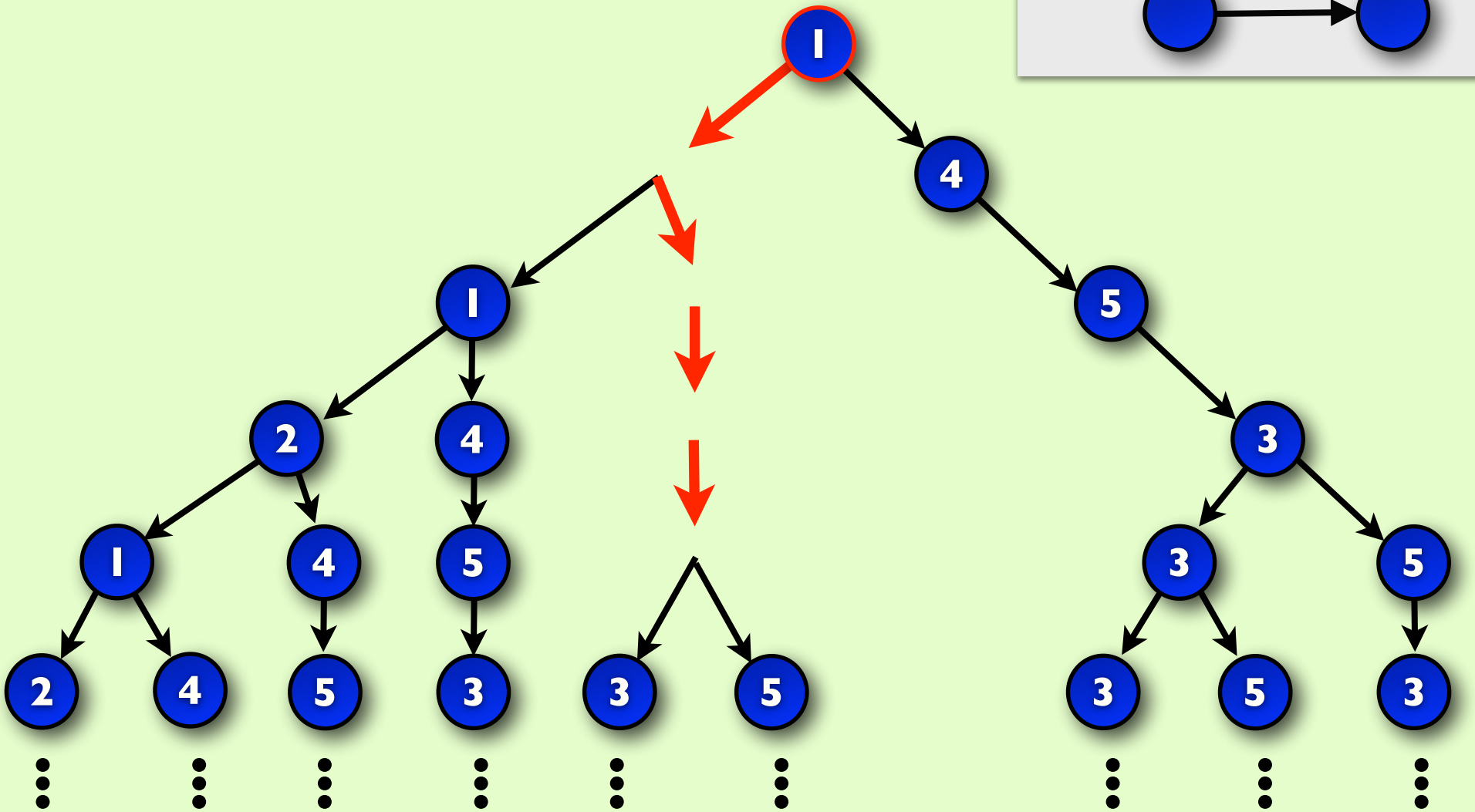
```
If (Knoten not visited) then
  bearbeite
  markiere „visited“
```


© 2006-2024 Jörg Striegnitz

Beispiel: Tiefensuche

B Tiefensuche

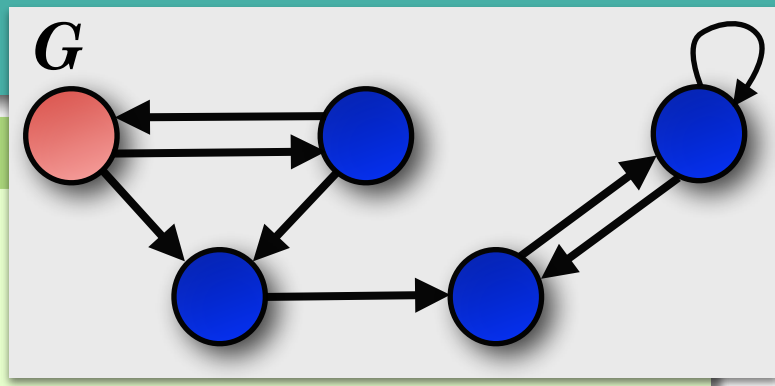




© 2006-2024 Jörg Striegnitz

Beispiel: Breitensuche

B Breitensuche



© 2006-2024 Jörg Striegnitz

Interpretation Graph-Durchlauf

- **Während des Durchlaufs werden Knoten in drei Klassen eingeteilt:**
  - **Ungesehene Knoten** (*unseen nodes*)
    Knoten, die noch nicht erreicht worden sind
  - **Baumknoten** (*tree vertices / visited nodes*)
    Knoten, die schon besucht und abgearbeitet worden sind. Diese Knoten ergeben den Suchbaum.
  - **Randknoten** (*fringe nodes / nodes in scope*)
    Knoten, die über eine Kante mit einem Baum-Knoten verbunden sind

© 2006-2024 Jörg Striegnitz

Traversal: Implementierung in JAVA I

- **Generischer Traversal als `interface`**

```
public interface Traversal<T> {
  // z.B. initialisiere Stack / Queue
  public void initTraversal();
  // Besucht den Knoten mit der id "KnotenId" und alle
  // von diesem erreichbaren Knoten (je nach Strategie)
  public void visit(TraversableGraph<T> g, int KnotenId);
  // Führt eine Operation beim Knoten aus
  public void doSomethingWith(NodeWithId<T> t);
}
```


© 2006-2024 Jörg Striegnitz

Traversal: Implementierung in JAVA II

- **Wir spalten die Vorgabeimplementierung in ein `interface` und eine abstrakte Basisklasse:**

```
public interface TraversableGraph<V> extends Graph<V> {

  public enum nodeState { visited, notseen, inscope };

  // Knoten müssen Auskunft über ihren Zustand geben können
  public abstract nodeState getState(int id);
  public abstract void setState(int id,nodeState s);
  public void traverse(Traversal<V> t);
}
```


© 2006-2024 Jörg Striegnitz

Traversal: Implementierung in JAVA II

- **Wir spalten die Vorgabeimplementierung in ein `interface` und eine abstrakte Basisklasse:**

```
public interface TraversableGraph<V> extends Graph<V> {

  public enum nodeState { visited, notseen, inscope };

  // Knoten müssen Auskunft über ihren Zustand geben können
  public abstract nodeState getState(int id);
  public abstract void setState(int id,nodeState s);
  public void traverse(Traversal<V> t);
}
```

- **Ein traversierbarer, gewichteter Graph kann dann sowohl das interface `TraversableGraph`, als auch das Interface `WeightedGraph` implementieren**

© 2006-2024 Jörg Striegnitz

Traversal: Implementierung in JAVA II

- **Das Traversieren wird in einer abstrakten Basisklasse realisiert** (die Verwaltung des State obliegt dem Graph)**:**

```
public abstract class StdTraversableGraph<V>
                      implements TraversableGraph<T> {

  // Knoten müssen Auskunft über ihren Zustand geben können
  public abstract nodeState getState(int id);
  public abstract void setState(int id,nodeState s);

  void traverse(Traversal<V> t) {
    // Traversal initialisieren (z.B. Stack / Queue)
    t.initTraversal();
    for (int i=0 ; i<numNodes() ; ++i)
      setState(i,nodeState.notseen); // Noch kein Knoten besucht
    // Alle Knoten durchprobieren; notwendig, da nicht alle
    // Knoten vom Ausgangsknoten erreichbar sein müssen.
    for (int i=0 ; i<numNodes() ; ++i)
      if (getState(i)==nodeState.notseen)
        t.visit(this,i); // visit-Methode steuert Traversal
  }
}
```


© 2006-2024 Jörg Striegnitz

Traversal: Implementierung in JAVA II

- **Das Traversieren wird in einer abstrakten Basisklasse realisiert:**

  - Traversal initialisieren
  - Alle Knoten erhalten Status „`notseen`“
  - `visit` bearbeitet nur zusammenhängende Teilgraphen
  - Daher: Schleife über alle Knoten

```
  public abstract void setState(int id,nodeState s);

  void traverse(Traversal<V> t) {
    // Traversal initialisieren (z.B. Stack / Queue)
    t.initTraversal();
    for (int i=0 ; i<numNodes() ; ++i)
      setState(i,nodeState.notseen); // Noch kein Knoten besucht
    // Alle Knoten durchprobieren; notwendig, da nicht alle
    // Knoten vom Ausgangsknoten erreichbar sein müssen.
    for (int i=0 ; i<numNodes() ; ++ i)
      if (getState(i)==nodeState.notseen)
        t.visit(this,i); // visit-Methode steuert Traversal
  }
}
```


© 2006-2024 Jörg Striegnitz

Traversal: Implementierung in JAVA III

- **Tiefensuche :**

```
public abstract class DepthFirst<T> implements Traversal<T> {

  public enum strategies { preorder, // Knoten erst bearbeiten
                           postorder // Erst Kindknoten bearbeiten
                         };

  private strategies strategy; // Traversal-Strategie
  public DepthFirst(strategies s) { strategy=s; }

  public void initTraversal() {  } // nichts zu initialisieren

  // "visit" ist abhängig von "strategy" ...
  public void visit(TraversableGraph<T> g,int k) {
    if (strategy==strategies.preorder)
      visitPreOrder(g,k);
    else if (strategy==strategies.postorder)
        visitPostOrder(g,k);
  }
```

... Fortsetzung auf nächster Folie ...

© 2006-2024 Jörg Striegnitz

Traversal: Implementierung in JAVA III

- **Tiefensuche :**

```
public abstract class DepthFirst<T> implements Traversal<T> {
  public enum strategies { preorder, // Knoten erst bearbeiten
                           postorder // Erst Kindknoten bearbeiten
                         };

  private strategies strategy; // Traversal-Strategie
```

Wir wollen *preorder-* und *postorder-traversal* unterstützen

```
  public void initTraversal() {  } // nichts zu initialisieren

  // "visit" ist abhŠngig von "strategy" ...
  public void visit(TraversableGraph<T> g,int k) {
    if (strategy==strategies.preorder)
      visitPreOrder(g,k);
    else if (strategy==strategies.postorder)
        visitPostOrder(g,k);
  }
```

... Fortsetzung auf nächster Folie ...

© 2006-2024 Jörg Striegnitz

Traversal: Implementierung in JAVA IV

- **Tiefensuche (*postorder*):**

... Fortsetzung von vorheriger Folie ...

```
public void visitPostOrder(TraversableGraph<T> g, int k) {
  // Postorder -> erst "absteigen". Um Endlossschleife
  // zu vermeiden, Knotenstatus auf "inscope" setzen.
  g.setState(k,TraversableGraph.nodeState.inscope);
  for (int t=0 ; t<g.numNodes() ; ++t) {
    if ((g.connected(k,t)) &&
        (g.getState(t)==TraversableGraph.nodeState.notseen))
        visit(g,t);
  }
  doSomethingWith(g.getNodeWithId(k)); // Knoten bearbeiten
  g.setState(k,TraversableGraph.nodeState.visited);
}
```

... Fortsetzung auf nächster Folie ...

© 2006-2024 Jörg Striegnitz

Traversal: Implementierung in JAVA V

- **Tiefensuche (*preorder*):**

... Fortsetzung von vorheriger Folie ...

```
  public void visitPreOrder(TraversableGraph<T> g, int k) {
    // Preorder -> erst bearbeiten ...
    doSomethingWith(g.getNodeWithId(k));
    g.setState(k, TraversableGraph.nodeState.visited);
    // ... dann "absteigen"
    for (int t=0 ; t<g.numNodes() ; ++t) {
      if ((g.connected(k,t)) &&
          (g.getState(t)==TraversableGraph.nodeState.notseen))
          visit(g,t);
    }
  }
}
```


© 2006-2024 Jörg Striegnitz

Traversal: Implementierung in JAVA VI

- **Breitensuche:**

```
import java.util.LinkedList;

public abstract class BreadthFirst<T> implements Traversal<T> {
  LinkedList<Integer> queue;
  public void initTraversal() {queue = new LinkedList<Integer>();}

  public void visit(TraversableGraph<T> g, int k) {
    g.setState(k,TraversableGraph.nodeState.inscope);
    queue.offer(k);
    do { // nächsten Knoten aus Queue und bearbeiten
      k=queue.remove() ;  doSomethingWith(g.getNodeWithId(k));
      g.setState(k,TraversableGraph.nodeState.visited);
      // Knoten in unmittelbarer Reichweite zur Queue hinzu
      for (int t=0 ; t<g.numNodes(); ++t)
        if ((g.connected(k, t)) &&
            (g.getState(t)==TraversableGraph.nodeState.notseen)) {
          g.setState(t, TraversableGraph.nodeState.inscope);
          queue.offer(t);
        }
    } while (! queue.isEmpty());
  }
}
```


© 2006-2024 Jörg Striegnitz

VII. Graphen

7. Graphen

7.1. Einführung und Begriffsbestimmungen
7.2. Traversierung
7.3. Topologisches Sortieren
7.4. Kürzeste Wege
7.4.1. Einführung
7.4.2. *Single-Source Best Path* Algorithmen
7.4.3. *All-Pairs Best Path* Algorithmen
7.5. Maximaler Fluss
7.6. Minimaler Spannbaum

© 2006-2023 Jörg Striegnitz

Abhängigkeitsgraphen (*Dependency Graphs*)

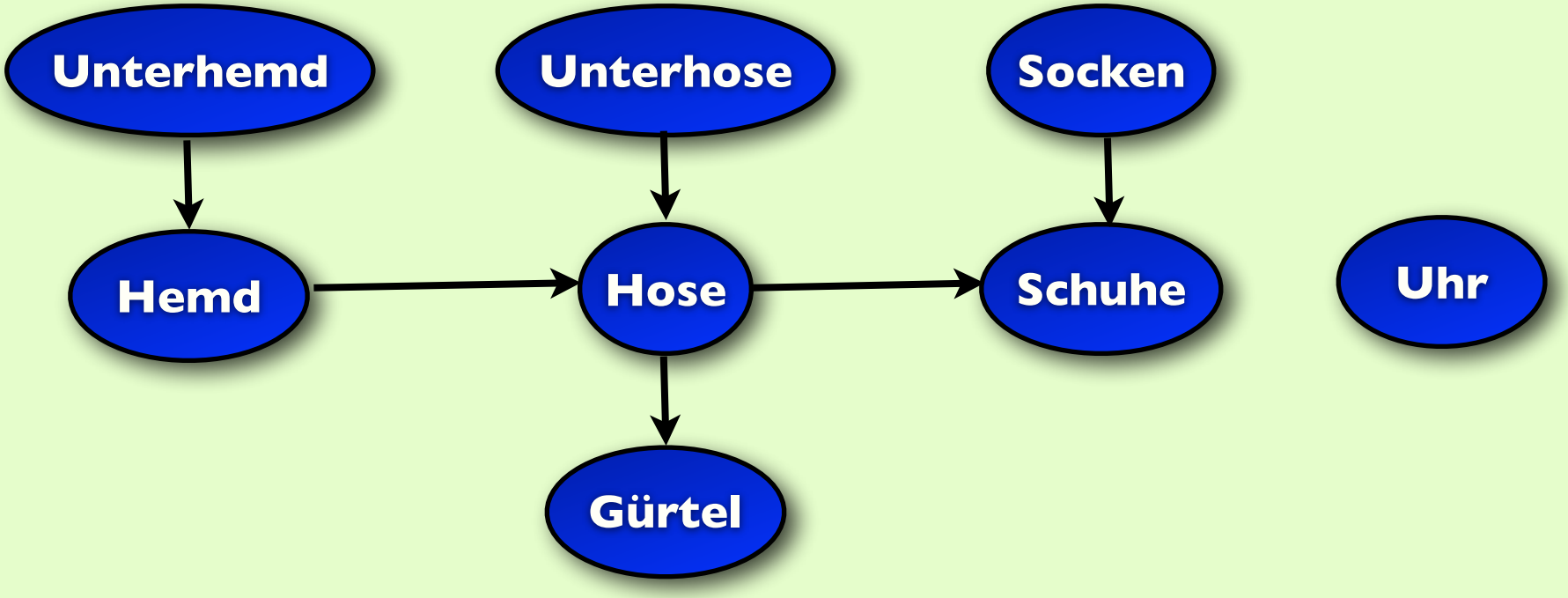
- **In einem Graph lassen sich Abhängigkeiten darstellen:**

© 2006-2024 Jörg Striegnitz

Abhängigkeitsgraphen (*Dependency Graphs*)

- **In einem Graph lassen sich Abhängigkeiten darstellen:**

B **Abhängigkeitsgraph (*Dependency Graph*)**



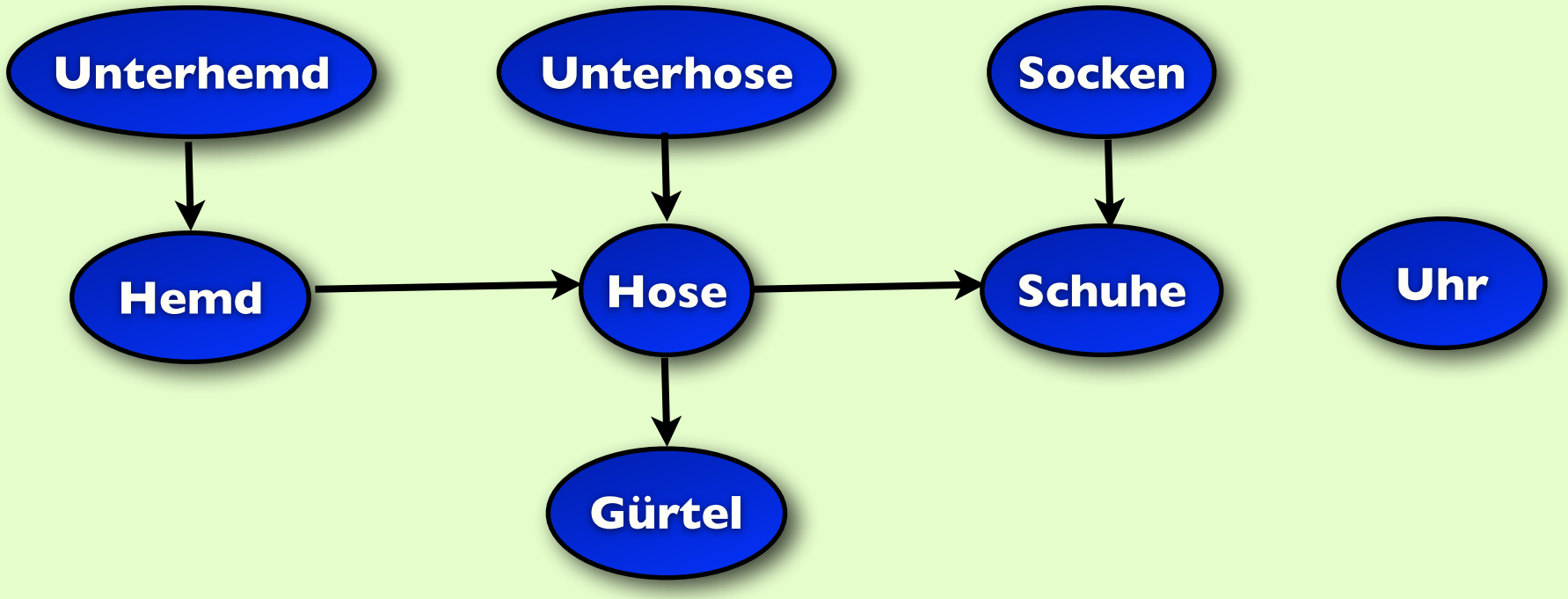
Kantensemantik: „Muss angezogen werden vor“

© 2006-2024 Jörg Striegnitz

Abhängigkeitsgraphen (*Dependency Graphs*)

- **In einem Graph lassen sich Abhängigkeiten darstellen:**

B Abhängigkeitsgraph (*Dependency Graph*)



Kantensemantik: „Muss angezogen werden vor“

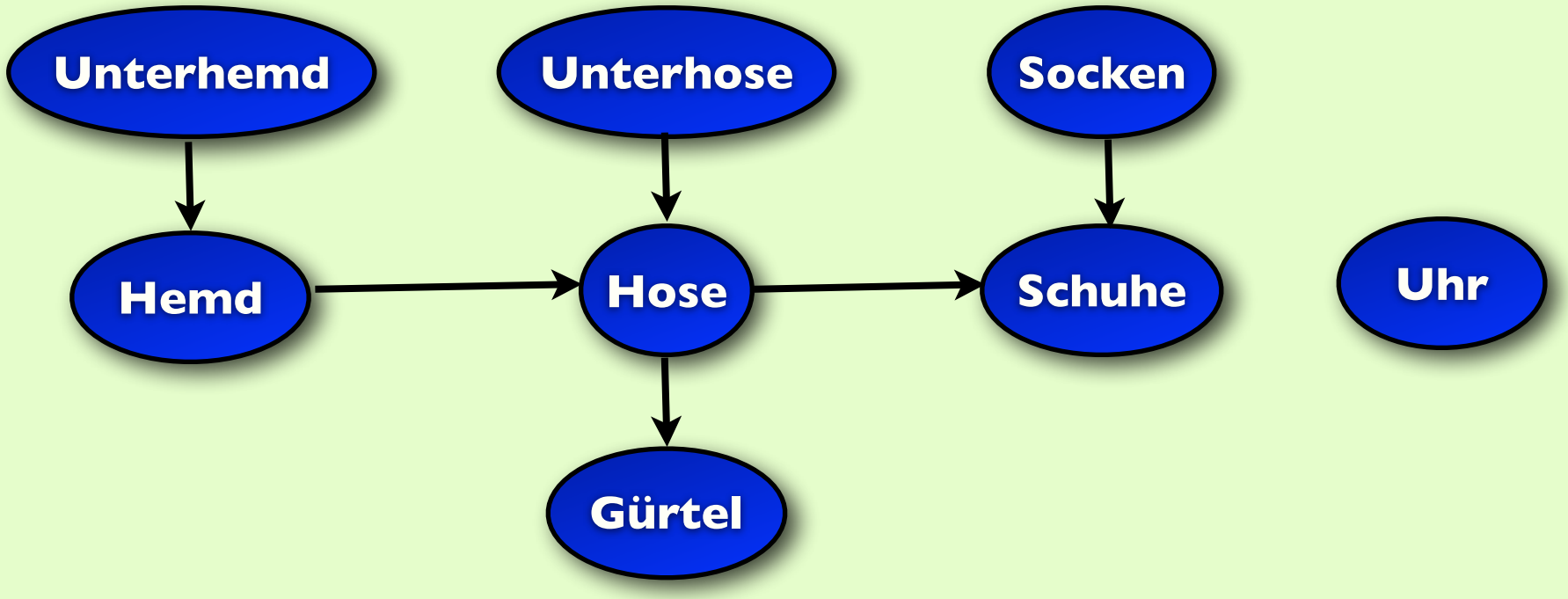
- **Frage:** in welcher Reihenfolge anziehen?

© 2006-2024 Jörg Striegnitz

Abhängigkeitsgraphen (*Dependency Graphs*)

- **In einem Graph lassen sich Abhängigkeiten darstellen:**

B Abhängigkeitsgraph (*Dependency Graph*)



Kantensemantik: „Muss angezogen werden vor“

- **Frage:** in welcher Reihenfolge anziehen?
- **Lösung:** **Topologisches Sortieren**

© 2006-2024 Jörg Striegnitz

D Topologische Sortierung

Sei $G = (V, E)$ ein gerichteter, zyklusfreier Graph. Eine **topologische Sortierung** der Knoten von G ist eine Aufzählung aller Knoten aus V:

$$v_1, v_2, \cdots, v_n$$

für die gilt:

$$(v_i, v_j) \in E \Rightarrow i \leq j$$

© 2006-2024 Jörg Striegnitz

Topologische Sortierung

D Topologische Sortierung

Sei $G = (V, E)$ ein gerichteter, zyklusfreier Graph. Eine **topologische Sortierung** der Knoten von G ist eine Aufzählung aller Knoten aus V:

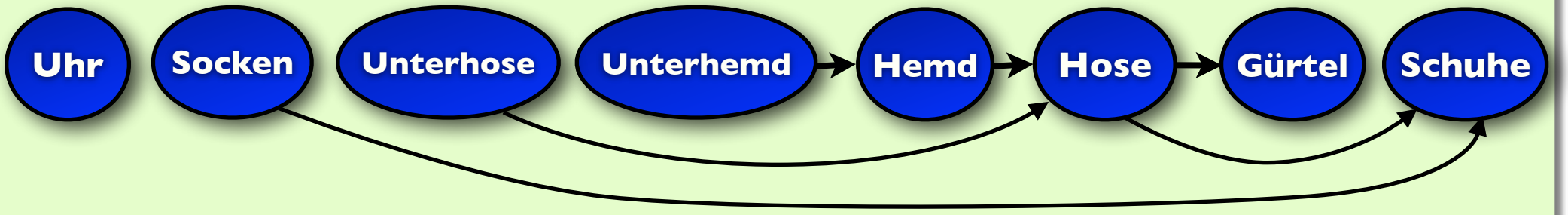
$$v_1, v_2, \cdots, v_n$$

für die gilt:

$$(v_i, v_j) \in E \Rightarrow i \leq j$$

- **Anschaulich:** in v_1,...v_n gehen Kanten nur **von links nach rechts**:

B Topologische Sortierung



© 2006-2024 Jörg Striegnitz

Topologisches Sortieren: Implementierung (JAVA)

- **Algorithmus:**
  - Tiefensuche (*postorder*)
  - Operation beim Knoten:
    - Knoten an den Anfang einer linearen Liste einfügen
  - Dann: Ausgabe der Liste ist topologische Sortierung!

© 2006-2024 Jörg Striegnitz

Topologisches Sortieren: Implementierung (JAVA)

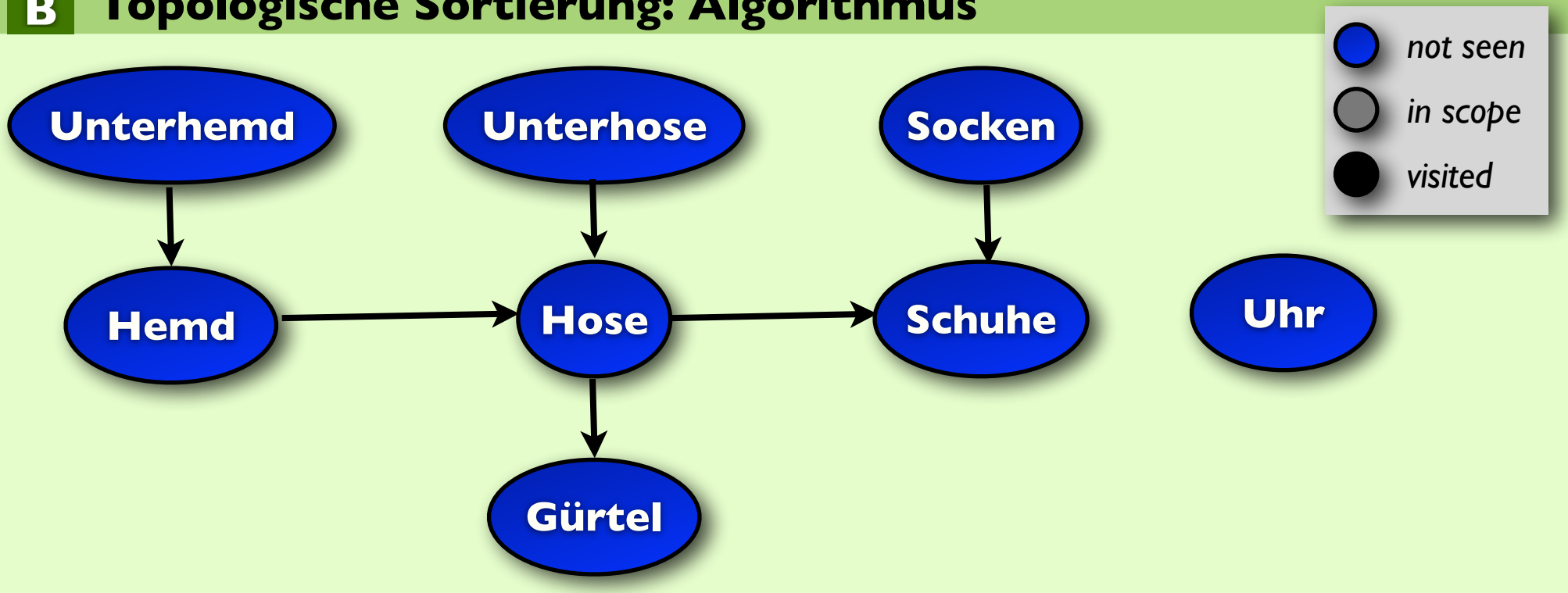
- **Algorithmus:**
  - Tiefensuche (*postorder*)
  - Operation beim Knoten:
    - Knoten an den Anfang einer linearen Liste einfügen
  - Dann: Ausgabe der Liste ist topologische Sortierung!

```
import java.util.LinkedList;

public class TopologicSort<T> extends DepthFirst<T> {
  LinkedList<T> sortedList;

  TopologicSort() { super(DepthFirst.strategies.postorder);
                    sortedList = new LinkedList<T>(); }

  public void doSomethingWith(NodeWithId<T> t) {
    // Wird ausgeführt, nachdem alle Nachfolger von
    // T besucht wurden -> Knoten an Anfang der Liste
    sortedList.addFirst(t.vertex);
  }
  LinkedList<T> result() { return sortedList; }
}
```


© 2006-2024 Jörg Striegnitz

Topologisches Sortieren: Beispiel

B Topologische Sortierung: Algorithmus

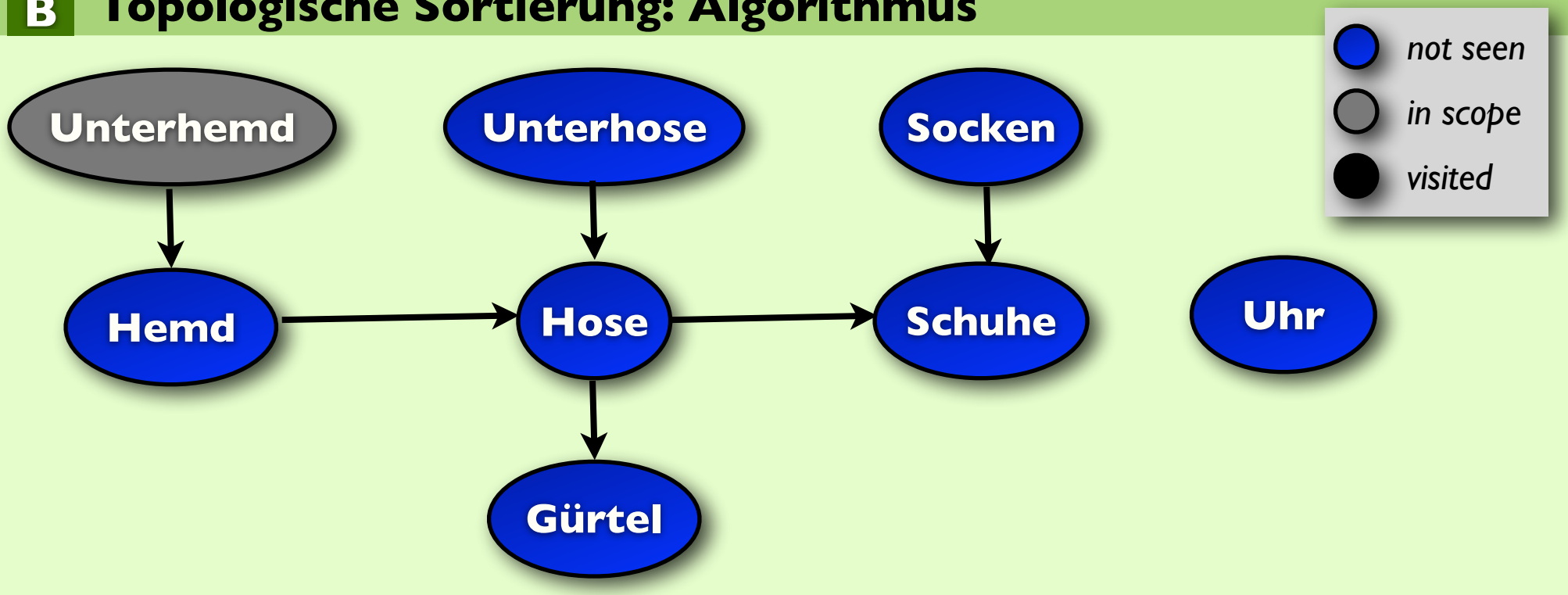


Wir wählen einen beliebigen Startknoten; z.B. das **Unterhemd**; dieser Knoten habe die id 0 - siehe `TraversableGraph.traverse`

© 2006-2024 Jörg Striegnitz

Topologisches Sortieren: Beispiel

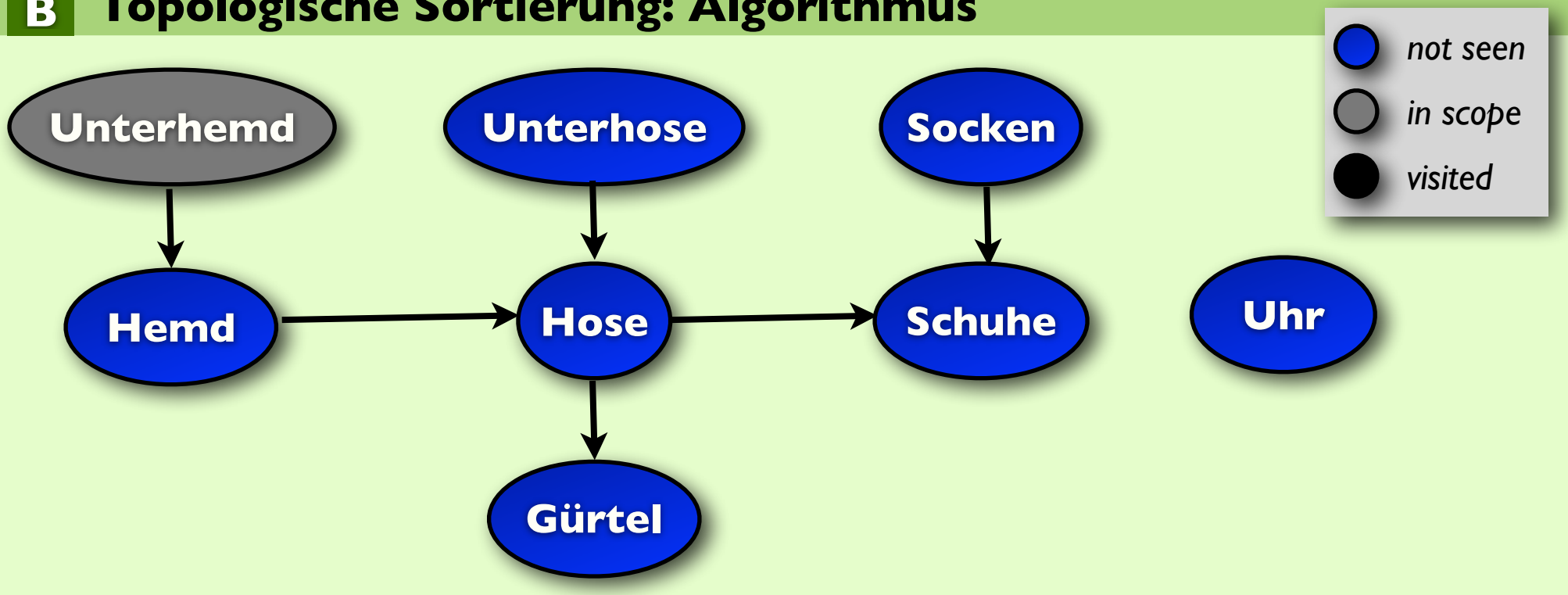
B Topologische Sortierung: Algorithmus



© 2006-2024 Jörg Striegnitz

Topologisches Sortieren: Beispiel

B Topologische Sortierung: Algorithmus

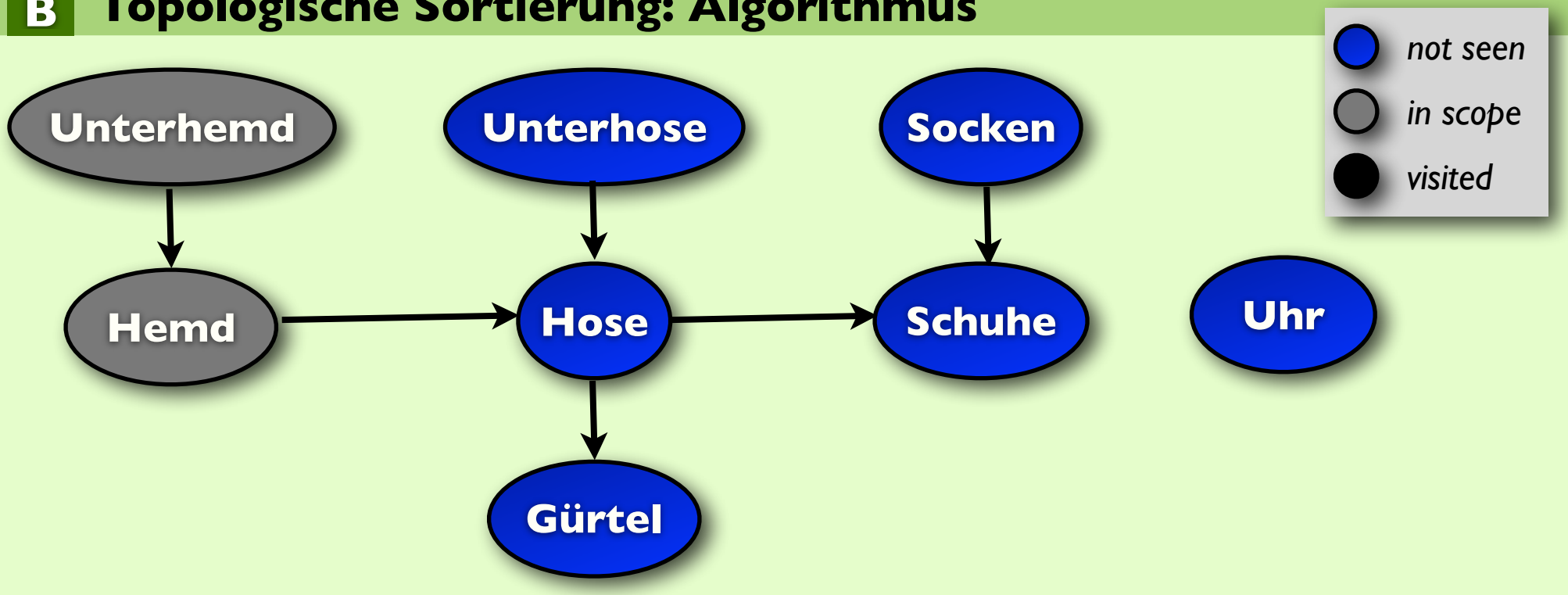


Von hier aus suchen wir solange nach Nachbarknoten, bis wir in eine „Sackgasse“ laufen - siehe `DepthFirst.visitPostOrder`

© 2006-2024 Jörg Striegnitz

Topologisches Sortieren: Beispiel

B Topologische Sortierung: Algorithmus

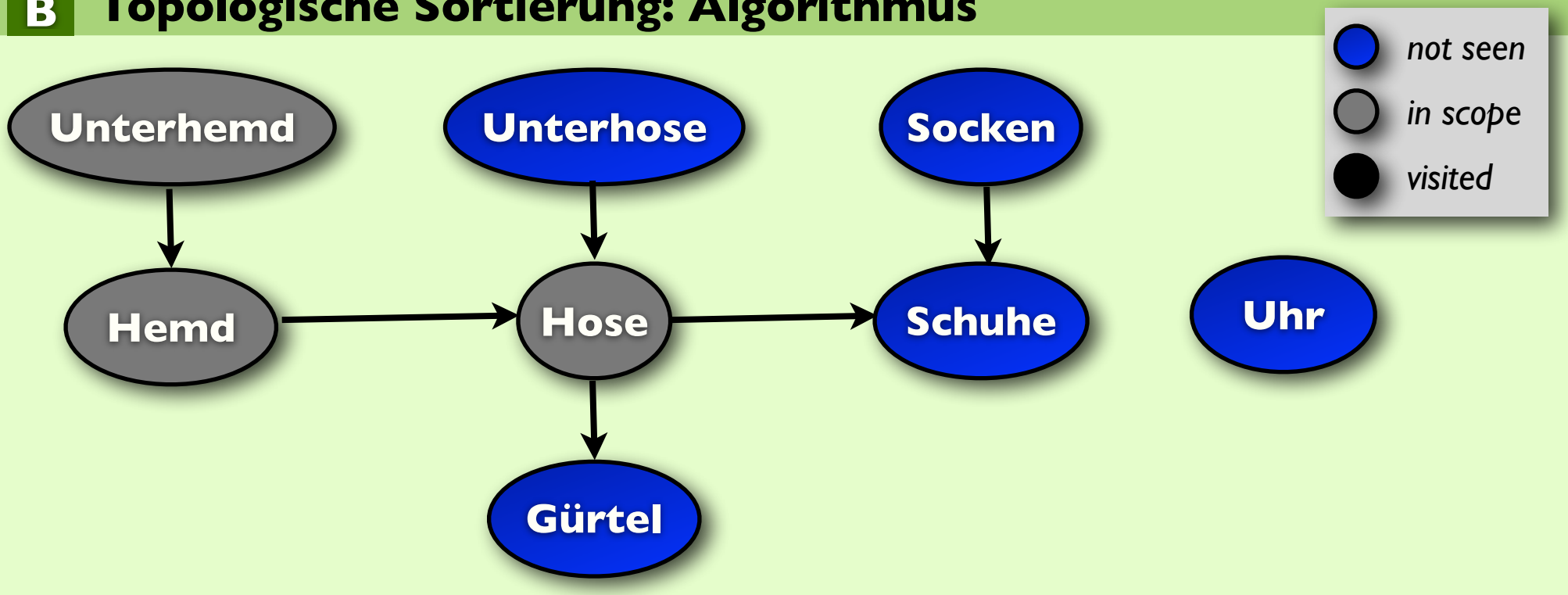


Von hier aus suchen wir solange nach Nachbarknoten, bis wir in eine „Sackgasse“ laufen - siehe `DepthFirst.visitPostOrder`

© 2006-2024 Jörg Striegnitz

Topologisches Sortieren: Beispiel

B Topologische Sortierung: Algorithmus

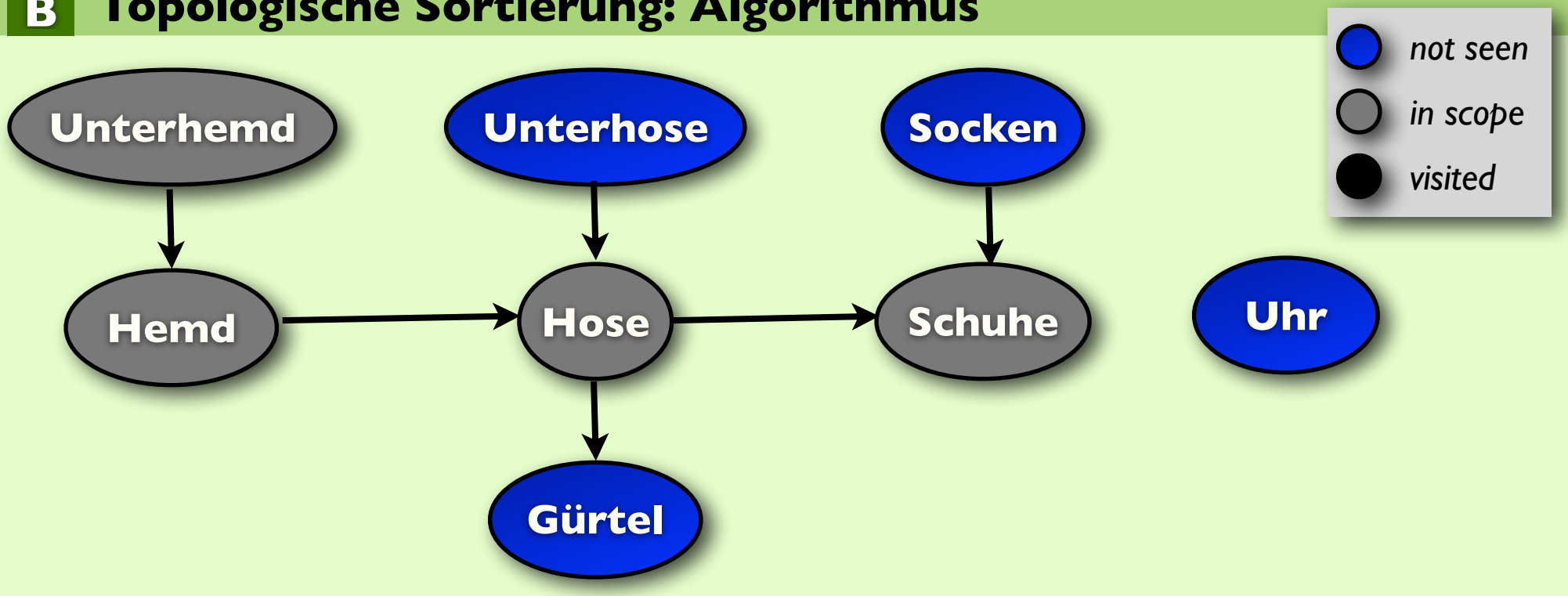


Von hier aus suchen wir solange nach Nachbarknoten, bis wir in eine „Sackgasse“ laufen - siehe `DepthFirst.visitPostOrder`

© 2006-2024 Jörg Striegnitz

Topologisches Sortieren: Beispiel

B Topologische Sortierung: Algorithmus

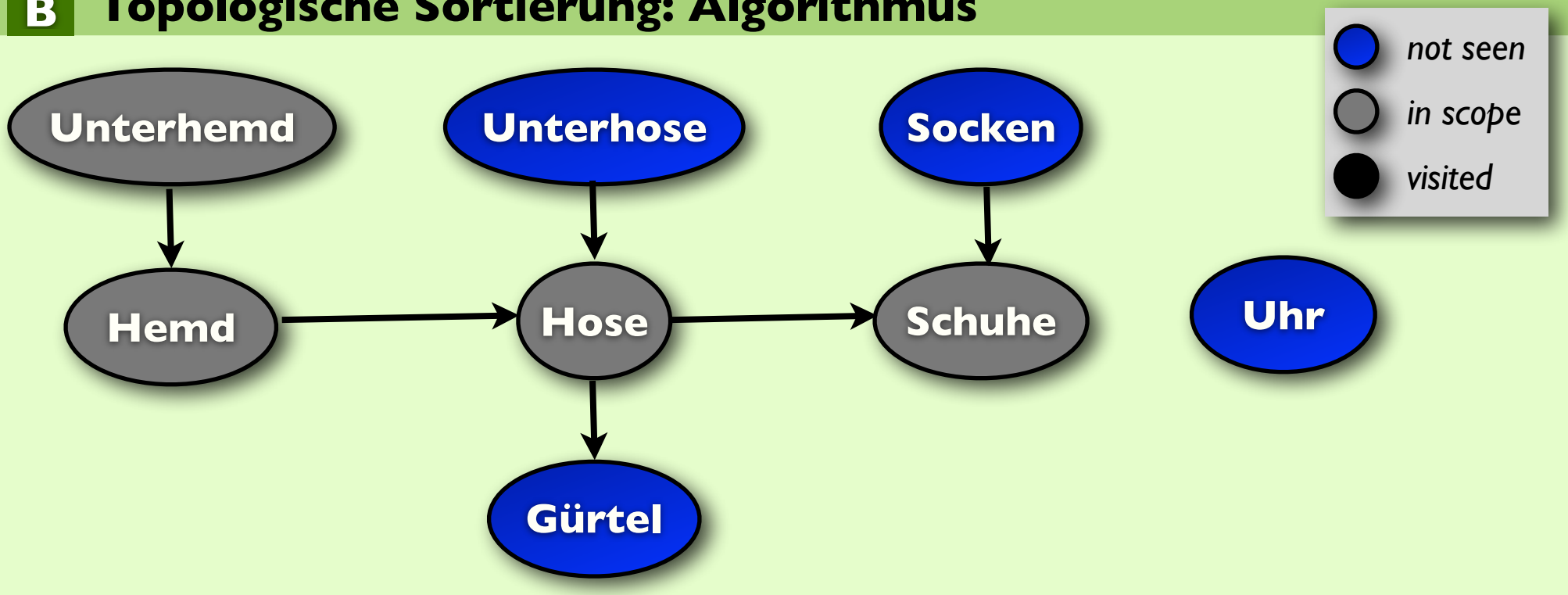


Von hier aus suchen wir solange nach Nachbarknoten, bis wir in eine „Sackgasse“ laufen - siehe `DepthFirst.visitPostOrder`

© 2006-2024 Jörg Striegnitz

Topologisches Sortieren: Beispiel

B Topologische Sortierung: Algorithmus

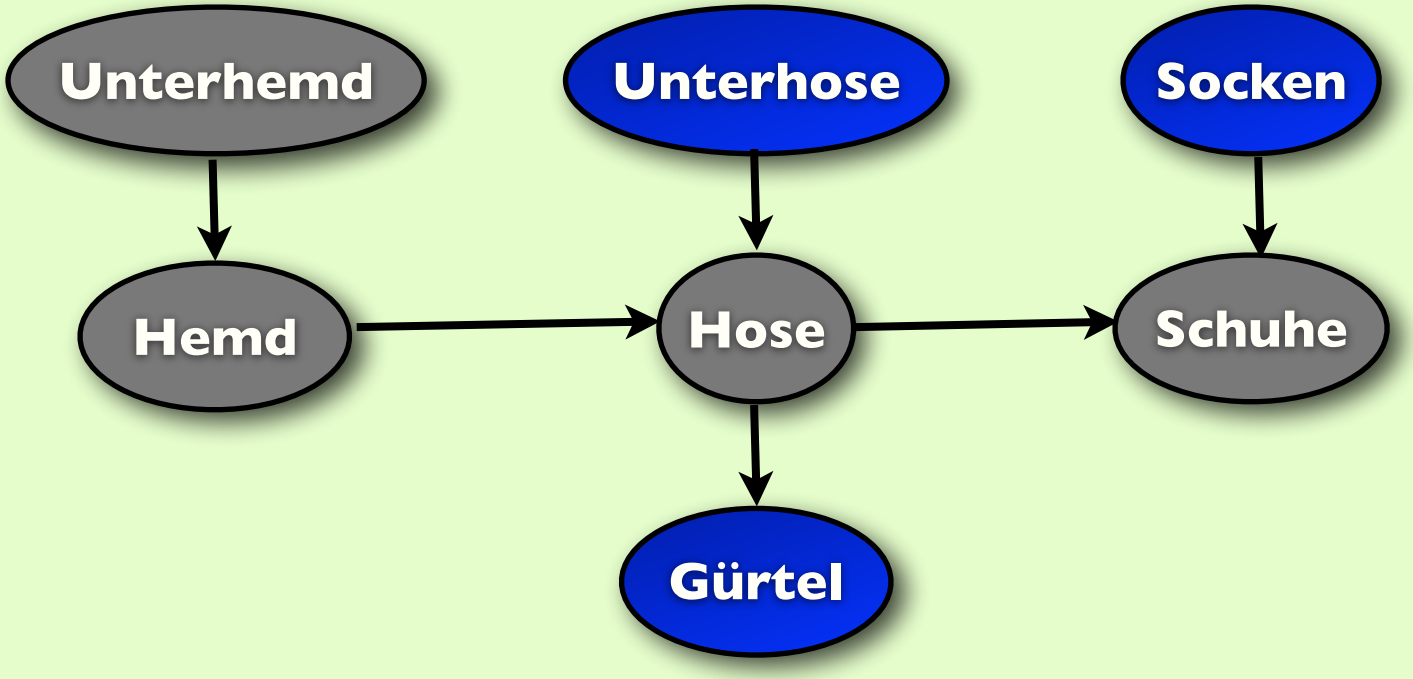


Von hier aus suchen wir solange nach Nachbarknoten, bis wir in eine „Sackgasse“ laufen - die **Schuhe**. Von den Schuhen hängt nichts weiter ab - wir können sie hinzufügen (siehe `TopologicSort.doSomething`) ...

© 2006-2024 Jörg Striegnitz

Topologisches Sortieren: Beispiel

B Topologische Sortierung: Algorithmus

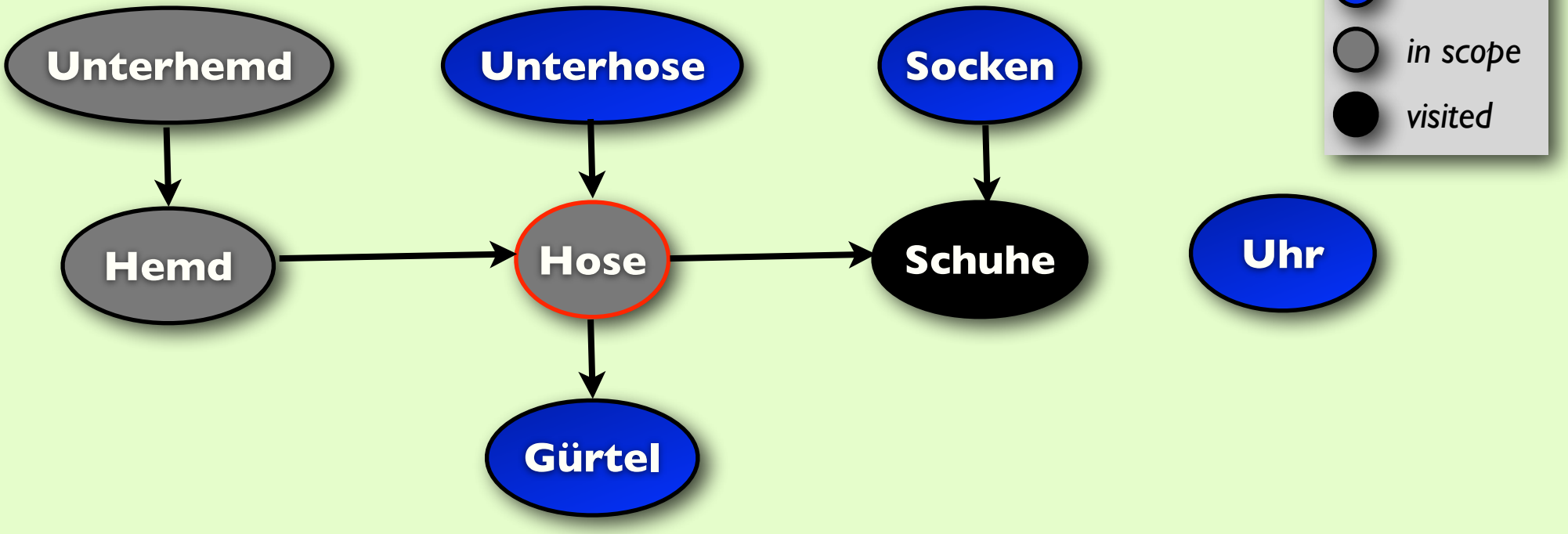


Von hier aus suchen wir solange nach Nachbarknoten, bis wir in eine „Sackgasse“ laufen - die **Schuhe**. Von den Schuhen hängt nichts weiter ab - wir können sie hinzufügen (siehe `TopologicSort.doSomething`) ...

© 2006-2024 Jörg Striegnitz

Topologisches Sortieren: Beispiel

B Topologische Sortierung: Algorithmus

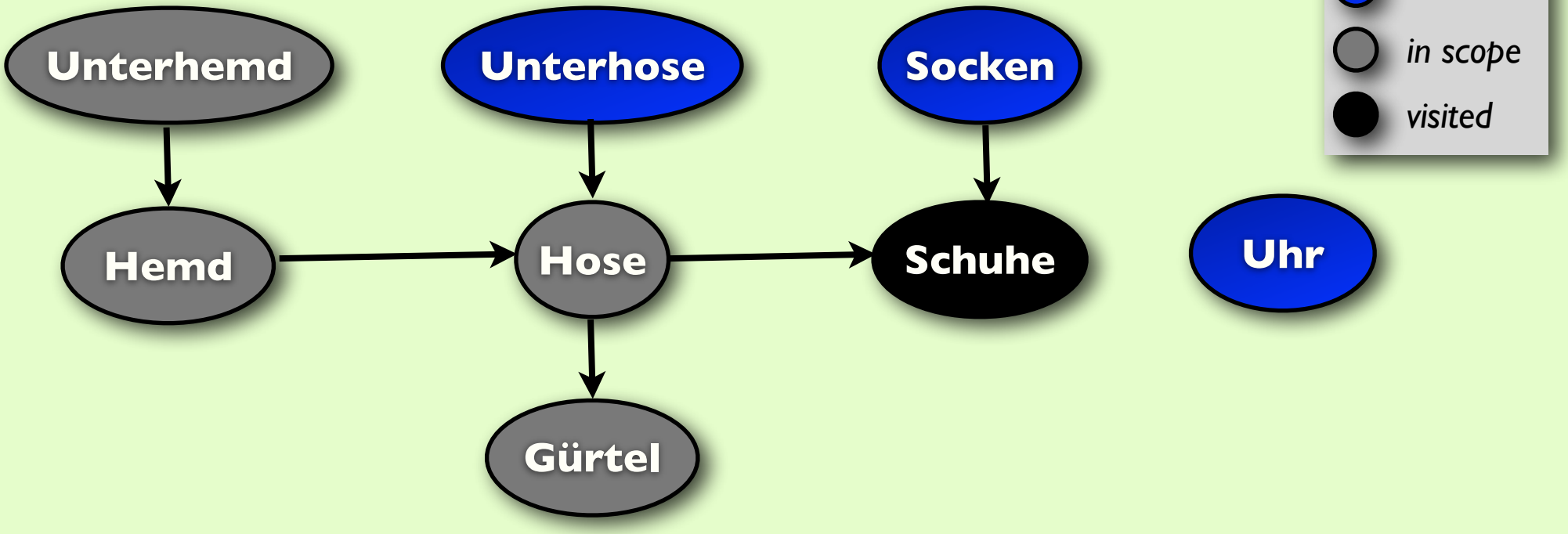


... und kehren zurück zur **Hose** (*backtrack*) Dort setzen wir die Suche nach Nachfolgern fort - bis zur nächsten Sackgasse ...

© 2006-2024 Jörg Striegnitz

Topologisches Sortieren: Beispiel

B Topologische Sortierung: Algorithmus

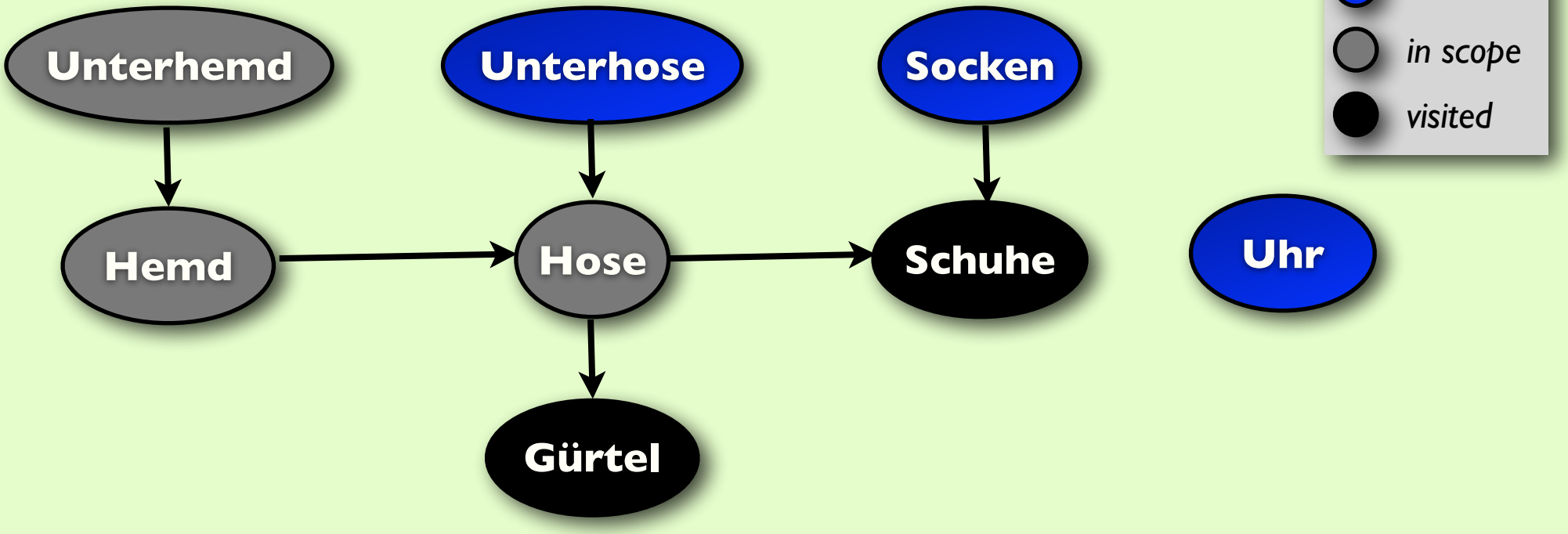


Die ist schnell gefunden - der **Gürtel**!

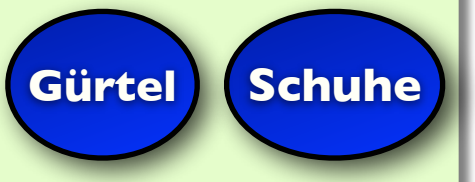
© 2006-2024 Jörg Striegnitz

Topologisches Sortieren: Beispiel

B Topologische Sortierung: Algorithmus



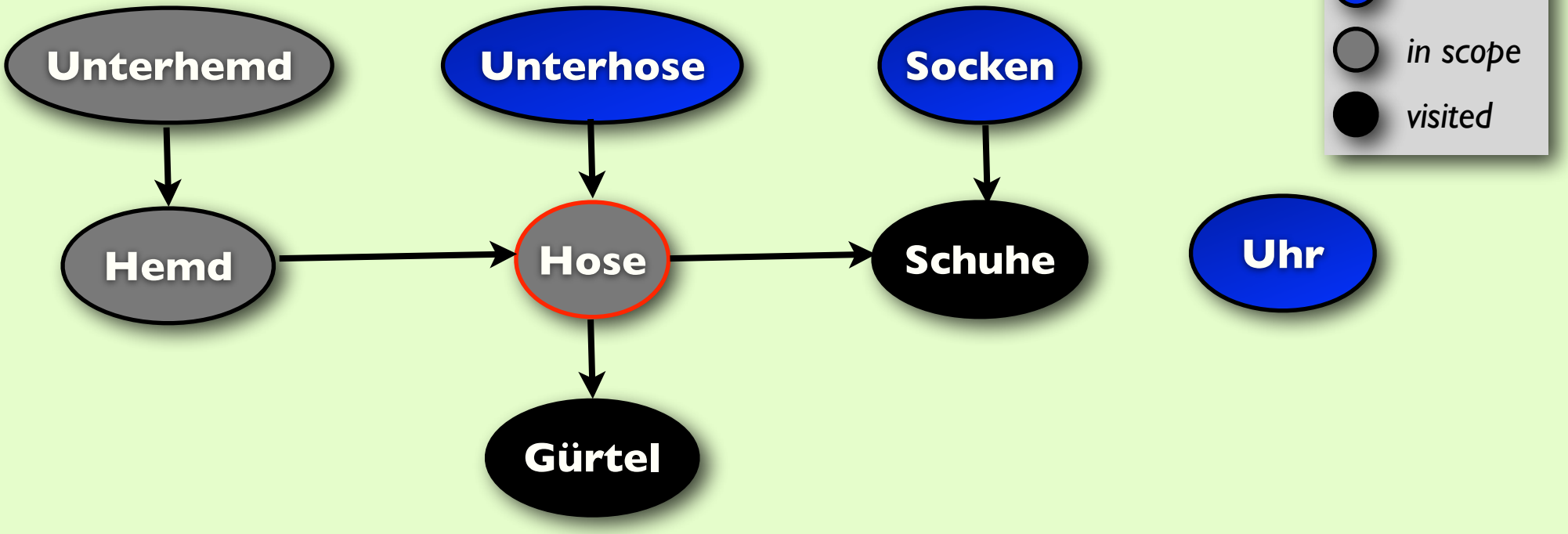
Die ist schnell gefunden - der **Gürtel**! Wir fügen ihn der Liste hinzu - er *kann* vor den Schuhen angezogen werden.



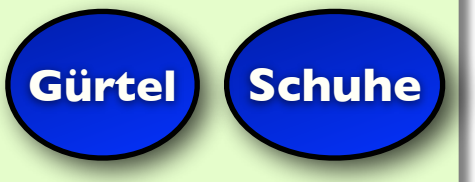
© 2006-2024 Jörg Striegnitz

Topologisches Sortieren: Beispiel

B Topologische Sortierung: Algorithmus



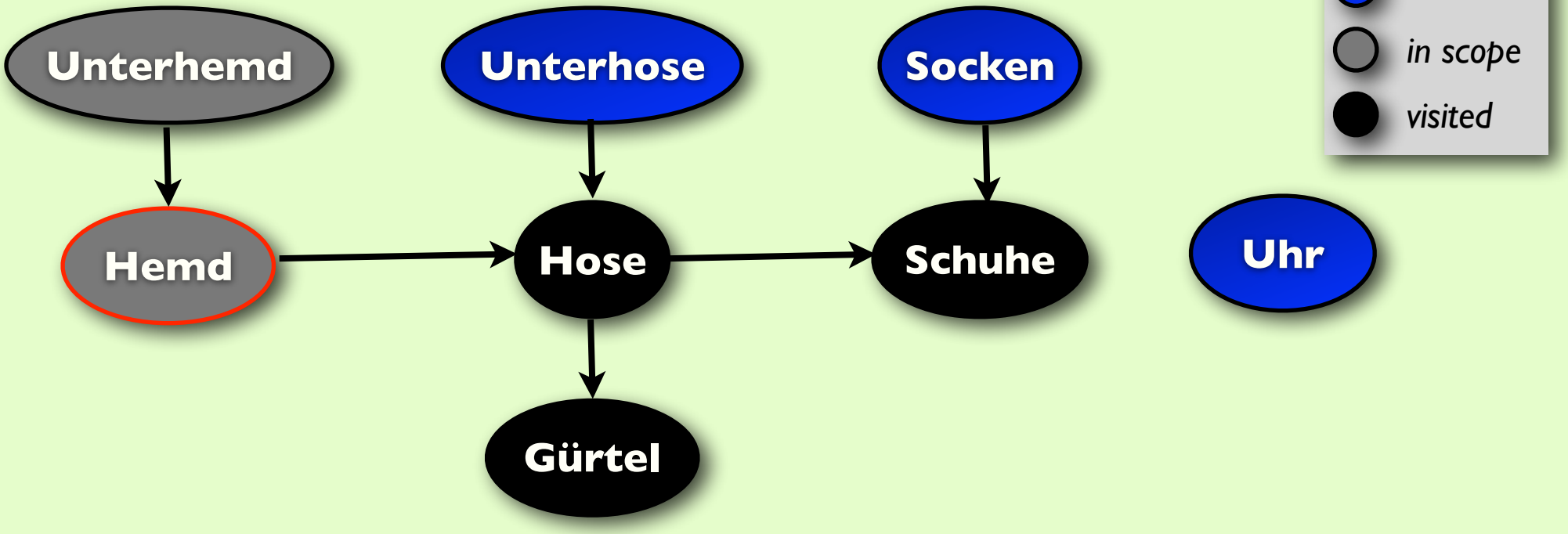
Wieder zurück zur **Hose** (*backtrack*). Wir haben alle Alternativen geprüft!



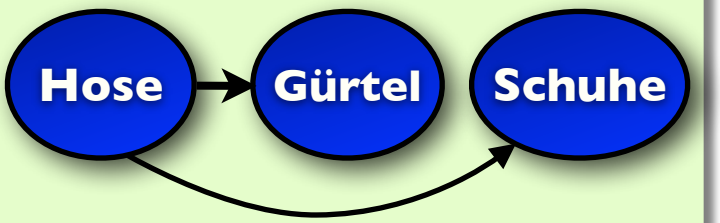
© 2006-2024 Jörg Striegnitz

Topologisches Sortieren: Beispiel

B Topologische Sortierung: Algorithmus



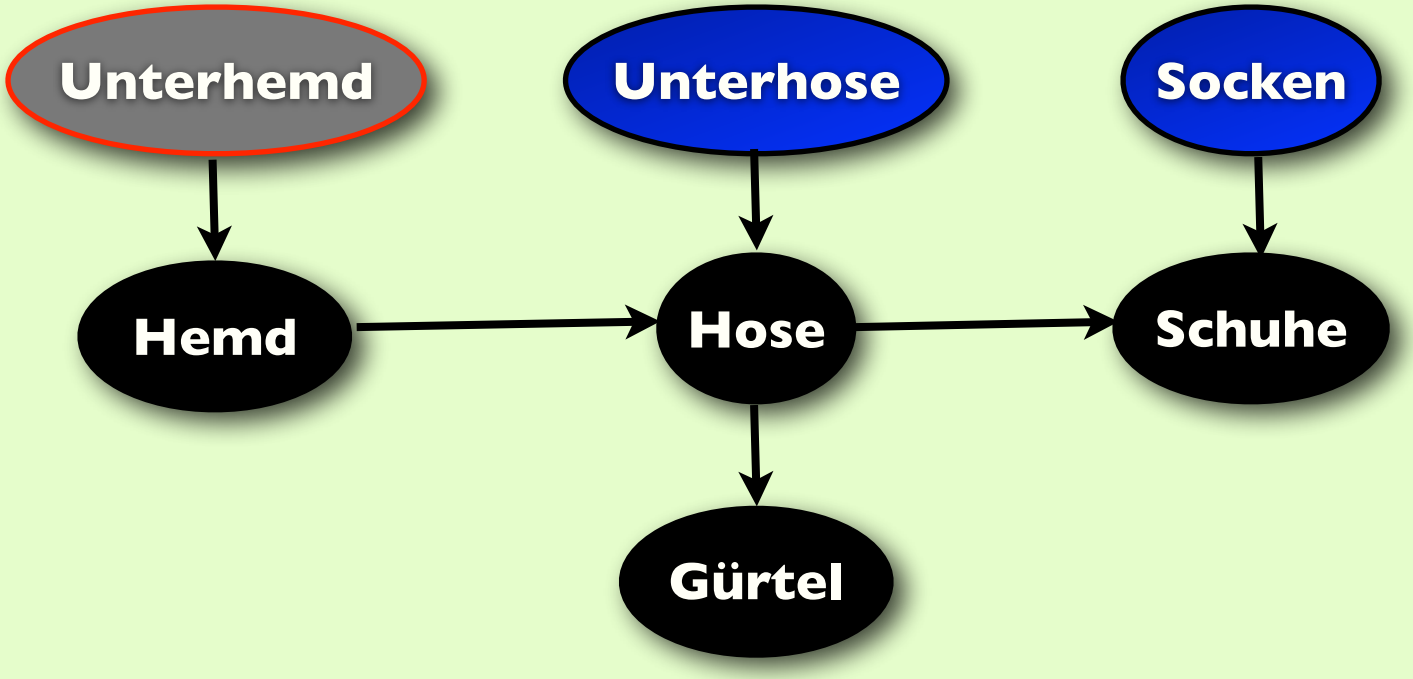
Wir fügen der Liste die **Hose** hinzu - sie muss *vor* Schuhen und Gürtel angezogen werden - und gehen zurück zum **Hemd** (*backtrack*). Auch hier haben wir alle Alternativen geprüft!



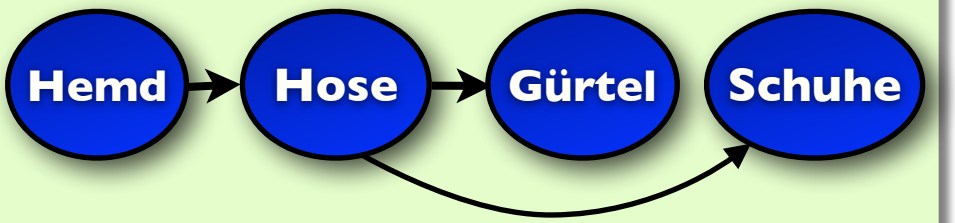
© 2006-2024 Jörg Striegnitz

Topologisches Sortieren: Beispiel

B Topologische Sortierung: Algorithmus



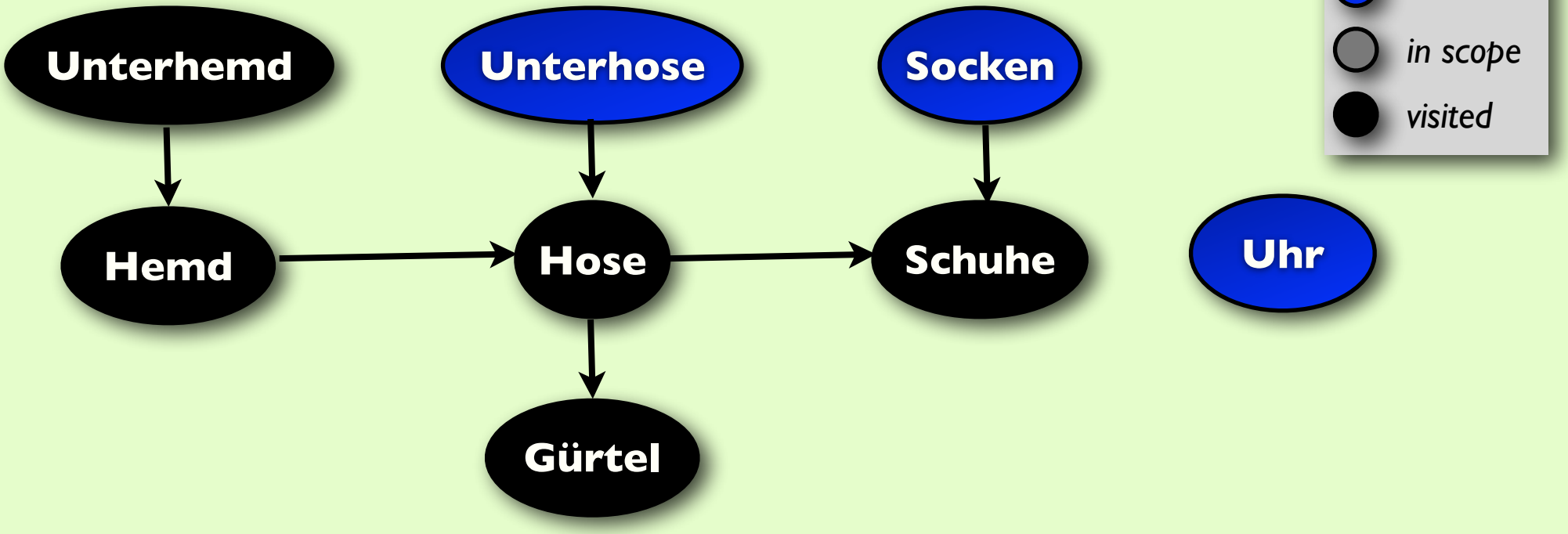
Wir fügen der Liste das Hemd hinzu und gehen zurück zum **Unterhemd** (*backtrack*). Auch hier: keine Alternativen mehr!



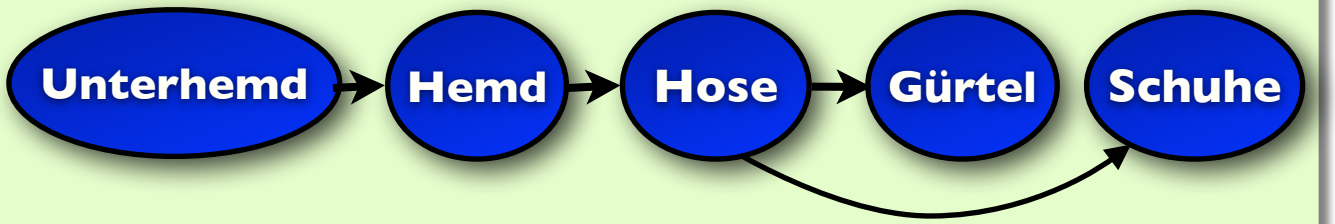
© 2006-2024 Jörg Striegnitz

Topologisches Sortieren: Beispiel

B Topologische Sortierung: Algorithmus



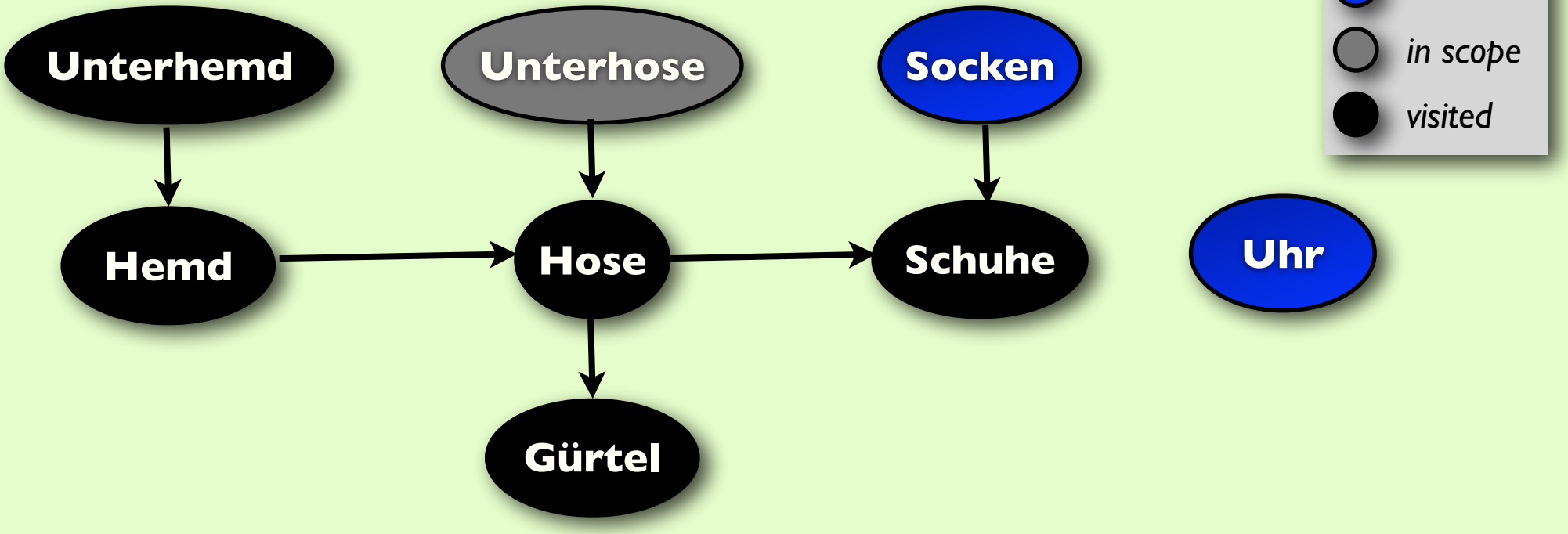
Wir fügen der Liste das **Unterhemd** hinzu. Wir sind mit dem *backtrack* am Ende suchen daher einen neuen Startknoten; .z.B. die **Unterhose**.



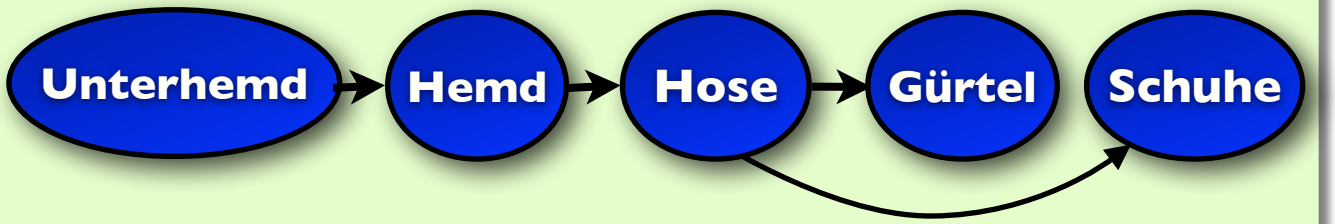
© 2006-2024 Jörg Striegnitz

Topologisches Sortieren: Beispiel

B Topologische Sortierung: Algorithmus



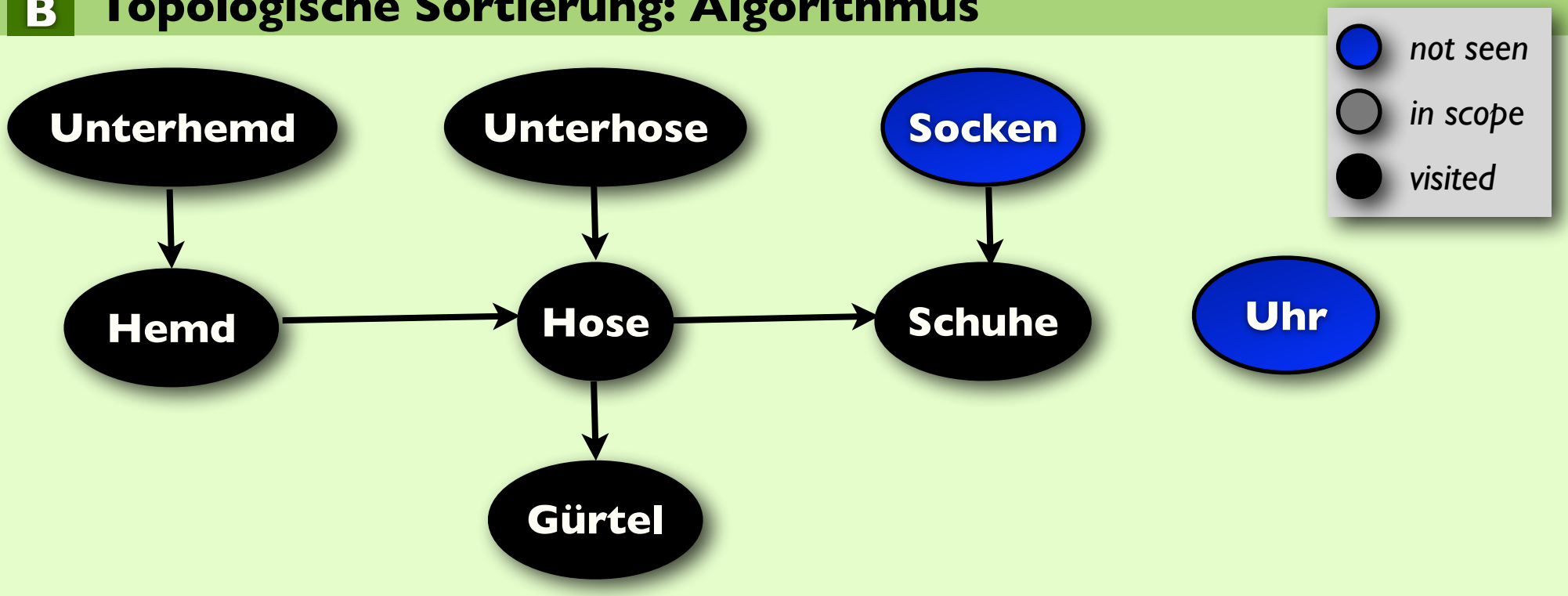
Wir haben bereits alle Nachbarn der **Unterhose** betrachtet. und fügen Sie daher der Liste zu.



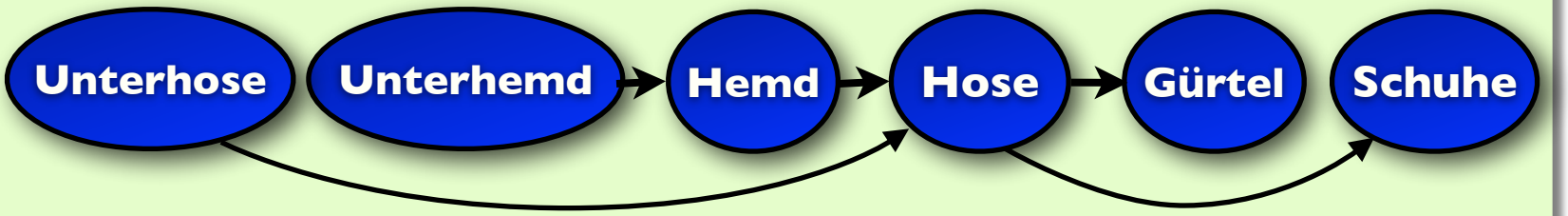
© 2006-2024 Jörg Striegnitz

Topologisches Sortieren: Beispiel

B Topologische Sortierung: Algorithmus



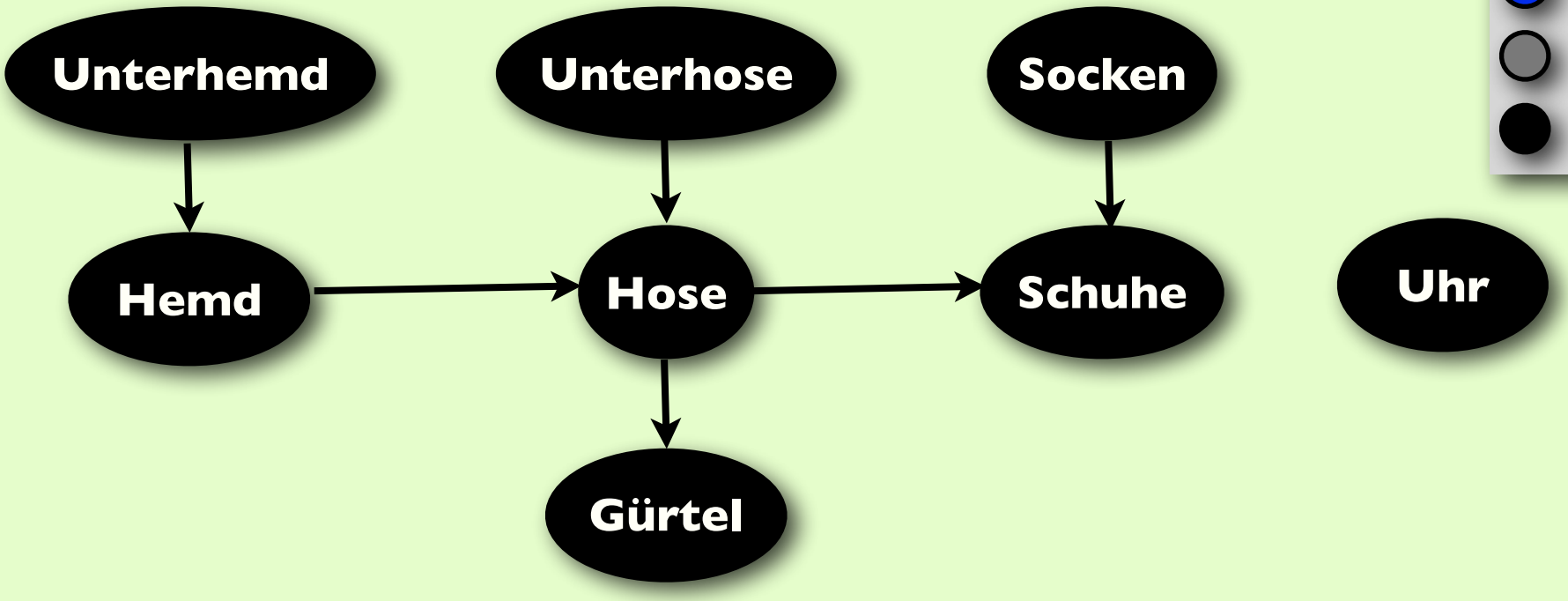
Wir haben bereits alle Nachbarn der **Unterhose** betrachtet. und fügen Sie daher der Liste zu. Analog verfahren wir mit Socken und Uhr ...



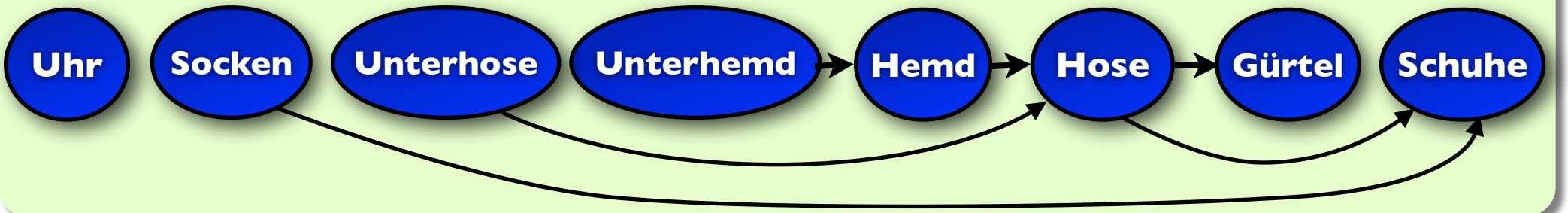
© 2006-2024 Jörg Striegnitz

Topologisches Sortieren: Beispiel

B Topologische Sortierung: Algorithmus



... und erhalten schließlich die folgende topologische Sortierung der Knoten:



© 2006-2024 Jörg Striegnitz

VII. Graphen

7. Graphen

7.1. Einführung und Begriffsbestimmungen
7.2. Traversierung
7.3. Topologisches Sortieren
7.4. Kürzeste Wege
7.4.1. Einführung
7.4.2. *Single-Source Best Path* Algorithmen
7.4.3. *All-Pairs Best Path* Algorithmen
7.5. Maximaler Fluss
7.6. Minimaler Spannbaum

© 2006-2023 Jörg Striegnitz

VII. Graphen

7. Graphen

7.1. Einführung und Begriffsbestimmungen
7.2. Traversierung
7.3. Topologisches Sortieren
7.4. Kürzeste Wege
7.4.1. Einführung
7.4.2. *Single-Source Best Path* Algorithmen
7.4.3. *All-Pairs Best Path* Algorithmen
7.5. Maximaler Fluss
7.6. Minimaler Spannbaum

© 2006-2023 Jörg Striegnitz

Problemstellung

- **Suche den kürzesten Pfad**
  - von einem Knoten zu einem anderen
  - von einem Knoten zu allen anderen (***Single Source Best Path***)
  - von allen Knoten zu allen anderen (***All Pairs Best Path***)

© 2006-2024 Jörg Striegnitz

Problemstellung

- **Suche den kürzesten Pfad**
  - von einem Knoten zu einem anderen
  - von einem Knoten zu allen anderen (***Single Source Best Path***)
  - von allen Knoten zu allen anderen (***All Pairs Best Path***)

- **Etwas verwirrend ...**
  - ... aber leider in der Literatur nahezu durchgängig so zu finden:
  - Gemeint ist hier **nicht** die Pfadlänge, sondern das **minimale Gewicht** eines Pfades (manchmal auch das maximale Gewicht)

© 2006-2024 Jörg Striegnitz

Problemstellung

- **Suche den kürzesten Pfad**
  - von einem Knoten zu einem anderen
  - von einem Knoten zu allen anderen (***Single Source Best Path***)
  - von allen Knoten zu allen anderen (***All Pairs Best Path***)

- **Etwas verwirrend ...**
  - ... aber leider in der Literatur nahezu durchgängig so zu finden:
  - Gemeint ist hier **nicht** die Pfadlänge, sondern das **minimale Gewicht** eines Pfades (manchmal auch das maximale Gewicht)

D **Pfadgewicht**

Sei $G = (V, E)$ ein gewichteter Graph mit Gewichtsfunktion weight : $E \to \mathbb{R}$ und $p = \langle v_1, \ldots, v_n \rangle$ ein Pfad. Dann ist das **Pfadgewicht** weight definiert durch

$$\text{weight}(p) = \sum_{i=1}^{n-1} \text{weight}(v_i, v_{i+1})$$

© 2006-2024 Jörg Striegnitz

Formalisierung der Aufgabe

- **Gegeben:** Startknoten (Source) v_s und Endknoten (Target) v_t

© 2006-2024 Jörg Striegnitz

Formalisierung der Aufgabe

- **Gegeben:** Startknoten (Source) v_s und Endknoten (Target) v_t
- **Aufgabe:** finde den Pfad mit minimalen Kosten von v_s nach v_t; d.h. den Pfad

$$u_0^j = \langle u_0, u_2, \ldots, u_j \rangle$$

mit Startknoten v_s=u_0 und Endknoten v_t = u_j mit

$$C(v_s, v_t) = \min_{j, u_0^j : u_0 = v_s, u_j = v_t} \sum_{i=1}^{j} \text{weight}(u_{i-1}, u_i)$$

Lesart: Minimum über alle Pfadlängen (j) und alle Pfade der Länge j (u_0^j), für die gilt, dass sie vom Startknoten v_s ($u_0 = v_s$) zum Endknoten v_t ($u_j = v_t$) führen.

© 2006-2024 Jörg Striegnitz

Formalisierung der Aufgabe

- **Gegeben:** Startknoten (Source) v_s und Endknoten (Target) v_t
- **Aufgabe:** finde den Pfad mit minimalen Kosten von v_s nach v_t; d.h. den Pfad

$$u_0^j = \langle u_0, u_2, \ldots, u_j \rangle$$

mit Startknoten v_s=u_0 und Endknoten v_t = u_j mit

$$C(v_s, v_t) = \min_{j, u_0^j : u_0 = v_s, u_j = v_t} \sum_{i=1}^{j} \text{weight}(u_{i-1}, u_i)$$

Lesart: Minimum über alle Pfadlängen (j) und alle Pfade der Länge j (u_0^j), für die gilt, dass sie vom Startknoten v_s ($u_0 = v_s$) zum Endknoten v_t ($u_j = v_t$) führen.

- Es wird sich zeigen, dass die allgemeinere Aufgabe, die besten Pfade von v_s zu **allen Knoten** zu bestimmen, nicht mehr Aufwand erfordert!

© 2006-2024 Jörg Striegnitz

VII. Graphen

7. Graphen

7.1. Einführung und Begriffsbestimmungen
7.2. Traversierung
7.3. Topologisches Sortieren
7.4. Kürzeste Wege
7.4.1. Einführung
7.4.2. Single-Source Best Path Algorithmen
7.4.3. *All-Pairs Best Path* Algorithmen
7.5. Maximaler Fluss
7.6. Minimaler Spannbaum

© 2006-2023 Jörg Striegnitz

Die Operation „*Relax*“

- Die hier vorgestellten Algorithmen verwenden im Kern eine Operation, die **Relaxation** (Entspannung nach einer Anspannung) genannt wird.

© 2006-2024 Jörg Striegnitz

Die Operation „*Relax*“

- Die hier vorgestellten Algorithmen verwenden im Kern eine Operation, die **Relaxation** (Entspannung nach einer Anspannung) genannt wird.
- Jeder Knoten v erhält ein Attribut $\boldsymbol{d[v]}$
  Obergrenze der Kosten eines Pfades vom Startknoten v_s nach v

© 2006-2024 Jörg Striegnitz

Die Operation „*Relax*“

- Die hier vorgestellten Algorithmen verwenden im Kern eine Operation, die **Relaxation** (Entspannung nach einer Anspannung) genannt wird.
- Jeder Knoten v erhält ein Attribut $\boldsymbol{d[v]}$
  Obergrenze der Kosten eines Pfades vom Startknoten v_s nach v
- Relaxieren (Entspannen) einer Kante (u, v) bedeutet zu prüfen, ob man einen günstigeren Pfad von v_s nach v erhält, wenn man über u geht
  (Kosten=Grad der Anspannung - Relaxation=Lösen der Anspannung)

© 2006-2024 Jörg Striegnitz

Die Operation „*Relax*“

- Die hier vorgestellten Algorithmen verwenden im Kern eine Operation, die **Relaxation** (Entspannung nach einer Anspannung) genannt wird.
- Jeder Knoten v erhält ein Attribut $\boldsymbol{d[v]}$
  Obergrenze der Kosten eines Pfades vom Startknoten v_s nach v
- Relaxieren (Entspannen) einer Kante (u, v) bedeutet zu prüfen, ob man einen günstigeren Pfad von v_s nach v erhält, wenn man über u geht
  (Kosten=Grad der Anspannung - Relaxation=Lösen der Anspannung)

Funktionaler Teil von **relax**

$$\text{relax}(u, v, \text{weight}) = \begin{cases} d[u] + \text{weight}(u, v) & \text{falls } d[u] + \text{weight}(u, v) < d[v] \\ d[v] & \text{sonst} \end{cases}$$

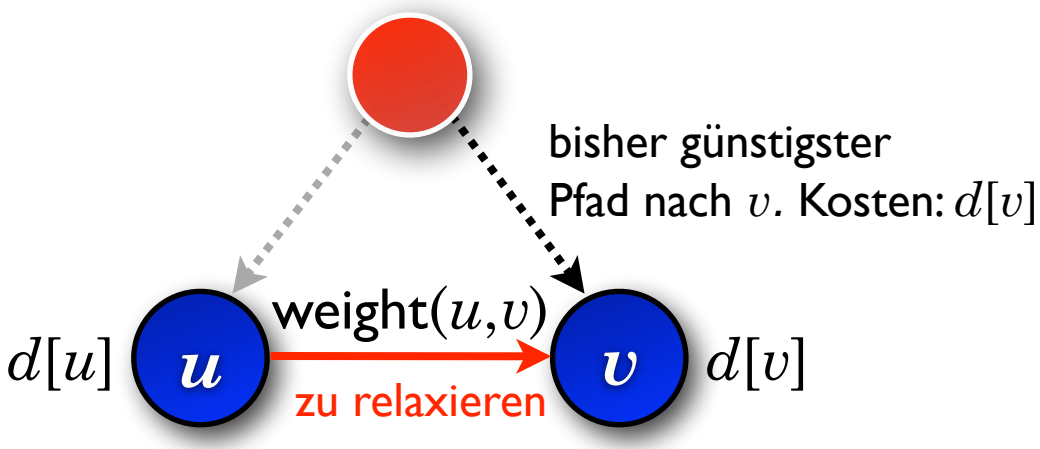
© 2006-2024 Jörg Striegnitz

Die Operation „*Relax*“

- Die hier vorgestellten Algorithmen verwenden im Kern eine Operation, die **Relaxation** (Entspannung nach einer Anspannung) genannt wird.
- Jeder Knoten v erhält ein Attribut $\boldsymbol{d[v]}$
  Obergrenze der Kosten eines Pfades vom Startknoten v_s nach v
- Relaxieren (Entspannen) einer Kante (u, v) bedeutet zu prüfen, ob man einen günstigeren Pfad von v_s nach v erhält, wenn man über u geht
  (Kosten=Grad der Anspannung - Relaxation=Lösen der Anspannung)

Funktionaler Teil von **relax**

$$\text{relax}(u, v, \text{weight}) = \begin{cases} d[u] + \text{weight}(u, v) & \text{falls } d[u] + \text{weight}(u, v) < d[v] \\ d[v] & \text{sonst} \end{cases}$$



Setze $d[v]$=relax$(u,v,$weight$)$
(ggf. entsteht neue Obergrenze)

© 2006-2024 Jörg Striegnitz

Implementierung von `relax` (JAVA)

```
public class SingleSourceBestPath<R extends Weight> {
  public ArrayList<R> d = new ArrayList<R>(); // Kosten
  int sourceId=0;       // id des Startknotens

  public SingleSourceBestPath(int nodeCount, int sId,
                              R initialDist, R inftyVal) {
    // Zu Beginn sind Kosten maximal (wir wissen es nicht besser …)
    for (int i=0 ; i<nodeCount ; ++i) d.add(inftyVal);
    sourceId=sId ;  d.set(sourceId, initialDist);
  }

  public <V> boolean relax(int u, int v, WeightedGraph<V,R> g) {
    if (g.connected(u, v)) {
      R c_u = d.get(u), c_v = d.get(v); // Aktuelle Obergrenze für u & v
      R c_uv = g.getWeight(c_u, c_v);   // Kantengewicht u->v
      R newDist = c_u.add( c_uv );
      if ( newDist.compareTo(c_v) < 0 ) { // Weg über u günstiger?
        d.set(c_v, newDist );             // ... dann nehmen wir den!
        return true;
      }
    }
    return false;
  }
}
```


© 2006-2024 Jörg Striegnitz

Implementierung von `relax` (JAVA)

```
public class SingleSourceBestPath<R extends Weight> {
  public ArrayList<R> d = new ArrayList<R>(); // Kosten
  int sourceId=0;       // id des Startknotens

  public SingleSourceBestPath(int nodeCount, int sId,
                              R initialDist, R inftyVal) {
    // Zu Beginn sind Kosten maximal (wir wissen es nicht besser …)
    for (int i=0 ; i<nodeCount ; ++i) d.add(inftyVal);
    sourceId=sId ;  d.set(sourceId, initialDist);
  }

  public <V> boolean relax(int u, int v, WeightedGraph<V,R> g) {
    if (g.connected(u, v)) {
      R c_u = d.get(u), c_v = d.get(v); // Aktuelle Obergrenze für u & v
      R c_uv = g.getWeight(c_u, c_v);   // Kantengewicht u->v
      R newDist = c_u.add( c_uv );
      if ( newDist.compareTo(c_v) < 0 ) { // Weg über u günstiger?
        d.set(c_v, newDist );             // ... dann nehmen wir den!
        return true;
      }
    }
    return false;
  }
}
```

Wichtig! d wird angepasst!

© 2006-2024 Jörg Striegnitz

Ein Algorithmus für azyklische Graphen

- **Idee:** Kombiniere topologisches Sortieren / Relaxation
  - **Schritt I:** Initialisiere $d[v]$
    - an der Stelle v_s Abstand 0; ansonsten Abstand ∞ *(die größtmögliche Obergrenze)*
  - **Schritt 2:** topologisches Sortieren von G liefere

$$u_{\mathrm{o}},..,u_{\mathrm{n}}$$

  - **Schritt 3:**
    - Für i=0 bis n *(* Knoten in topologischer Reihenfolge *)*
    - Für alle Nachbarn v von u_{i} *(* v = alle von u_i erreichbaren Knoten *)*
      (Anmerkung: diese stehen alle **rechts von u_i**! *)*
    - $\mathrm{d}[v]$=$\mathrm{relax}(u_i,v,$weight) *(* Besser über u_i nach v? *)*

© 2006-2024 Jörg Striegnitz

Ein Algorithmus für azyklische Graphen

- **Idee:** Kombiniere topologisches Sortieren / Relaxation
  - **Schritt I:** Initialisiere $d[v]$
    - an der Stelle v_s Abstand 0; ansonsten Abstand ∞ *(die größtmögliche Obergrenze)*
  - **Schritt 2:** topologisches Sortieren von G liefere

$$u_0,..,u_n$$

  - **Schritt 3:**
    - Für i=0 bis n *(* Knoten in topologischer Reihenfolge *)*
    - Für alle Nachbarn v von u_i *(* v = alle von u_i erreichbaren Knoten *)*
      (Anmerkung: diese stehen alle **rechts von u_i**! *)*
    - $d[v]=\mathrm{relax}(u_i,v,\text{weight})$ *(* Besser über u_i nach v? *)*

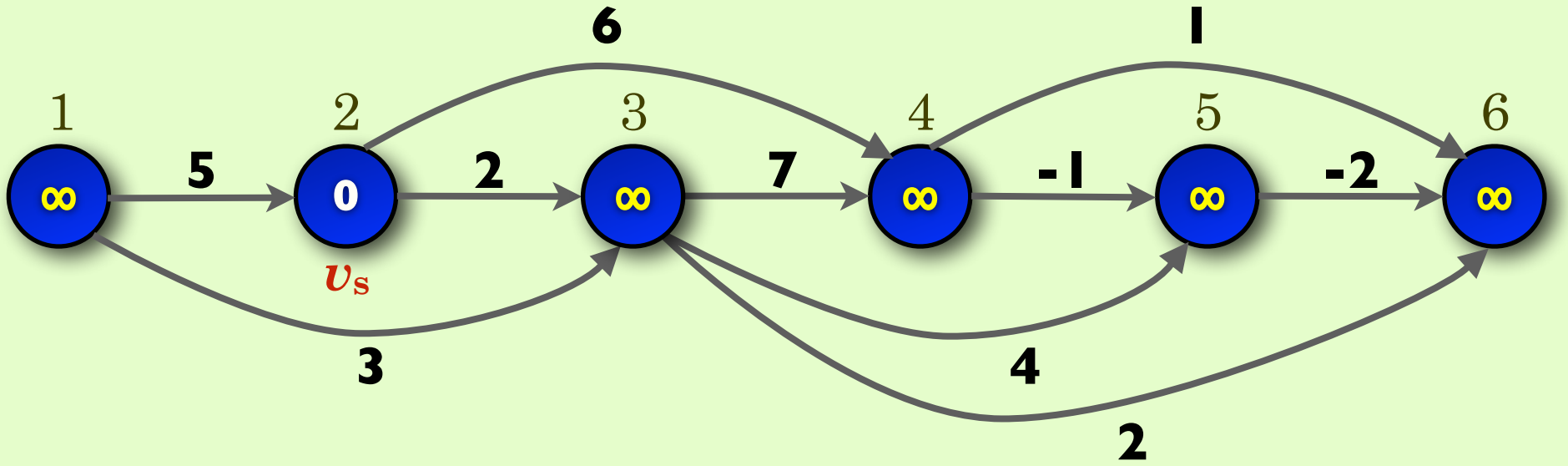
- **Aufwand**
  - Initialisierung: $\Theta(|V|)$
  - topologisches Sortieren $\Theta(|V|+|E|)$
  - 3. Schritt ebenfalls $\Theta(|V|+|E|)$
    - Jeder Knoten wird einmal betrachtet (äußere Schleife)
    - Jede Kante, die von diesem Knoten abgeht wird genau einmal betrachtet; folglich wird jede Kante des Graphen genau einmal durchlaufen
  - Also: insgesamt

$$\boxed{\Theta(|V|+|E|)}$$

© 2006-2024 Jörg Striegnitz

Kürzeste Pfade mit topologischem Sortieren I

B *Single Source Best Path* mit Topologischem Sortieren

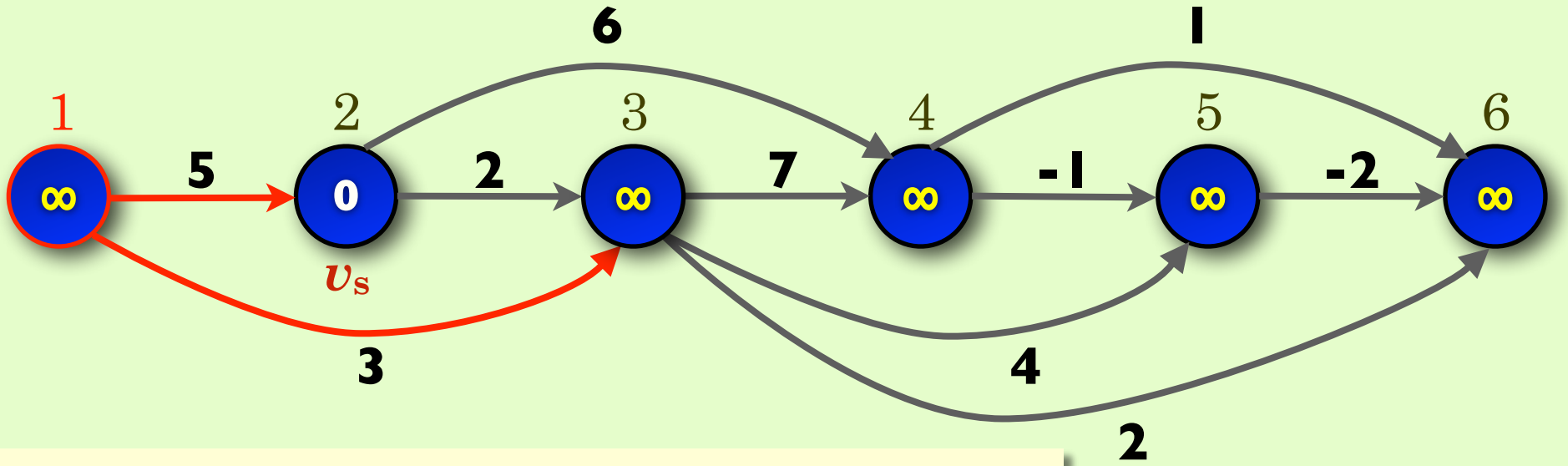


$$\text{relax}(u, v, \text{weight}) = \begin{cases} d[u] + \text{weight}(u, v) & \text{falls } d[u] + \text{weight}(u, v) < d[v] \\ d[v] & \text{sonst} \end{cases}$$

© 2006-2024 Jörg Striegnitz

Kürzeste Pfade mit topologischem Sortieren I

B *Single Source Best Path* mit Topologischem Sortieren



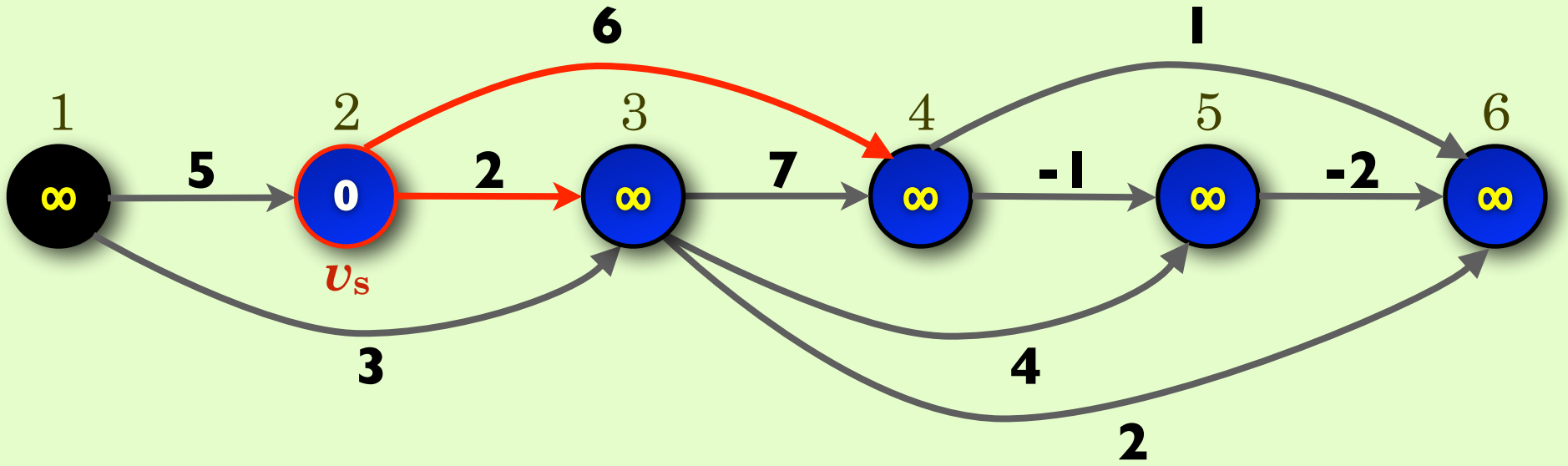
Relaxieren bringt keine weitere Einschränkung. Knoten 1 ist vom Startknoten v_s (2) nicht erreichbar!

$$\text{relax}(u, v, \text{weight}) = \begin{cases} d[u] + \text{weight}(u, v) & \text{falls } d[u] + \text{weight}(u, v) < d[v] \\ d[v] & \text{sonst} \end{cases}$$

© 2006-2024 Jörg Striegnitz

Kürzeste Pfade mit topologischem Sortieren I

B *Single Source Best Path* mit Topologischem Sortieren



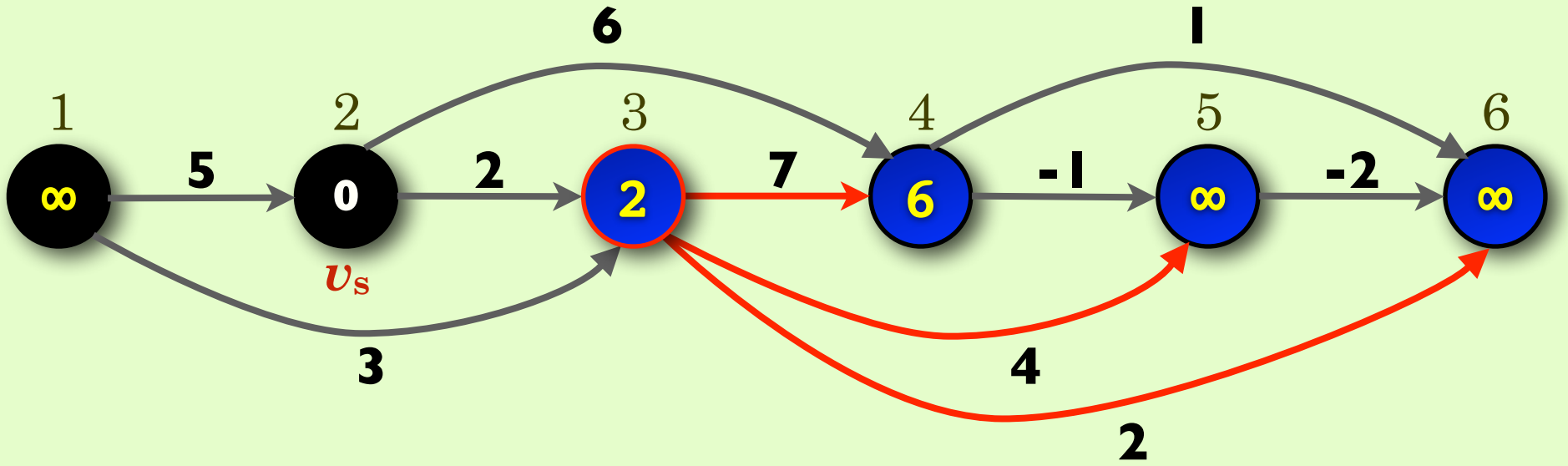
Knoten 3 und 4 können günstiger erreicht werden!

$$\text{relax}(u, v, \text{weight}) = \begin{cases} d[u] + \text{weight}(u, v) & \text{falls } d[u] + \text{weight}(u, v) < d[v] \\ d[v] & \text{sonst} \end{cases}$$

© 2006-2024 Jörg Striegnitz

Kürzeste Pfade mit topologischem Sortieren I

B *Single Source Best Path* mit Topologischem Sortieren



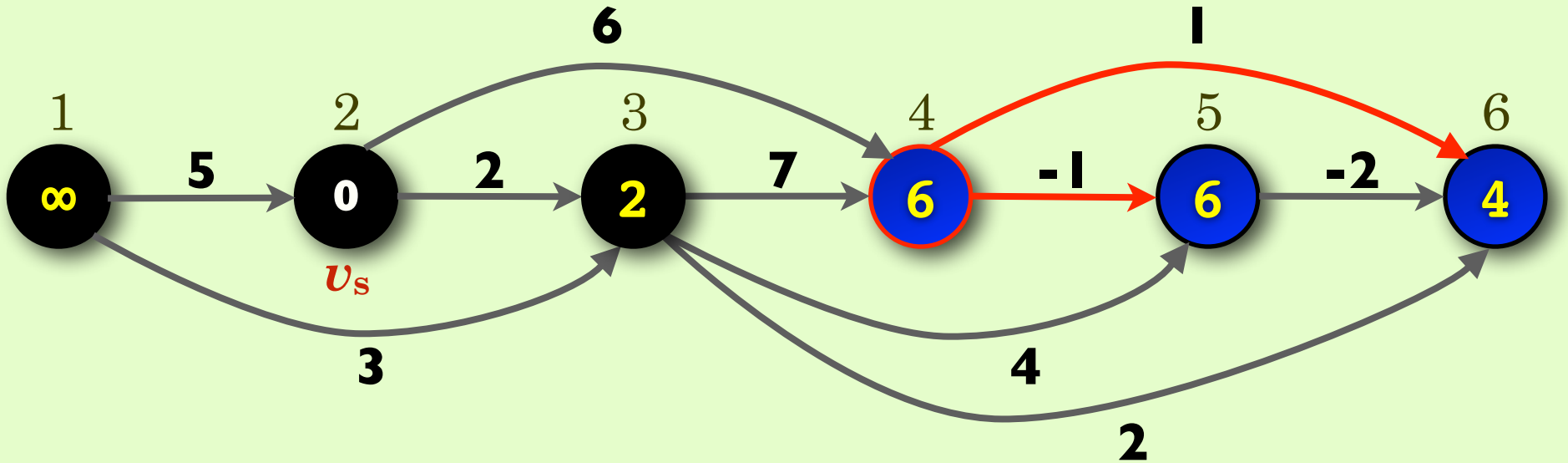
Knoten 4 kann via Knoten 2 mit Kosten 2+7=9 erreicht werden - das ist teurer! Knoten 5 und 6 sind jetzt aber erreichbar ...

$$\text{relax}(u, v, \text{weight}) = \begin{cases} d[u] + \text{weight}(u, v) & \text{falls } d[u] + \text{weight}(u, v) < d[v] \\ d[v] & \text{sonst} \end{cases}$$

© 2006-2024 Jörg Striegnitz

Kürzeste Pfade mit topologischem Sortieren I

B *Single Source Best Path* mit Topologischem Sortieren



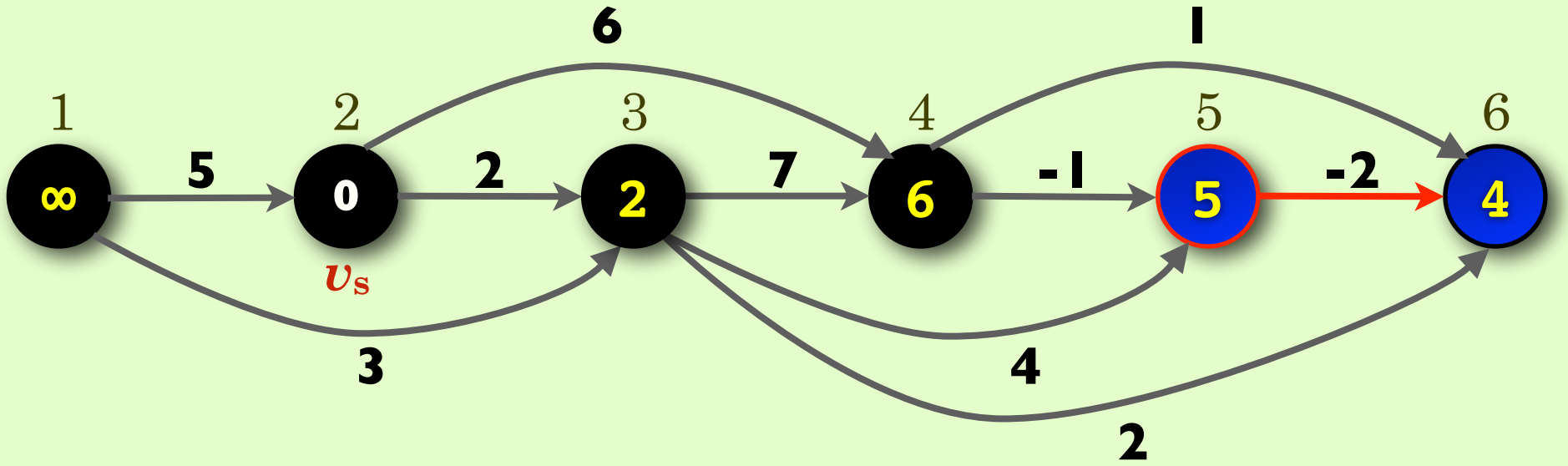
Wir können sparen! Knoten 5 kann via Konten 4 mit Kosten 6-1=5 erreicht werden. Knoten 6 wird via Knoten 4 mit Kosten 6+1=7 jedoch teurer ...

$$\text{relax}(u, v, \text{weight}) = \begin{cases} d[u] + \text{weight}(u, v) & \text{falls } d[u] + \text{weight}(u, v) < d[v] \\ d[v] & \text{sonst} \end{cases}$$

© 2006-2024 Jörg Striegnitz

Kürzeste Pfade mit topologischem Sortieren I

B *Single Source Best Path* mit Topologischem Sortieren



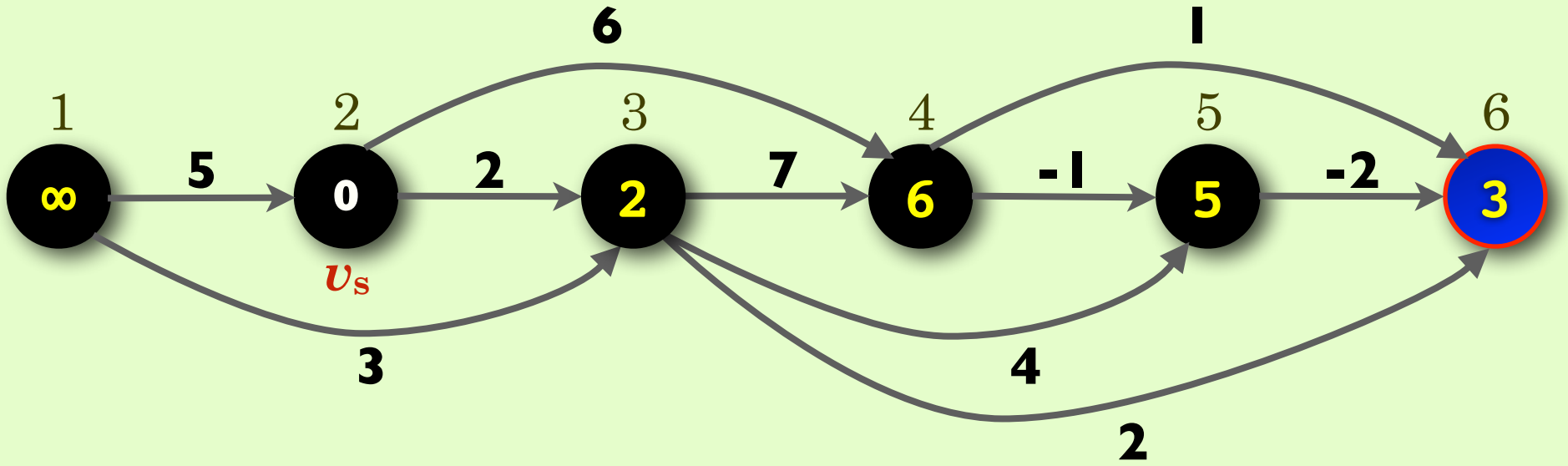
Erneutes Sparpotential. Knoten 6 kann via Knoten 5 mit kosten 5-2=3 erreicht werden.

$$\text{relax}(u, v, \text{weight}) = \begin{cases} d[u] + \text{weight}(u, v) & \text{falls } d[u] + \text{weight}(u, v) < d[v] \\ d[v] & \text{sonst} \end{cases}$$

© 2006-2024 Jörg Striegnitz

Kürzeste Pfade mit topologischem Sortieren I

B *Single Source Best Path* mit Topologischem Sortieren



Fertig!

$$\text{relax}(u, v, \text{weight}) = \begin{cases} d[u] + \text{weight}(u, v) & \text{falls } d[u] + \text{weight}(u, v) < d[v] \\ d[v] & \text{sonst} \end{cases}$$

© 2006-2024 Jörg Striegnitz

Implementierung (JAVA)

```
public class SSBP_TopologicSort_Relax<E extends Weight>
       extends SingleSourceBestPath<E> {

  public SSBP_TopologicSort_Relax(int nodeCount, int sId,
                                  E initialDist,E infty) {
    super(nodeCount, sId, initialDist, infty);
  }

  public <V> void run(WeightedTraversableGraph<V,E> g) {
    TopologicSort<V> ts = new TopologicSort<V>();
    g.traverse(ts); // topologisch sortieren

    // Knoten in topologischer Sortierung abarbeiten:
    for (int u : ts.result()) {
      for (int v=0 ; v<g.numNodes(); ++v) {
        relax(u,v,g); // Kante relaxieren
      }
    }
  }
}
```


© 2006-2024 Jörg Striegnitz

Der Dijkstra Algorithmus

- **Kürzester Weg von einem Knoten zu allen anderen (*Single Source Best Path*)**
- **funktioniert auch mit zyklischen Graphen**
- **Dijkstra publizierte die 3-seitige (!) Originalarbeit 1959**

Edsgar Wybe Dÿkstra

★1930-†2002

© 2006-2024 Jörg Striegnitz

Der Dijkstra Algorithmus

- **Kürzester Weg von einem Knoten zu allen anderen (*Single Source Best Path*)**
- **funktioniert auch mit zyklischen Graphen**
- **Dijkstra publizierte die 3-seitige (!) Originalarbeit 1959**

Edsgar Wybe Dÿkstra

★1930-†2002

„This was the first graph problem I ever posed myself and solved. The amazing thing was that I didn't publish it. It was not amazing at the time. At the time, algorithms were hardly considered a scientific topic."

Dijkstra über seinen Algorithmus

© 2006-2024 Jörg Striegnitz

Die Grundidee 1

- **Voraussetzung: die Gewichtsfunktion erfüllt die Eigenschaft**

$$\text{weight}(v,w) = \begin{cases} \geq 0 & \text{falls } (v,w) \in E \\ \infty & \text{falls } (v,w) \notin E \\ 0 & \text{falls } v = w \end{cases}$$

© 2006-2024 Jörg Striegnitz

Die Grundidee 1

- **Voraussetzung: die Gewichtsfunktion erfüllt die Eigenschaft**

$$\text{weight}(v,w) = \begin{cases} \geq 0 & \text{falls } (v,w) \in E \\ \infty & \text{falls } (v,w) \notin E \\ 0 & \text{falls } v = w \end{cases}$$

Gewicht ≥ 0!

© 2006-2024 Jörg Striegnitz

Die Grundidee 1

- **Voraussetzung: die Gewichtsfunktion erfüllt die Eigenschaft**

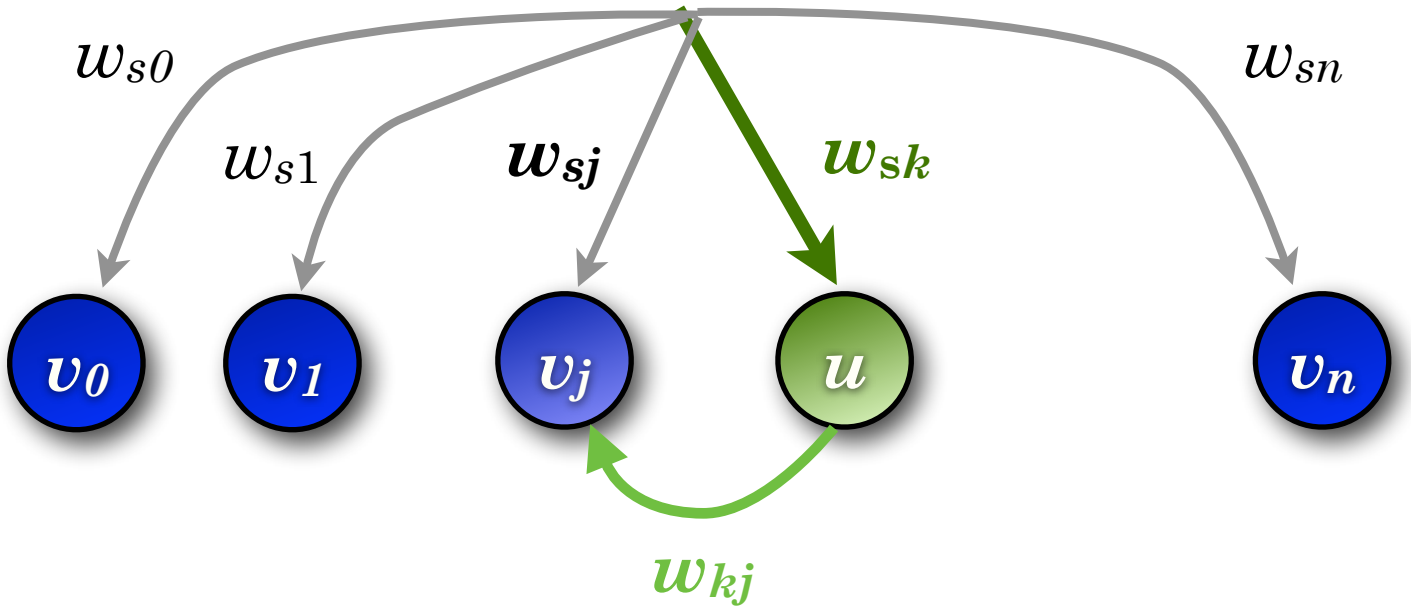
$$\text{weight}(v,w) = \begin{cases} \geq 0 & \text{falls } (v,w) \in E \\ \infty & \text{falls } (v,w) \notin E \\ 0 & \text{falls } v = w \end{cases}$$

Gewicht ≥ 0!

- **Idee von Dijkstra: erweitere sukzessiv bekannte beste Pfade**
  - **Hinzunahme einer Kante kann das Gewicht nicht verringern!**
  - Menge S enthält Knoten, zu denen beste Pfade bereits bekannt
    Initial: Knoten, der am günstigsten von v_s erreicht werden kann
  - Wähle günstigsten (kleinstes $d[]$) Knoten u aus V - S
    organisiere Knoten aus V-S in Priority-Queue für effiziente Auswahl
  - Relaxiere **alle** von u ausgehenden Kanten und füge u zu S hinzu

© 2006-2024 Jörg Striegnitz

Die Grundidee 2:



- w_{si} $(0 \leq i \leq n)$ sind optimale Kosten für Pfad der Länge 1
- Falls $w_{sk} = \min\{w_{s0},\ldots,w_{sn}\}$, so ist v_s, u der günstigste Pfad von v_s nach u
  negative Kanten sind ausgeschlossen - Hinzunahme einer Kante kann Kosten nur erhöhen!
- Ein „Umweg“ über u könnte günstigeren Pfad zu anderem Knoten liefern; z.B. zu v_j, wenn $w_{sk} + w_{kj} < w_{sj}$
- **Daher**: relaxiere alle v_i mit u und erhalte so günstigste Pfade der Länge 2.

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Dijkstra - Pseudocode

Initialisiere $d[v]$ *(* an der Stelle v_s Abstand 0; ansonsten Abstand ∞ (die größtmögliche Obergrenze) *)*

$S \leftarrow \emptyset$ *(* Liste der Knoten, für die optimaler Pfad bestimmt wurde *)*

$Q \leftarrow$ V(G) *(* Alle Knoten des Graphen G in die Queue *)*

while Q ≠ ∅

  do $u \leftarrow$ min Q *(* Knoten mit geringsten Pfadkosten auswählen ... *)*

    $Q \leftarrow Q \setminus \{u\}$ *(* ... und aus Queue entfernen *)*

    $S \leftarrow S \cup \{u\}$ *(* Pfad zu diesem Knoten ist optimal - könnte nur länger werden! *)*

    (Für die teureren Pfade könnte der Weg über u eine Abkürzung sein - also alle *)*

    (abgehenden Kanten relaxieren ... *)*

    for each vertex v in Q that is reachable from u

      relax(u,v) *(* **Vorsicht!** implizite Reorganisation der Queue! *)*

© 2006-2024 Jörg Striegnitz

Implementierung (JAVA) I

- **Nicht optimal, da Reorganisation der Queue nicht gut unterstützt wird!**

© 2006-2024 Jörg Striegnitz

Implementierung (JAVA) I

- **Nicht optimal, da Reorganisation der Queue nicht gut unterstützt wird!**

```
public class SSBP_Dijkstra<E extends Weight>
             extends SingleSourceBestPath<E> {

  public SSBP_Dijkstra(int nodeCount, int sId,
                       E initialDist,E infty) {
    super(nodeCount, sId, initialDist, infty);
  }

  public <V> void run(WeightedTraversableGraph<V,E> g) {
    // Knoten werden anhand der Pfadkosten sortiert - diese Info steht im Feld d
    // der Oberklasse. Vorsicht: PriorityQueue braucht strikte Ordnung!
    PriorityQueue<NodeWithId<V> > Q=new PriorityQueue<NodeWithId<V> >(
        g.numNodes(),
        new Comparator<NodeWithId<V> >() {
          public int compare(NodeWithId<V> v1,NodeWithId<V> v2) {
            if (d.get(v1.id).compareTo(d.get(v2.id)) == 0)
               return v1.id - v2.id; // Gleich? Dann entscheidet id
            else
               return d.get(v1.id).compareTo(d.get(v2.id));
          }
       });
```

... Fortsetzung auf nächster Folie ...

© 2006-2024 Jörg Striegnitz

Implementierung (JAVA) II

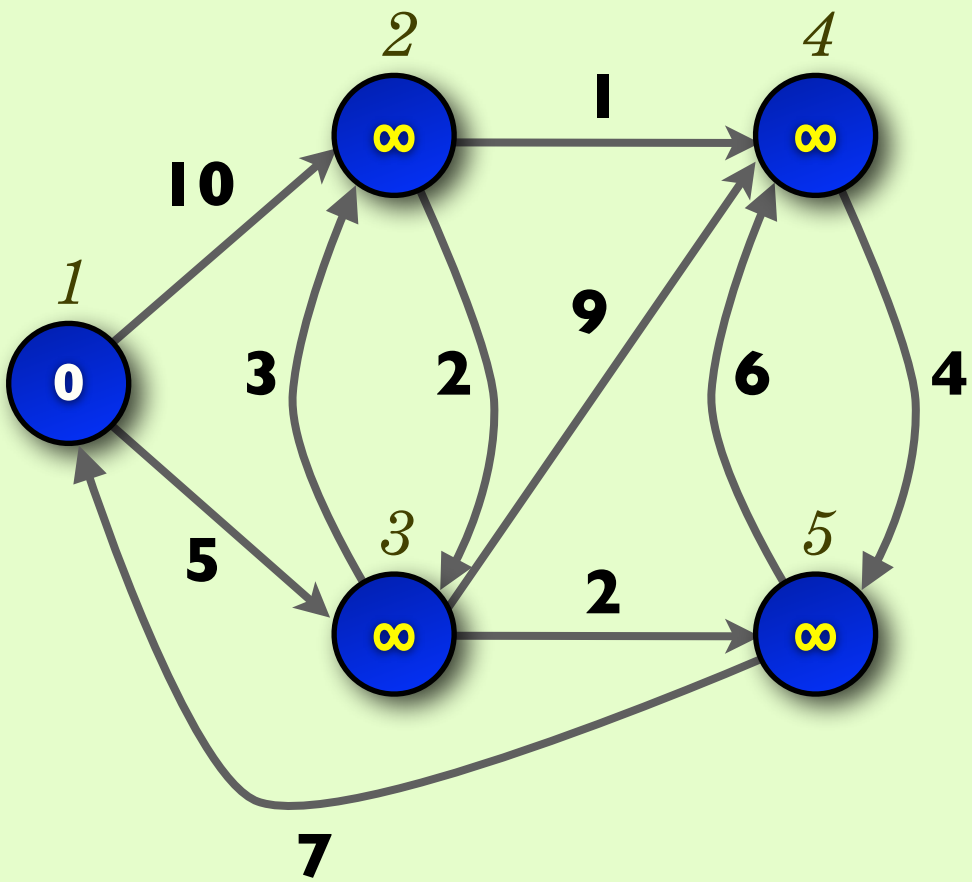
... Fortsetzung von vorheriger Folie ...

```
    // Wird zum Reorganisieren der Queue benötigt
    Vector<NodeWithId<V> > reOrg = new Vector<NodeWithId<V> >();
    // Queue mit allen Knoten füllen
    for (int n=0 ; n<g.numNodes() ; ++n) Q.offer(g.getNodeWithId(n));
    while (! Q.isEmpty()) {
      NodeWithId<V> u = Q.poll();
      reOrg.clear(); // noch nichts zu re-organisieren.
      for (NodeWithId<V> v : Q) {
        if (relax(u.id,v.id,g))
          // Wenn sich der Wert d[v] geändert hat, muss die Queue
          // angepasst werden. Leider fehlt eine Methode „changeKey“
          reOrg.add(v);
      }
      // Queue reorganisieren (besser: zwei Schleifen!)
      for (NodeWithId<V> v : reOrg) {
        Q.remove(v) ; Q.offer(v);
      }
    }
  }
}
```


© 2006-2024 Jörg Striegnitz

Der Algorithmus von Dijkstra: Beispiel

B *Single Source Best Path*: Algorithmus von Dijkstra



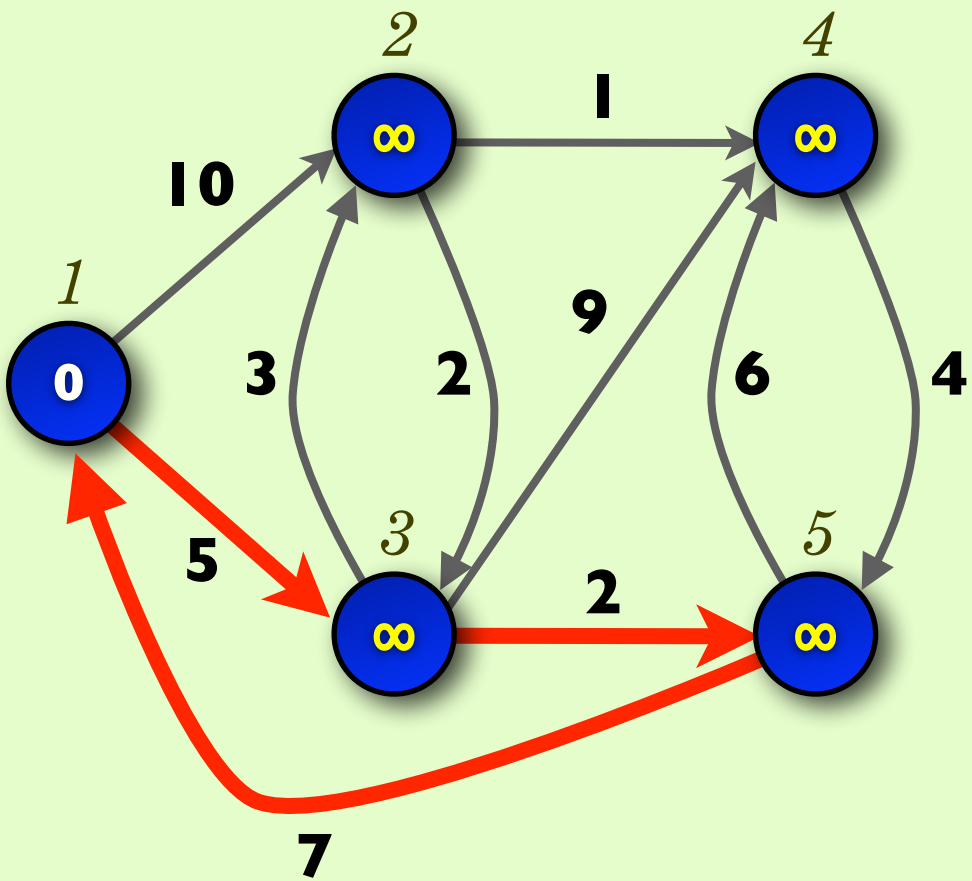
Queue

1	2	3	4	5

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Dijkstra: Beispiel

B *Single Source Best Path*: Algorithmus von Dijkstra



Queue

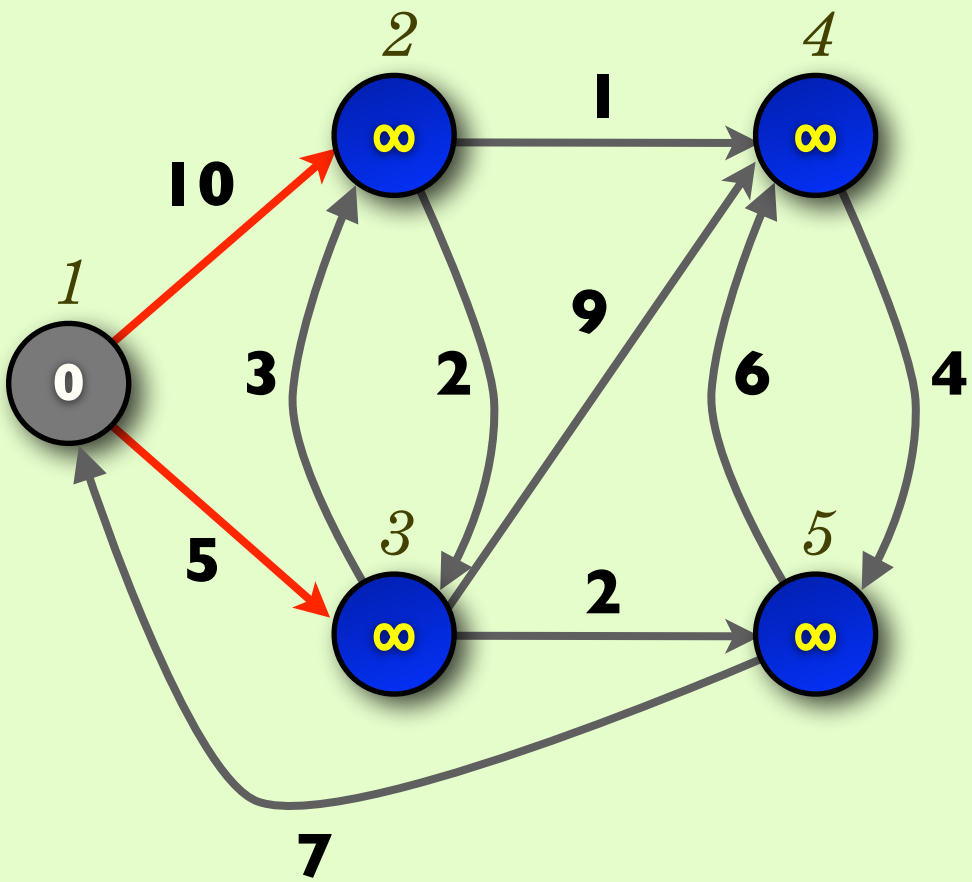
1	2	3	4	5

Beachte: der Graph ist **zyklisch**!

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Dijkstra: Beispiel

B *Single Source Best Path*: Algorithmus von Dijkstra



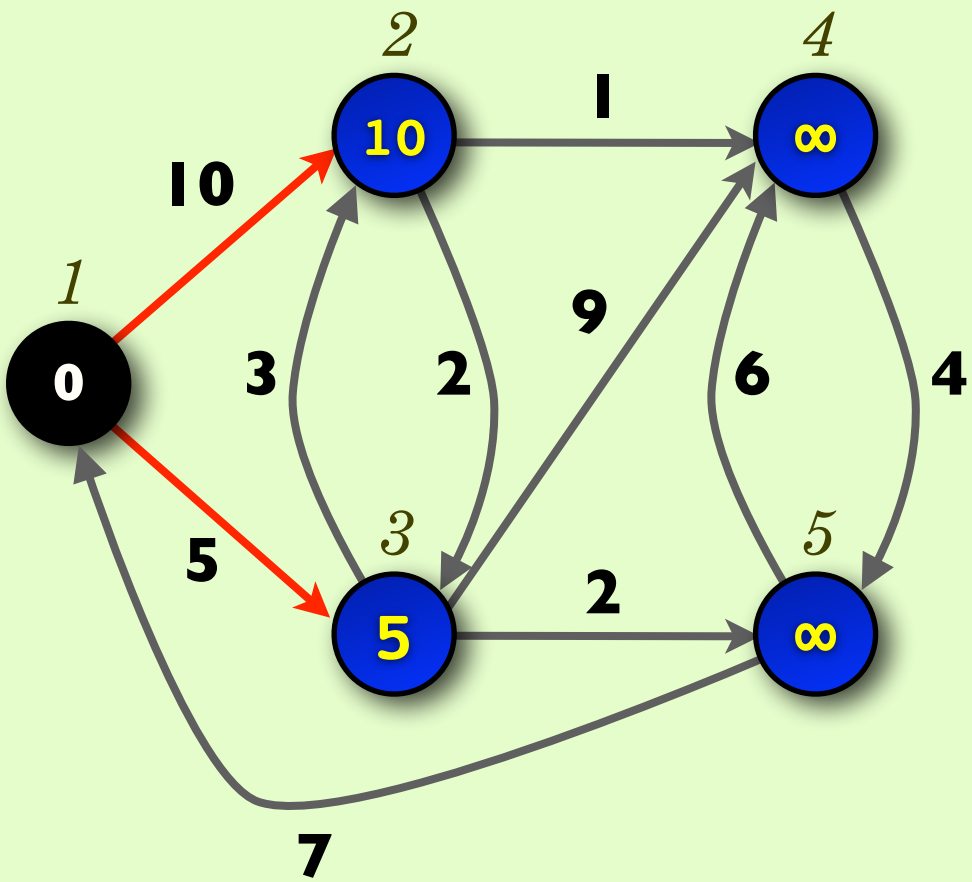
Queue

2	3	4	5

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Dijkstra: Beispiel

B *Single Source Best Path*: Algorithmus von Dijkstra



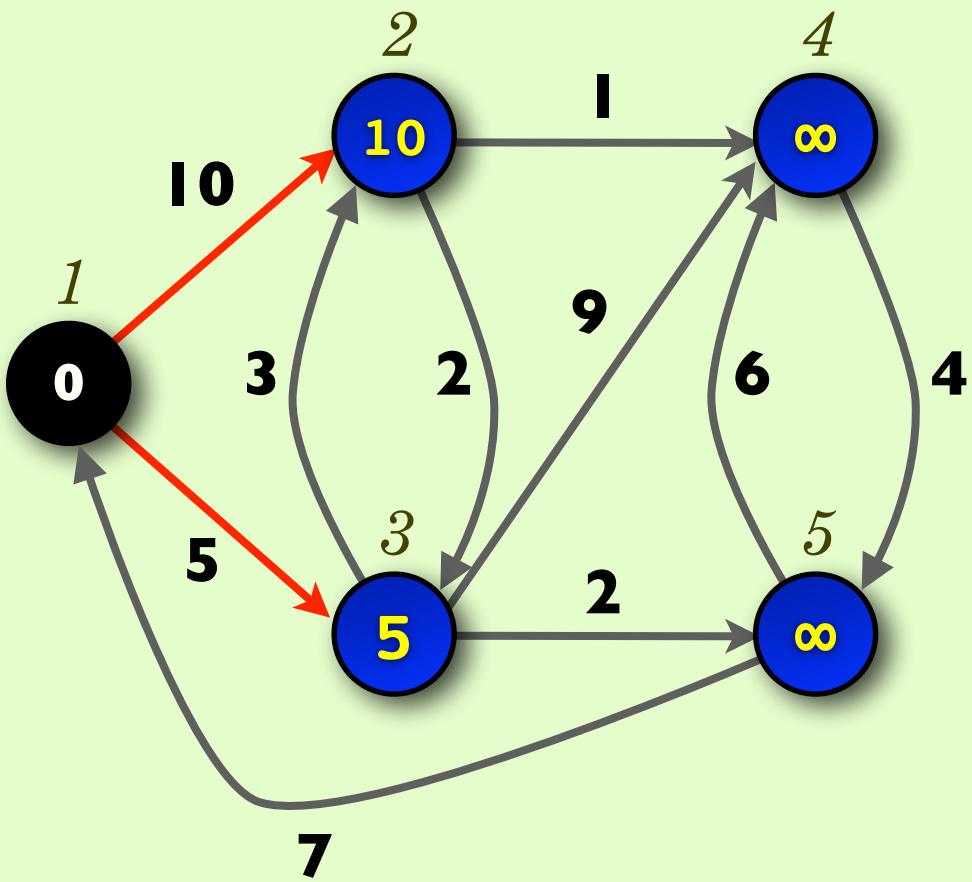
Queue

2	3	4	5

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Dijkstra: Beispiel

B *Single Source Best Path*: Algorithmus von Dijkstra



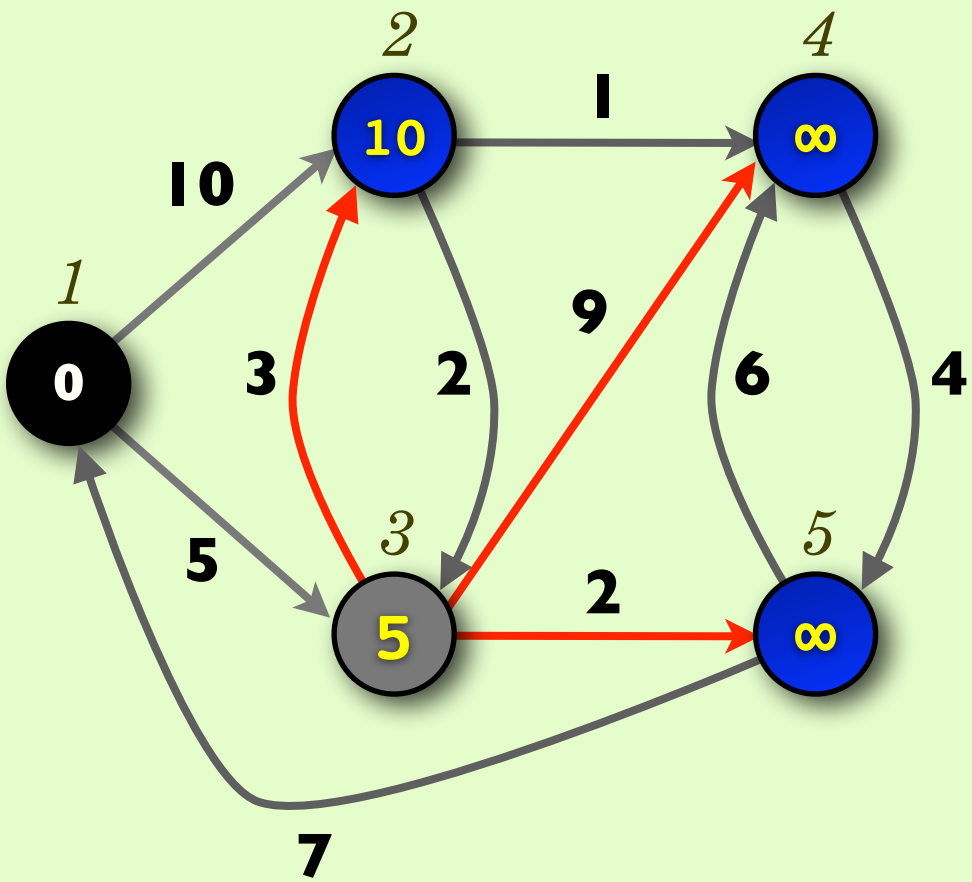
Queue

3	2	4	5

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Dijkstra: Beispiel

B *Single Source Best Path*: Algorithmus von Dijkstra



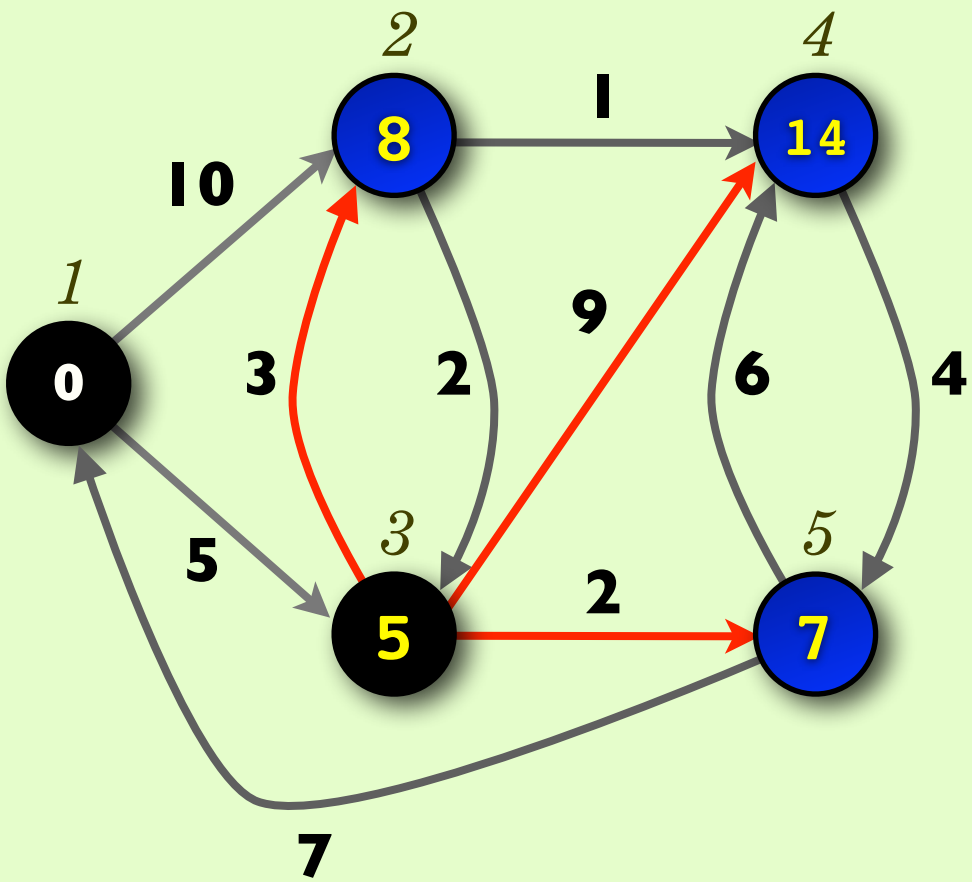
Queue

2	4	5

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Dijkstra: Beispiel

B *Single Source Best Path*: Algorithmus von Dijkstra



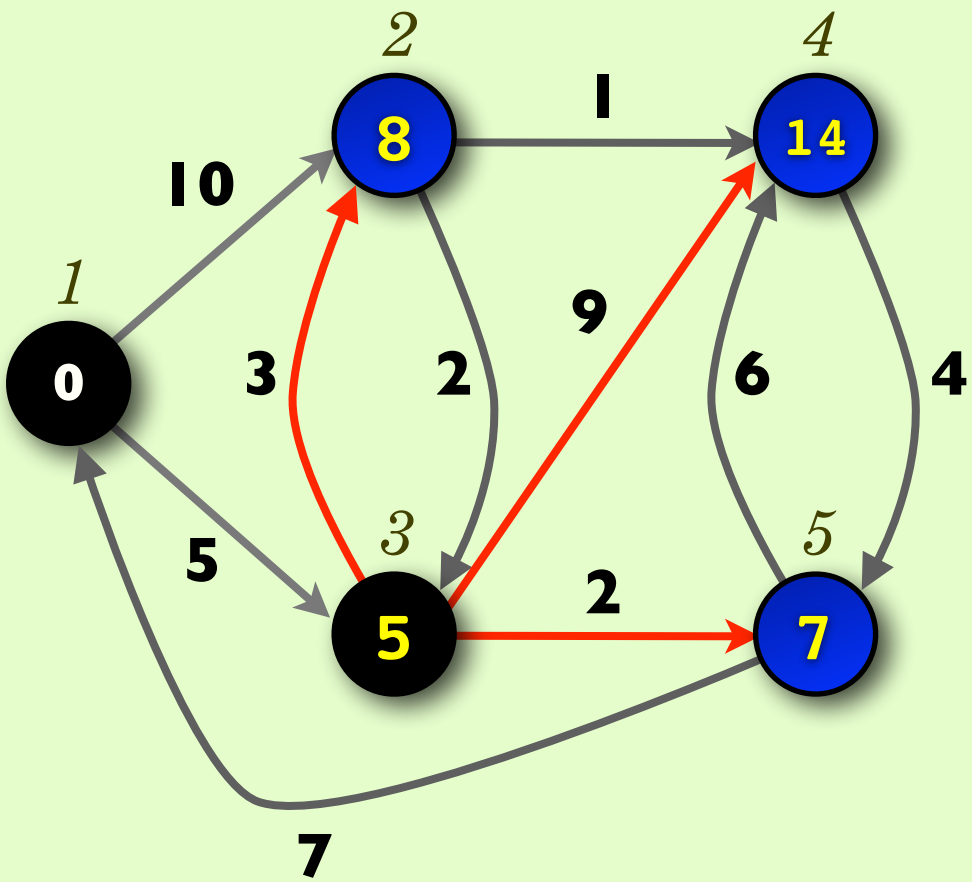
Queue

2	4	5

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Dijkstra: Beispiel

B *Single Source Best Path*: Algorithmus von Dijkstra



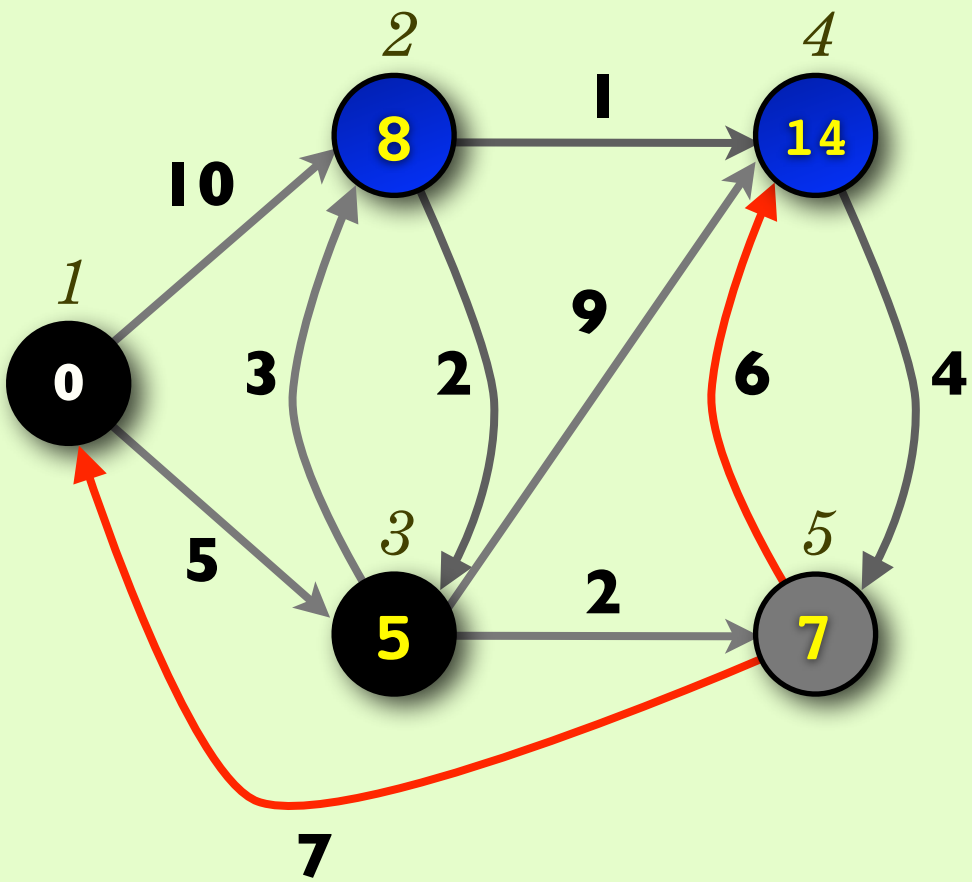
Queue

5	2	4

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Dijkstra: Beispiel

B *Single Source Best Path*: Algorithmus von Dijkstra



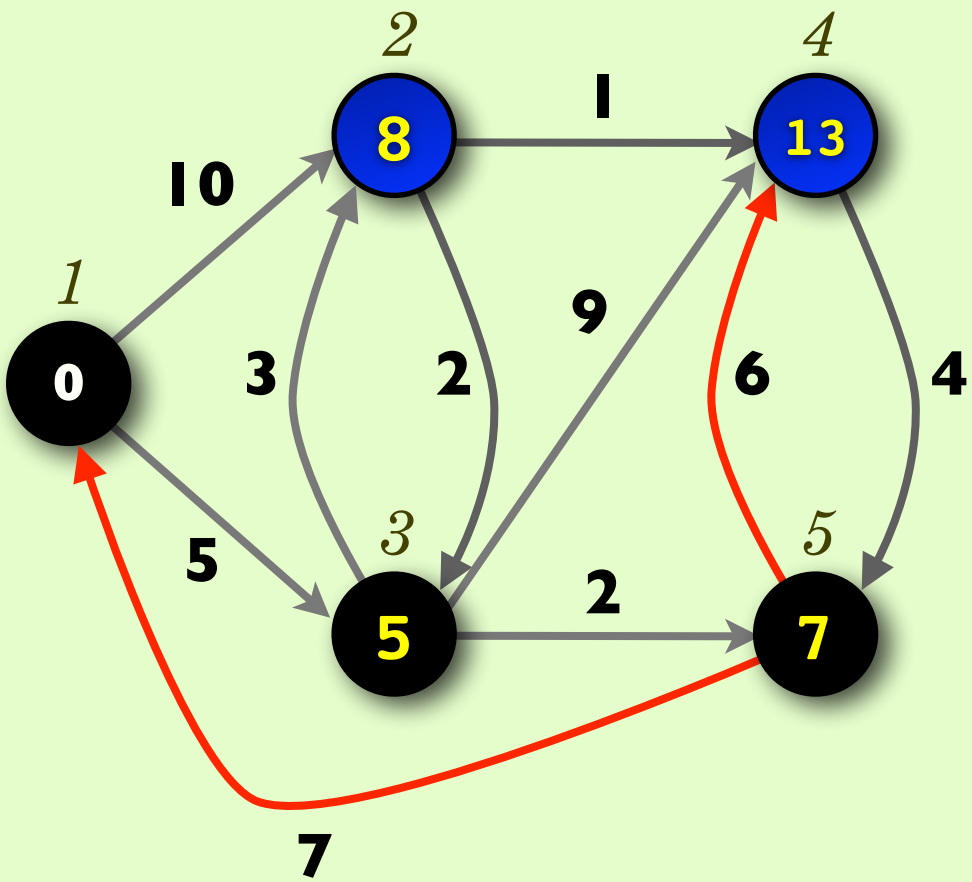
Queue

2	4

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Dijkstra: Beispiel

B *Single Source Best Path*: Algorithmus von Dijkstra



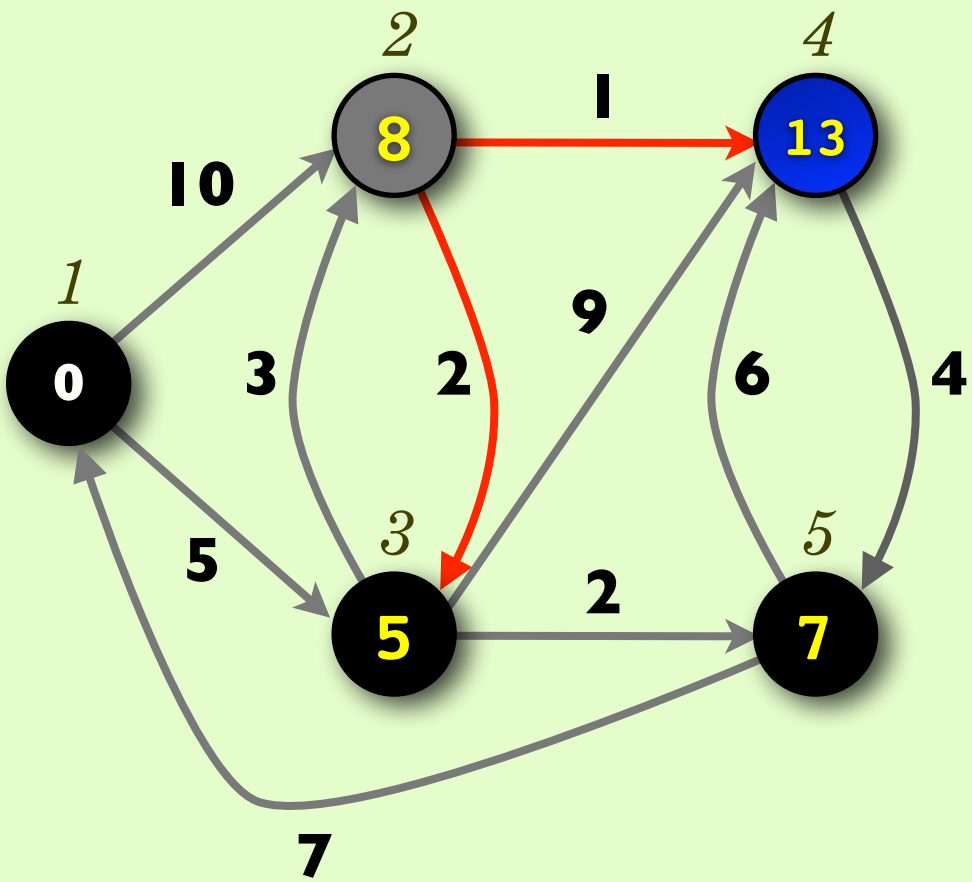
Queue

2	4

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Dijkstra: Beispiel

B *Single Source Best Path*: Algorithmus von Dijkstra

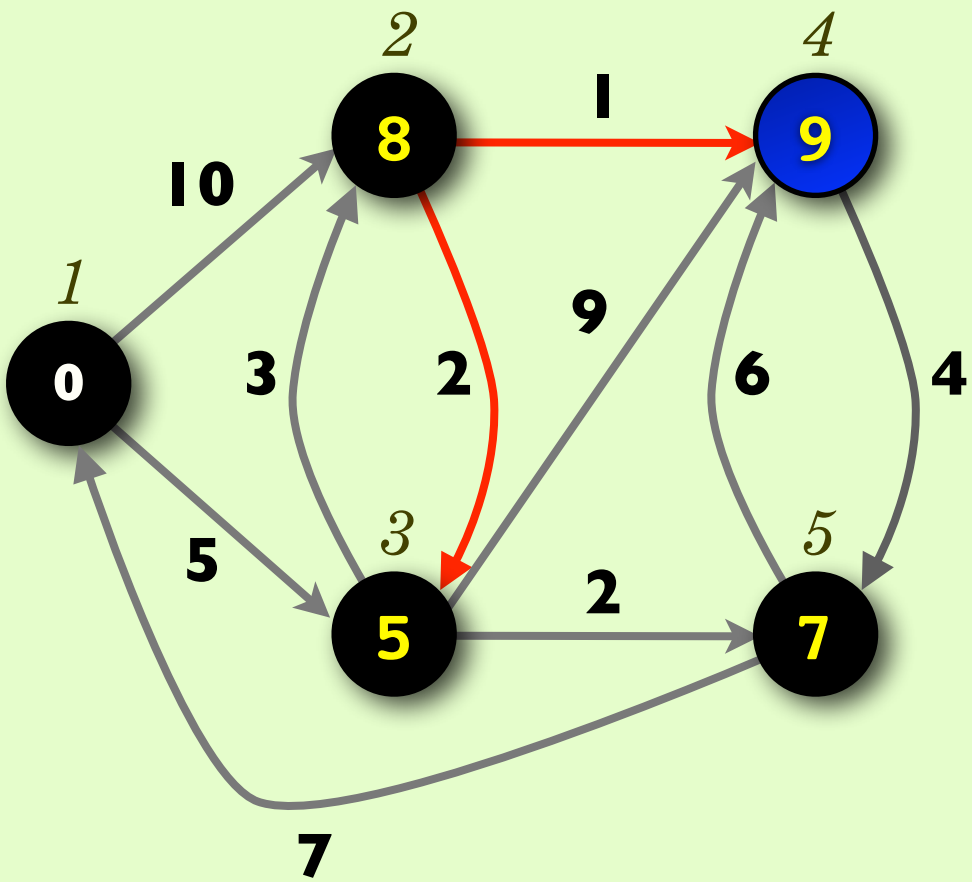


Queue

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Dijkstra: Beispiel

B *Single Source Best Path*: Algorithmus von Dijkstra

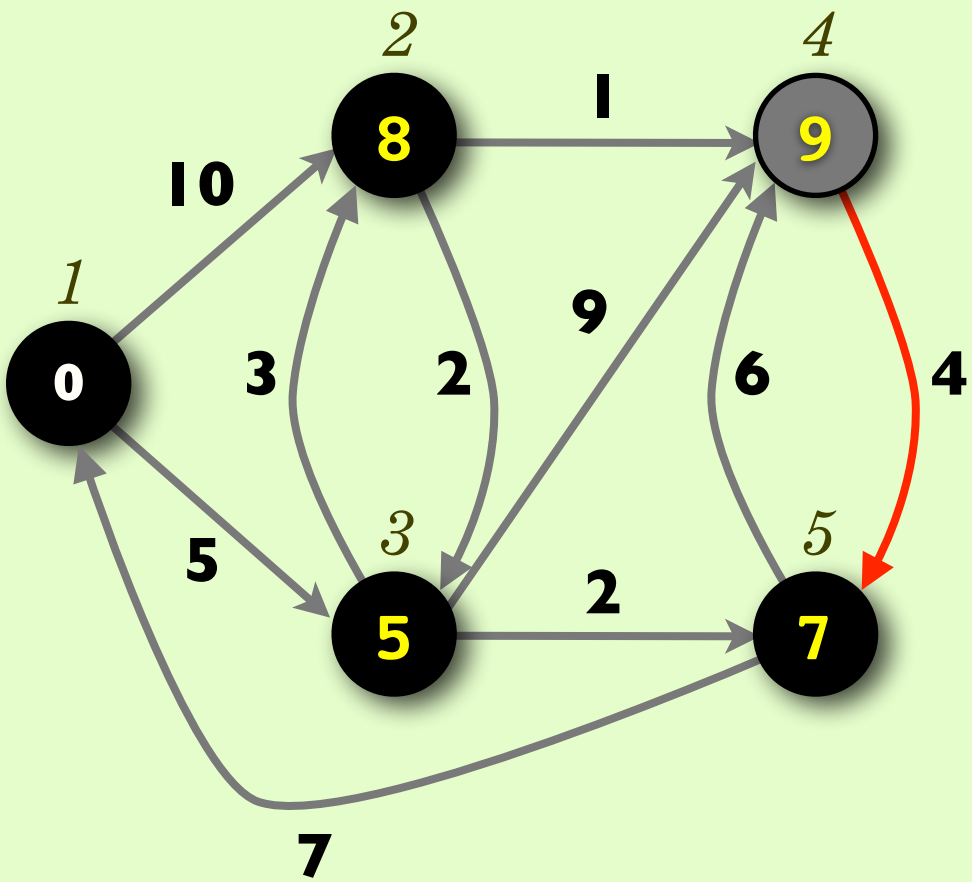


Queue

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Dijkstra: Beispiel

B *Single Source Best Path*: Algorithmus von Dijkstra



Queue

© 2006-2024 Jörg Striegnitz

Algorithmus von Dijkstra: Analyse

- **Man kann zeigen, dass der Dijkstra-Algorithmus immer die optimale Lösung liefert**

- **Der Aufwand des Algorithmus wird wesentlich durch die Verwaltung des Heap bestimmt**
  insbesondere dessen **Reorganisation**!

- **Die Schleife wird für jede Kante einmal durchlaufen:**
  $$O(|E|)$$

- **simple Queue: Array mit linearer Suche nach Minimum:**
  $$O(|V|^2 + |E|)$$

- **Queue mit Fibonacci-Heap** (hier nicht gezeigt)**:**
  $$O(|V| \cdot \mathrm{ld}(|V|) + |E|)$$

© 2006-2024 Jörg Striegnitz

Zusammenfassung: *Single-Source Best Path*

- **Topologisches Sortieren + Relax**
  - nur **azyklische Graphen**, negative Kantengewichte erlaubt
  - Komplexität: $\Theta(\ |V| + |E|\ )$

- **Algorithmus von Dijkstra**
  - **keine negativen Kantengewichte**, zyklische Graphen erlaubt
  - Komplexität stark Abhängig von Implementierung der Queue
    - $O(|V|^2 + |E|)$ - Queue als Array mit linearer Suche
    - $O(|E| \cdot \mathrm{ld}(|V|))$ - Heap
    - $O(|V| \cdot \mathrm{ld}(|V|) + |E|)$ - Fibonacci Heap

© 2006-2024 Jörg Striegnitz

Exkurs: Der A*-Algorithmus

… wenn man doch nur von **A** nach **B** will

© 2006-2022 Jörg Striegnitz

Motivation

- **Dijkstra: Optimale Pfade zu allen Knoten**

© 2006-2024 Jörg Striegnitz

Motivation

- **Dijkstra: Optimale Pfade zu allen Knoten**

- **Manchmal interessiert man sich nur für einen Zielknoten**
  - **„Problem“ mit Dijkstra**: es wird immer der aktuell kürzeste Pfad „weiterentwickelt“ - der kann sich aber durchaus vom Zielknoten **weg** bewegen
  - **Worst Case**: Zielknoten wird zuletzt aus Queue genommen
  - **Mögliche Verbesserung**: entwickle optimalen Pfad **in Richtung des Zielknotens**
    - Dazu: Führe Heuristik ein, die Algorithmus steuert

© 2006-2024 Jörg Striegnitz

A*-Algorithmus

- **Heuristik h schätzt Pfadkosten zum Zielknoten**
  - Darf tatsächliche Kosten nicht „überschätzen“!
  - Beispiele: Euklidischer Abstand / Manhattan-Distanz

© 2006-2024 Jörg Striegnitz

A*-Algorithmus

- **Heuristik h schätzt Pfadkosten zum Zielknoten**
  - Darf tatsächliche Kosten nicht „überschätzen“!
  - Beispiele: Euklidischer Abstand / Manhattan-Distanz
- **In Queue wird sortiert nach**

$$d(u) + h(u, target)$$

  - Also: Bisher bekannte Minimalkosten vom Startknoten zu u plus geschätzte Kosten von u zum Zielknoten $target$.

© 2006-2024 Jörg Striegnitz

A*-Algorithmus

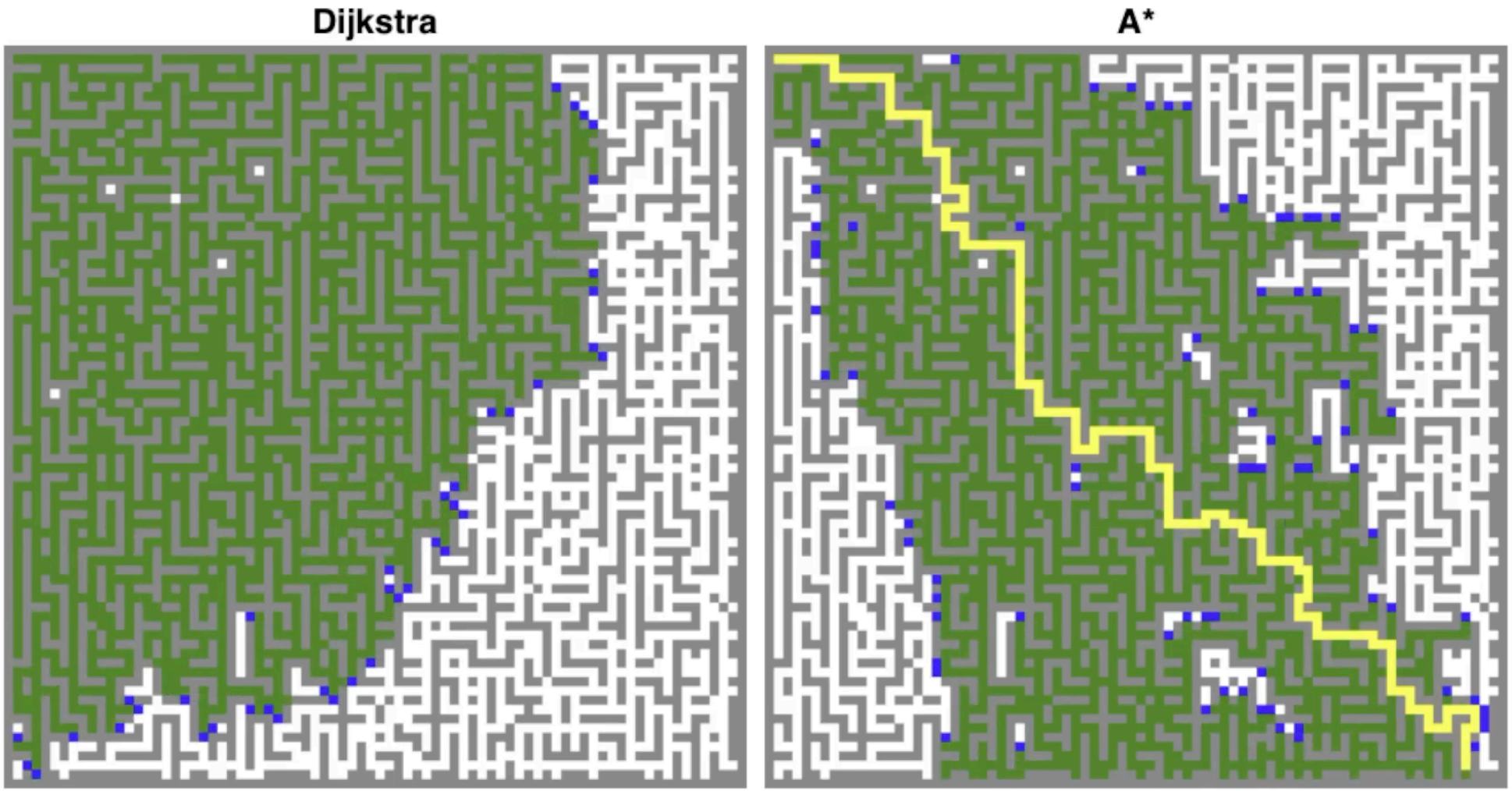
- **Heuristik h schätzt Pfadkosten zum Zielknoten**
  - Darf tatsächliche Kosten nicht „überschätzen“!
  - Beispiele: Euklidischer Abstand / Manhattan-Distanz
- **In Queue wird sortiert nach**

$$d(u) + h(u, target)$$

  - Also: Bisher bekannte Minimalkosten vom Startknoten zu u plus geschätzte Kosten von u zum Zielknoten $target$.
- **Anmerkung:** Dijkstra ist eine Spezialform von A* mit $h(u,target) = 0$

© 2006-2024 Jörg Striegnitz

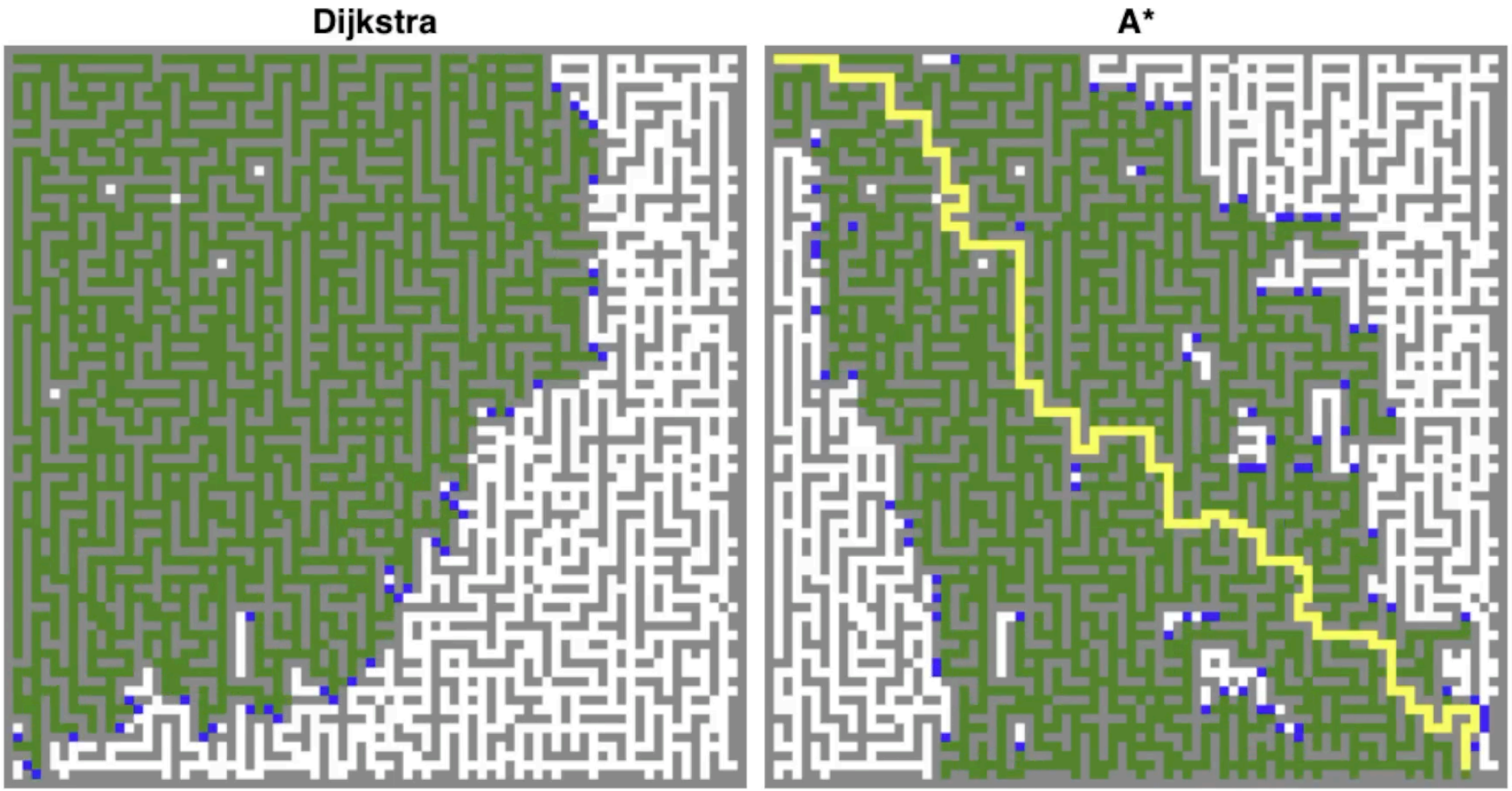
Dijkstra vs. A*



169 Schritte

© 2006-2024 Jörg Striegnitz

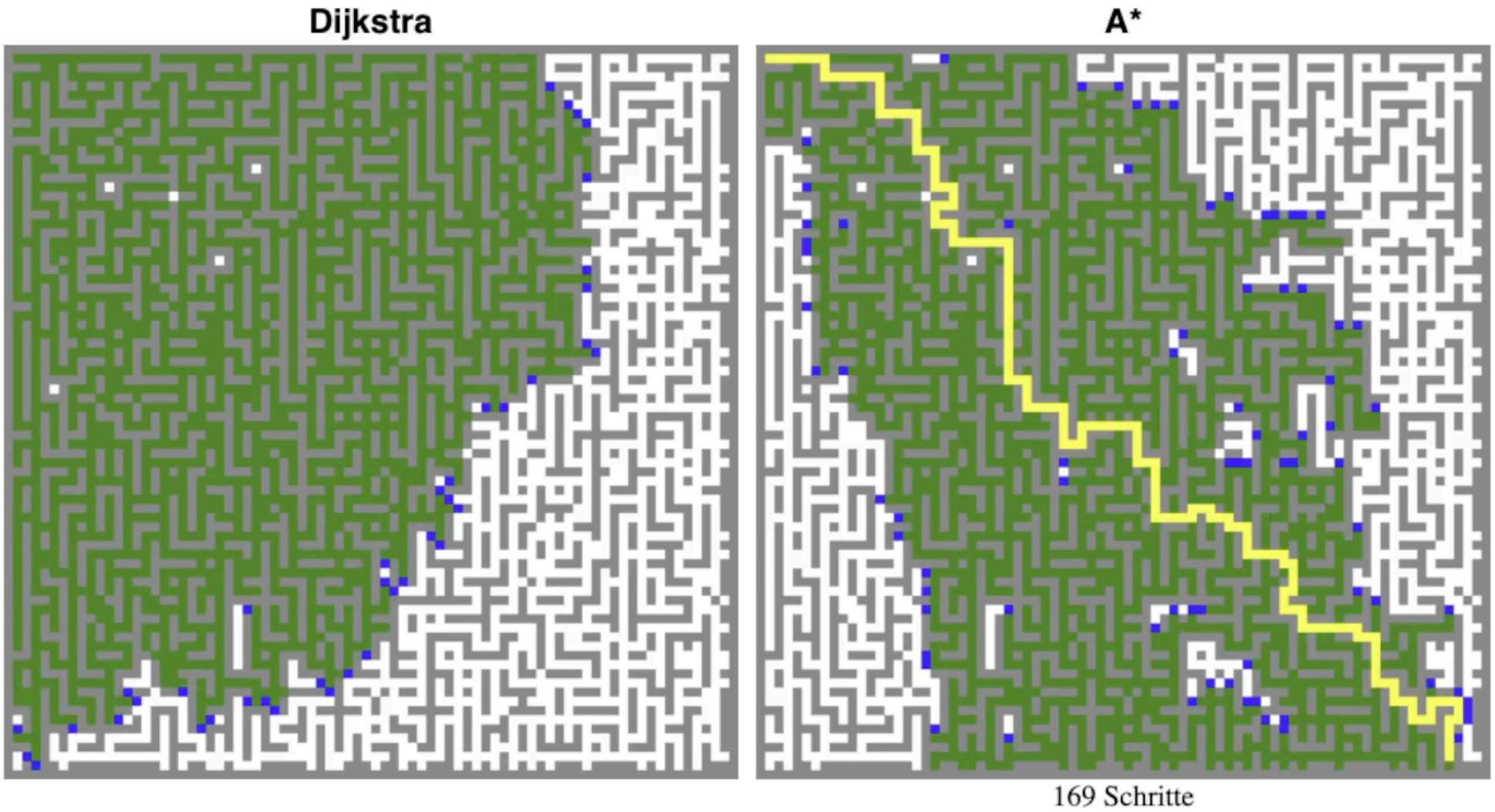
Dijkstra vs. A*



169 Schritte

© 2006-2024 Jörg Striegnitz

Dijkstra vs. A*



Beachte: Optimaler Pfad ist nicht eindeutig!

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Bellman und Ford

- **Anwendbar auf Graphen mit negativen Kantengewichten**
  - Negative Zyklen werden erkannt

- **Idee:**
  - Ignoriert man negative Zyklen: optimaler Pfad ist **einfacher** Pfad mit **höchstens** $|V|$-1 Kanten!
  - Daher: Relaxiere alle Kanten $|V|$-1 mal!
  - Erkenne negative Zyklen durch weitere „Test-Relaxation“

- **Geeignet für verteilte Systeme**
  - Siehe Vorlesung „Kommunikationssysteme"

Richard Bellman
★1920-†1984

Lester R. Ford jr.
★1927-†2017

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Bellman und Ford

- **Anwendbar auf Graphen mit negativen Kantengewichten**
  - Negative Zyklen werden erkannt

- **Idee:**
  - Ignoriert man negative Zyklen: optimaler Pfad ist **einfacher** Pfad mit **höchstens** $|V|$-1 Kanten!
  - Daher: Relaxiere alle Kanten $|V|$-1 mal!
  - Erkenne negative Zyklen durch weitere „Test-Relaxation“

- **Geeignet für verteilte Systeme**
  - Siehe Vorlesung „Kommunikationssysteme"

Richard Bellman
★1920-†1984

Lester R. Ford jr.
★1927-†2017

Bellman gilt als Begründer der dynamischen Programmierung

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Bellman und Ford - Pseudocode

Initialisiere $\boldsymbol{d[v]}$ *(* an der Stelle v_s Abstand 0; ansonsten Abstand ∞ (die größtmögliche Obergrenze) *)*

for k=1 to |V|-1. *(* k: Länge eines Pfades (# durchlaufene Kanten *)*

  for each edge (u, v) in G

    relax(u, v, weight)

(Test auf negativen Zyklus …. *)*

for each edge (u, v) in G

  If $\boldsymbol{d}[v] > \boldsymbol{d}[u]$ + weight(u, v) *(* bringt Zyklus Verbesserung? *)*

    return false

return true

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Bellman und Ford - Pseudocode

Initialisiere $d[v]$ *(* an der Stelle v_s Abstand 0; ansonsten Abstand ∞ (die größtmögliche Obergrenze) *)*

for k=1 to |V|-1. *(* k: Länge eines Pfades (# durchlaufene Kanten *)*
  for each edge (u, v) in G
    relax(u, v, weight)

Phase 1

(Test auf negativen Zyklus …. *)*

for each edge (u, v) in G
  If $d[v] > d[u]$ + weight(u, v) *(* bringt Zyklus Verbesserung? *)*
    return false

Phase 2

return true

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Bellman und Ford - Pseudocode

Initialisiere $d[v]$ *(* an der Stelle v_s Abstand 0; ansonsten Abstand ∞ (die größtmögliche Obergrenze) *)*

for k=1 to |V|-1. *(* k: Länge eines Pfades (# durchlaufene Kanten *)*
  for each edge (u, v) in G
    relax(u, v, weight)

Phase 1

(Test auf negativen Zyklus …. *)*

for each edge (u, v) in G
  If $d[v] > d[u]$ + weight(u, v) *(* bringt Zyklus Verbesserung? *)*
    return false

Phase 2

return true

in der k-ten Iteration wird d_k berechnet

$d_k[v]$: günstigste Kosten von v_s nach v mit **maximal** k Kanten

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Bellman und Ford - Pseudocode

Initialisiere $d[v]$ *(* an der Stelle v_s Abstand 0; ansonsten Abstand ∞ (die größtmögliche Obergrenze) *)*

for k=1 to |V|-1. *(* k: Länge eines Pfades (# durchlaufene Kanten *)*

  for each edge (u, v) in G

    relax(u, v, weight)

$\Theta(|V| \cdot |E|)$

Phase 1

(Test auf negativen Zyklus …. *)*

for each edge (u, v) in G

  If $d[v] > d[u]$ + weight(u, v) *(* bringt Zyklus Verbesserung? *)*

    return false

Phase 2

return true

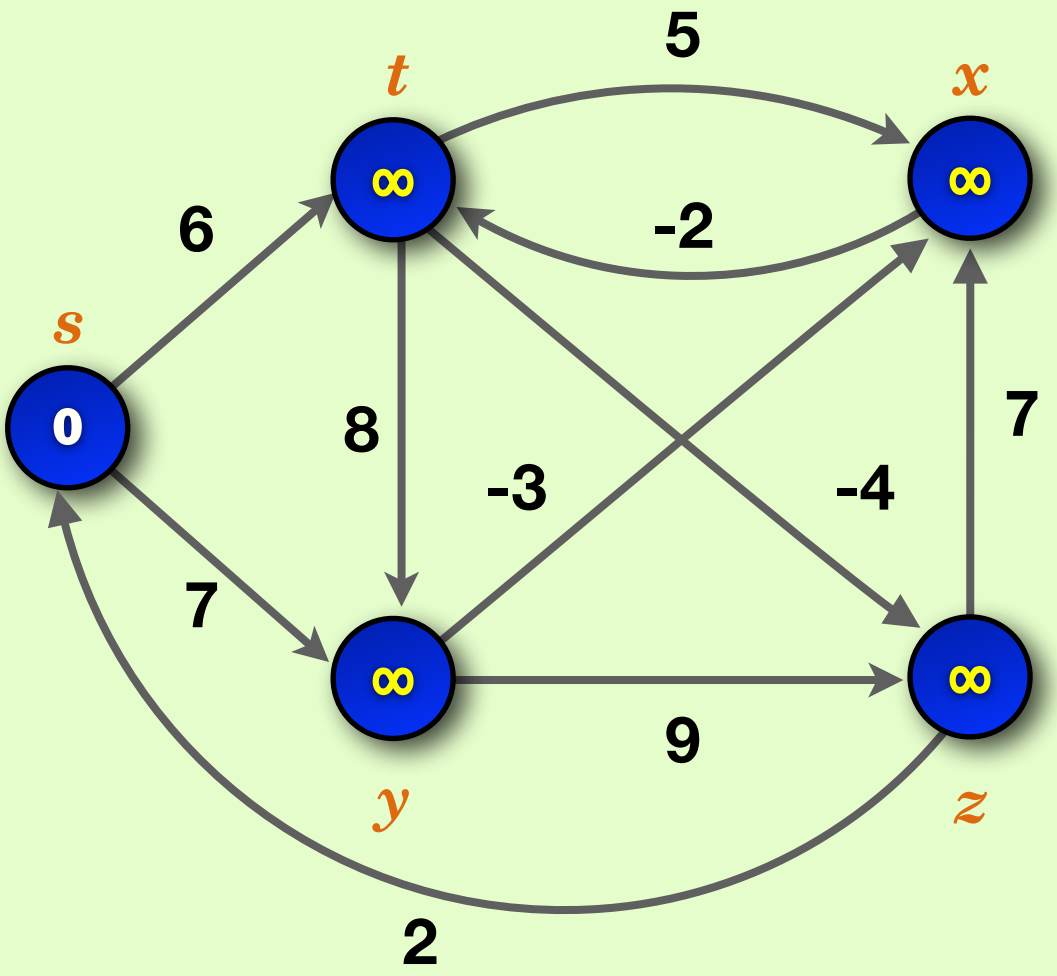
in der k-ten Iteration wird d_k berechnet

$d_k[v]$: günstigste Kosten von v_s nach v mit **maximal** k Kanten

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Bellman und Ford: Beispiel

B *Single Source Best Path*: Algorithmus von Bellman und Ford

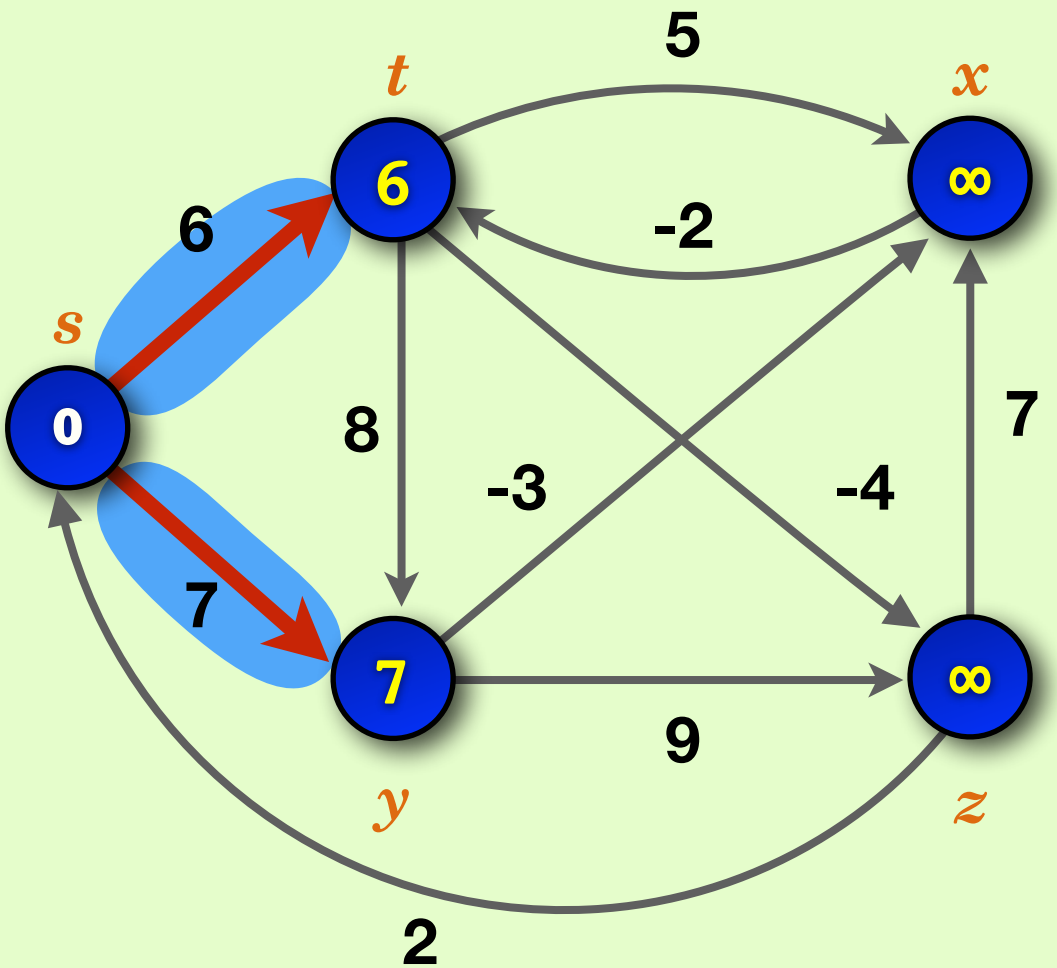


- Im Knoten v steht der Wert d[v]
- Die Kanten werden immer in derselben Reihenfolge relaxiert!
- Es werden Kanten **rot** hervorgehoben, die finale Änderung bewirken
- Blau hinterlegte Kanten: *predecessor subgraph*

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Bellman und Ford: Beispiel

B *Single Source Best Path*: Algorithmus von Bellman und Ford



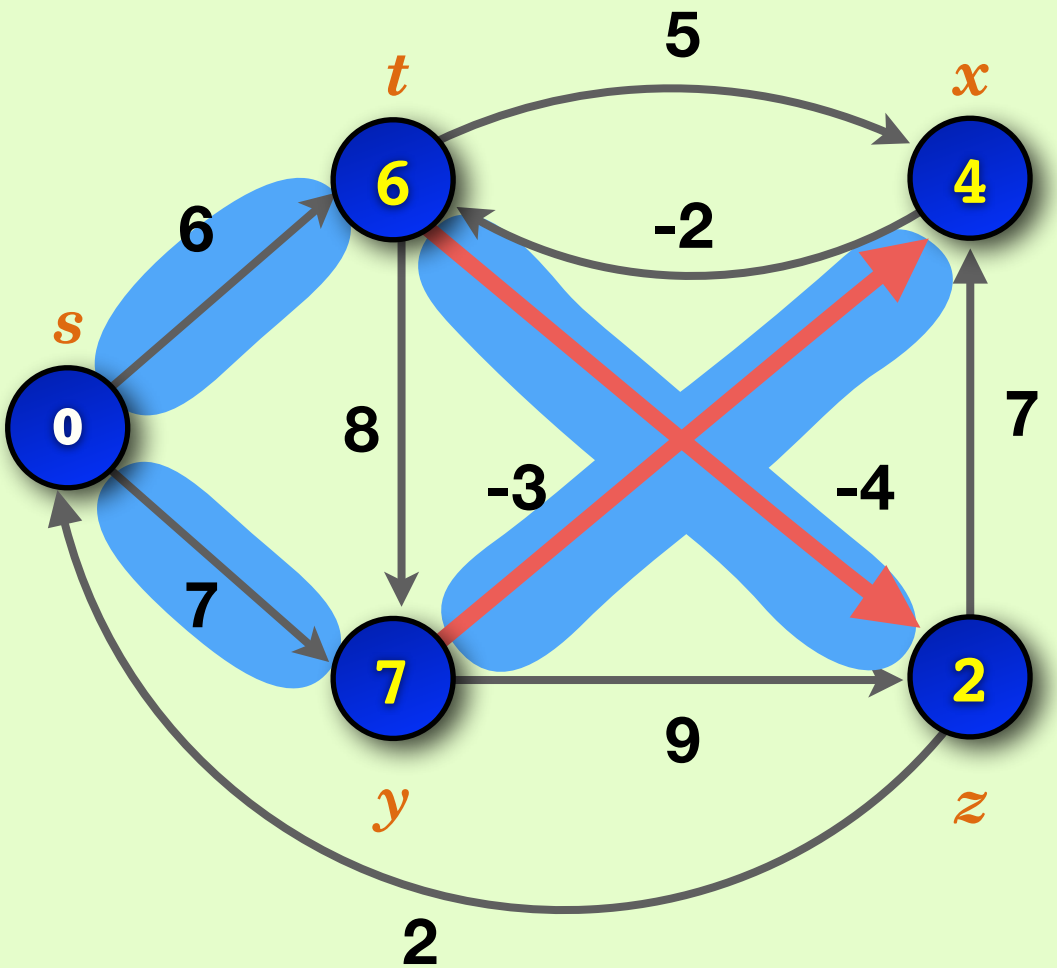
1. Iteration

- Im Knoten v steht der Wert d[v]
- Die Kanten werden immer in derselben Reihenfolge relaxiert!
- Es werden Kanten **rot** hervorgehoben, die finale Änderung bewirken
- Blau hinterlegte Kanten: *predecessor subgraph*

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Bellman und Ford: Beispiel

B *Single Source Best Path*: Algorithmus von Bellman und Ford



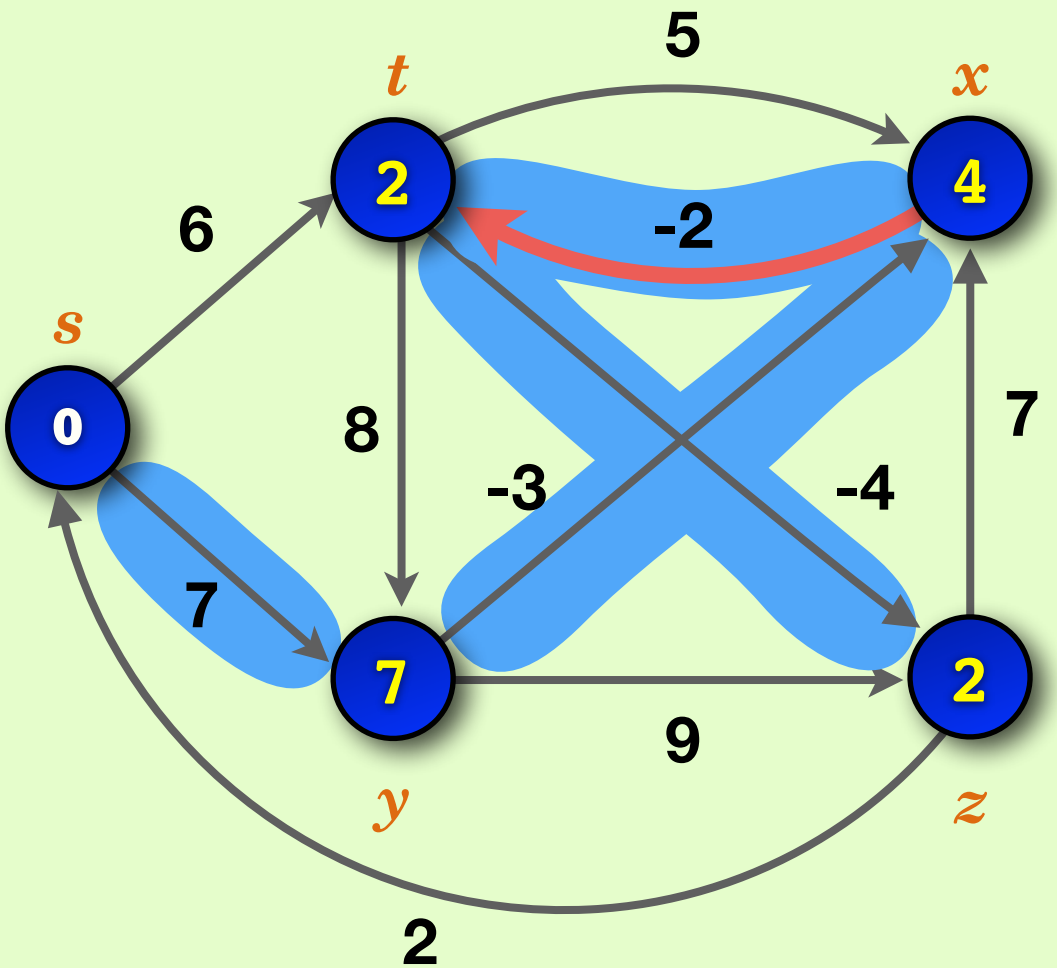
- Im Knoten v steht der Wert d[v]
- Die Kanten werden immer in derselben Reihenfolge relaxiert!
- Es werden Kanten **rot** hervorgehoben, die finale Änderung bewirken
- Blau hinterlegte Kanten: *predecessor subgraph*

2. Iteration

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Bellman und Ford: Beispiel

B *Single Source Best Path*: Algorithmus von Bellman und Ford



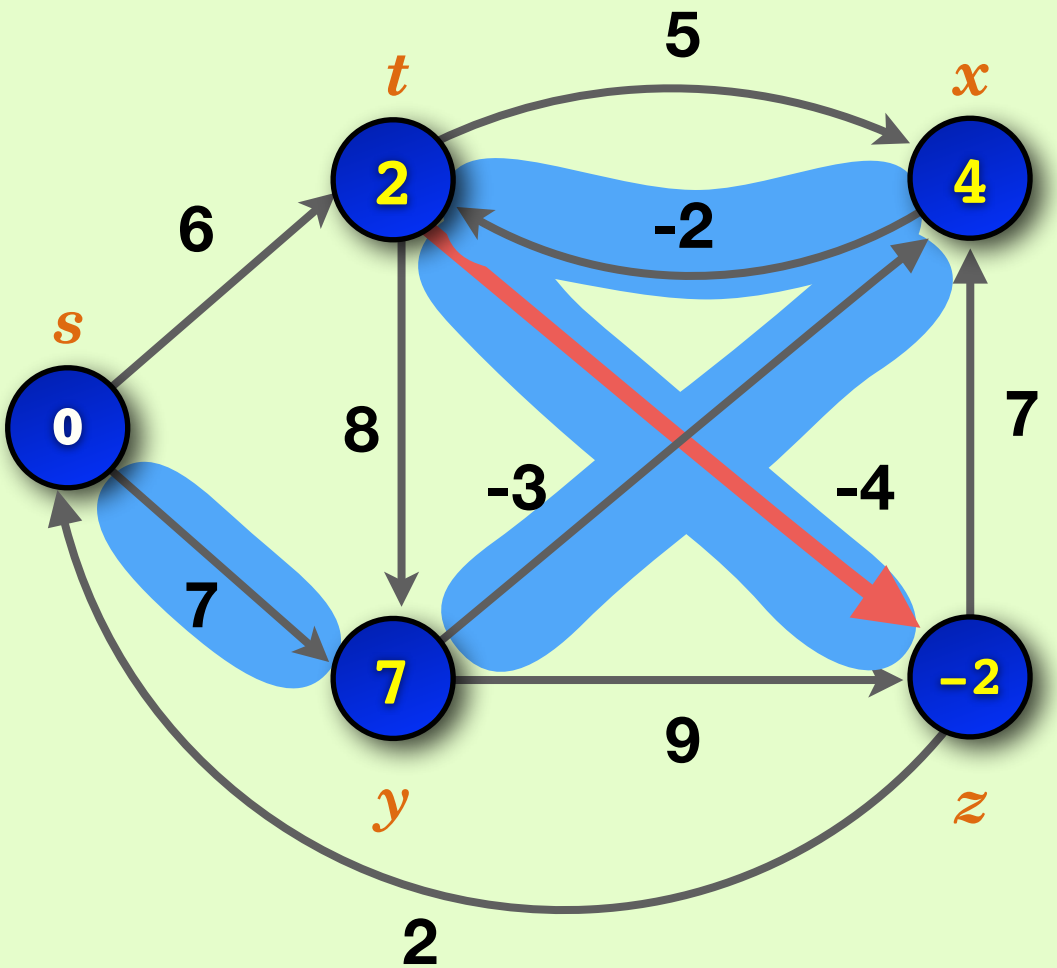
- Im Knoten v steht der Wert d[v]
- Die Kanten werden immer in derselben Reihenfolge relaxiert!
- Es werden Kanten **rot** hervorgehoben, die finale Änderung bewirken
- Blau hinterlegte Kanten: *predecessor subgraph*

3. Iteration

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Bellman und Ford: Beispiel

B *Single Source Best Path*: Algorithmus von Bellman und Ford



- Im Knoten v steht der Wert d[v]
- Die Kanten werden immer in derselben Reihenfolge relaxiert!
- Es werden Kanten **rot** hervorgehoben, die finale Änderung bewirken
- Blau hinterlegte Kanten: *predecessor subgraph*

4. Iteration

© 2006-2024 Jörg Striegnitz

Ein anderer Blickwinkel ...

- **Bellman-Ford entspricht topologischem Sortieren + relax auf einer wie folgt transformierten Variante von G**
  Graph Duplication

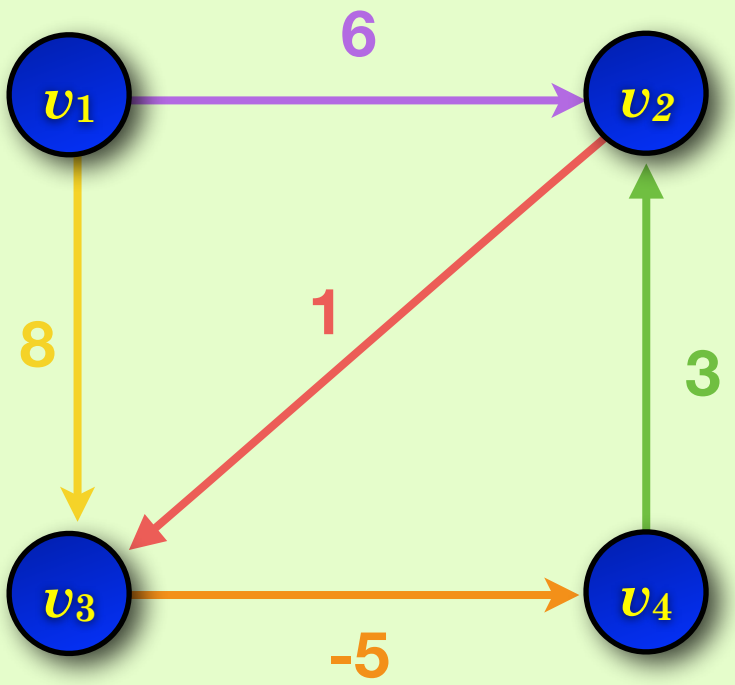
  Sei $G = (V, E)$ ein gerichteter Graph mit $V = \{v_1, \cdots, v_n\}$, der eventuell Zyklen enthält. Erhalte azyklischen gerichteten Graphen $G' = (V', E')$ wie folgt:

  - $V' = \{v_{1,0}, v_{1,1}, \ldots, v_{1,n}, \cdots, v_{2,0}, \cdots, v_{2,n}, \ldots, v_{n,n}\}$
    Alle Knoten n-mal duplizieren. Knoten $v_{i,k}$ ist der zu v_i korrespondierende Knoten auf Ebene k. Keine Kanten in G' innerhalb einer Ebene!
  - $\forall i, k \in \{0, \ldots, n-1\} : (v_{i,k}, v_{i,k+1}) \in E'$
    Korrespondierende Knoten in aufeinanderfolgenden Ebenen verbinden.
  - $(v_i, v_j) \in E \Rightarrow \forall k \in \{0, \ldots, n-1\} : (v_{i,k}, v_{j,k+1}) \in E$
    Kanten aus G verbinden Ebenen in G'.

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Bellman und Ford: Beispiel

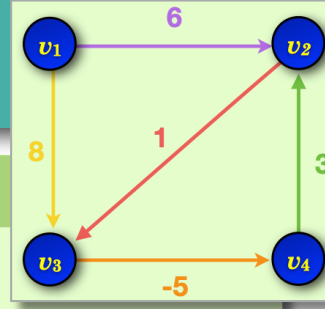
B Algorithmus von Bellman und Ford mit neg. Zyklus



- Startknoten sei v_1
- Zuerst wird Transformation durchgeführt

© 2006-2024 Jörg Striegnitz

Bellman und Ford: Beispiel-Transformation 1



B Algorithmus von Bellman und Ford mit neg. Zyklus

$v_{1,0}$

$v_{2,0}$

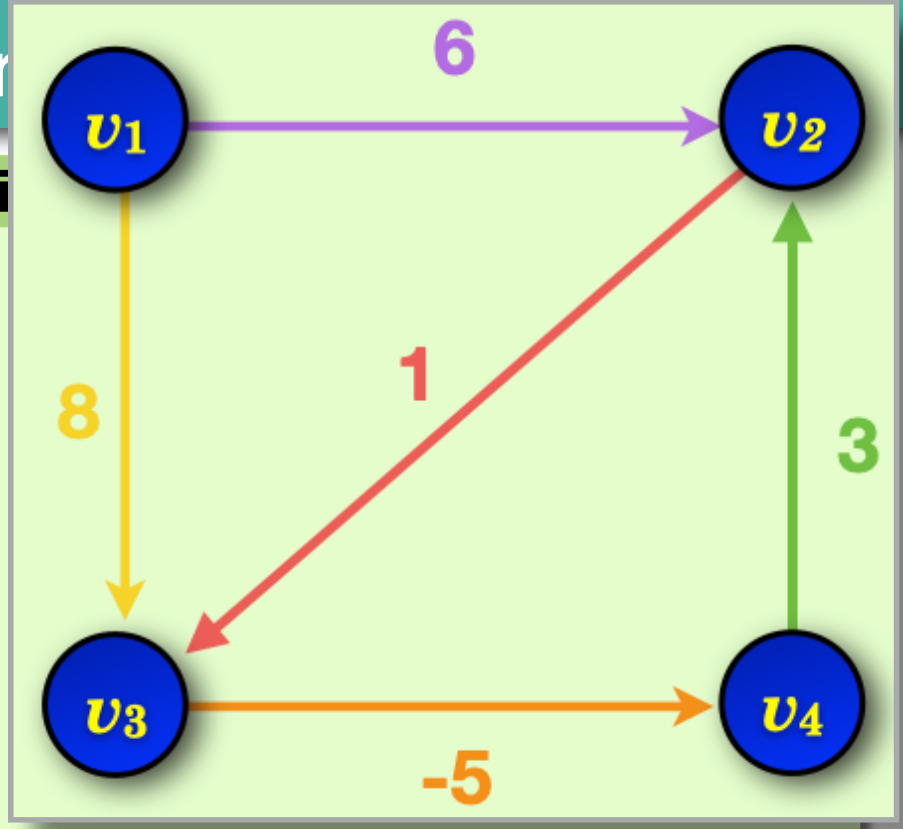
$v_{3,0}$

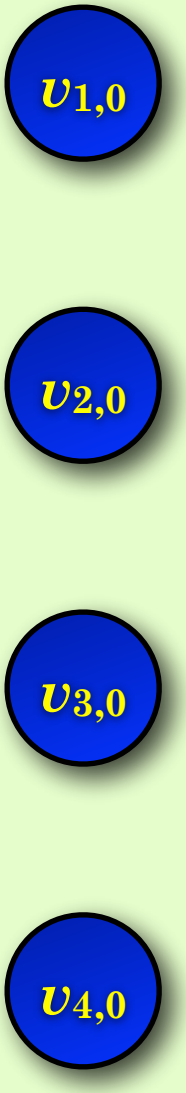
$v_{4,0}$

© 2006-2024 Jörg Striegnitz

Bellman und Ford: Beispiel-Tran

B Algorithmus von Bellman und Ford mi

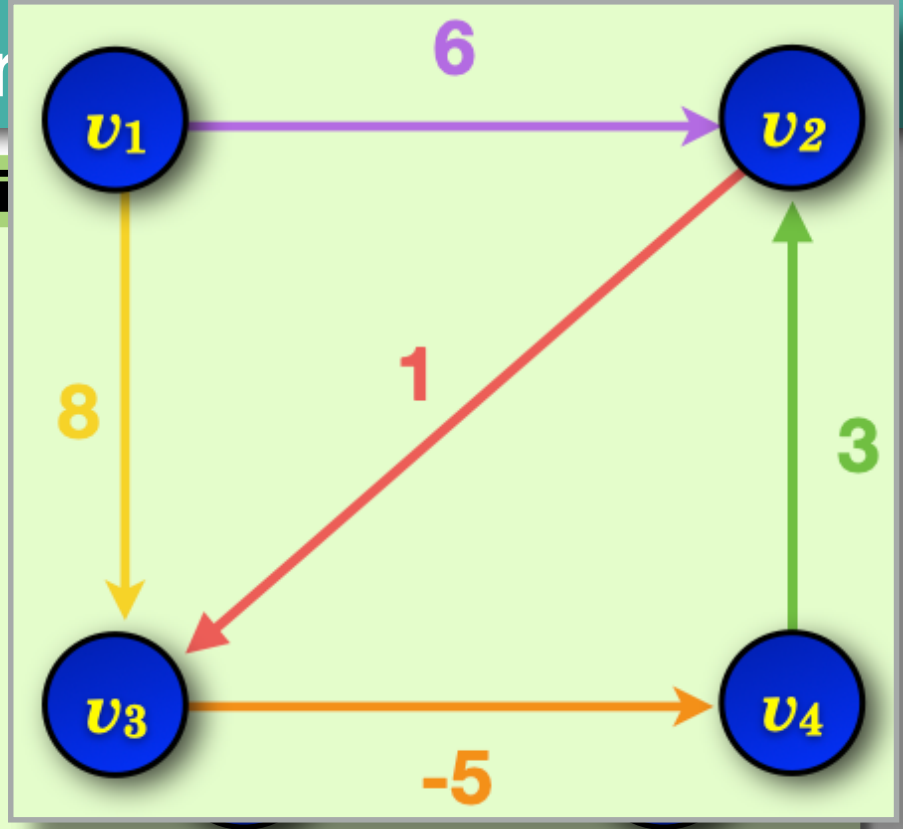


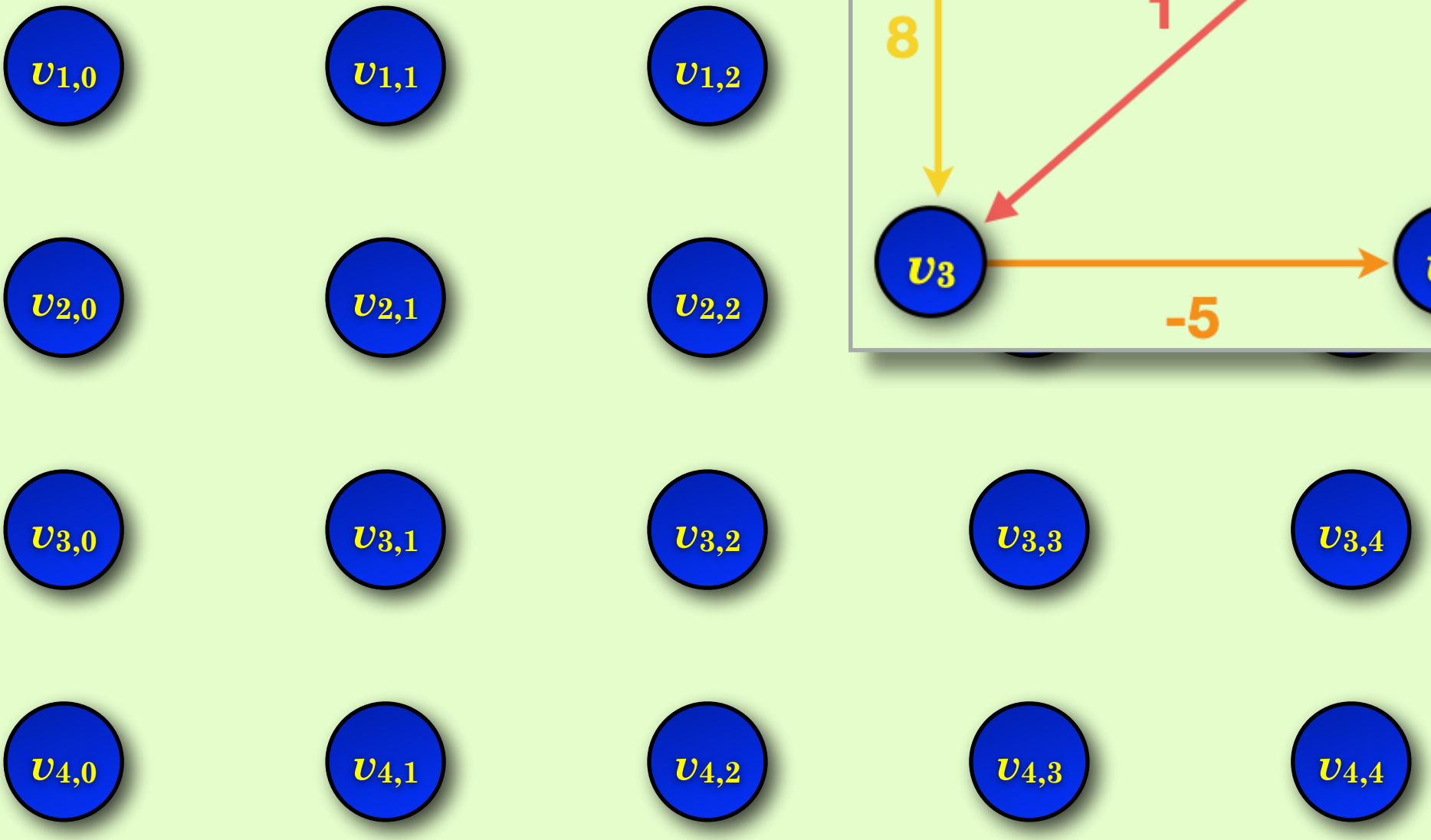


© 2006-2024 Jörg Striegnitz

Bellman und Ford: Beispiel-Tran

B Algorithmus von Bellman und Ford mi

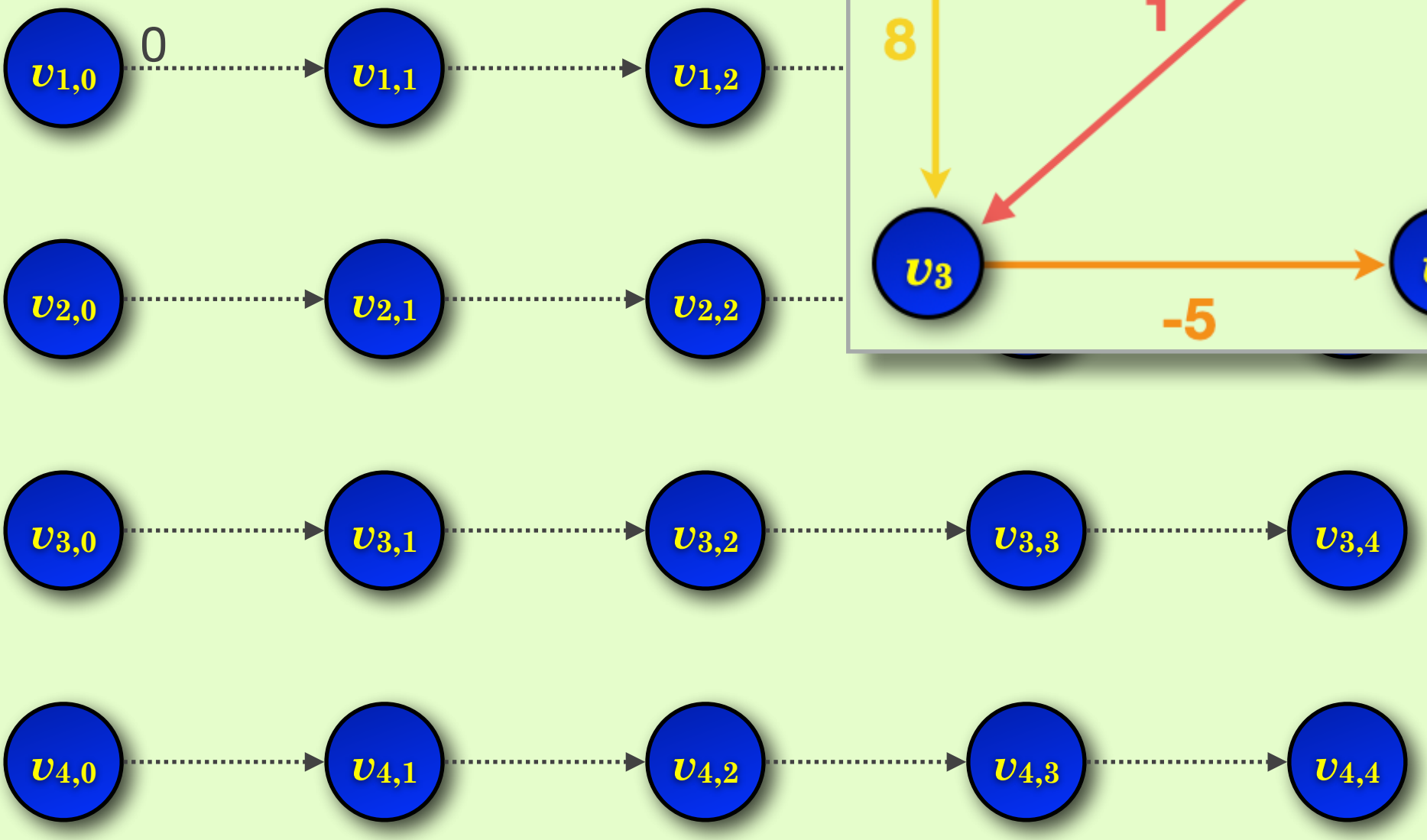


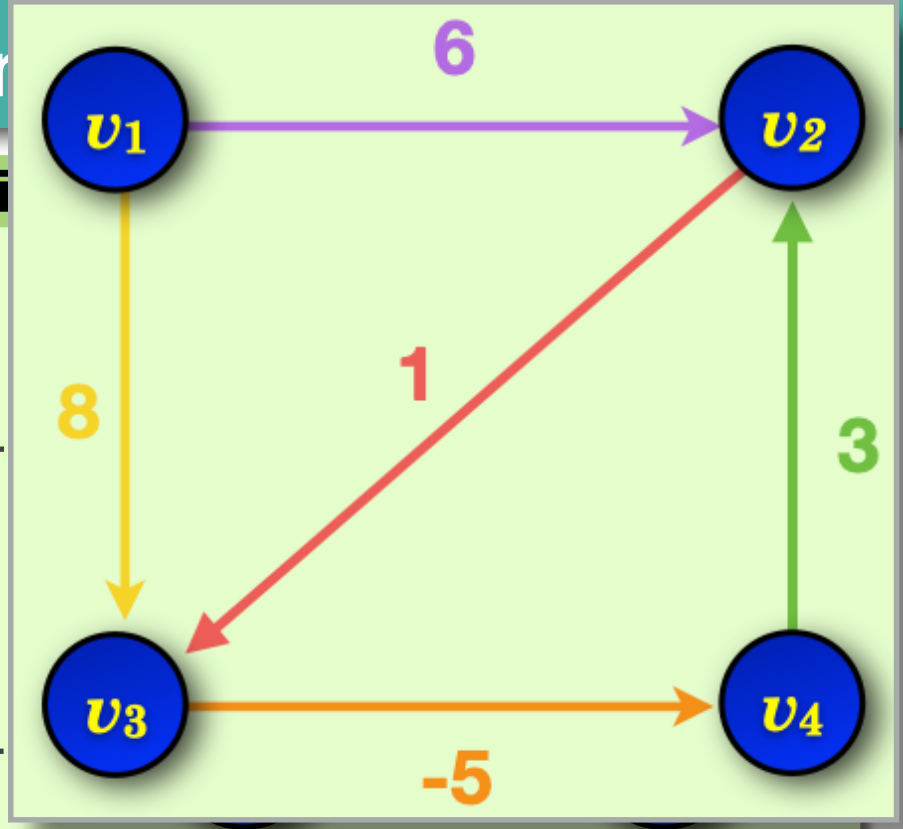


© 2006-2024 Jörg Striegnitz

Bellman und Ford: Beispiel-Tran

B Algorithmus von Bellman und Ford mi

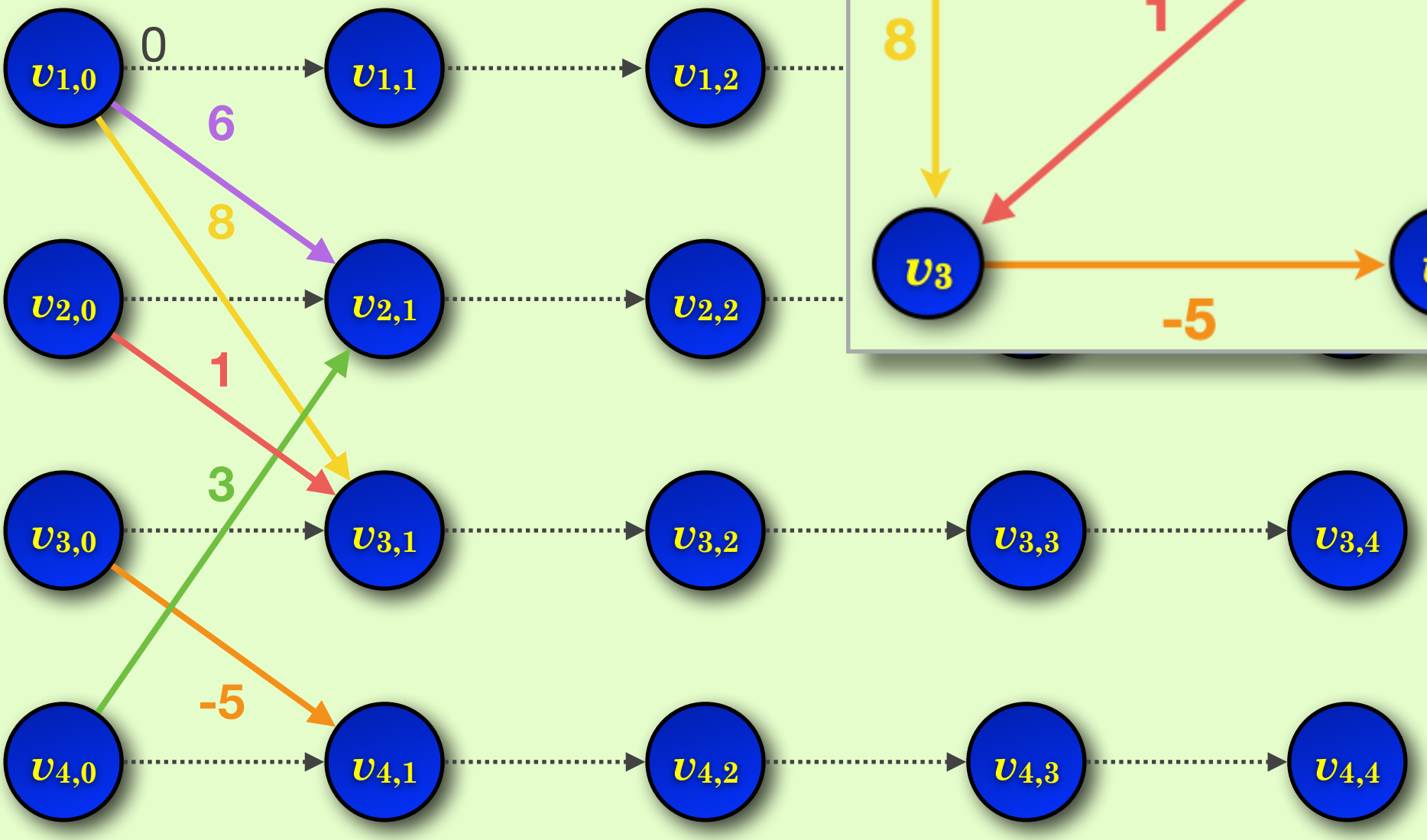


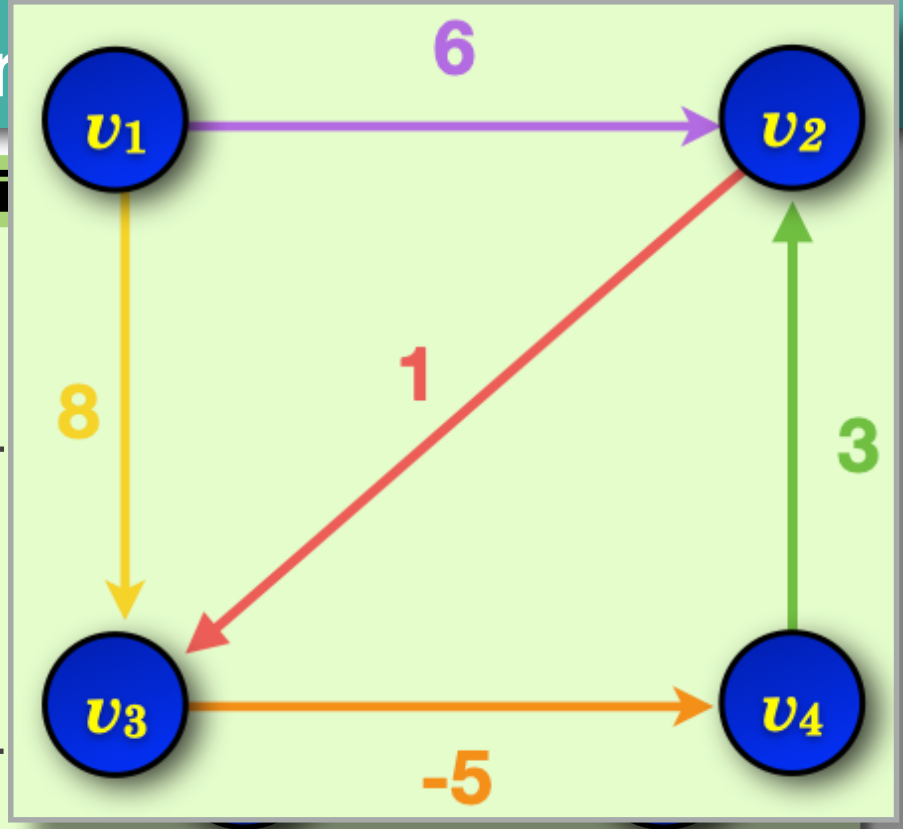


© 2006-2024 Jörg Striegnitz

Bellman und Ford: Beispiel-Tran

B Algorithmus von Bellman und Ford mi

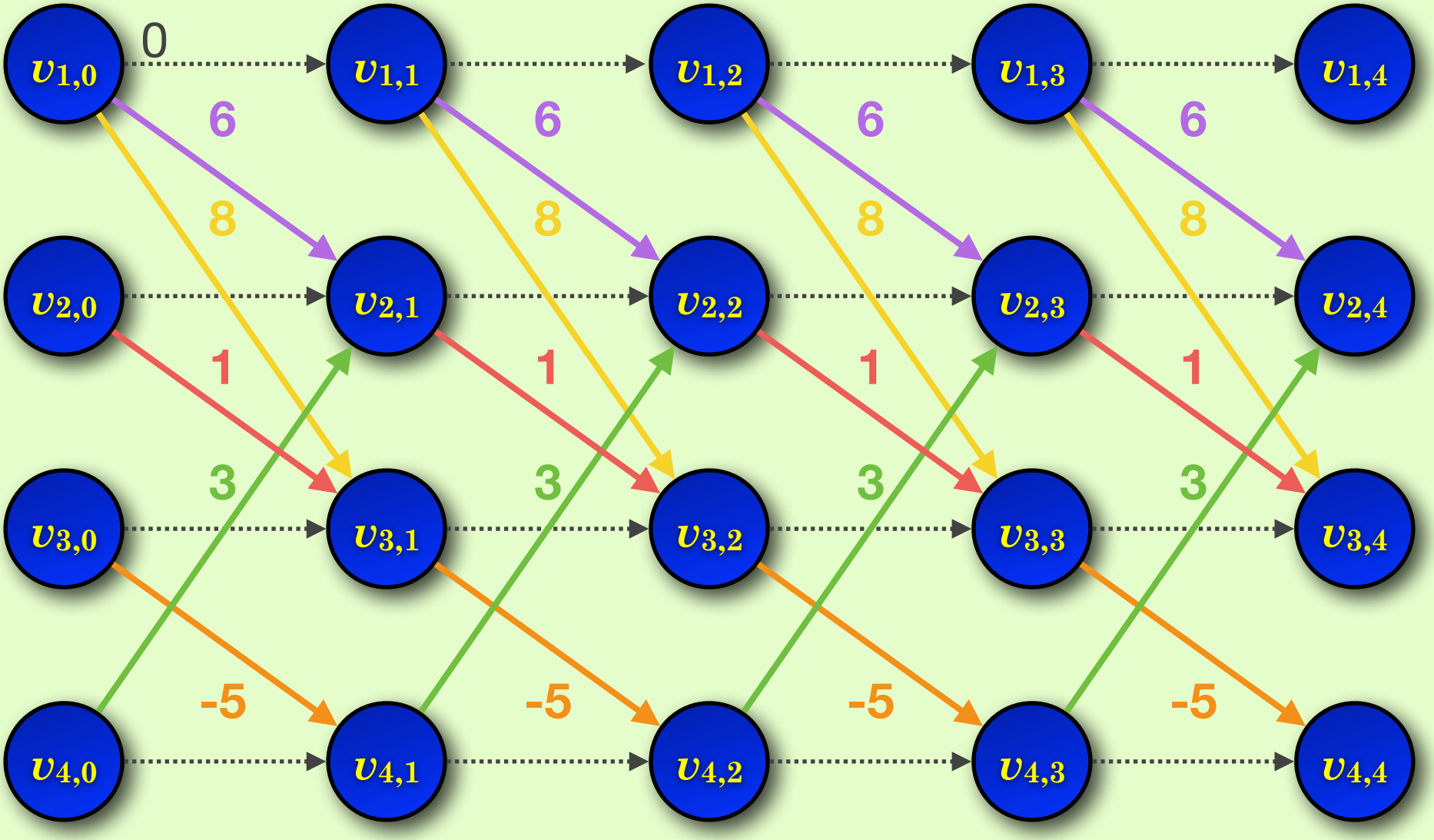




© 2006-2024 Jörg Striegnitz

Bellman und Ford: Beispiel-Transformation 1

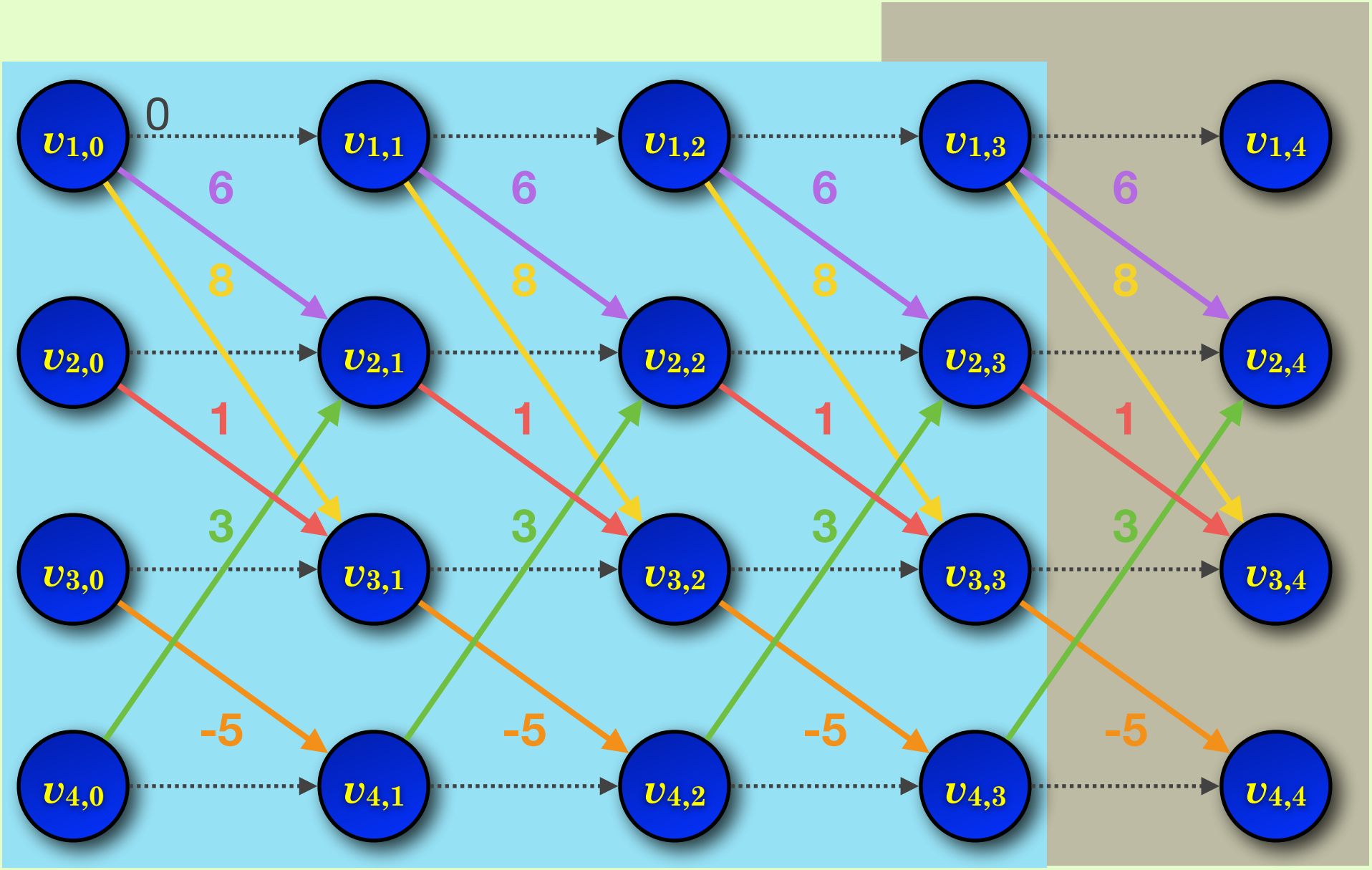
B Algorithmus von Bellman und Ford mit neg. Zyklus



© 2006-2024 Jörg Striegnitz

Bellman und Ford: Beispiel-Transformation 1

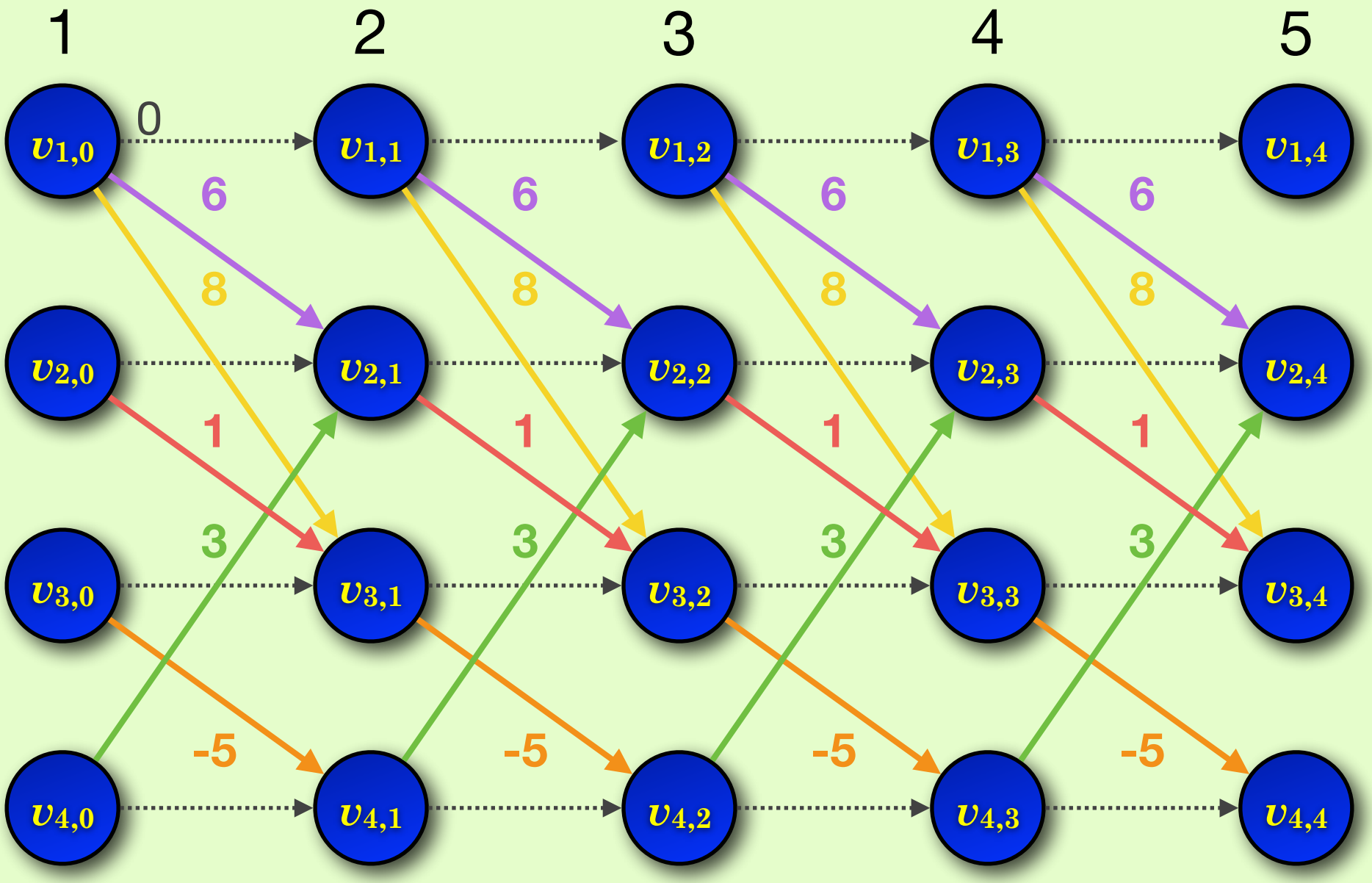
B Algorithmus von Bellman und Ford mit neg. Zyklus



© 2006-2024 Jörg Striegnitz

Bellman und Ford: Beispiel-Transformation 1

B Algorithmus von Bellman und Ford mit neg. Zyklus



© 2006-2024 Jörg Striegnitz

Bellman und Ford: Beispiel-Transformation 1

B Algorithmus von Bellman und Ford mit neg. Zyklus

Im entstandenen **DAG** (***D**irected **A**cyclic **G**raph*)
ist SSBP mit
topologischem Sortieren + Relax
in $O(|V| \cdot |E|)$
lösbar!

Aber: Was ist mit negativem Zyklus?

© 2006-2024 Jörg Striegnitz

Die Funktionen δ und δ_k - I

D **Minimales Pfadgewicht: δ und δ_k**

Sei $G = (V, E)$ ein gewichteter Graph mit Gewichtsfunktion weight : $E \to \mathbb{R}$. Dann verwenden wir für zwei beliebige Knoten s,t aus V folgende Bezeichnungen für Pfadkosten von s nach t:

- $\delta_k(s,t)$: Kosten eines günstigsten Pfades, der **höchstens** k **Kanten** durchläuft
- $\delta(s,t)$: Kosten eines günstigste Pfades, der **beliebig viele Kanten** durchläuft.

Existiert keine Verbindung von s nach t in G, so ist $\delta(s,t) = \infty$ bzw. $\forall k \in \mathbb{N}_0 : \delta_k(s,t) = \infty$

© 2006-2024 Jörg Striegnitz

Die Funktionen δ und δ_k - I

D **Minimales Pfadgewicht: δ und δ_k**

Sei $G = (V, E)$ ein gewichteter Graph mit Gewichtsfunktion weight : $E \to \mathbb{R}$. Dann verwenden wir für zwei beliebige Knoten s,t aus V folgende Bezeichnungen für Pfadkosten von s nach t:

- $\delta_k(s,t)$: Kosten eines günstigsten Pfades, der **höchstens** k **Kanten** durchläuft
- $\delta(s,t)$: Kosten eines günstigste Pfades, der **beliebig viele Kanten** durchläuft.

Existiert keine Verbindung von s nach t in G, so ist $\delta(s,t) = \infty$ bzw. $\forall k \in \mathbb{N}_0 : \delta_k(s,t) = \infty$

- Beobachtung: **ohne** negative Zyklen gilt für alle Knotenpaare s, t:

$$\delta_{|V|-1}(s,t) \quad = \quad \delta(s,t)$$

optimale Pfade sind dann immer **einfache Pfade**

© 2006-2024 Jörg Striegnitz

L 7.1 Unterpfade von kürzesten Pfaden sind kürzeste Pfade

Sei $G = (V, E)$ ein gewichteter Graph mit Gewichtsfunktion weight : $E \to \mathbb{R}$ und ferner $p = \langle v_1, v_2, \ldots, v_k \rangle$ ein kürzester Pfad von v_1 nach v_k. Für beliebige i und j mit $1 \leq i \leq j \leq k$, sei $p_{i,j} = \langle v_i, \ldots v_j \rangle$ ein **Unterpfad** von p. Es gilt: $p_{i,j}$ ist ein kürzester Pfad von v_i nach v_j.

© 2006-2024 Jörg Striegnitz

Optimale Pfade und Unterpfade

L 7.1 Unterpfade von kürzesten Pfaden sind kürzeste Pfade

Sei $G = (V, E)$ ein gewichteter Graph mit Gewichtsfunktion weight : $E \to \mathbb{R}$ und ferner $p = \langle v_1, v_2, \ldots, v_k \rangle$ ein kürzester Pfad von v_1 nach v_k. Für beliebige i und j mit $1 \leq i \leq j \leq k$, sei $p_{i,j} = \langle v_i, \ldots v_j \rangle$ ein **Unterpfad** von p. Es gilt: $p_{i,j}$ ist ein kürzester Pfad von v_i nach v_j.

Beachte: negative Kantengewichte sind ***nicht*** ausgeschlossen

© 2006-2024 Jörg Striegnitz

L 7.1 Unterpfade von kürzesten Pfaden sind kürzeste Pfade

Sei $G = (V, E)$ ein gewichteter Graph mit Gewichtsfunktion weight : $E \to \mathbb{R}$ und ferner $p = \langle v_1, v_2, \ldots, v_k \rangle$ ein kürzester Pfad von v_1 nach v_k. Für beliebige i und j mit $1 \leq i \leq j \leq k$, sei $p_{i,j} = \langle v_i, \ldots v_j \rangle$ ein **Unterpfad** von p. Es gilt: $p_{i,j}$ ist ein kürzester Pfad von v_i nach v_j.

Beachte: negative Kantengewichte sind ***nicht*** ausgeschlossen

Beweis:

Wenn wir den Pfad p in

$$v_1 \overset{p_{1,i}}{\rightsquigarrow} v_i \overset{p_{i,j}}{\rightsquigarrow} v_j \overset{p_{j,k}}{\rightsquigarrow} v_k$$

zerlegen, dann gilt: $\text{weight}(p) = \text{weight}(p_{1,i}) + \text{weight}(p_{i,j}) + \text{weight}(p_{j,k})$.

Angenommen es existiert ein Pfad $p'_{i,j}$ von v_i nach v_j mit $\text{weight}(p'_{i,j}) < \text{weight}(i, j)$. Dann ist

$$v_1 \overset{p_{1,i}}{\rightsquigarrow} v_i \overset{p'_{i,j}}{\rightsquigarrow} v_j \overset{p_{j,k}}{\rightsquigarrow} v_k$$

ein Pfad von v_1 nach v_k, dessen Gewicht $\text{weight}(p_{1,i}) + \text{weight}(p'_{i,j}) + \text{weight}(p_{j,k})$ kleiner ist als $\text{weight}(p)$, was der Annahme widerspricht, das p der kürzeste Pfad von v_i nach v_k war.

© 2006-2024 Jörg Striegnitz

δ erfüllt eine Dreiecksungleichung

L 7.2 Dreiecksungleichung für δ

Seien $G = (V, E)$ ein gerichteter Graph mit Gewichtsfunktion weight : $E \to \mathbb{R}$ und s ein beliebiger Startknoten aus V. Dann gilt

$$\forall (u, v) \in E \times E \ : \ \delta(s, v) \leq \delta(s, u) + \text{weight}(u, v)$$

© 2006-2024 Jörg Striegnitz

δ erfüllt eine Dreiecksungleichung

L 7.2 Dreiecksungleichung für δ

Seien $G = (V, E)$ ein gerichteter Graph mit Gewichtsfunktion weight : $E \to \mathbb{R}$ und s ein beliebiger Startknoten aus V. Dann gilt

$$\forall (u, v) \in E \times E \ : \ \delta(s, v) \leq \delta(s, u) + \text{weight}(u, v)$$

Beweis:

Es gelte $\forall c \in \mathbb{R} : \infty + c = \infty$ bzw. $-\infty + c = -\infty$.

Wir müssen unterscheiden, ob ein kürzester einfachen Pfad p von s nach v einfach ist, zyklisch ist, oder es gar keinen gibt:

1. p ist einfach: da p ein kürzester Pfad ist, folgt das Gewünschte sofort.
2. p ist zyklisch: dann ist $\delta(s, v) = -\infty$ und die Aussage somit korrekt.
3. p existiert nicht: dann ist $\delta(s, v) = \infty$ und es kann auch keinen Pfad von s zu u geben (sonst wäre v durch (u, v) erreichbar) - somit ist $\delta(s, u) = \infty$.

© 2006-2024 Jörg Striegnitz

Die Funktionen δ und δ_k - II

D Minimales Pfadgewicht: δ und δ_k

Sei $G = (V, E)$ ein gewichteter Graph mit Gewichtsfunktion weight : $E \to \mathbb{R}$. Dann verwenden wir für zwei beliebige Knoten s,t aus V folgende Bezeichnungen für Pfadkosten von s nach t:

- $\delta_k(s,t)$: Kosten eines günstigsten Pfades, der **höchstens** k **Kanten** durchläuft
- $\delta(s,t)$: Kosten eines günstigste Pfades, der **beliebig viele Kanten** durchläuft.

Existiert keine Verbindung von s nach t in G, so ist $\delta(s,t) = \infty$ bzw. $\forall k \in \mathbb{N}_0 : \delta_k(s,t) = \infty$

- Angenommen

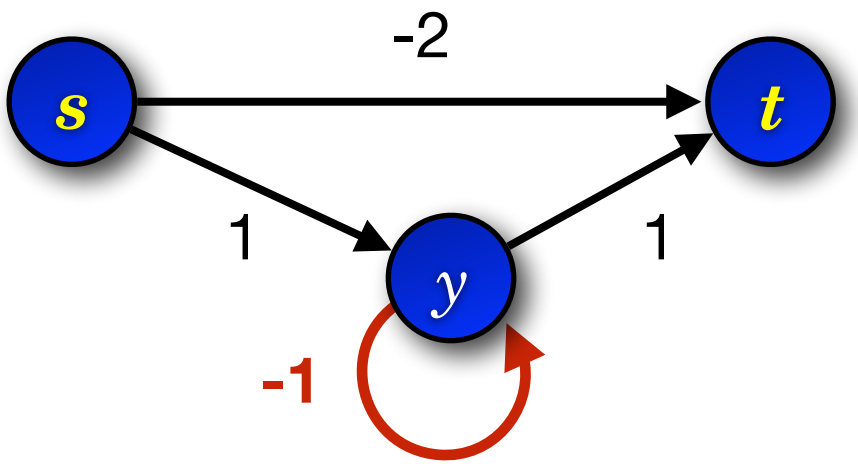
$$\delta_{|V|}(s,t) \quad < \quad \delta_{|V|-1}(s,t)$$

  - dann gibt es einen Pfad von s nach t **mit Zyklus**, der günstiger ist und es folgt $\delta(s,t) = -\infty$
  - t heißt dann **Zeuge** für einen negativen Zyklus

© 2006-2024 Jörg Striegnitz

Die Funktionen δ und δ_k - III

- **Beachte:** gilt $\delta(s,t) = -\infty$ muss t nicht zwingend ein Zeuge für einen negativen Zyklus sein!

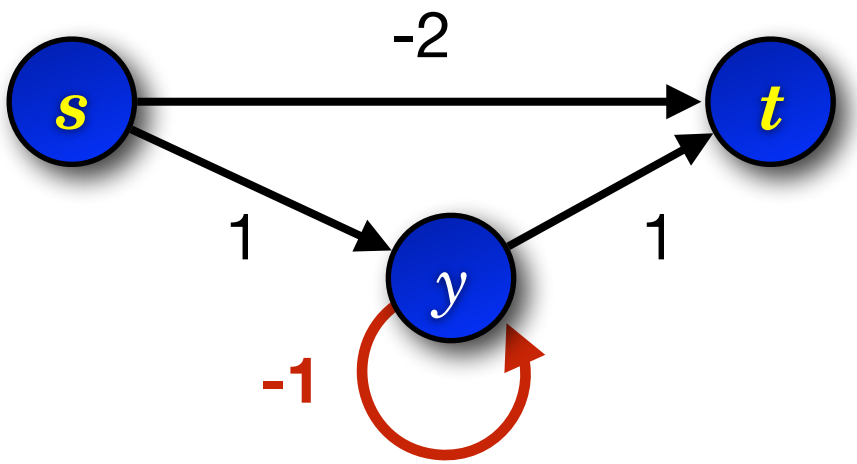


v	$\delta_0(s,\underline{v})$	$\delta_1(s,\underline{v})$	$\delta_2(s,\underline{v})$	$\delta_3(s,\underline{v})$	$\delta(s,\underline{v})$
s	0	0	0	0	0
y	∞	1	0	-1	$-\infty$
t	∞	-2	-2	-2	$-\infty$

© 2006-2024 Jörg Striegnitz

Die Funktionen δ und δ_k - III

- **Beachte:** gilt $\delta(s,t) = -\infty$ muss t nicht zwingend ein Zeuge für einen negativen Zyklus sein!



v	$\delta_0(s,\underline{v})$	$\delta_1(s,\underline{v})$	$\delta_2(s,\underline{v})$	$\delta_3(s,\underline{v})$	$\delta(s,\underline{v})$
s	0	0	0	0	0
y	∞	1	0	-1	-∞
t	∞	-2	-2	-2	-∞

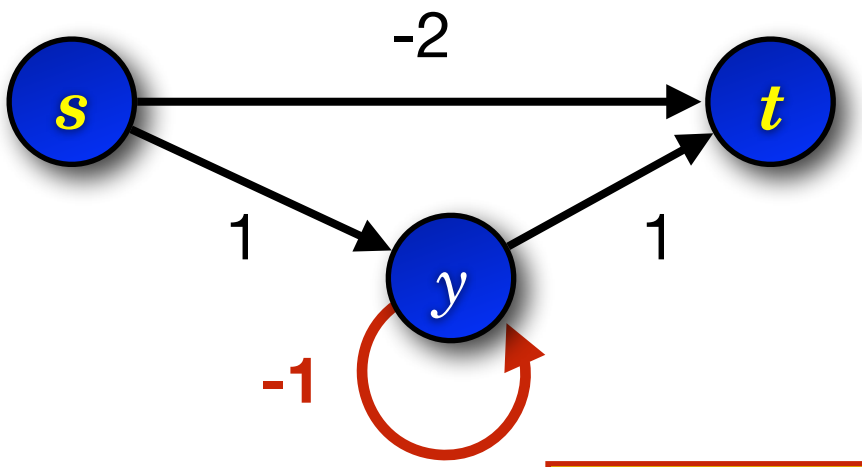
Offensichtlich ist $\langle s, y, t\rangle$ ein gülitger Pfad und wegen weight$(y, y) < 0$ folgt sofort $\delta(s, t) = -\infty$. t st aber **kein Zeuge** für einen negativen Zyklus, denn

$$\delta_{|V|-1} = \delta_2(s,t) = \delta_3(s,t) = \delta_{|V|}(s,t) = -2$$

© 2006-2024 Jörg Striegnitz

Die Funktionen δ und δ_k - III

- **Beachte:** gilt $\delta(s,t) = -\infty$ muss t nicht zwingend ein Zeuge für einen negativen Zyklus sein!



v	$\delta_0(s,\underline{v})$	$\delta_1(s,\underline{v})$	$\delta_2(s,\underline{v})$	$\delta_3(s,\underline{v})$	$\delta(s,\underline{v})$
s	0	0	0	0	0
y	∞	1	0	-1	-∞
t	∞	-2	-2	-2	-∞

Beachte: y ist Zeuge!

Offensichtlich ist $\langle s, y, t\rangle$ ein gülitger Pfad und wegen weight$(y, y) < 0$ folgt sofort $\delta(s, t) = -\infty$. t st aber **kein Zeuge** für einen negativen Zyklus, denn

$$\delta_{|V|-1} \quad = \quad \delta_2(s,t) \quad = \quad \delta_3(s,t) \quad = \quad \delta_{|V|}(s,t) \quad = \quad -2$$

© 2006-2024 Jörg Striegnitz

Die Funktionen δ und δ_k - IV

L **Negative Zyklen und Zeugen**

Seien $G = (V, E)$ ein gewichteter Graph mit Gewichtsfunktion weight : $E \to \mathbb{R}$, $s \in V$ ein Startknoten, dann gilt für alle Knoten $v \in V$:

$$\delta(s, v) = -\infty \quad \Rightarrow v' \rightsquigarrow v$$

wobei v' Zeuge für einen negativen Zyklus ist.

© 2006-2024 Jörg Striegnitz

Die Funktionen δ und δ_k - IV

L Negative Zyklen und Zeugen

Seien $G = (V, E)$ ein gewichteter Graph mit Gewichtsfunktion weight : $E \to \mathbb{R}$, $s \in V$ ein Startknoten, dann gilt für alle Knoten $v \in V$:

$$\delta(s, v) = -\infty \quad \Rightarrow v' \rightsquigarrow v$$

wobei v' Zeuge für einen negativen Zyklus ist.

Beweis:

- Wir beweisen eine strengere Aussage: Jeder negative Zyklus enthält einen Zeugen v' für einen negativen Zyklus
  denn dann gilt selbstverständlich für jeden Knoten v, der von v' aus erreichbar ist: $\delta(s,v)=-\infty$
  (nächste Folie)

© 2006-2024 Jörg Striegnitz

Die Funktionen δ und δ_k - V

Beweis:

© 2006-2024 Jörg Striegnitz

Die Funktionen δ und δ_k - V

Beweis:

Betrachte negativen Zylkus $c = \langle v_0, \cdots, v_k \rangle$, also weight$(c) < 0$. Offenbar gilt $\forall i \in \{0, \cdots, k-1\}$

$$\delta_{|V|}(s, v_i) \leq \delta_{|V|-1}(s, v_{i+1}) \; + \; \text{weight}(v_i, v_{i+1})$$

Wenn wir dieses Argument auf den gesamten Zyklus anwenden, ergibt sich

$$\sum_{i=0}^{k-1} \delta_{|V|}(s, v_i) \leq \sum_{i=0}^{k-1} \delta_{|V|-1}(s, v_{i+1}) + \sum_{i=0}^{k-1} \text{weight}(v_i, v_{i+1}) = \text{weight}(c)$$

und wegen weight$(c) < 0$ erhalten wir

$$\sum_{i=0}^{k-1} \delta_{|V|}(s, v_i) < \sum_{i=0}^{k-1} \delta_{|V|-1}(s, v_{i+1})$$

Sollte c keinen Zeugen enthalten, so würde für alle $i \in \{0, \ldots, k-1\}$ $\delta_{|V|}(s, v_i) \geq \delta_{|V|+1}(s, v_i)$, was zu dieser Beobachtung im Widerspruch stünde.

© 2006-2024 Jörg Striegnitz

Bellmann-Ford (Variante)

- **Erzeuge G' durch *Graph Duplication*** (erhalte DAG)

$O(|V| \cdot (|V|+1))$ **Knoten**

$O(|V| \cdot (|V|+|E|))$ **Kanten**

© 2006-2024 Jörg Striegnitz

Bellmann-Ford (Variante)

- **Erzeuge G' durch *Graph Duplication*** (erhalte DAG)

  $O(|V| \cdot (|V|+1))$ **Knoten** $O(|V| \cdot (|V|+|E|))$ **Kanten**

- **Berechne $\delta(s, v_{i,k})$ durch Relaxation** (nach top.-Sort)

  $O(|V| \cdot (|V|+|E|))$ **Kanten**

© 2006-2024 Jörg Striegnitz

Bellmann-Ford (Variante)

- **Erzeuge G' durch *Graph Duplication*** (erhalte DAG)

  $O(|V| \cdot (|V|+1))$ **Knoten** $O(|V| \cdot (|V|+|E|))$ **Kanten**

- **Berechne $\delta(s, v_{i,k})$ durch Relaxation** (nach top.-Sort)

  $O(|V| \cdot (|V|+|E|))$ **Kanten**

- **Setze $d[v_i] = \delta(s, v_{i,|V|-1})$**

© 2006-2024 Jörg Striegnitz

Bellmann-Ford (Variante)

- **Erzeuge G' durch *Graph Duplication*** (erhalte DAG)

  $O(|V| \cdot (|V|+1))$ **Knoten** $O(|V| \cdot (|V|+|E|))$ **Kanten**

- **Berechne $\delta(s, v_{i,k})$ durch Relaxation** (nach top.-Sort)

  $O(|V| \cdot (|V|+|E|))$ **Kanten**

- **Setze $d[v_i] = \delta(s, v_{i,|V|-1})$**

- **Für alle Zeugen in G': Ermittle in G erreichbare Knoten v und setzte $d[v]=-\infty$**

  $O(|V| \cdot |E|)$ **Tiefensuche für $|V|$ Knoten**

© 2006-2024 Jörg Striegnitz

Bellmann-Ford (Variante)

- **Erzeuge G' durch *Graph Duplication*** (erhalte DAG)

$O(|V| \cdot (|V|+1))$ **Knoten** $O(|V| \cdot (|V|+|E|))$ **Kanten**

- **Berechne $\delta(s, v_{i,k})$ durch Relaxation** (nach top.-Sort)

$O(|V| \cdot (|V|+|E|))$ **Kanten**

- **Setze $d[v_i] = \delta(s, v_{i,|V|-1})$**

- **Für alle Zeugen in G': Ermittle in G erreichbare Knoten v und setzte $d[v]=-\infty$**

$O(|V| \cdot |E|)$ **Tiefensuche für $|V|$ Knoten**

Entfernt man in G zu Beginn alle Knoten (und zugehörige) Kanten, die von v_s aus nicht erreichbar sind, erhält man einen gerichteten Graphen $G^*(V^*, E^*)$ für den $|V^*| \in O(|E^*|+1) = O(|E^*|)$ gilt, da er zusammenhängend ist.
Ein Algorithmus mit Laufzeit in $O(|V| \cdot (|V|+|E|))$ gehört dann zur Komplexitätsklasse $O(|V| \cdot |E|)$.

$O(|V| \cdot |E|)$ **Insgesamt**

© 2006-2024 Jörg Striegnitz

VII. Graphen

7. Graphen

7.1. Einführung und Begriffsbestimmungen
7.2. Traversierung
7.3. Topologisches Sortieren
7.4. Kürzeste Wege
7.4.1. Einführung
7.4.2. *Single-Source Best Path* Algorithmen
7.4.3. All-Pairs Best Path Algorithmen
7.5. Maximaler Fluss
7.6. Minimaler Spannbaum

© 2006-2023 Jörg Striegnitz

All Pairs Best Path

- **Optimale Pfade zwischen allen Knotenpaaren eines Graphen**

  (Beispiel: Entfernungstabelle)

© 2006-2024 Jörg Striegnitz

All Pairs Best Path

- **Optimale Pfade zwischen allen Knotenpaaren eines Graphen**
  (Beispiel: Entfernungstabelle)

- **Einfache Strategie:**
  Single-Source Best Path-Methode auf jeden Knoten aus V anwenden!

© 2006-2024 Jörg Striegnitz

All Pairs Best Path

- **Optimale Pfade zwischen allen Knotenpaaren eines Graphen**
  (Beispiel: Entfernungstabelle)

- **Einfache Strategie:**
  Single-Source Best Path-Methode auf jeden Knoten aus V anwenden!

  - **Komplexität:**
    - $\Theta(|V| \cdot (|V| \cdot |E|))$ (Bellman-Ford)

© 2006-2024 Jörg Striegnitz

All Pairs Best Path

- **Optimale Pfade zwischen allen Knotenpaaren eines Graphen**
  (Beispiel: Entfernungstabelle)

- **Einfache Strategie:**
  Single-Source Best Path-Methode auf jeden Knoten aus V anwenden!

Laufzeit: $O(|V|^3)$
für allg. zusammenhängenden, dichten Graph

- **Komplexität:**
  - $\Theta(|V| \cdot (|V| \cdot |E|))$ (Bellman-Ford)

© 2006-2024 Jörg Striegnitz

All Pairs Best Path

- **Optimale Pfade zwischen allen Knotenpaaren eines Graphen**
  (Beispiel: Entfernungstabelle)

- **Einfache Strategie:**
  Single-Source Best Path-Methode auf jeden Knoten aus V anwenden!

Laufzeit: $O(|V|^3)$
für allg. zusammenhängenden, dichten Graph

- **Komplexität:**
  - $\Theta(|V| \cdot (|V| \cdot |E|))$ (Bellman-Ford)
  - $\Theta(|V| \cdot (|V| + |E|))$ (Topologisches Sortieren + Relax)
  - $O(|V| \cdot ((|V| \cdot \mathrm{ld}(|V|) + |E|))$ (Dijkstras Algorithmus)

- **Aber: Art der Graphen für schnellere Verfahren eingeschränkt!**
  (azyklischer Graph / Graph ohne negative Kanten ist Voraussetzung)

© 2006-2024 Jörg Striegnitz

All Pairs Best Path

- **Optimale Pfade zwischen allen Knotenpaaren eines Graphen**
  (Beispiel: Entfernungstabelle)

- **Einfache Strategie:**
  Single-Source Best Path-Methode auf jeden Knoten aus V anwenden!

Laufzeit: $O(|V|^3)$
für allg. zusammenhängenden, dichten Graph

- **Komplexität:**
  - $\Theta(|V| \cdot (|V| \cdot |E|))$ (Bellman-Ford)
  - $\Theta(|V| \cdot (|V| + |E|))$ (Topologisches Sortieren + Relax)
  - $O(|V| \cdot ((|V| \cdot \mathrm{ld}(|V|) + |E|))$ (Dijkstras Algorithmus)

- **Aber: Art der Graphen für schnellere Verfahren eingeschränkt!**
  (azyklischer Graph / Graph ohne negative Kanten ist Voraussetzung)

- **Es geht schneller …**

© 2006-2024 Jörg Striegnitz

Vorüberlegungen I

- **Im Ergebnis möchten wir eine Entfernungstabelle erhalten; also für einen Graphen $G=(V,E)$ mit $|V|=n$ eine $n \times n$ Matrix δ.**
  - $\delta(v_i, v_j)$ bezeichnet die minimalen Kosten, um von v_i nach v_j zu kommen.

© 2006-2024 Jörg Striegnitz

Vorüberlegungen I

- **Im Ergebnis möchten wir eine Entfernungstabelle erhalten; also für einen Graphen $G=(V,E)$ mit $|V|=n$ eine $n \times n$ Matrix δ.**
  - $\delta(v_i, v_j)$ bezeichnet die minimalen Kosten, um von v_i nach v_j zu kommen.
- **Diese Information reicht aber nicht aus, da wir so den eigentlichen Pfad nicht (direkt) rekonstruieren können!**

© 2006-2024 Jörg Striegnitz

Vorüberlegungen I

- **Im Ergebnis möchten wir eine Entfernungstabelle erhalten; also für einen Graphen $G=(V,E)$ mit $|V|=n$ eine $n \times n$ Matrix δ.**
  - $\delta(v_i, v_j)$ bezeichnet die minimalen Kosten, um von v_i nach v_j zu kommen.

- **Diese Information reicht aber nicht aus, da wir so den eigentlichen Pfad nicht (direkt) rekonstruieren können!**

- **Zur Pfadrekonstruktion führen wir eine Vorgängermatrix $\Pi=(\pi_{ij})$ ein:**
  - π_{ij} = `nil`, falls es **keinen** Pfad von i nach j gibt, oder $i=j$
  - ansonsten ist π_{ij} der **Vorgänger von Knoten j** auf einem kürzesten Pfad von Knoten i

© 2006-2024 Jörg Striegnitz

Vorüberlegungen II

- **Die i-te Zeile der Vorgängermatrix induziert einen Teilgraph, den *Predecessor Subgraph***

$$V_{\pi,i} = \{j \in V \,|\, \pi_{i,j} \neq \texttt{nil}\} \cup \{i\}$$

$$E_{\pi,i} = \{(\pi_{i,j}, j) \,|\, j \in V_{\pi,i} \setminus \{i\}\}$$

© 2006-2024 Jörg Striegnitz

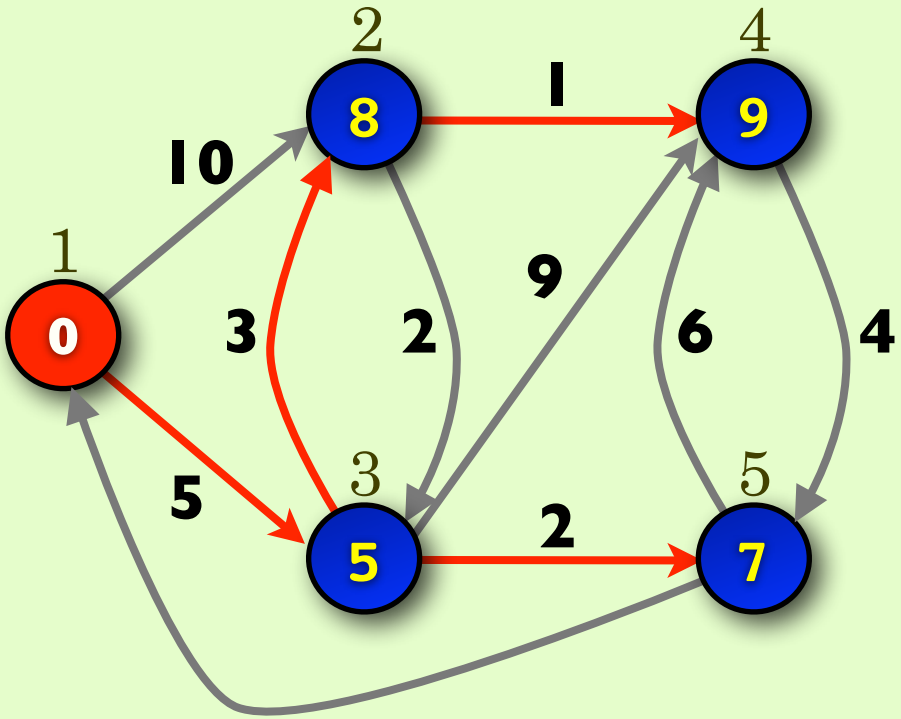
Vorüberlegungen II

- **Die i-te Zeile der Vorgängermatrix induziert einen Teilgraph, den *Predecessor Subgraph***

$$V_{\pi,i} = \{j \in V \,|\, \pi_{i,j} \neq \texttt{nil}\} \cup \{i\}$$

$$E_{\pi,i} = \{(\pi_{i,j}, j) \,|\, j \in V_{\pi,i} \setminus \{i\}\}$$

B *Predecessor Subgraph*



	1	2	3	4	5
π_{1j}	nil	3	1	2	3

Kürzeste Pfade vom Knoten 1 *(siehe Dijkstra-Algoritmus)*

© 2006-2024 Jörg Striegnitz

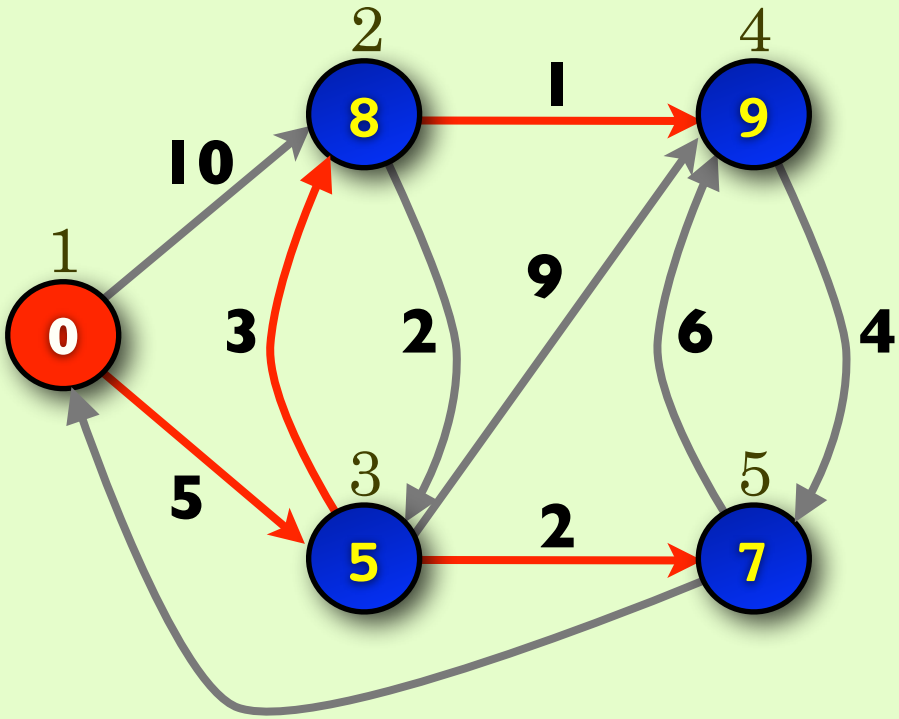
Vorüberlegungen II

- **Die i-te Zeile der Vorgängermatrix induziert einen Teilgraph, den *Predecessor Subgraph***

$$V_{\pi,i} = \{j \in V \,|\, \pi_{i,j} \neq \texttt{nil}\} \cup \{i\}$$

$$E_{\pi,i} = \{(\pi_{i,j}, j) \,|\, j \in V_{\pi,i} \setminus \{i\}\}$$

B *Predecessor Subgraph*



	1	2	3	4	5
π_{1j}	**nil**	**3**	**1**	**2**	**3**

Offenbar lassen sich beste Pfade leicht aus der Vorgängermatrix rekonstruieren!

Kürzeste Pfade vom Knoten 1 *(siehe Dijkstra-Algoritmus)*

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Dijkstra - Pseudocode

Initialisiere $d[v]$ *(* an der Stelle v_s Abstand 0; ansonsten Abstand ∞ (die größtmögliche Obergrenze) *)*

$S \leftarrow \emptyset$ *(* Liste der Knoten, für die optimaler Pfad bestimmt wurde *)*

$Q \leftarrow$ V(G) *(* Alle Knoten des Graphen G in die Queue *)*

while Q ≠ ∅

  do $u \leftarrow$ min Q *(* Knoten mit geringsten Pfadkosten auswählen ... *)*

    $Q \leftarrow Q \setminus \{u\}$ *(* ... und aus Queue entfernen *)*

    $S \leftarrow \{u\}$ *(* Pfad zu diesem Knoten ist optimal - könnte nur länger werden! *)*

    (Für die teureren Pfade könnte der Weg über u eine Abkürzung sein - also alle *)*

    (abgehenden Kanten relaxieren ... *)*

    for each vertex v in Q that leaves u

      if relax(u,v, weight) *(* **Vorsicht!** implizite Reorganisation der Queue! *)*

        $\pi_v = u$ *(* Relaxieren erfolgreich? Ja: u als Vorgänger auf Pfad nach v *)*

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Dijkstra - Pseudocode

Initialisiere $d[v]$ *(* an der Stelle v_s Abstand 0; ansonsten Abstand ∞ (die größtmögliche Obergrenze) *)*

$S \leftarrow \emptyset$ *(* Liste der Knoten, für die optimaler Pfad bestimmt wurde *)*

$Q \leftarrow$ V(G) *(* Alle Knoten des Graphen G in die Queue *)*

while Q ≠ ∅

  do $u \leftarrow$ min Q *(* Knoten mit geringsten Pfadkosten auswählen ... *)*

    $Q \leftarrow Q \setminus \{u\}$ *(* ... und aus Queue entfernen *)*

    $S \leftarrow \{u\}$ *(* Pfad zu diesem Knoten ist optimal - könnte nur länger werden! *)*

    (Für die teureren Pfade könnte der Weg über u eine Abkürzung sein - also alle *)*

    (abgehenden Kanten relaxieren ... *)*

    for each vertex v in Q that leaves u

      if relax(u,v, weight) *(* **Vorsicht!** implizite Reorganisation der Queue! *)*

        $\pi_v = u$ *(* Relaxieren erfolgreich? Ja: u als Vorgänger auf Pfad nach v *)*

© 2006-2024 Jörg Striegnitz

Optimale Pfade und Unterpfade

L **Unterpfade von kürzesten Pfaden sind kürzeste Pfade**

Sei $G = (V, E)$ ein gewichteter Graph mit Gewichtsfunktion weight : $E \to \mathbb{R}$ und ferner $p = \langle v_1, v_2, \ldots, v_k \rangle$ ein kürzester Pfad von v_1 nach v_k. Für beliebige i und j mit $1 \leq i \leq j \leq k$, sei $p_{i,j} = \langle v_i, \ldots v_j \rangle$ ein **Unterpfad** von p. Es gilt: $p_{i,j}$ ist ein kürzester Pfad von v_i nach v_j.

© 2006-2024 Jörg Striegnitz

Optimale Pfade und Unterpfade

L **Unterpfade von kürzesten Pfaden sind kürzeste Pfade**

Sei $G = (V, E)$ ein gewichteter Graph mit Gewichtsfunktion weight : $E \to \mathbb{R}$ und ferner $p = \langle v_1, v_2, \ldots, v_k \rangle$ ein kürzester Pfad von v_1 nach v_k. Für beliebige i und j mit $1 \leq i \leq j \leq k$, sei $p_{i,j} = \langle v_i, \ldots v_j \rangle$ ein **Unterpfad** von p. Es gilt: $p_{i,j}$ ist ein kürzester Pfad von v_i nach v_j.

Beachte: negative Kantengewichte sind ***nicht*** ausgeschlossen

© 2006-2024 Jörg Striegnitz

L **Unterpfade von kürzesten Pfaden sind kürzeste Pfade**

Sei $G = (V, E)$ ein gewichteter Graph mit Gewichtsfunktion weight : $E \to \mathbb{R}$ und ferner $p = \langle v_1, v_2, \ldots, v_k \rangle$ ein kürzester Pfad von v_1 nach v_k. Für beliebige i und j mit $1 \leq i \leq j \leq k$, sei $p_{i,j} = \langle v_i, \ldots v_j \rangle$ ein **Unterpfad** von p. Es gilt: $p_{i,j}$ ist ein kürzester Pfad von v_i nach v_j.

Beachte: negative Kantengewichte sind ***nicht*** ausgeschlossen

Beweis:

Wenn wir den Pfad p in

$$v_1 \overset{p_{1,i}}{\rightsquigarrow} v_i \overset{p_{i,j}}{\rightsquigarrow} v_j \overset{p_{j,k}}{\rightsquigarrow} v_k$$

zerlegen, dann gilt: $\text{weight}(p) = \text{weight}(p_{1,i}) + \text{weight}(p_{i,j}) + \text{weight}(p_{j,k})$.

Angenommen es existiert ein Pfad $p'_{i,j}$ von v_i nach v_j mit $\text{weight}(p'_{i,j}) < \text{weight}(i, j)$. Dann ist

$$v_1 \overset{p_{1,i}}{\rightsquigarrow} v_i \overset{p'_{i,j}}{\rightsquigarrow} v_j \overset{p_{j,k}}{\rightsquigarrow} v_k$$

ein Pfad von v_1 nach v_k, dessen Gewicht $\text{weight}(p_{1,i}) + \text{weight}(p'_{i,j}) + \text{weight}(p_{j,k})$ kleiner ist als $\text{weight}(p)$, was der Annahme widerspricht, das p der kürzeste Pfad von v_i nach v_k war.

© 2006-2024 Jörg Striegnitz

Optimale Pfade und Unterpfade

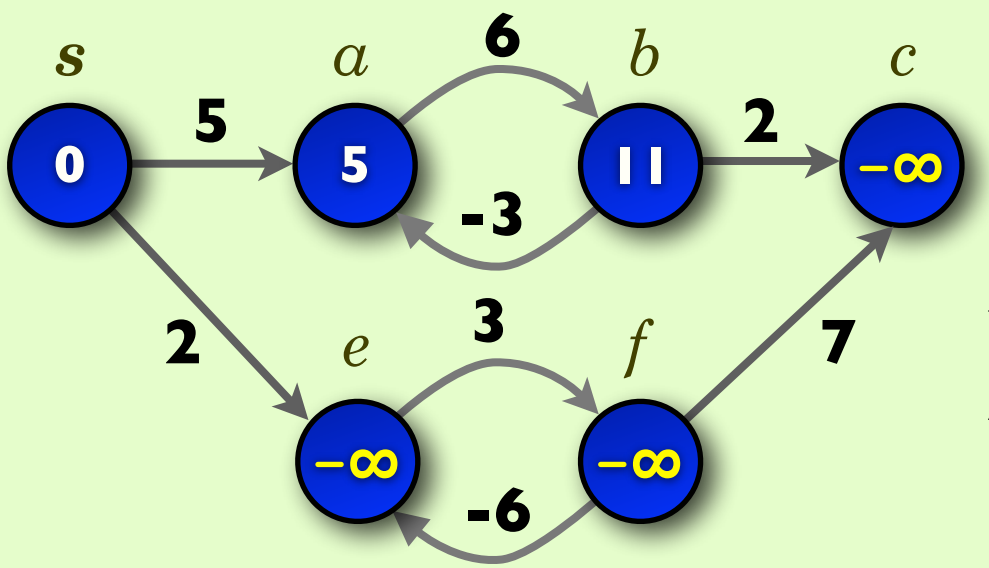
Anmerkung: In einem Graph, der eine negative Kante in einem Kreis hat, muss es nicht immer für jedes Knotenpaar einen optimalen (kürzesten) Pfad geben!

© 2006-2024 Jörg Striegnitz

Optimale Pfade und Unterpfade

Anmerkung: In einem Graph, der eine negative Kante in einem Kreis hat, muss es nicht immer für jedes Knotenpaar einen optimalen (kürzesten) Pfad geben!

B Negative Kanten: Auswirkungen



Kürzeste Pfade, ausgehend von Knoten s.

Die negative Kante zwischen a und b ist unkritisch, da sie die Kosten von s nach a nicht reduziert - der Umweg über b ist teurer als der Gewinn!

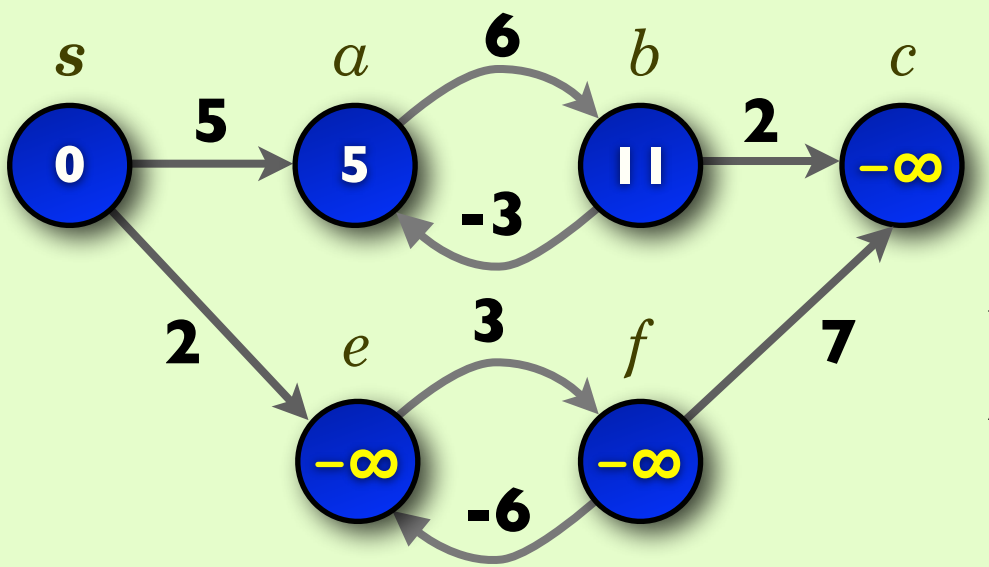
Anders beim Kreis zwischen Knoten e und f. Dort werden die Kosten kontinuierlich reduziert.

© 2006-2024 Jörg Striegnitz

Optimale Pfade und Unterpfade

Anmerkung: In einem Graph, der eine negative Kante in einem Kreis hat, muss es nicht immer für jedes Knotenpaar einen optimalen (kürzesten) Pfad geben!

B Negative Kanten: Auswirkungen



Kürzeste Pfade, ausgehend von Knoten s.

Die negative Kante zwischen a und b ist unkritisch, da sie die Kosten von s nach a nicht reduziert - der Umweg über b ist teurer als der Gewinn!

Anders beim Kreis zwischen Knoten e und f. Dort werden die Kosten kontinuierlich reduziert.

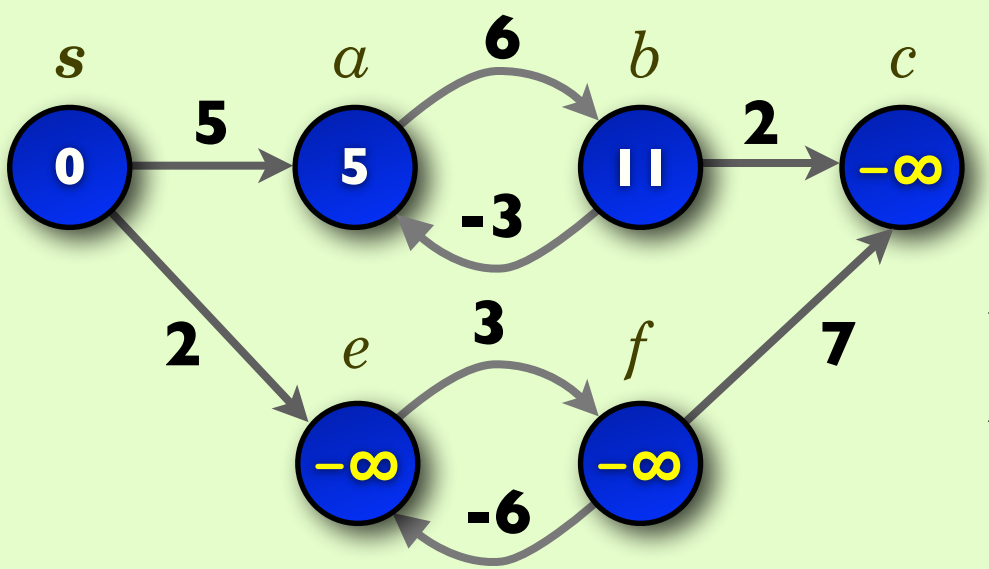
Wir betrachten von nun an Graphen ohne Kreise mit negativer Kantensumme (negative Kreise / Zyklen)**, lassen negative Kanten jedoch zu!**

© 2006-2024 Jörg Striegnitz

Optimale Pfade und Unterpfade

Anmerkung: In einem Graph, der eine negative Kante in einem Kreis hat, muss es nicht immer für jedes Knotenpaar einen optimalen (kürzesten) Pfad geben!

B Negative Kanten: Auswirkungen



Kürzeste Pfade, ausgehend von Knoten s.

Die negative Kante zwischen a und b ist unkritisch, da sie die Kosten von s nach a nicht reduziert - der Umweg über b ist teurer als der Gewinn!

Anders beim Kreis zwischen Knoten e und f. Dort werden die Kosten kontinuierlich reduziert.

Wir betrachten von nun an Graphen ohne Kreise mit negativer Kantensumme (negative Kreise / Zyklen)**, lassen negative Kanten jedoch zu!**

Konsequenz: Algorithmen arbeiten mit allgemeineren Graphen, als ein auf Dijkstra oder topologischer Sortierung basierender *All-Pairs Shortest Path*-Algorithmus!

© 2006-2024 Jörg Striegnitz

Probieren wir die dynamische Programmierung …

- **Warum? Unterpfade von optimalen Pfaden sind optimale Pfade!**
  - Erinnern Sie sich? Teilausdrücke eines optimal geklammerten Matrixettenproduktes sind optimal geklammert!

- **Schritte der dynamische Programmierung:**
  1. Charakterisiere die optimale Lösung
  2. Finde eine rekursive Definition der optimalen Lösung
  3. Berechne die Optimale Lösung „*bottom up*“
  4. Konstruiere die optimale Lösung aus den Zwischenergebnissen

© 2006-2024 Jörg Striegnitz

1.) Charakterisieren der optimalen Lösung

© 2006-2024 Jörg Striegnitz

1.) Charakterisieren der optimalen Lösung

- **Wie sieht die Struktur der optimalen Lösung aus?**

© 2006-2024 Jörg Striegnitz

1.) Charakterisieren der optimalen Lösung

- **Wie sieht die Struktur der optimalen Lösung aus?**

- **Betrachte günstigsten Pfad $p_{i,j}$ von v_i nach v_j. Dieser durchlaufe m Kanten** (**Anmerkung**: m ist finit, da wir negative Kreise verboten haben!)

© 2006-2024 Jörg Striegnitz

1.) Charakterisieren der optimalen Lösung

- **Wie sieht die Struktur der optimalen Lösung aus?**

- **Betrachte günstigsten Pfad $p_{i,j}$ von v_i nach v_j. Dieser durchlaufe m Kanten** (**Anmerkung**: m ist finit, da wir negative Kreise verboten haben!)

- **Wir unterscheiden zwei Fälle:**
  - $i = j$: der Pfad durchläuft keine Kante und hat demnach Gewicht 0

© 2006-2024 Jörg Striegnitz

1.) Charakterisieren der optimalen Lösung

- **Wie sieht die Struktur der optimalen Lösung aus?**

- **Betrachte günstigsten Pfad $p_{i,j}$ von v_i nach v_j. Dieser durchlaufe m Kanten** (**Anmerkung**: m ist finit, da wir negative Kreise verboten haben!)

- **Wir unterscheiden zwei Fälle:**
  - $i = j$: der Pfad durchläuft keine Kante und hat demnach Gewicht 0
  - $i \neq j$: dann kann $p_{i,j}$ wie folgt zerlegt werden

$$v_i \overset{p_{i,k}}{\rightsquigarrow} v_k \longrightarrow v_j$$

© 2006-2024 Jörg Striegnitz

1.) Charakterisieren der optimalen Lösung

- **Wie sieht die Struktur der optimalen Lösung aus?**

- **Betrachte günstigsten Pfad $p_{i,j}$ von v_i nach v_j. Dieser durchlaufe m Kanten** (**Anmerkung**: m ist finit, da wir negative Kreise verboten haben!)

- **Wir unterscheiden zwei Fälle:**
  - $i = j$: der Pfad durchläuft keine Kante und hat demnach Gewicht 0
  - $i \neq j$: dann kann $p_{i,j}$ wie folgt zerlegt werden

$$v_i \overset{p_{i,k}}{\rightsquigarrow} v_k \longrightarrow v_j$$

$p_{i,k}$ ist ein **Pfad mit m-1 Kanten**. $p_{i,k}$ ist aber auch ein günstigster Pfad von v_i nach v_k! (**Grund**: Unterpfade von optimalen Pfaden sind optimal!)

© 2006-2024 Jörg Striegnitz

1.) Charakterisieren der optimalen Lösung

- **Wie sieht die Struktur der optimalen Lösung aus?**

- **Betrachte günstigsten Pfad $p_{i,j}$ von v_i nach v_j. Dieser durchlaufe m Kanten** (**Anmerkung**: m ist finit, da wir negative Kreise verboten haben!)

- **Wir unterscheiden zwei Fälle:**

  - $i = j$: der Pfad durchläuft keine Kante und hat demnach Gewicht 0

  - $i \neq j$: dann kann $p_{i,j}$ wie folgt zerlegt werden

$$v_i \overset{p_{i,k}}{\rightsquigarrow} v_k \longrightarrow v_j$$

$p_{i,k}$ ist ein **Pfad mit m-1 Kanten**. $p_{i,k}$ ist aber auch ein günstigster Pfad von v_i nach v_k! (**Grund**: Unterpfade von optimalen Pfaden sind optimal!)

Folglich gilt für die Entfernungstabelle δ:

$$\delta(v_i, v_j) = \delta(v_i, v_k) + \mathrm{weight}(v_k, v_j)$$

© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung I

- **Nutze Struktur der optimalen Lösung zur Aufstellung einer Rekursionsgleichung, mit der diese berechnet werden kann.**

© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung I

- **Nutze Struktur der optimalen Lösung zur Aufstellung einer Rekursionsgleichung, mit der diese berechnet werden kann.**

- **Betrachte Matrix $L^{(m)}$ mit den Einträgen $l_{ij}{}^{(m)}$**

© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung I

- **Nutze Struktur der optimalen Lösung zur Aufstellung einer Rekursionsgleichung, mit der diese berechnet werden kann.**

- **Betrachte Matrix $L^{(m)}$ mit den Einträgen $l_{ij}{}^{(m)}$**

- **$l_{ij}{}^{(m)}$ bezeichne minimales Gewicht eines Pfades von v_i nach v_j mit höchstens m Kanten**

© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung I

- **Nutze Struktur der optimalen Lösung zur Aufstellung einer Rekursionsgleichung, mit der diese berechnet werden kann.**

- **Betrachte Matrix $L^{(m)}$ mit den Einträgen $l_{ij}{}^{(m)}$**

- **$l_{ij}{}^{(m)}$ bezeichne minimales Gewicht eines Pfades von v_i nach v_j mit höchstens m Kanten**

- **Berechne $l_{ij}{}^{(m)}$ wie folgt:**

© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung I

- **Nutze Struktur der optimalen Lösung zur Aufstellung einer Rekursionsgleichung, mit der diese berechnet werden kann.**

- **Betrachte Matrix $L^{(m)}$ mit den Einträgen $l_{ij}^{(m)}$**

- **$l_{ij}^{(m)}$ bezeichne minimales Gewicht eines Pfades von v_i nach v_j mit höchstens m Kanten**

- **Berechne $l_{ij}^{(m)}$ wie folgt:**
  - m=0: einen Pfad von v_i nach v_j, der keine Kanten enthält, gibt es nur für den Fall $v_i = v_j$

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{falls } i = j \\ \infty & \text{sonst } (i \neq j) \end{cases}$$

© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung II

- $m \geq 1$: wir setzen voraus, dass wir $L^{(m\text{-}1)}$, also die geringsten Kosten für Pfade mit maximaler Länge $m\text{-}1$, kennen (induktive Definition).

© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung II

- $m \geq 1$: wir setzen voraus, dass wir $L^{(m-1)}$, also die geringsten Kosten für Pfade mit maximaler Länge $m-1$, kennen (induktive Definition).
  - Aus der Charakterisierung der optimalen Lösung wissen wir, dass sich ein optimaler Pfad $p_{i,j}$ mit höchstens m Kanten zerlegen lässt in

© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung II

- $m \geq 1$: wir setzen voraus, dass wir $L^{(m\text{-}1)}$, also die geringsten Kosten für Pfade mit maximaler Länge $m\text{-}1$, kennen (induktive Definition).
  - Aus der Charakterisierung der optimalen Lösung wissen wir, dass sich ein optimaler Pfad $p_{i,j}$ mit höchstens m Kanten zerlegen lässt in

$$v_i \overset{p_{i,k}}{\rightsquigarrow} v_k \longrightarrow v_j$$

    wobei $p_{i,k}$ ein optimaler Pfad von v_i nach v_k mit höchstens $m\text{-}1$ Kanten ist - dessen Kosten kennen wir!

© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung II

- $m \geq 1$: wir setzen voraus, dass wir $L^{(m\text{-}1)}$, also die geringsten Kosten für Pfade mit maximaler Länge $m\text{-}1$, kennen (induktive Definition).
  - Aus der Charakterisierung der optimalen Lösung wissen wir, dass sich ein optimaler Pfad $p_{i,j}$ mit höchstens m Kanten zerlegen lässt in

    $$v_i \overset{p_{i,k}}{\rightsquigarrow} v_k \longrightarrow v_j$$

    wobei $p_{i,k}$ ein optimaler Pfad von v_i nach v_k mit höchstens $m\text{-}1$ Kanten ist - dessen Kosten kennen wir!
  - um $l_{ij}^{(m)}$ zu bestimmen, müssen für alle möglichen v_k prüfen, ob ein „Umweg“ über v_k die Kosten senkt; also:

© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung II

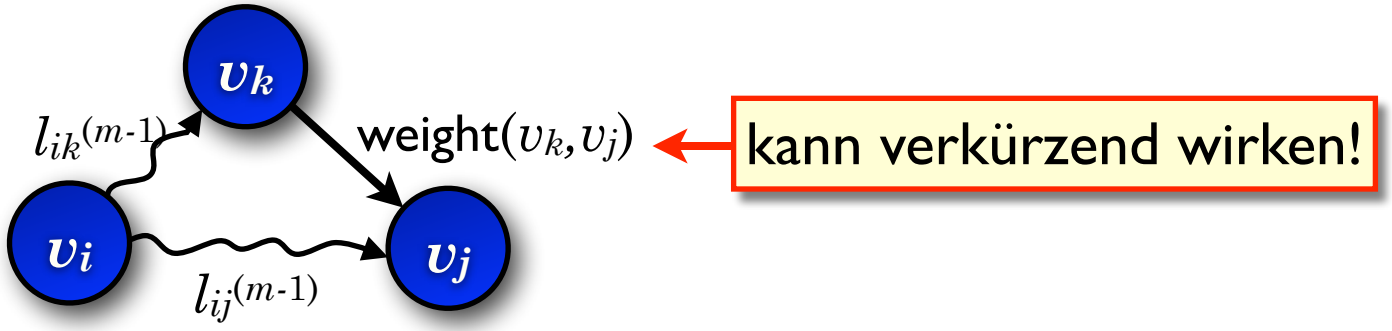
- $m \geq 1$: wir setzen voraus, dass wir $L^{(m-1)}$, also die geringsten Kosten für Pfade mit maximaler Länge m-1, kennen (induktive Definition).
  - Aus der Charakterisierung der optimalen Lösung wissen wir, dass sich ein optimaler Pfad $p_{i,j}$ mit höchstens m Kanten zerlegen lässt in

$$v_i \overset{p_{i,k}}{\rightsquigarrow} v_k \longrightarrow v_j$$

    wobei $p_{i,k}$ ein optimaler Pfad von v_i nach v_k mit höchstens m-1 Kanten ist - dessen Kosten kennen wir!

  - um $l_{ij}^{(m)}$ zu bestimmen, müssen für alle möglichen v_k prüfen, ob ein „Umweg“ über v_k die Kosten senkt; also:

$$l_{ij}^{(m)} = \min\left(l_{ij}^{(m-1)}, \min_{1 \leq k \leq n}\left\{l_{ik}^{(m-1)} + \text{weight}(v_k, v_j)\right\}\right)$$



© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung III

- $m \geq 1$: (Fortsetzung)

© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung III

- $m \geq 1$: (Fortsetzung)
  - um $l_{ij}{}^{(m)}$ zu bestimmen, müssen wir für alle möglichen v_k prüfen, ob ein „Umweg“ über v_k die Kosten senkt; also:

$$l_{ij}^{(m)} = \min\left(l_{ij}^{(m-1)}, \min_{1 \leq k \leq n}\left\{l_{ik}^{(m-1)} + \mathsf{weight}(v_k, v_j)\right\}\right)$$

© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung III

- $m \geq 1$: (Fortsetzung)
  - um $l_{ij}^{(m)}$ zu bestimmen, müssen wir für alle möglichen v_k prüfen, ob ein „Umweg“ über v_k die Kosten senkt; also:

$$l_{ij}^{(m)} = \min\left(l_{ij}^{(m-1)}, \min_{1 \leq k \leq n}\left\{l_{ik}^{(m-1)} + \text{weight}(v_k, v_j)\right\}\right)$$

  - wegen weight(v_i,v_i)=0 gilt für k=j:

$$l_{ij}^{(m-1)} + \text{weight}(v_j, v_j) = l_{ij}^{(m-1)}$$

© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung III

- $m \geq 1$: (Fortsetzung)
  - um $l_{ij}^{(m)}$ zu bestimmen, müssen wir für alle möglichen v_k prüfen, ob ein „Umweg“ über v_k die Kosten senkt; also:

$$l_{ij}^{(m)} = \min\left(l_{ij}^{(m-1)}, \min_{1 \leq k \leq n}\left\{l_{ik}^{(m-1)} + \text{weight}(v_k, v_j)\right\}\right)$$

  - wegen weight(v_i,v_i)=0 gilt für k=j:

$$l_{ij}^{(m-1)} + \text{weight}(v_j, v_j) = l_{ij}^{(m-1)}$$

  - und $l_{ij}^{(m-1)}$ wurde im **inneren Minimum** bereits betrachtet; es folgt daher:

$$l_{ij}^{(m)} = \min_{1 \leq k \leq n}\left\{l_{ik}^{(m-1)} + \text{weight}(v_k, v_j)\right\}$$

© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung IV

- **Es fehlt eine Abbruchbedingung**

© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung IV

- **Es fehlt eine Abbruchbedingung**
- **Negative Zyklen haben wir ausgeschlossen!**

© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung IV

- **Es fehlt eine Abbruchbedingung**
- **Negative Zyklen haben wir ausgeschlossen!**
  - Es kann daher keinen zyklischen Pfad geben, bei dem das Gewicht nicht kontinuierlich wächst!

© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung IV

- **Es fehlt eine Abbruchbedingung**
- **Negative Zyklen haben wir ausgeschlossen!**
  - Es kann daher keinen zyklischen Pfad geben, bei dem das Gewicht nicht kontinuierlich wächst!
  - Folglich kann ein optimaler Pfad von v_i nach v_j höchstens n-1 Kanten haben ($n = |V|$)
  - Daher gilt:

© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung IV

- **Es fehlt eine Abbruchbedingung**
- **Negative Zyklen haben wir ausgeschlossen!**
  - Es kann daher keinen zyklischen Pfad geben, bei dem das Gewicht nicht kontinuierlich wächst!
  - Folglich kann ein optimaler Pfad von v_i nach v_j höchstens n-1 Kanten haben ($n = |V|$)
  - Daher gilt:

$$\delta(v_i, v_j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} \; . \; . \; .$$

© 2006-2024 Jörg Striegnitz

3.) Berechne die optimale Lösung „*bottom up*“ I

Vorgehen:

- Aus der Eingabe-Matrix $W=w_{ij}$ mit w_{ij}=weight(v_i,v_j) berechnen wir eine Serie von Matrizen

$$L^{(1)},L^{(2)},\ldots,L^{(n-1)}$$

- wobei für $1 \leq m \leq n-1$ gilt

$$L^{(m)}=(l_{ij}^{(m)})$$

- $L^{(n-1)}$ wird dann die minimalen Pfadgewichte enthalten
  (Anmerkung: Offenbar gilt $L^{(1)}=W$)

- Wir entwickeln zunächst eine Funktion, die $L^{(m)}$ aus $L^{(m-1)}$ unter Verwendung der Gleichung

ermittelt

© 2006-2024 Jörg Striegnitz

3.) Berechne die optimale Lösung „*bottom up*“ I

Vorgehen:

- Aus der Eingabe-Matrix $W=w_{ij}$ mit w_{ij}=weight(v_i,v_j) berechnen wir eine Serie von Matrizen

$$L^{(1)},L^{(2)},\ldots,L^{(n-1)}$$

- wobei für $1 \leq m \leq n-1$ gilt

$$L^{(m)}=(l_{ij}^{(m)})$$

- $L^{(n-1)}$ wird dann die minimalen Pfadgewichte enthalten
  (Anmerkung: Offenbar gilt $L^{(1)}=W$)

- Wir entwickeln zunächst eine Funktion, die $L^{(m)}$ aus $L^{(m-1)}$ unter Verwendung der Gleichung

$$l_{ij}^{(m)} = \min_{1 \leq k \leq n} \left\{ l_{ik}^{(m-1)} + \text{weight}(v_k, v_j) \right\}$$

ermittelt

© 2006-2024 Jörg Striegnitz

3.) Berechne die optimale Lösung „*bottom up*“ II

$$l_{ij}^{(m)} = \min_{1 \le k \le n} \left\{ l_{ik}^{(m-1)} + \text{weight}(v_k, v_j) \right\}$$

Extend-Shortest-Paths(L,W):

Eingabe: Matrizen L und W

(W: Gewichtsmatrix; L minimales Gewicht der Pfade mit m-1 Knoten; also $L=L^{(m-1)}$ *)*

$n \leftarrow |V|$ *(* Anzahl der Knoten des Graphen *)*

$L^{(m)}$ **sei eine $n \times n$ Matrix** *(* Ergebnis: $L^{(m)}$ *)*

for $i \leftarrow 1$ to n

for $j \leftarrow 1$ to n

$l^{(m)}_{ij} \leftarrow \infty$ *(* l'_{ik}: Kosten bei Pfadlänge m; zunächst „exorbitante“ Kosten .... *)*

for $k \leftarrow 1$ to n

$l^{(m)}_{ij} \leftarrow \min(\, l^{(m)}_{ij},\, l_{ik} + w_{kj})$ *(* l_{ik}: geringste Kosten mit Pfadlänge m-1 *)*

(w_{kj}: gewichtsreduzierende Kante? *)*

return $L^{(m)}$

© 2006-2024 Jörg Striegnitz

3.) Berechne die optimale Lösung „*bottom up*“ II

$$l_{ij}^{(m)} = \min_{1 \le k \le n} \left\{ l_{ik}^{(m-1)} + \text{weight}(v_k, v_j) \right\}$$

Extend-Shortest-Paths(L,W):

Eingabe: Matrizen L und W

(W: Gewichtsmatrix; L minimales Gewicht der Pfade mit m-1 Knoten; also $L=L^{(m-1)}$ *)*

$n \leftarrow |V|$ *(* Anzahl der Knoten des Graphen *)*

$L^{(m)}$ **sei eine $n \times n$ Matrix** *(* Ergebnis: $L^{(m)}$ *)*

for $i \leftarrow 1$ to n

for $j \leftarrow 1$ to n

$l^{(m)}_{ij} \leftarrow \infty$ *(* l'_{ik}: Kosten bei Pfadlänge m; zunächst „exorbitante“ Kosten .... *)*

for $k \leftarrow 1$ to n

$l^{(m)}_{ij} \leftarrow \min(\, l^{(m)}_{ij},\, l_{ik} + w_{kj})$ *(* l_{ik}: geringste Kosten mit Pfadlänge m-1 *)*

(w_{kj}: gewichtsreduzierende Kante? *)*

return $L^{(m)}$

Laufzeit: $O(|V|^3)$

© 2006-2024 Jörg Striegnitz

3.) Berechne die optimale Lösung „*bottom up*“ III

- **Die Berechnung von $L^{(n\text{-}1)}$ ist jetzt leicht:**

© 2006-2024 Jörg Striegnitz

3.) Berechne die optimale Lösung „*bottom up*“ III

- **Die Berechnung von $L^{(n-1)}$ ist jetzt leicht:**

```
Slow-All-Pairs-Shortest-Path(G):
```

Eingabe: Graph $G=(V,E)$, Gewichtsmatrix W

(W: Gewichtsmatrix; L minimales Gewicht der Pfade mit m-1 Knoten; also $L=L^{(m-1)}$ *)*

$n \leftarrow |V|$ *(* Anzahl der Knoten des Graphen *)*

$L^{(1)}=W$

for $m \leftarrow 2$ **to** n-1

$L^{(m)}=$ `Extend-Shortest-Paths`$(L^{(m-1)},W)$

return $L^{(m)}$

© 2006-2024 Jörg Striegnitz

3.) Berechne die optimale Lösung „*bottom up*“ III

- **Die Berechnung von $L^{(n-1)}$ ist jetzt leicht:**

```
Slow-All-Pairs-Shortest-Path(G):
```

Eingabe: Graph $G=(V,E)$, Gewichtsmatrix W

(W: Gewichtsmatrix; L minimales Gewicht der Pfade mit m-1 Knoten; also $L=L^{(m-1)}$ *)*

$n \leftarrow |V|$ *(* Anzahl der Knoten des Graphen *)*

$L^{(1)}=W$

for $m \leftarrow 2$ **to** n-1

$L^{(m)}=$ `Extend-Shortest-Paths`$(L^{(m-1)},W)$

return $L^{(m)}$

Laufzeit: $O(|V|^4)$

© 2006-2024 Jörg Striegnitz

Optimierung I

- **Woran erinnert uns diese Funktion?**

```
Extend-Shortest-Paths(L,W):
  Eingabe: Matrizen L und W
        (* W: Gewichtsmatrix; L minimales Gewicht der Pfade mit m-1 Knoten; also L=L^(m-1) *)
    n ← |V|                                  (* Anzahl der Knoten des Graphen *)
    L^(m) sei eine n×n Matrix  (* Ergebnis: L^(m)                    *)
    for i←1 to n
      for j←1 to n
        l^(m)_ij ←∞        (* l'_ik: Kosten bei Pfadlänge m; zunächst „exorbitante" Kosten .... *)
        for k←1 to n
          l^(m)_ij ←min( l^(m)_ij, l_ik +w_kj)(* l_ik: geringste Kosten mit Pfadlänge m-1 *)
                                                (* w_kj: gewichtsreduzierende Kante?      *)
    return L^(m)
```


© 2006-2024 Jörg Striegnitz

Optimierung I

- **Woran erinnert uns diese Funktion?**

```
Extend-Shortest-Paths(L,W):
  Eingabe: Matrizen L und W
        (* W: Gewichtsmatrix; L minimales Gewicht der Pfade mit m-1 Knoten; also L=L^(m-1) *)
    n ← |V|                                  (* Anzahl der Knoten des Graphen *)
    L^(m) sei eine n×n Matrix (* Ergebnis: L^(m)                    *)
    for i←1 to n
      for j←1 to n
        l^(m)_ij ←∞        (* l'_ik: Kosten bei Pfadlänge m; zunächst „exorbitante" Kosten .... *)
        for k←1 to n
          l^(m)_ij ←min( l^(m)_ij, l_ik +w_kj)(* l_ik: geringste Kosten mit Pfadlänge m-1 *)
                                                (* w_kj: gewichtsreduzierende Kante?      *)
    return L^(m)
```

- **Die Matrixmultiplikation $c=a\cdot b$! Setze**
  - a → $l^{(m-1)}$
  - b → W
  - c → $l^{(m)}$
  - + → min
  - * → +

© 2006-2024 Jörg Striegnitz

Optimierung II

- Offenbar berechnet **`Slow-All-Pairs-Shortest-Path`** die folgende Sequenz (wobei für „$\cdot$“ **`Extend-Shortest-Paths`** steht)

$$
\begin{array}{lclcl}
L^{(1)} & = & L^{(0)} \cdot W & = & W \\
L^{(2)} & = & L^{(1)} \cdot W & = & W^2 \\
L^{(3)} & = & L^{(2)} \cdot W & = & W^3 \\
 & & \vdots & & \\
L^{(n-1)} & = & L^{(n-2)} \cdot W & = & W^{n-1}
\end{array}
$$

© 2006-2024 Jörg Striegnitz

Optimierung II

- Offenbar berechnet **`Slow-All-Pairs-Shortest-Path`** die folgende Sequenz (wobei für „$\cdot$“ **`Extend-Shortest-Paths`** steht)

$$
\begin{array}{lclcl}
L^{(1)} & = & L^{(0)} \cdot W & = & W \\
L^{(2)} & = & L^{(1)} \cdot W & = & W^2 \\
L^{(3)} & = & L^{(2)} \cdot W & = & W^3 \\
 & & \vdots & & \\
L^{(n-1)} & = & L^{(n-2)} \cdot W & = & W^{n-1}
\end{array}
$$

- Uns interessieret nur $L^{(n\text{-}1)}$!

© 2006-2024 Jörg Striegnitz

Optimierung II

- Offenbar berechnet **`Slow-All-Pairs-Shortest-Path`** die folgende Sequenz (wobei für „$\cdot$“ **`Extend-Shortest-Paths`** steht)

$$
\begin{array}{lclcl}
L^{(1)} & = & L^{(0)} \cdot W & = & W \\
L^{(2)} & = & L^{(1)} \cdot W & = & W^2 \\
L^{(3)} & = & L^{(2)} \cdot W & = & W^3 \\
 & & \vdots & & \\
L^{(n-1)} & = & L^{(n-2)} \cdot W & = & W^{n-1}
\end{array}
$$

- Uns interessieret nur $L^{(n\text{-}1)}$!

- Die „*Matrix-Multiplikation*“ von **`Extent-ShortestPaths`** ist assoziativ! Wir können daher auch folgende Sequenz berechnen:

$$
\begin{array}{lclcl}
L^{(1)} & = & W & & \\
L^{(2)} & = & W^2 & = & W \cdot W \\
L^{(4)} & = & W^4 & = & W^2 \cdot W^2 \\
 & & \vdots & & \\
L^{(2^{\lceil \mathrm{ld}\ (n-1) \rceil})} & = & W^{(2^{\lceil \mathrm{ld}\ (n-1) \rceil})} & = & W^{(2^{\lceil \mathrm{ld}\ (n-1) \rceil})-1} \cdot W^{(2^{\lceil \mathrm{ld}\ (n-1) \rceil}-1)}
\end{array}
$$

© 2006-2024 Jörg Striegnitz

Optimierung III

- **Diese Idee führt zu:**

```
Faster-All-Pairs-Shortest-Path(G):
```

Eingabe: Graph $G=(V,E)$, Gewichtsmatrix W

(W: Gewichtsmatrix; L minimales Gewicht der Pfade mit m-1 Knoten; also $L=L^{(m-1)}$ *)*

$n \leftarrow |V|$ *(* Anzahl der Knoten des Graphen *)*

$L^{(1)}=W$ *(* Ergebnis: $L^{(m)}$ *)*

while $m<n-1$

$L^{(2m)}=$ `Extend-Shortest-Paths`$(L^{(m)}, L^{(m)})$

$m \leftarrow 2m$

return $L^{(m)}$

© 2006-2024 Jörg Striegnitz

Optimierung III

- **Diese Idee führt zu:**

```
Faster-All-Pairs-Shortest-Path(G):
```

Eingabe: Graph $G=(V,E)$, Gewichtsmatrix W

(W: Gewichtsmatrix; L minimales Gewicht der Pfade mit m-1 Knoten; also $L=L^{(m-1)}$ *)*

$n \leftarrow |V|$ *(* Anzahl der Knoten des Graphen *)*

$L^{(1)}=W$ *(* Ergebnis: $L^{(m)}$ *)*

while $m<n-1$

$L^{(2m)}=$ `Extend-Shortest-Paths`$(L^{(m)}, L^{(m)})$

$m \leftarrow 2m$

return $L^{(m)}$

Laufzeit: $O(\mathrm{ld}\ |V| \bullet |V|^3)$

© 2006-2024 Jörg Striegnitz

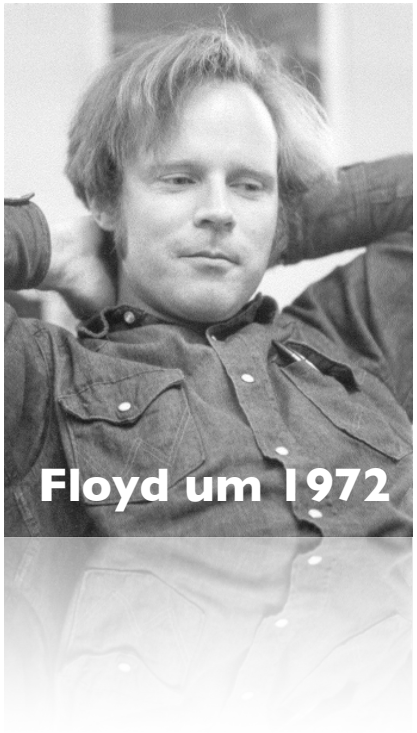
Floyd-Warshall-Algorithmus - Motivation

- **Erstmalig veröffentlicht von Robert Floyd und Stephen Warshall 1962**
- ***All-Pairs Shortest Path* Algorithmus, der auf den Ansatz der dynamischen Programmierung beruht**
- **Gut: Algorithmus erkennt Graphen mit negativen Zyklen!**

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus - Motivation

- **Erstmalig veröffentlicht von Robert Floyd und Stephen Warshall 1962**
- ***All-Pairs Shortest Path* Algorithmus, der auf den Ansatz der dynamischen Programmierung beruht**
- **Gut: Algorithmus erkennt Graphen mit negativen Zyklen!**



© 2006-2024 Jörg Striegnitz

1.) Charakterisieren der optimalen Lösung I

© 2006-2024 Jörg Striegnitz

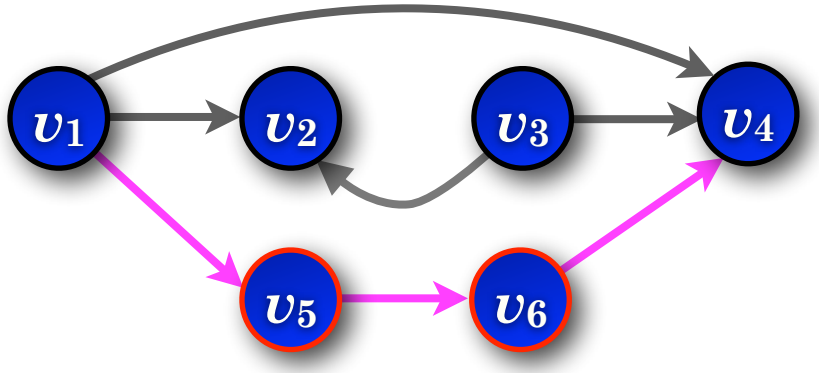
1.) Charakterisieren der optimalen Lösung I

- Sei $p=v_1,\ldots,v_n$ ein Pfad. Die Knoten $v_2,\ldots,v_{n-1}$ heißen **innere Knoten** des Pfades p

© 2006-2024 Jörg Striegnitz

1.) Charakterisieren der optimalen Lösung I

- Sei $p=v_1,\ldots,v_n$ ein Pfad. Die Knoten $v_2,\ldots,v_{n-1}$ heißen **innere Knoten** des Pfades p



v_5 und v_6 sind innere Knoten des Pfades

$p= v_1, v_5, v_6, v_4$

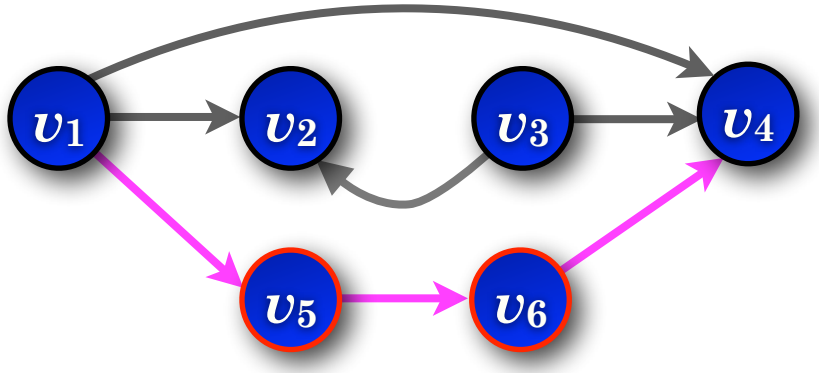
Wir gehen einen „**Umweg**“ über v_5 und v_6

(Umweg i.S.v. # durchlaufene Kanten)

© 2006-2024 Jörg Striegnitz

1.) Charakterisieren der optimalen Lösung I

- Sei $p=v_1,\ldots,v_n$ ein Pfad. Die Knoten $v_2,\ldots,v_{n-1}$ heißen **innere Knoten** des Pfades p



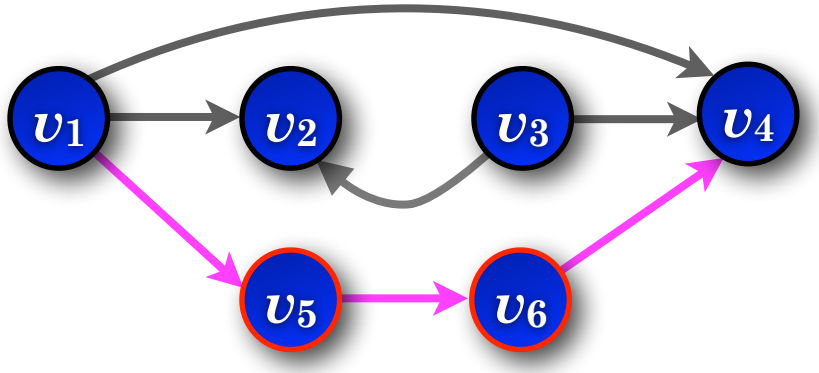
v_5 und v_6 sind innere Knoten des Pfades
$p= v_1, v_5, v_6, v_4$
Wir gehen einen „**Umweg**“ über v_5 und v_6
(Umweg i.S.v. # durchlaufene Kanten)

- Sei $V=\{v_1,\ldots,v_n\}$ und $V_k=\{v_1,v_2,\ldots,v_k\}$ eine Teilmenge von V ($k\leq n$) und seien $p_{i,j,k}$ Pfade von v_i nach v_j, deren innere Knoten nur in V_k liegen.

© 2006-2024 Jörg Striegnitz

1.) Charakterisieren der optimalen Lösung I

- Sei $p=v_1,\ldots,v_n$ ein Pfad. Die Knoten $v_2,\ldots,v_{n-1}$ heißen **innere Knoten** des Pfades p



v_5 und v_6 sind innere Knoten des Pfades
$p= v_1, v_5, v_6, v_4$
Wir gehen einen „**Umweg**“ über v_5 und v_6
(Umweg i.S.v. # durchlaufene Kanten)

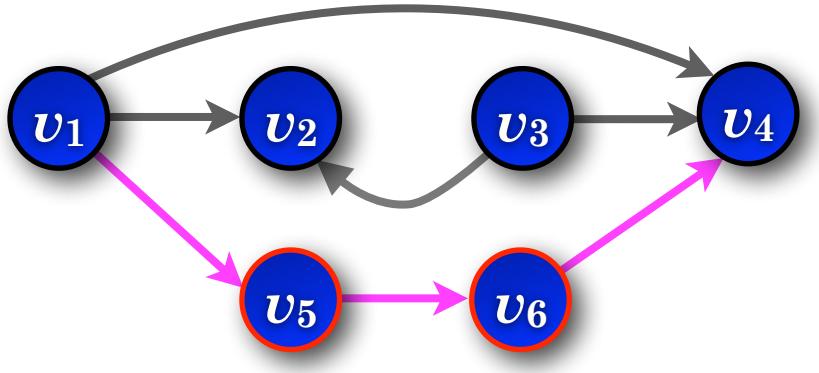
- Sei $V=\{v_1,\ldots,v_n\}$ und $V_k=\{v_1,v_2,\ldots,v_k\}$ eine Teilmenge von V ($k\leq n$) und seien $p_{i,j,k}$ Pfade von v_i nach v_j, deren innere Knoten nur in V_k liegen.

- Betrachte $p_{i,j,k,Opt}$: optimaler Pfad mit inneren Knoten nur aus V_k
  (also: $p_{i,j,k,Opt}$ = Optimaler Pfad mit „Umweg“ über Knoten aus V_k)

© 2006-2024 Jörg Striegnitz

1.) Charakterisieren der optimalen Lösung I

- Sei $p=v_1,\ldots,v_n$ ein Pfad. Die Knoten $v_2,\ldots,v_{n-1}$ heißen **innere Knoten** des Pfades p



v_5 und v_6 sind innere Knoten des Pfades
$p= v_1, v_5, v_6, v_4$
Wir gehen einen „**Umweg**“ über v_5 und v_6
(Umweg i.S.v. # durchlaufene Kanten)

- Sei $V=\{v_1,\ldots,v_n\}$ und $V_k=\{v_1,v_2,\ldots,v_k\}$ eine Teilmenge von V ($k\leq n$) und seien $p_{i,j,k}$ Pfade von v_i nach v_j, deren innere Knoten nur in V_k liegen.

- Betrachte $p_{i,j,k,Opt}$: optimaler Pfad mit inneren Knoten nur aus V_k
  (also: $p_{i,j,k,Opt}$ = Optimaler Pfad mit „Umweg“ über Knoten aus V_k)

- Der Floyd-Warshall Algorithmus nutzt einen Zusammenhang zwischen $p_{i,j,k,Opt}$ (mit inneren Knoten aus V_k) und den kürzesten Pfaden mit inneren Knoten aus $V_{k-1}=\{v_1,\ldots,v_{k-1}\}$.

© 2006-2024 Jörg Striegnitz

1.) Charakterisieren der optimalen Lösung II

© 2006-2024 Jörg Striegnitz

1.) Charakterisieren der optimalen Lösung II

- **Der Beziehung zwischen $p_{i,j,k,Opt}$ und $V_{k\text{-}1}$ hängt davon ab, ob v_k ein innerer Knoten von $p_{i,j,k,Opt}$ ist.**

© 2006-2024 Jörg Striegnitz

1.) Charakterisieren der optimalen Lösung II

- **Der Beziehung zwischen $p_{i,j,k,Opt}$ und $V_{k\text{-}1}$ hängt davon ab, ob v_k ein innerer Knoten von $p_{i,j,k,Opt}$ ist.**
  - Ist v_k ***kein*** **innerer Knoten** von $p_{i,j,k,Opt}$, liegen alle inneren Knoten von $p_{i,j,k,Opt}$, bereits in $V_{k\text{-}1}$. Folglich ist der kürzeste Pfad $p_{i,j,k\text{-}1,Opt}$ (von v_i nach v_j mit inneren Knoten aus $V_{k\text{-}1}$) gleichzeitig ein kürzester Pfad $p_{i,j,k,Opt}$ mit inneren Knoten aus V_k - ein „Umweg“ über v_k brachte keine Verbesserung!

© 2006-2024 Jörg Striegnitz

1.) Charakterisieren der optimalen Lösung II

- **Der Beziehung zwischen $p_{i,j,k,Opt}$ und $V_{k\text{-}1}$ hängt davon ab, ob v_k ein innerer Knoten von $p_{i,j,k,Opt}$ ist.**
  - Ist v_k ***kein*** **innerer Knoten** von $p_{i,j,k,Opt}$, liegen alle inneren Knoten von $p_{i,j,k,Opt}$, bereits in $V_{k\text{-}1}$. Folglich ist der kürzeste Pfad $p_{i,j,k\text{-}1,Opt}$ (von v_i nach v_j mit inneren Knoten aus $V_{k\text{-}1}$) gleichzeitig ein kürzester Pfad $p_{i,j,k,Opt}$ mit inneren Knoten aus V_k - ein „Umweg“ über v_k brachte keine Verbesserung!
  - Ist v_k ***ein*** **innerer Knoten** von $p_{i,j,k,Opt}$, dann hat der Pfad folgende Gestalt:

$$p_{i,j,k,Opt} = v_i \overset{p_1}{\rightsquigarrow} v_k \overset{p_2}{\rightsquigarrow} v_j$$

© 2006-2024 Jörg Striegnitz

1.) Charakterisieren der optimalen Lösung II

- **Der Beziehung zwischen $p_{i,j,k,Opt}$ und $V_{k\text{-}1}$ hängt davon ab, ob v_k ein innerer Knoten von $p_{i,j,k,Opt}$ ist.**
  - Ist v_k ***kein*** **innerer Knoten** von $p_{i,j,k,Opt}$, liegen alle inneren Knoten von $p_{i,j,k,Opt}$, bereits in $V_{k\text{-}1}$. Folglich ist der kürzeste Pfad $p_{i,j,k\text{-}1,Opt}$ (von v_i nach v_j mit inneren Knoten aus $V_{k\text{-}1}$) gleichzeitig ein kürzester Pfad $p_{i,j,k,Opt}$ mit inneren Knoten aus V_k - ein „Umweg“ über v_k brachte keine Verbesserung!
  - Ist v_k ***ein*** **innerer Knoten** von $p_{i,j,k,Opt}$, dann hat der Pfad folgende Gestalt:

$$p_{i,j,k,Opt} = v_i \overset{p_1}{\rightsquigarrow} v_k \overset{p_2}{\rightsquigarrow} v_j$$

  - Es gilt, da Unterpfade von kürzesten Pfaden kürzeste Pfade sind - (s.o.):
    - $p_1 = p_{i,k,k\text{-}1,Opt}$ (kürzester Pfad von v_i nach v_k mit inneren Knoten aus $V_{k\text{-}1}$)
    - $p_2 = p_{k,j,k\text{-}1,Opt}$ (kürzester Pfad von v_k nach v_j mit inneren Knoten aus $V_{k\text{-}1}$)

© 2006-2024 Jörg Striegnitz

1.) Charakterisieren der optimalen Lösung II

- **Der Beziehung zwischen $p_{i,j,k,Opt}$ und $V_{k\text{-}1}$ hängt davon ab, ob v_k ein innerer Knoten von $p_{i,j,k,Opt}$ ist.**
  - Ist v_k ***kein*** **innerer Knoten** von $p_{i,j,k,Opt}$, liegen alle inneren Knoten von $p_{i,j,k,Opt}$, bereits in $V_{k\text{-}1}$. Folglich ist der kürzeste Pfad $p_{i,j,k\text{-}1,Opt}$ (von v_i nach v_j mit inneren Knoten aus $V_{k\text{-}1}$) gleichzeitig ein kürzester Pfad $p_{i,j,k,Opt}$ mit inneren Knoten aus V_k - ein „Umweg“ über v_k brachte keine Verbesserung!
  - Ist v_k ***ein*** **innerer Knoten** von $p_{i,j,k,Opt}$, dann hat der Pfad folgende Gestalt:

$$p_{i,j,k,Opt} = v_i \overset{p_1}{\rightsquigarrow} v_k \overset{p_2}{\rightsquigarrow} v_j$$

  - Es gilt, da Unterpfade von kürzesten Pfaden kürzeste Pfade sind - (s.o.):
    - $p_1 = p_{i,k,k\text{-}1,Opt}$ (kürzester Pfad von v_i nach v_k mit inneren Knoten aus $V_{k\text{-}1}$)
    - $p_2 = p_{k,j,k\text{-}1,Opt}$ (kürzester Pfad von v_k nach v_j mit inneren Knoten aus $V_{k\text{-}1}$)

Fazit: kennen wir einen kürzesten Pfad $p_{i,j,k\text{-}1,Opt}$ von v_i nach v_j, mit inneren Knoten aus $V_{k\text{-}1}$, können wir leicht einen kürzesten Pfad konstruieren, der zusätzlich v_k als „Umweg“ nutzt.

© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung

- $d_{ij}^{(k)}$: Gewicht eines kürzesten Pfades von v_i nach v_j, bei dem alle inneren Knoten aus $V_k=\{v_1,\ldots,v_k\}$ stammen

© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung

- $d_{ij}^{(k)}$: **Gewicht eines kürzesten Pfades von v_i nach v_j, bei dem alle inneren Knoten aus $V_k=\{v_1,\ldots,v_k\}$ stammen**

- **Offenbar gilt: $d_{ij}^{(0)}=w_{ij}$;**
  ansonsten ergibt sich $d_{ij}^{(k)}$ wie folgt *(siehe obige Überlegungen)*:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{falls } k = 0 \\ \min\left(d_{ij}^{(k-1)} \;,\; d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{falls } k \geq 1 \end{cases}$$

© 2006-2024 Jörg Striegnitz

2.) Rekursive Definition der optimalen Lösung

- $d_{ij}^{(k)}$: **Gewicht eines kürzesten Pfades von v_i nach v_j, bei dem alle inneren Knoten aus $V_k=\{v_1,\ldots,v_k\}$ stammen**

- **Offenbar gilt: $d_{ij}^{(0)}=w_{ij}$;**
  ansonsten ergibt sich $d_{ij}^{(k)}$ wie folgt *(siehe obige Überlegungen)***:**

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{falls } k = 0 \\ \min\left(d_{ij}^{(k-1)} \;,\; d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{falls } k \geq 1 \end{cases}$$

- **Für alle Pfade gilt, dass die inneren Knoten in der Menge V_n liegen** *(negative Kreise sind ja ausgeschlossen)***. Es folgt:**

$$\delta(v_i, v_j) = d_{ij}^{(n)} \text{ für alle } i, j \in V$$

© 2006-2024 Jörg Striegnitz

3.) Berechne die optimale Lösung „*bottom up*“

```
Floyd-Warshall(W)
 Eingabe: Matrix W                          (* W: Gewichtsmatrix                  *)
   n ← |V|                                  (* Anzahl der Knoten des Graphen *)
   D(0) = W
   for k←1 to n     (* „Umweg“ zusätzlich über v_k        *)
    for i←1 to n    (* Alle Pfade von v_i ....            *)
        for j←1 to n (*  .... nach v_j                     *)
          (* Liefert zusätzlicher Umweg über v_k echte Verbesserung ?      *)
          d_ij(k) ←min( d_ij(k-1), d_ik(k-1)+ d_kj(k-1))
   return D(n)
```


© 2006-2024 Jörg Striegnitz

3.) Berechne die optimale Lösung „*bottom up*“

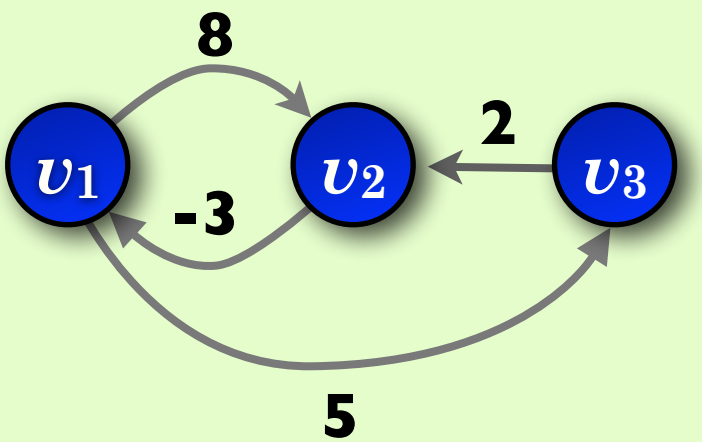
```
Floyd-Warshall(W)
 Eingabe: Matrix W                          (* W: Gewichtsmatrix                  *)
   n ← |V|                                  (* Anzahl der Knoten des Graphen *)
   D(0) = W
   for k←1 to n      (* „Umweg“ zusätzlich über v_k        *)
    for i←1 to n     (* Alle Pfade von v_i ....            *)
        for j←1 to n (* .... nach v_j                      *)
          (* Liefert zusätzlicher Umweg über v_k echte Verbesserung ?      *)
          d_ij(k) ←min( d_ij(k-1), d_ik(k-1)+ d_kj(k-1))
   return D(n)
```

Laufzeit: $O(|V|^3)$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



Beachte: negative Kante, kein negativer Zyklus; „*loops*“ wurden **nicht** eingezeichnet …

$D^{(0)}$		nach		
		v_1	v_2	v_3
von	v_1	**0**	8	5
	v_2	-3	**0**	∞
	v_3	∞	2	**0**

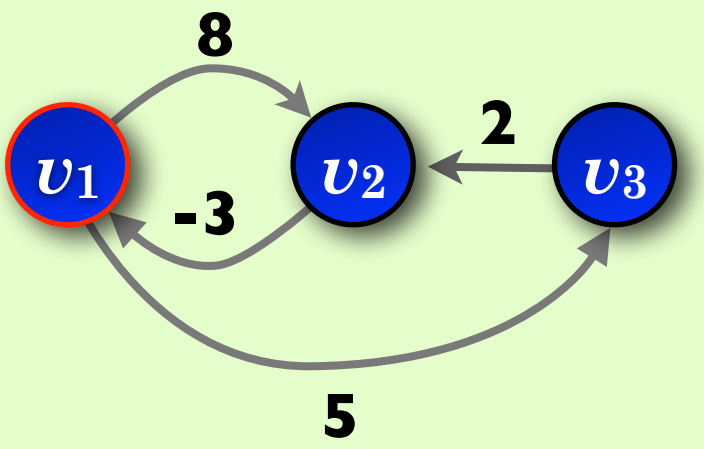
$D^{(0)}$: direkter Weg

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{falls } k = 0 \\ \min\left(\left(d_{ij}^{(k-1)} \;,\; d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)\right. & \text{falls } k \geq 1) \end{cases}$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(1)}$	nach			
von		v_1	v_2	v_3
	v_1	**0**	8	5
	v_2	-3	**0**	∞
	v_3	∞	2	**0**

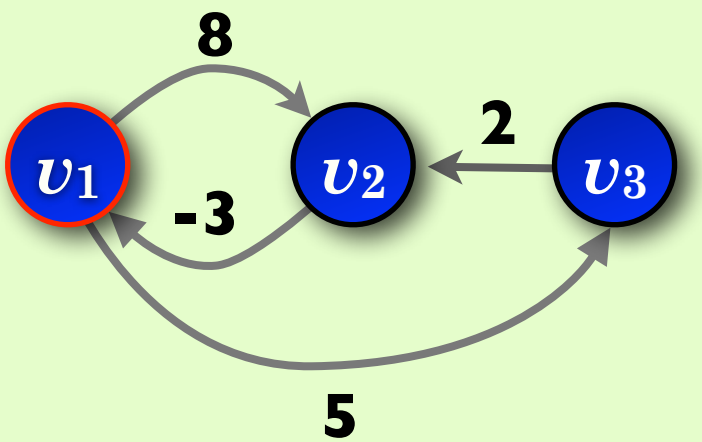
$D^{(1)}$: Umweg über $\{v_1\}$

$$d_{ij}^{(1)} = \min\left(d_{ij}^{(0)} \;,\; d_{i1}^{(0)} + d_{1j}^{(0)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(1)}$		nach		
		v_1	v_2	v_3
von	v_1	**0**	8	5
	v_2	-3	**0**	∞
	v_3	∞	2	**0**

von v_1 nach v_2 via v_1

➡ keine Verbesserung!

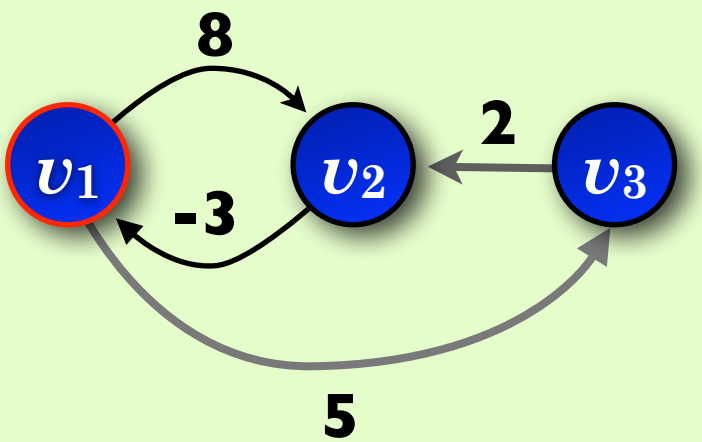
$D^{(1)}$: Umweg über $\{v_1\}$

$$d_{ij}^{(1)} = \min\left(d_{ij}^{(0)} \;,\; d_{i1}^{(0)} + d_{1j}^{(0)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



von v_1 nach v_3 via v_1
➡ keine Verbesserung!

$D^{(1)}$		nach		
		v_1	v_2	v_3
von	v_1	0	8	5
	v_2	-3	0	∞
	v_3	∞	2	0

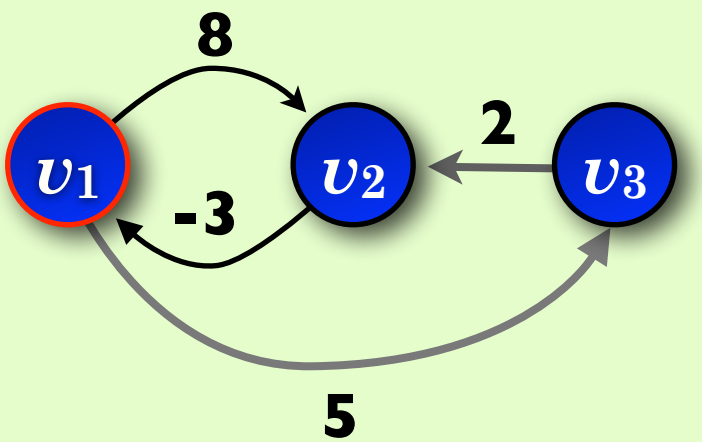
$D^{(1)}$: Umweg über $\{v_1\}$

$$d_{ij}^{(1)} = \min\left(d_{ij}^{(0)} \;,\; d_{i1}^{(0)} + d_{1j}^{(0)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(1)}$		nach		
		v_1	v_2	v_3
von	v_1	**0**	8	5
	v_2	-3	**0**	∞
	v_3	∞	2	**0**

von v_2 nach v_1 via v_1

➡ keine Verbesserung!

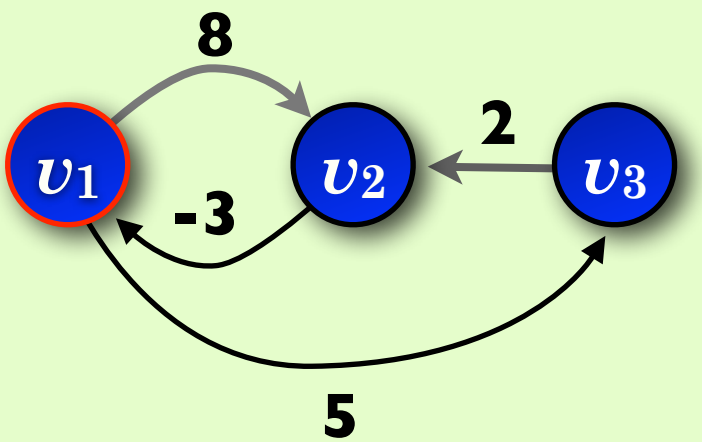
$D^{(1)}$: Umweg über $\{v_1\}$

$$d_{ij}^{(1)} = \min\left(d_{ij}^{(0)} \;,\; d_{i1}^{(0)} + d_{1j}^{(0)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(1)}$		nach		
		v_1	v_2	v_3
von	v_1	**0**	8	5
	v_2	-3	**0**	∞
	v_3	∞	2	**0**

von v_2 nach v_2 via v_1

➡ keine Verbesserung!

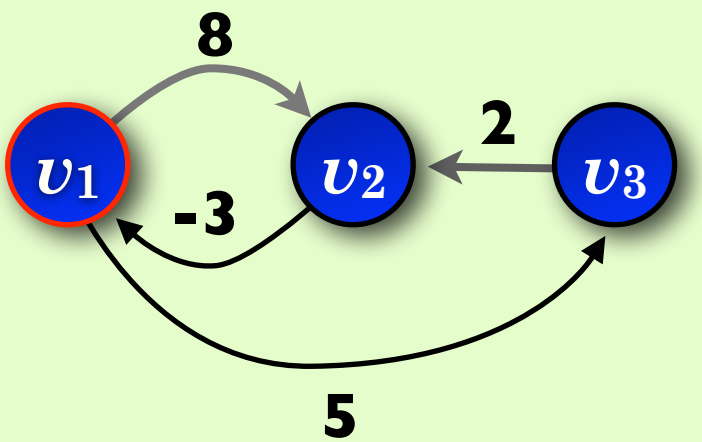
$D^{(1)}$: Umweg über $\{v_1\}$

$$d_{ij}^{(1)} = \min\left(d_{ij}^{(0)} \;,\; d_{i1}^{(0)} + d_{1j}^{(0)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(1)}$		nach		
		v_1	v_2	v_3
von	v_1	**0**	8	5
	v_2	-3	**0**	∞
	v_3	∞	2	**0**

von v_2 nach v_3 via v_1: Kosten=2
➡ Verbesserung von ∞

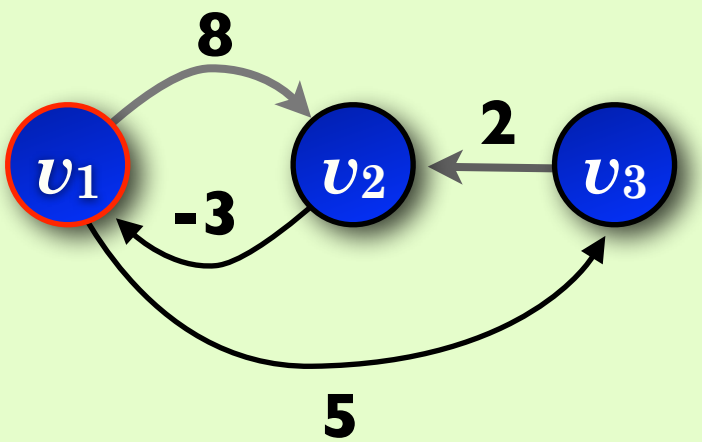
$D^{(1)}$: Umweg über $\{v_1\}$

$$d_{ij}^{(1)} = \min\left(d_{ij}^{(0)} \;,\; d_{i1}^{(0)} + d_{1j}^{(0)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(1)}$		nach		
		v_1	v_2	v_3
von	v_1	**0**	8	5
	v_2	-3	**0**	2
	v_3	∞	2	**0**

von v_2 nach v_3 via v_1: Kosten=2
➡ Verbesserung von ∞

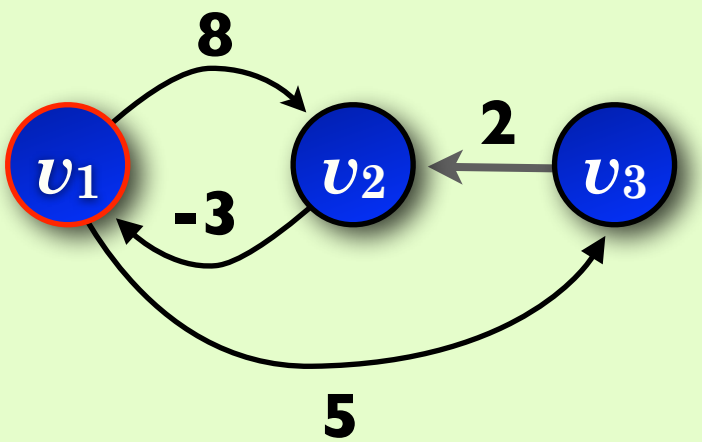
$D^{(1)}$: Umweg über $\{v_1\}$

$$d_{ij}^{(1)} = \min\left(d_{ij}^{(0)} \;,\; d_{i1}^{(0)} + d_{1j}^{(0)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(1)}$		nach		
		v_1	v_2	v_3
von	v_1	**0**	8	5
	v_2	-3	**0**	**2**
	v_3	**∞**	2	**0**

von v_3 nach v_1 via v_1

➡ keine Verbesserung!

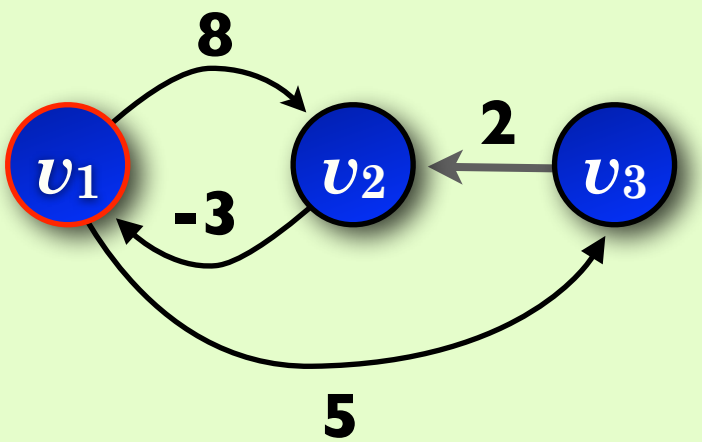
$D^{(1)}$: Umweg über $\{v_1\}$

$$d_{ij}^{(1)} = \min\left(d_{ij}^{(0)} \;,\; d_{i1}^{(0)} + d_{1j}^{(0)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(1)}$		nach		
		v_1	v_2	v_3
von	v_1	0	8	5
	v_2	-3	0	2
	v_3	∞	2	0

von v_3 nach v_2 via v_1

➡ keine Verbesserung!

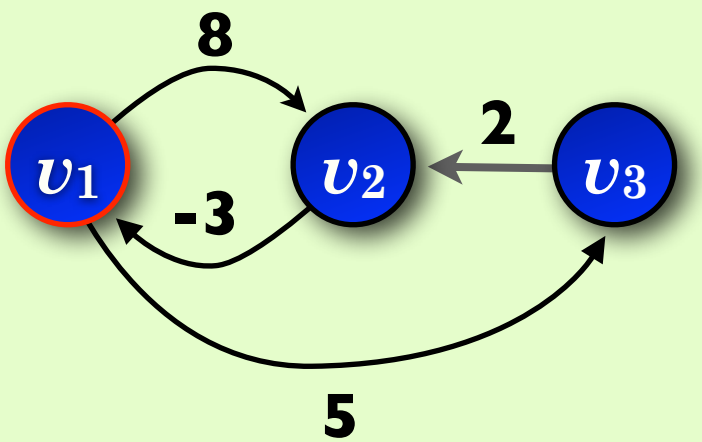
$D^{(1)}$: Umweg über $\{v_1\}$

$$d_{ij}^{(1)} = \min\left(d_{ij}^{(0)} \;,\; d_{i1}^{(0)} + d_{1j}^{(0)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(1)}$		nach		
		v_1	v_2	v_3
von	v_1	**0**	8	5
	v_2	-3	**0**	**2**
	v_3	∞	2	**0**

von v_3 nach v_3 via v_1
➡ keine Verbesserung!

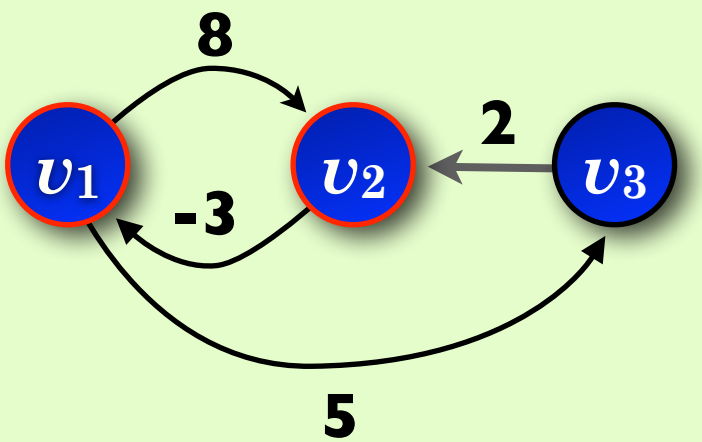
$D^{(1)}$: Umweg über $\{v_1\}$

$$d_{ij}^{(1)} = \min\left(d_{ij}^{(0)} \;,\; d_{i1}^{(0)} + d_{1j}^{(0)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(2)}$		nach		
		v_1	v_2	v_3
von	v_1	0	8	5
	v_2	-3	0	2
	v_3	∞	2	0

von v_1 nach v_1 via v_1, v_2

➡ keine Verbesserung!

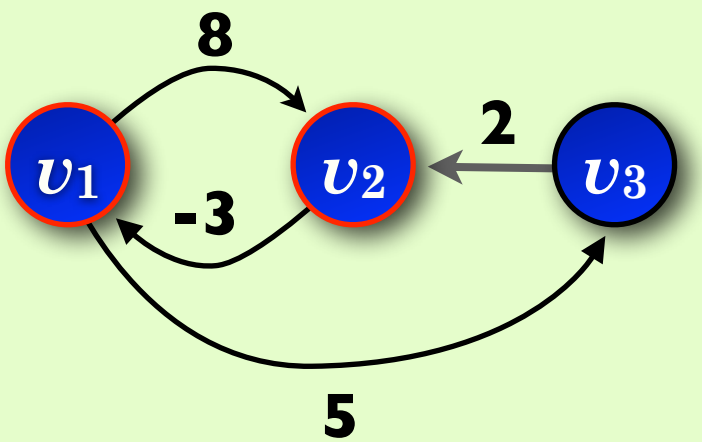
$D^{(2)}$: Umweg über $\{v_1, v_2\}$

$$d_{ij}^{(2)} = \min\left(d_{ij}^{(1)} \;,\; d_{i2}^{(1)} + d_{2j}^{(1)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(2)}$		nach		
		v_1	v_2	v_3
von	v_1	0	8	5
	v_2	-3	0	2
	v_3	∞	2	0

von v_1 nach v_2 via v_1, v_2

➡ keine Verbesserung!

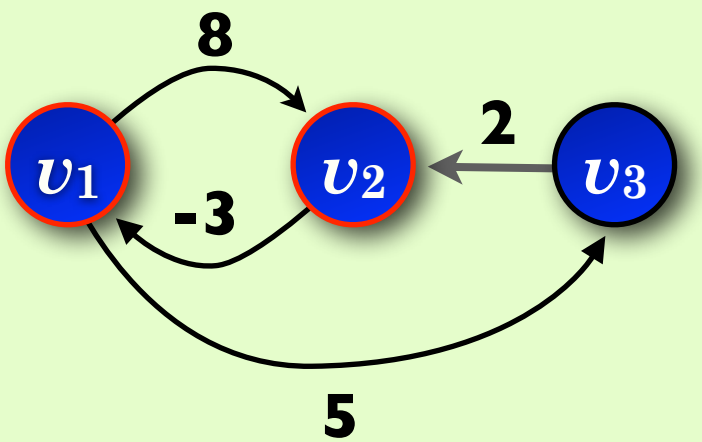
$D^{(2)}$: Umweg über $\{v_1, v_2\}$

$$d_{ij}^{(2)} = \min\left(d_{ij}^{(1)} \;,\; d_{i2}^{(1)} + d_{2j}^{(1)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(2)}$		nach		
		v_1	v_2	v_3
von	v_1	0	8	5
	v_2	-3	0	2
	v_3	∞	2	0

von v_1 nach v_3 via v_1, v_2

➡ keine Verbesserung!

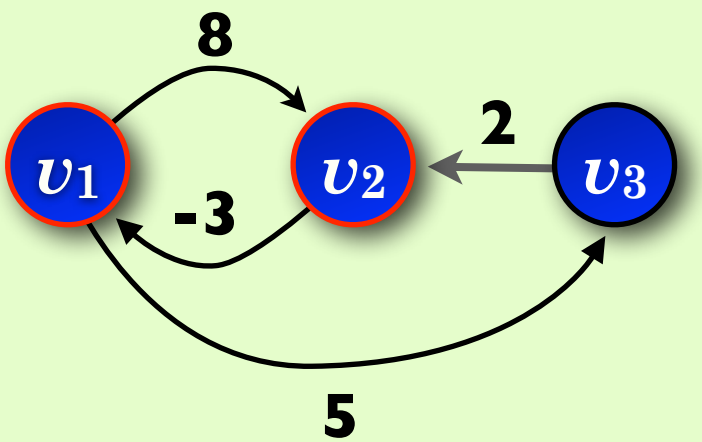
$D^{(2)}$: Umweg über $\{v_1, v_2\}$

$$d_{ij}^{(2)} = \min\left(d_{ij}^{(1)} \;,\; d_{i2}^{(1)} + d_{2j}^{(1)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(2)}$		nach		
		v_1	v_2	v_3
von	v_1	0	8	5
	v_2	-3	0	2
	v_3	∞	2	0

von v_2 nach v_1 via v_1, v_2

➡ keine Verbesserung!

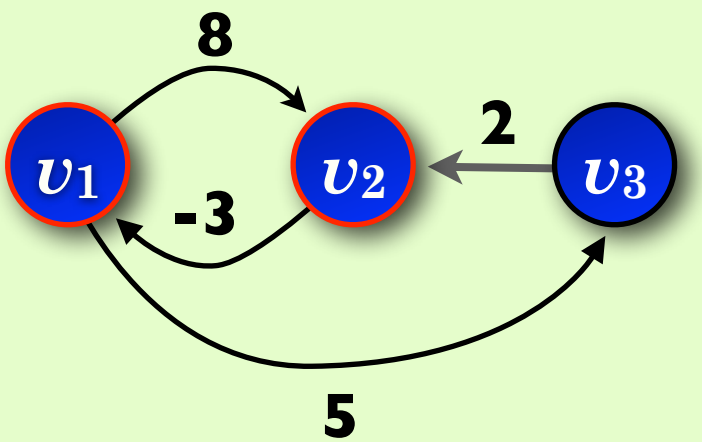
$D^{(2)}$: Umweg über $\{v_1, v_2\}$

$$d_{ij}^{(2)} = \min\left(d_{ij}^{(1)} \;,\; d_{i2}^{(1)} + d_{2j}^{(1)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(2)}$		nach		
		v_1	v_2	v_3
von	v_1	0	8	5
	v_2	-3	0	2
	v_3	∞	2	0

von v_2 nach v_3 via v_1, v_2

➡ keine Verbesserung!

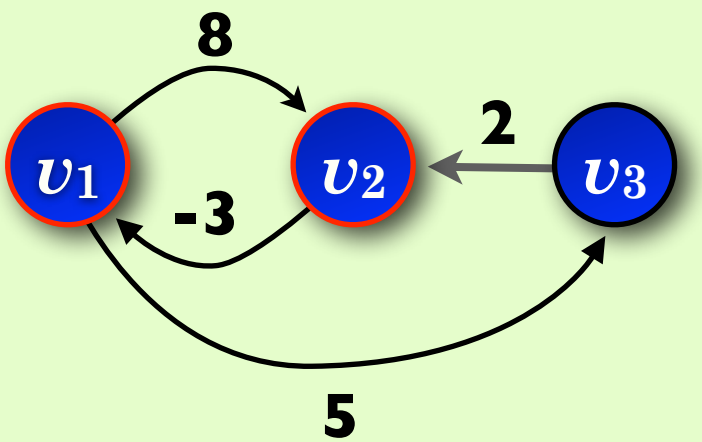
$D^{(2)}$: Umweg über $\{v_1, v_2\}$

$$d_{ij}^{(2)} = \min\left(d_{ij}^{(1)} \;,\; d_{i2}^{(1)} + d_{2j}^{(1)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(2)}$		nach		
		v_1	v_2	v_3
von	v_1	0	8	5
	v_2	-3	0	2
	v_3	∞	2	0

von v_3 nach v_1 via v_1,v_2: Kosten=-1
➡ Verbesserung von ∞

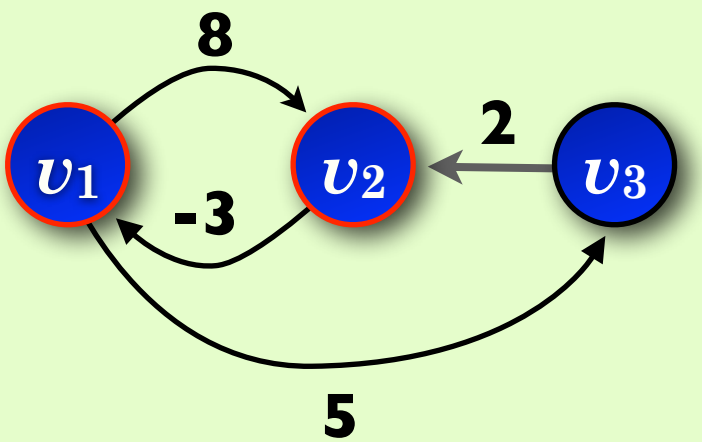
$D^{(2)}$: Umweg über $\{v_1, v_2\}$

$$d_{ij}^{(2)} = \min\left(d_{ij}^{(1)} \;,\; d_{i2}^{(1)} + d_{2j}^{(1)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(2)}$		nach		
		v_1	v_2	v_3
von	v_1	0	8	5
	v_2	-3	0	2
	v_3	-1	2	0

von v_3 nach v_1 via v_1,v_2: Kosten=-1
➡ Verbesserung von ∞

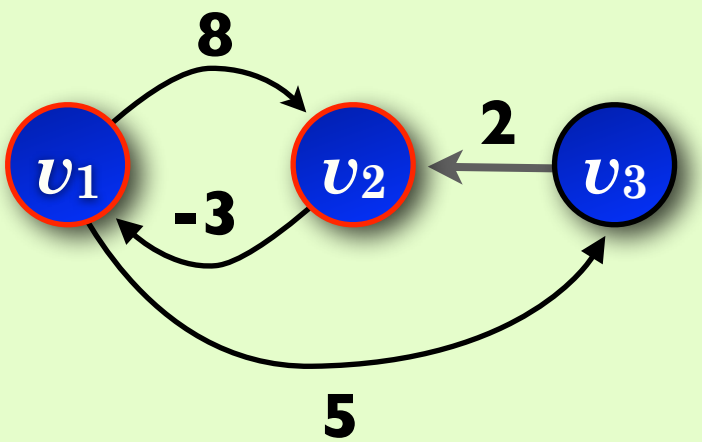
$D^{(2)}$: Umweg über $\{v_1, v_2\}$

$$d_{ij}^{(2)} = \min\left(d_{ij}^{(1)} \;,\; d_{i2}^{(1)} + d_{2j}^{(1)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(2)}$		nach		
		v_1	v_2	v_3
von	v_1	0	8	5
	v_2	-3	0	2
	v_3	-1	2	0

von v_3 nach v_2 via v_1, v_2

➡ keine Verbesserung!

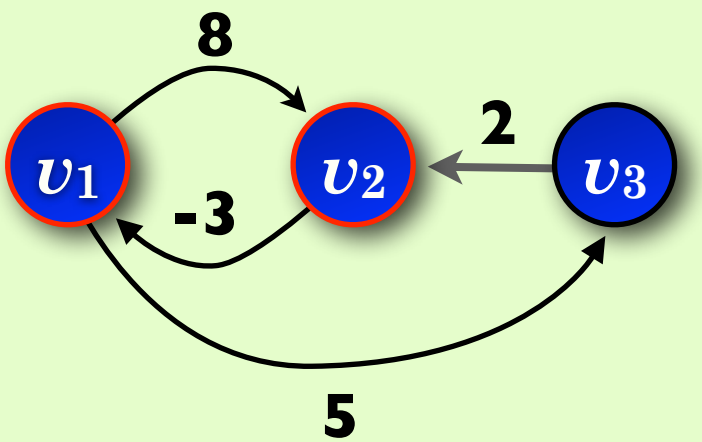
$D^{(2)}$: Umweg über $\{v_1, v_2\}$

$$d_{ij}^{(2)} = \min\left(d_{ij}^{(1)} \;,\; d_{i2}^{(1)} + d_{2j}^{(1)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(2)}$		nach		
		v_1	v_2	v_3
von	v_1	0	8	5
	v_2	-3	0	2
	v_3	-1	2	0

von v_3 nach v_3 via v_1, v_2

➡ keine Verbesserung!

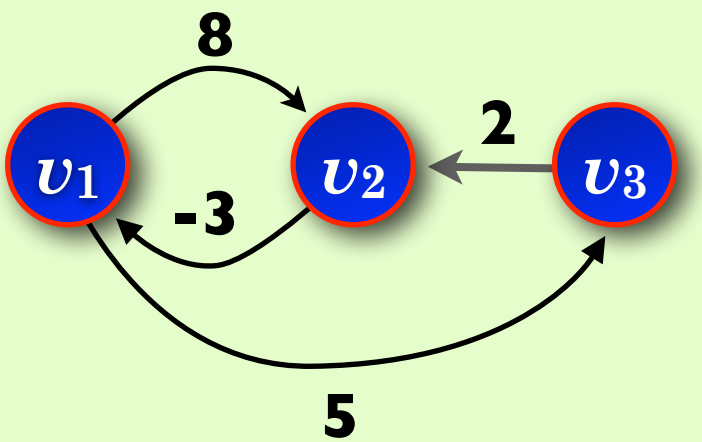
$D^{(2)}$: Umweg über $\{v_1, v_2\}$

$$d_{ij}^{(2)} = \min\left(d_{ij}^{(1)} \;,\; d_{i2}^{(1)} + d_{2j}^{(1)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(3)}$		nach		
		v_1	v_2	v_3
von	v_1	**0**	8	5
	v_2	-3	**0**	**2**
	v_3	**-1**	2	**0**

von v_1 nach v_1 via v_1, v_2, v_3

➡ keine Verbesserung!

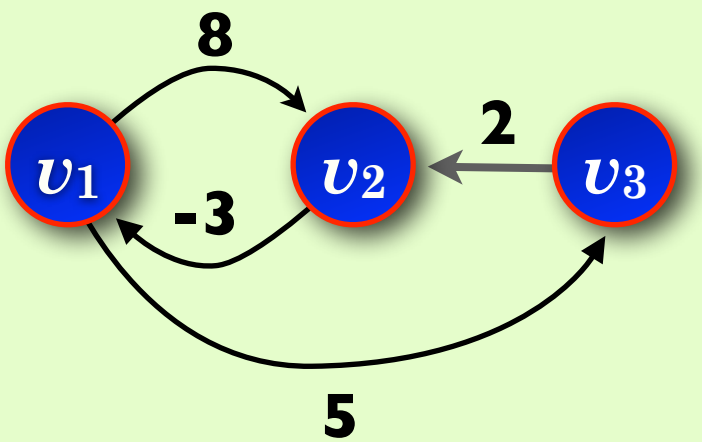
$D^{(3)}$: Umweg über $\{v_1, v_2, v_3\}$

$$d_{ij}^{(3)} = \min\left(d_{ij}^{(2)} \;,\; d_{i3}^{(2)} + d_{3j}^{(2)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(3)}$		nach		
		v_1	v_2	v_3
von	v_1	0	8	5
	v_2	-3	0	2
	v_3	-1	2	0

von v_1 nach v_2 via v_1, v_2, v_3 Kosten=7

➟ Verbesserung von 8

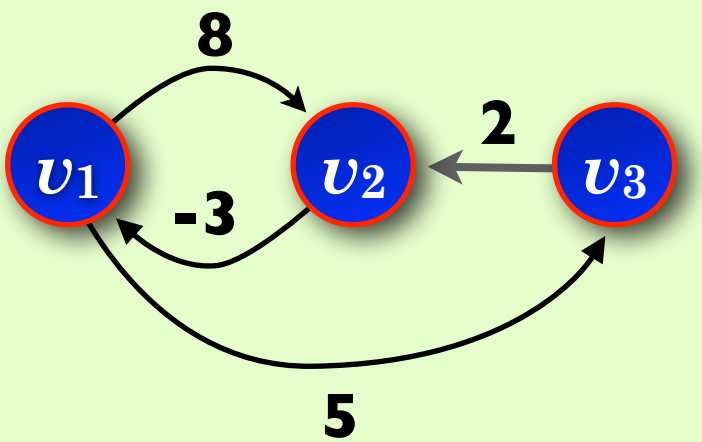
$D^{(3)}$: Umweg über $\{v_1, v_2, v_3\}$

$$d_{ij}^{(3)} = \min\left(d_{ij}^{(2)} \;,\; d_{i3}^{(2)} + d_{3j}^{(2)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(3)}$		nach		
		v_1	v_2	v_3
von	v_1	0	7	5
	v_2	-3	0	2
	v_3	-1	2	0

von v_1 nach v_2 via v_1, v_2, v_3 Kosten=7
➟ Verbesserung von 8

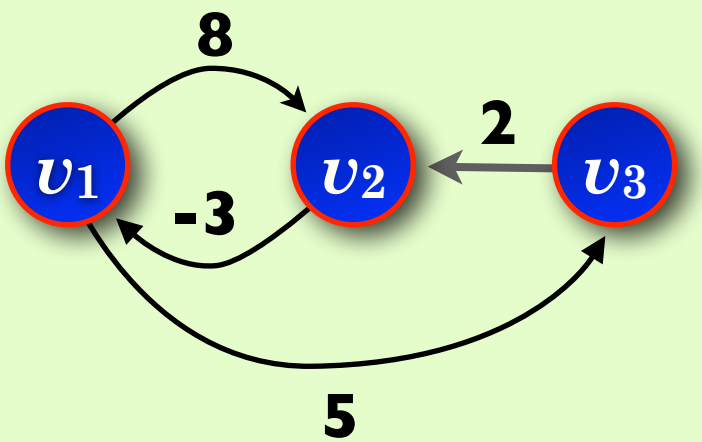
$D^{(3)}$: Umweg über $\{v_1, v_2, v_3\}$

$$d_{ij}^{(3)} = \min\left(d_{ij}^{(2)} \;,\; d_{i3}^{(2)} + d_{3j}^{(2)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(3)}$		nach		
		v_1	v_2	v_3
von	v_1	**0**	7	5
	v_2	**-3**	**0**	**2**
	v_3	**-1**	2	**0**

von v_2 nach v_1 via v_1, v_2, v_3

➡ keine Verbesserung!

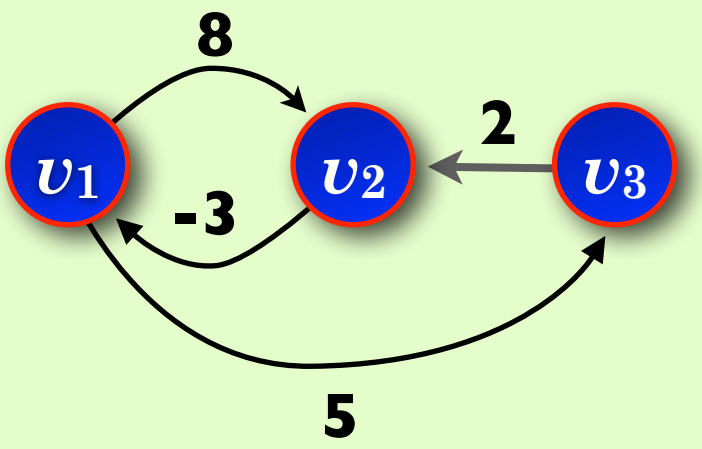
$D^{(3)}$: Umweg über $\{v_1, v_2, v_3\}$

$$d_{ij}^{(3)} = \min\left(d_{ij}^{(2)} \;,\; d_{i3}^{(2)} + d_{3j}^{(2)}\right)$$

© 2006-2024 Jörg Striegnitz

Floyd-Warshall-Algorithmus: Beispiel

B *All-Pairs Best Path*: Algorithmus von Floyd/Warshall



$D^{(3)}$		nach		
		v_1	v_2	v_3
von	v_1	0	7	5
	v_2	-3	0	2
	v_3	-1	2	0

von v_2 nach v_2 via v_1, v_2, v_3

➡ keine Verbesserung!

$D^{(3)}$: Umweg über $\{v_1, v_2, v_3\}$

$$d_{ij}^{(3)} = \min\left(d_{ij}^{(2)} \;,\; d_{i3}^{(2)} + d_{3j}^{(2)}\right)$$

© 2006-2024 Jörg Striegnitz

Erkennen von negativen Zyklen

- **Enthält ein Graph einen negativen Zyklus, so gilt am Ende des Algorithmus**

$$\exists i \in \{1, \cdots, |V|\} : \ \delta(i,i) < 0$$

- **Aber Vorsicht!**
  - Man kann zeigen, dass die Werte von $\delta^{(k)}$ sich dann während des Laufes exponentiell ändern können
  - **Es kann rasch zu einem Überlauf kommen!**

- **Daher:**
  - **Test** auf negative Zyklen immer **unmittelbar nach dem Update von $\delta_{ii}{}^{(k)}$** durchführen!

© 2006-2024 Jörg Striegnitz

Erkennen von negativen Zyklen

- **Enthält ein Graph einen negativen Zyklus, so gilt am Ende des Algorithmus**

$$\exists i \in \{1, \cdots, |V|\} : \ \delta(i,i) < 0$$

- **Aber Vorsicht!**
  - Man kann zeigen, dass die Werte von $\delta^{(k)}$ sich dann während des Laufes exponentiell ändern können
  - **Es kann rasch zu einem Überlauf kommen!**
- **Daher:**
  - **Test** auf negative Zyklen immer **unmittelbar nach dem Update von $\delta_{ii}{}^{(k)}$** durchführen!

© 2006-2024 Jörg Striegnitz

Transitive Hülle 1:

- **Manchmal interessiert nur Existenz einen Pfades von v_i nach v_j gibt** - Kosten spielen keine Rolle
  - Transitive Hülle

$$E^* = \{(v_i, v_j) \mid \text{es ex. ein Pfad von } v_i \text{ nach } v_j\}$$

  eines gerichteten Graphen

- **Einfacher Ansatz mit Laufzeitkomplexität $\Theta(n^3)$**
  - Setzte Kantengewichte 1 und wende Floyd-Warshall-Algorithmus an
  - ex. Pfad von v_i nach v_j, so ist $d_{ij}<n$, ansonsten ist $d_{ij}=\infty$

- **Optimierung**
  - Berechne boole'sches Feld $t^{(k)}_{ij}$: es existiert ein Pfad von v_i nach v_k mit inneren aus $v_1,\ldots,v_k$
  - Ersetze **min** und **+** durch logische Operatoren $\wedge$ und $\vee$

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{falls } i \neq j \text{ und } (v_i, v_j) \notin E \\ 1 & \text{falls } i = j \text{ oder } (v_i, v_j) \in E \end{cases}$$

Für $k \geq 1$:

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

© 2006-2024 Jörg Striegnitz

Transitive Hülle 2:

```
Transitive-Hülle( G=(V,E) )
    n ← |V|                                   (* Anzahl der Knoten des Graphen *)
    for k←1 to n                              (*  Initialisiere t_ij^(k)        *)
     for i←1 to n      (*  Alle Pfade von v_i ....                              *)
          for j←1 to n (*   .... nach v_j                                       *)
            t_ij^(0) ← i==j ∨ (v_i,v_j)∈E

    for k←1 to n                              (*  Umweg zusätzlich über v_k     *)
     for i←1 to n      (*  Alle Pfade von v_i ....                              *)
          for j←1 to n (*   .... nach v_j                                       *)
         (*  Liefert zusätzlicher Umweg über v_k eine Verbindung ?               *)
            t_ij^(k) ← t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
    return T^(n)
```


© 2006-2024 Jörg Striegnitz

Transitive Hülle 2:

Laufzeit: $O(|V|^3)$

```
Transitive-Hülle( G=(V,E) )
```

$n \leftarrow |V|$ (* Anzahl der Knoten des Graphen *)

for $k \leftarrow 1$ **to** n (* Initialisiere $t_{ij}^{(k)}$ *)

  for $i \leftarrow 1$ **to** n (* Alle Pfade von v_i .... *)

    for $j \leftarrow 1$ **to** n (* .... nach v_j *)

      $t_{ij}^{(0)} \leftarrow i==j \vee (v_i,v_j) \in E$

for $k \leftarrow 1$ **to** n (* Umweg zusätzlich über v_k *)

  for $i \leftarrow 1$ **to** n (* Alle Pfade von v_i .... *)

    for $j \leftarrow 1$ **to** n (* .... nach v_j *)

    (* Liefert zusätzlicher Umweg über v_k eine Verbindung ? *)

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

return $T^{(n)}$

© 2006-2024 Jörg Striegnitz

Transitive Hülle 2:

Laufzeit: $O(|V|^3)$

```
Transitive-Hülle( G=(V,E) )
```

$n \leftarrow |V|$ (* Anzahl der Knoten des Graphen *)

for $k \leftarrow 1$ **to** n (* Initialisiere $t_{ij}^{(k)}$ *)

  for $i \leftarrow 1$ **to** n (* Alle Pfade von v_i .... *)

    for $j \leftarrow 1$ **to** n (* .... nach v_j *)

      $t_{ij}^{(0)} \leftarrow i==j \vee (v_i,v_j) \in E$

for $k \leftarrow 1$ **to** n (* Umweg zusätzlich über v_k *)

  for $i \leftarrow 1$ **to** n (* Alle Pfade von v_i .... *)

    for $j \leftarrow 1$ **to** n (* .... nach v_j *)

(* Liefert zusätzlicher Umweg über v_k eine Verbindung ? *)

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

return $T^{(n)}$

Vergleiche mit Floyd-Warshall:

$$d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

Bit-Operationen auf vielen Rechnern schneller als aritmetische

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Johnson

- **1977 entdeckt von Donald Bruce Johnson**
- **Anwendbar auf alle Graphen**
  - Negative Zyklen werden erkannt
  - Falls negative Kanten: **Graph-Transformation**
- **Kombiniert**
  - Bellman-Ford-Algorithmus
    im Kontext der Graph-Transformation
  - Dijkstra-Algorithmus
    für jeden Knoten

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Johnson: Die Idee - 1

- **Keine negativen Kanten?**
  - Dann $|V|$-mal Dijkstra anwenden
- **Sonst: Transformiere Graphen G mit negativen Kanten in G' ohne**
  - **Ersetze negative Kanten geschickt durch positive.**
    „Geschickt“ meint: erhalte *predecessor subgraphs* für alle Knoten
  - **Dann: kürzeste Pfade in G' sind kürzeste Pfade in G!**

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Johnson: Die Idee - 2

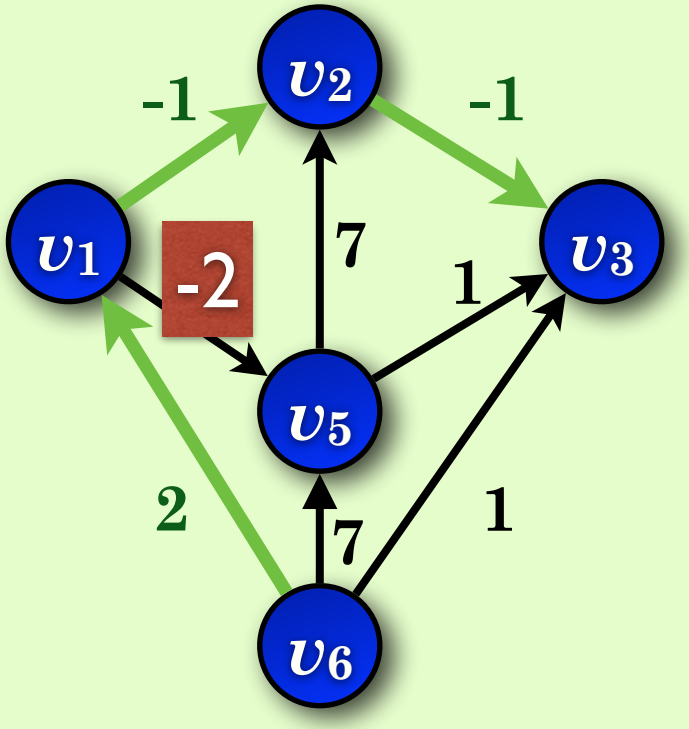
- **Was nicht geht:**
  - Finde **kleinste negative Kante** mit Gewicht min
  - Erhöhe Gewicht aller Kanten um -min

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Johnson: Die Idee - 2

- **Was nicht geht:**
  - Finde **kleinste negative Kante** mit Gewicht min
  - Erhöhe Gewicht aller Kanten um $-min$

B **Einfache Transformation funktioniert nicht**



$\delta(v_6,v_3)=0$

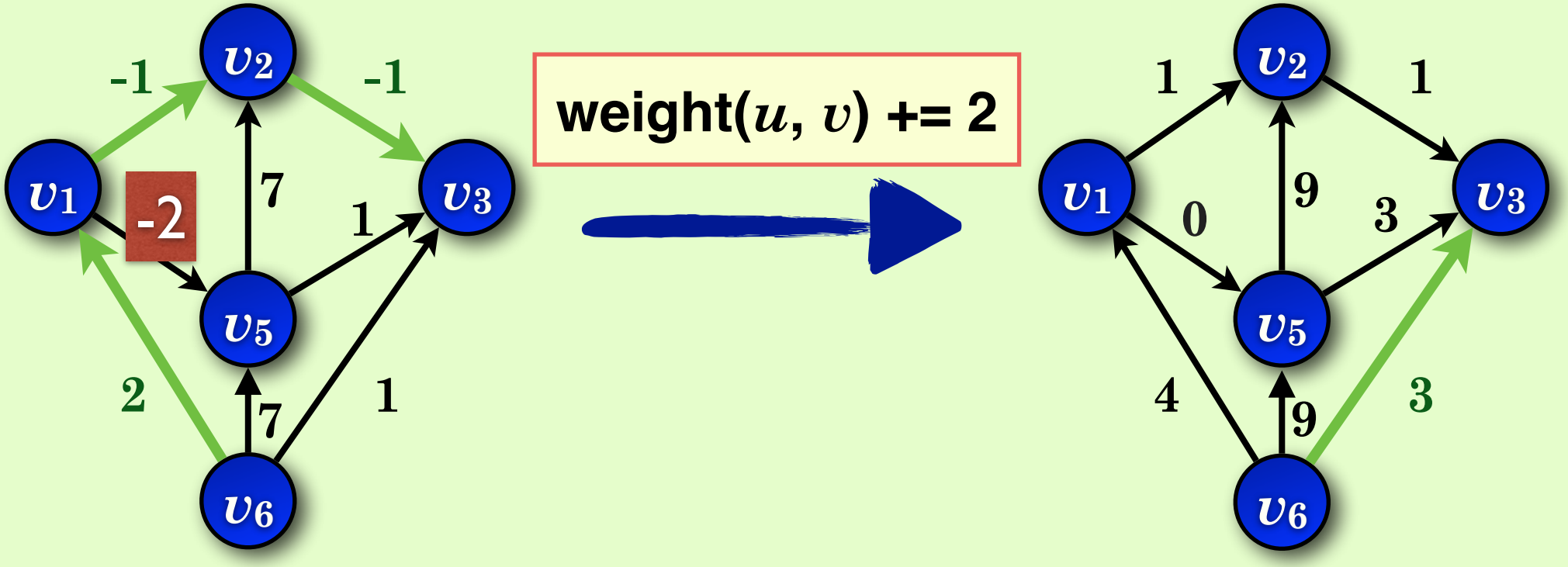
$\langle v_6,v_1,v_2,v_3\rangle$

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Johnson: Die Idee - 2

- **Was nicht geht:**
  - Finde **kleinste negative Kante** mit Gewicht min
  - Erhöhe Gewicht aller Kanten um $-min$

B **Einfache Transformation funktioniert nicht**



$\delta(v_6,v_3)=0$

$\langle v_6,v_1,v_2,v_3\rangle$

$\delta(v_6,v_3)=3$

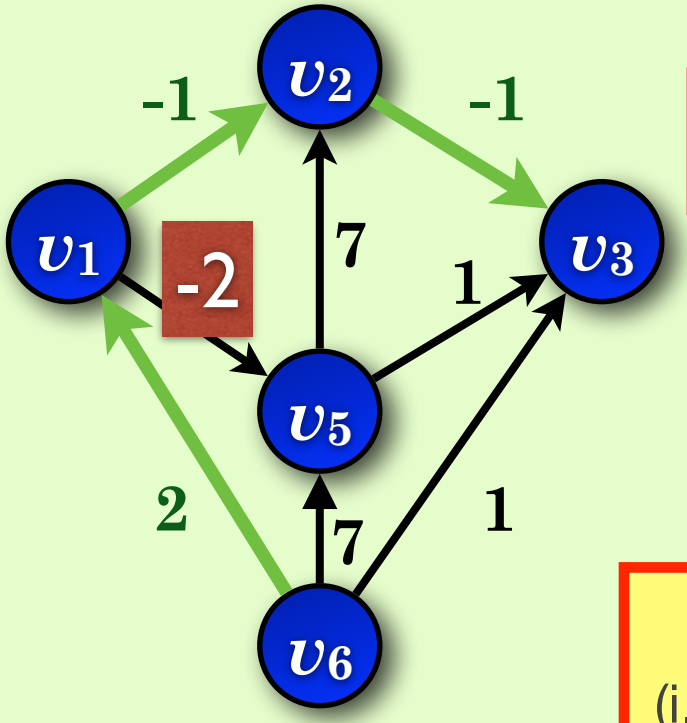
$\langle v_6,v_3\rangle$

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Johnson: Die Idee - 2

- **Was nicht geht:**
  - Finde **kleinste negative Kante** mit Gewicht min
  - Erhöhe Gewicht aller Kanten um $-min$

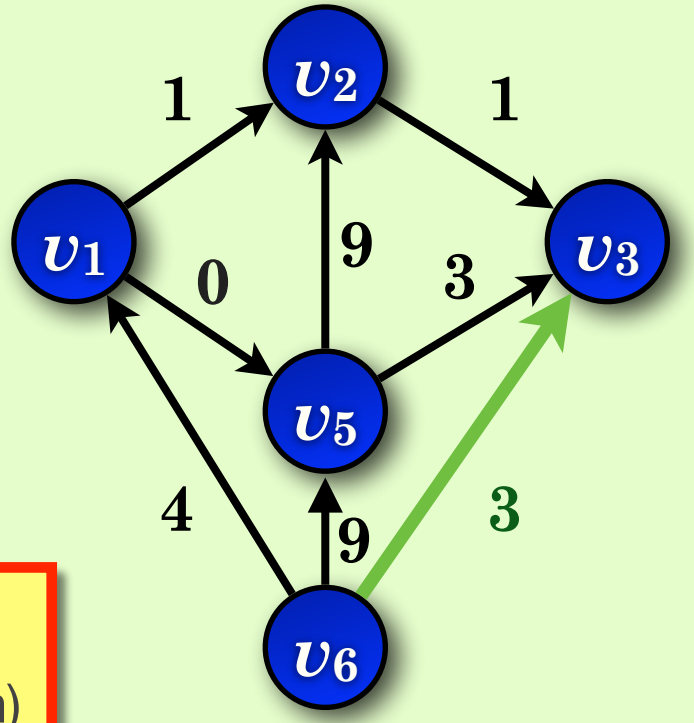
B **Einfache Transformation funktioniert nicht**



$\delta(v_6,v_3)=0$

$\langle v_6,v_1,v_2,v_3\rangle$

weight(u, v) += 2



$\delta(v_6,v_3)=3$

$\langle v_6,v_3\rangle$

Lange Pfade
(i.S.v. # durchlaufene Kanten)
werden benachteiligt

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Johnson: Die Idee - 3

- **Betrachte Knoten anstelle von Kanten**
  - **erhöhe** Gewicht aller **ausgehenden** Kanten von v um Wert x und (gleichzeitig!)
  - **erniedrige** Gewicht aller **eingehenden** Kanten von v um Wert x

- **Algorithmus**
  - Gewichtsänderung pro Knoten v mithilfe einer **Potenzialfunktion** $h(v)=x$
  - Optimale Pfade bleiben erhalten (**zu beweisen!**)
  - Wähle $h(v)$ so, dass alle Kanten am Ende positives Gewicht (**wie?**)
  - Wende Dijkstra auf jeden Knoten an

© 2006-2024 Jörg Striegnitz

Potentialfunktionen erhalten kürzeste Pfade - 1

L Potentialfunktionen erhalten kürzeste Pfade

Seien $G = (V, E)$ ein gerichteter Graph mit Gewichtsfunktion weight : $E \to \mathbb{R}$ und $h : V \to \mathbb{R}$ eine **Potenzialfunktion**. Erhalte aus h die Gewichtsfunktion $\text{weight}_h : (E \to \mathbb{R})$ wie folgt:

$$\forall (u, v) \in V \times V : \text{weight}_h(u, v) = \text{weight}(u, v) + h(u) - h(v)$$

Dann gilt: p ist ein optimaler Pfad in G mit weight genau dann, wenn p ein optimaler Pfad in G mit weight_h ist.

© 2006-2024 Jörg Striegnitz

Potentialfunktionen erhalten kürzeste Pfade - 1

L **Potentialfunktionen erhalten kürzeste Pfade**

Seien $G = (V, E)$ ein gerichteter Graph mit Gewichtsfunktion weight : $E \to \mathbb{R}$ und $h : V \to \mathbb{R}$ eine **Potenzialfunktion**. Erhalte aus h die Gewichtsfunktion $\text{weight}_h : (E \to \mathbb{R})$ wie folgt:

$$\forall (u, v) \in V \times V : \text{weight}_h(u, v) = \text{weight}(u, v) + h(u) - h(v)$$

Dann gilt: p ist ein optimaler Pfad in G mit weight genau dann, wenn p ein optimaler Pfad in G mit weight_h ist.

Beweis:

- Wir zeigen, dass für **alle** Pfade von einem Knoten s zu einem Knoten t in G die mit weight bzw. weight_h ermittelten Pfadkosten exakt um einen konstanten Wert abweichen.

  Kurz: Für alle Pfade $s \rightsquigarrow t$ gilt

  $$\text{weight}(s \rightsquigarrow t) = \text{weight}_h(s \rightsquigarrow t) + c$$

  Egal, welche Zwischenknoten besucht werden (c ist immer gleich!)

© 2006-2024 Jörg Striegnitz

Potentialfunktionen erhalten kürzeste Pfade - 2

- **Beweis**

  Sei $p = \langle v_0, v_1, \cdots, v_k \rangle$ ein beliebiger Pfad in G von v_0 nach v_k mit weight$(p) = c$. Dann ergibt sich für weight$_h(p)$:

$$
\begin{aligned}
\text{weight}_h(p) &= \sum_{i=0}^{k-1} \text{weight}_h(v_i, v_{i+1}) \\
&= \sum_{i=0}^{k-1} \text{weight}(v_i, v_{i+1}) + h(v_i) - h(v_{i+1}) \\
&= \text{weight}(p) + \sum_{i=0}^{k-1} h(v_i) - h(v_{i+1}) \\
&= \text{weight}(p) + h(v_0) - h(v_k)
\end{aligned}
$$

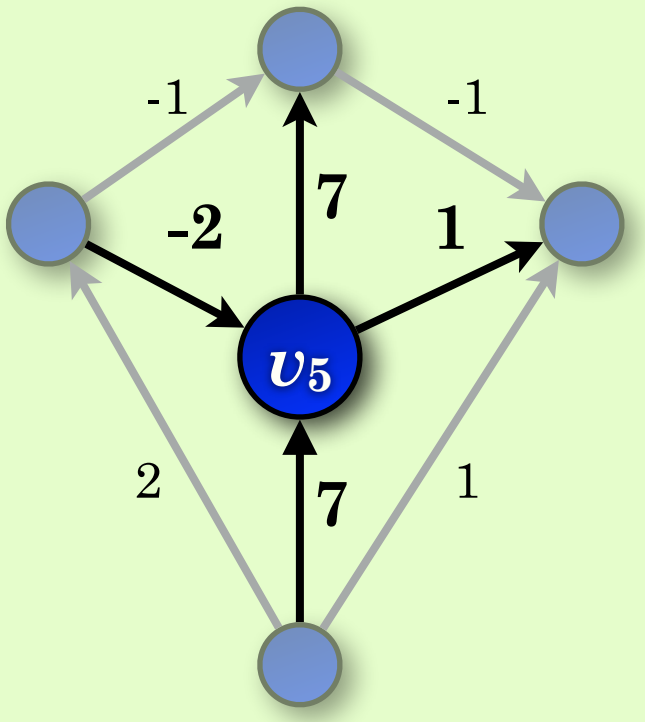
  Somit unterscheiden sich die Pfadkosten für alle Pfade von v_0 nach v_k um den Betrag $h(v_0) - h(v_k)$ und insbesondere bleibt ein kürzester Pfad ein kürzester Pfad, egal ob wir mit weight oder weight$_h$ arbeiten.

© 2006-2024 Jörg Striegnitz

Wie kommen wir zur Potenzialfunktion h?

B Wie finden wir ein geeignetes h?

Starte mit $\forall v \in V : h(v) = 0$



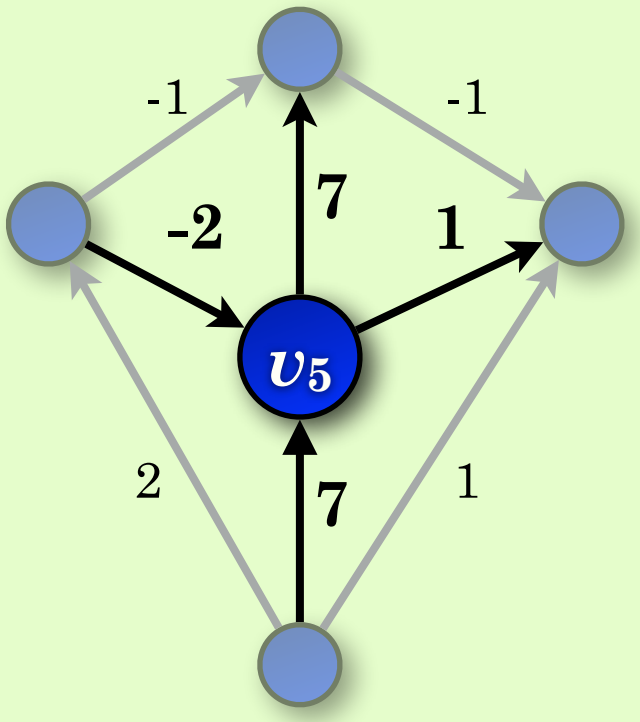
$$\forall (u,v) \in V \times V : \text{weight}_h(u,v) = \text{weight}(u,v) + h(u) - h(v)$$

© 2006-2024 Jörg Striegnitz

Wie kommen wir zur Potenzialfunktion h?

B Wie finden wir ein geeignetes h?

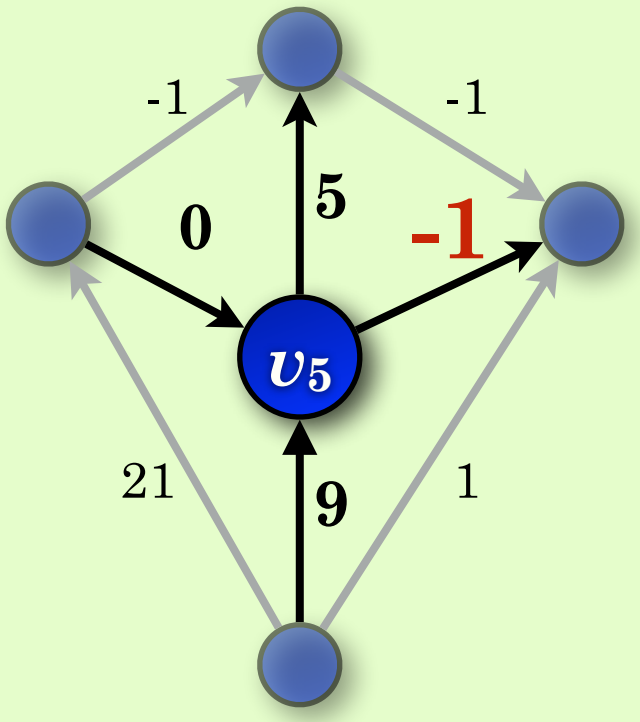
Starte mit $\forall v \in V : h(v) = 0$



$h(v_5)$ = -2

weight(u, v_5) -= -2

weight(v_5, u) += 2



$$\forall (u,v) \in V \times V : \text{weight}_h(u,v) = \text{weight}(u,v) + h(u) - h(v)$$

© 2006-2024 Jörg Striegnitz

Der Weg zu h ... - 1: *Bellman-Ford to the Rescue*

Ziel:

$$\forall (u,v) \in V \times V : \text{weight}_h(u,v) = \text{weight}(u,v) + h(u) - h(v) \geq 0$$

© 2006-2024 Jörg Striegnitz

Der Weg zu h ... - 1: *Bellman-Ford to the Rescue*

Ziel:

$$\forall (u,v) \in V \times V : \text{weight}_h(u,v) = \text{weight}(u,v) + h(u) - h(v) \geq 0$$

Stellen wir die Gleichung um:

$$\begin{aligned} & \text{weight}(u,v) + h(u) - h(v) \geq 0 \\ \Leftrightarrow \quad & h(v) \leq h(u) + \text{weight}(u,v) \end{aligned}$$

© 2006-2024 Jörg Striegnitz

Der Weg zu h ... - 1: *Bellman-Ford to the Rescue*

Ziel:

$$\forall (u,v) \in V \times V : \text{weight}_h(u,v) = \text{weight}(u,v) + h(u) - h(v) \geq 0$$

Stellen wir die Gleichung um:

$$\begin{aligned} & \text{weight}(u,v) + h(u) - h(v) \geq 0 \\ \Leftrightarrow \quad & \boxed{h(v) \leq h(u) + \text{weight}(u,v)} \end{aligned}$$

L 7.2 Dreiecksungleichung für δ

Seien $G = (V,E)$ ein gerichteter Graph mit Gewichtsfunktion weight : $E \to \mathbb{R}$ und s ein beliebiger Startknoten aus V. Dann gilt

$$\forall (u,v) \in E \times E \ : \boxed{\delta(s,v) \leq \delta(s,u) + \text{weight}(u,v)}$$

© 2006-2024 Jörg Striegnitz

Der Weg zu h ... - 1: *Bellman-Ford to the Rescue*

Ziel:

$$\forall (u,v) \in V \times V : \text{weight}_h(u,v) = \text{weight}(u,v) + h(u) - h(v) \geq 0$$

Stellen wir die Gleichung um:

$$\text{weight}(u,v) + h(u) - h(v) \geq 0$$
$$\Leftrightarrow \quad \boxed{h(v) \leq h(u) + \text{weight}(u,v)}$$

L 7.2 Dreiecksungleichung für δ

Seien $G = (V,E)$ ein gerichteter Graph mit Gewichtsfunktion weight : $E \to \mathbb{R}$ und s ein beliebiger Startknoten aus V. Dann gilt

$$\forall (u,v) \in E \times E \ : \boxed{\delta(s,v) \leq \delta(s,u) + \text{weight}(u,v)}$$

- **Idee**: Setze wir $\boldsymbol{h(v) = \delta(s,v)}$

© 2006-2024 Jörg Striegnitz

Der Weg zu h ... - 1: *Bellman-Ford to the Rescue*

Ziel:

$$\forall (u,v) \in V \times V : \text{weight}_h(u,v) = \text{weight}(u,v) + h(u) - h(v) \geq 0$$

Stellen wir die Gleichung um:

$$\text{weight}(u,v) + h(u) - h(v) \geq 0$$
$$\Leftrightarrow \quad \boxed{h(v) \leq h(u) + \text{weight}(u,v)}$$

L 7.2 Dreiecksungleichung für δ

Seien $G = (V,E)$ ein gerichteter Graph mit Gewichtsfunktion weight : $E \to \mathbb{R}$ und s ein beliebiger Startknoten aus V. Dann gilt

$$\forall (u,v) \in E \times E \ : \boxed{\delta(s,v) \leq \delta(s,u) + \text{weight}(u,v)}$$

- **Idee**: Setze wir $\boldsymbol{h(v) = \delta(s,v)}$
- δ können wir für beliebige Graphen berechnen (**Bellman-Ford!**)

© 2006-2024 Jörg Striegnitz

Der Weg zu h ... - 1: *Bellman-Ford to the Rescue*

Ziel:

$$\forall (u,v) \in V \times V : \text{weight}_h(u,v) = \text{weight}(u,v) + h(u) - h(v) \geq 0$$

Stellen wir die Gleichung um:

$$\text{weight}(u,v) + h(u) - h(v) \geq 0$$
$$\Leftrightarrow \quad \boxed{h(v) \leq h(u) + \text{weight}(u,v)}$$

L 7.2 Dreiecksungleichung für δ

Seien $G = (V,E)$ ein gerichteter Graph mit Gewichtsfunktion weight : $E \to \mathbb{R}$ und s ein beliebiger Startknoten aus V. Dann gilt

$$\forall (u,v) \in E \times E \ : \boxed{\delta(s,v) \leq \delta(s,u) + \text{weight}(u,v)}$$

- **Idee**: Setze wir $\boldsymbol{h(v) = \delta(s,v)}$
- δ können wir für beliebige Graphen berechnen (**Bellman-Ford!**)
- **Nur: woher nehmen wir den Startknoten s?**

© 2006-2024 Jörg Striegnitz

Der Weg zu h ... - 2: der Graph G_s

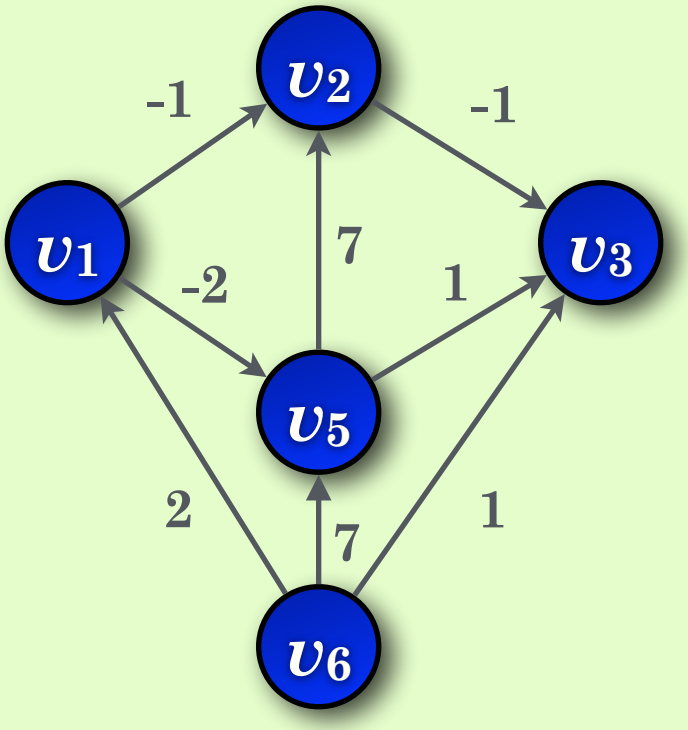
- **Führe neuen Knoten s ein**
  - s wird mit allen Knoten verbunden; Kantengewicht 0!

© 2006-2024 Jörg Striegnitz

Der Weg zu h ... - 2: der Graph G_s

- **Führe neuen Knoten s ein**
  - s wird mit allen Knoten verbunden; Kantengewicht 0!

B **Hinzunahme eines neuen „Startknotens“ - G_s**

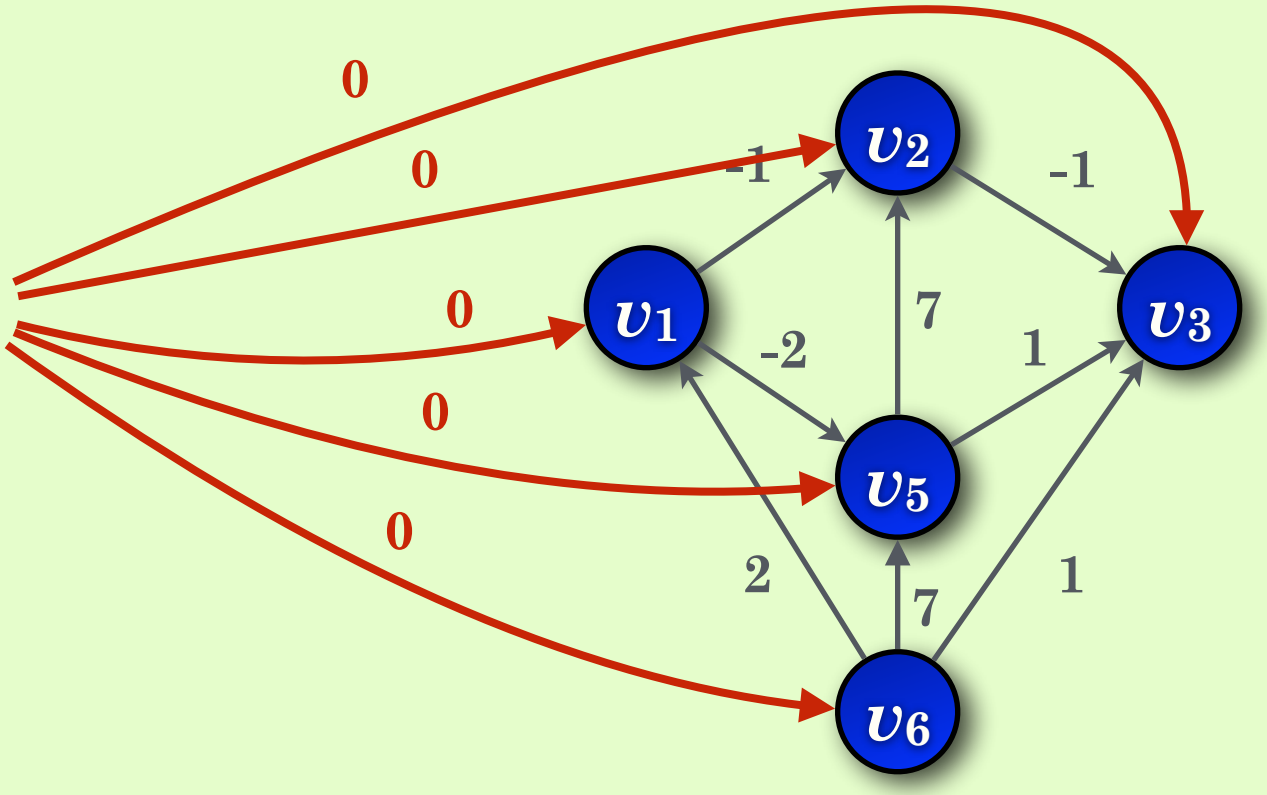


© 2006-2024 Jörg Striegnitz

Der Weg zu h … - 2: der Graph G_s

- **Führe neuen Knoten s ein**
  - s wird mit allen Knoten verbunden; Kantengewicht 0!

B Hinzunahme eines neuen „Startknotens“ - G_s

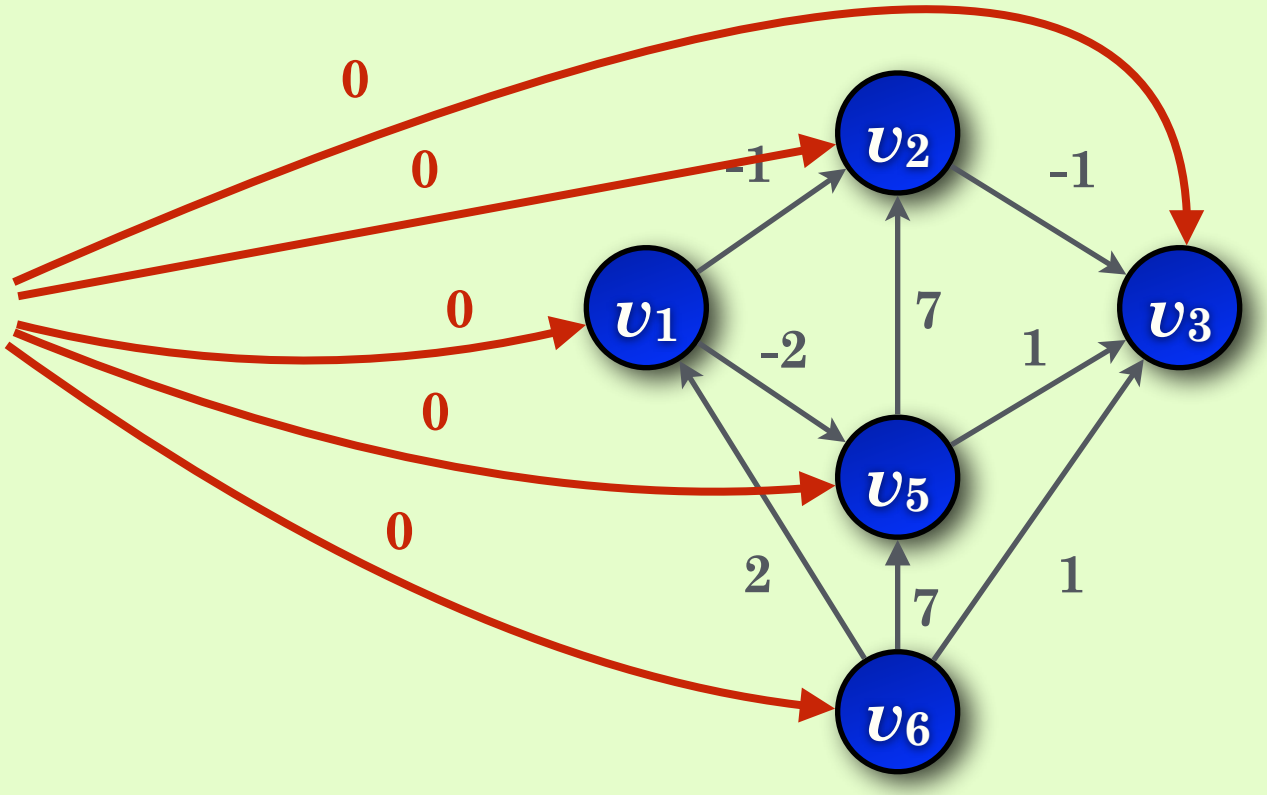


© 2006-2024 Jörg Striegnitz

Der Weg zu h … - 2: der Graph G_s

- **Führe neuen Knoten s ein**
  - s wird mit allen Knoten verbunden; Kantengewicht 0!

B **Hinzunahme eines neuen „Startknotens“ - G_s**



Beachte: da s nur ausgehende Kanten,
entsteht kein neuer Zyklus; (negative) Zyklen werden erhalten!

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Johnson

- **Erzeuge Graph G_s (neuer „Startknoten“)**
- **Berechne $\delta(s, v)$ für alle Knoten $v \in V$ mit Bellman-Ford**
- **Breche ab, wenn Bellman-Ford negativen Zyklus erkennt**
- **Sonst**
  - Berechne **weight$_h$** mithilfe von $h(v) = \delta(s, v)$
  - Berechne kürzeste Pfade für jeden Knoten mit Dijkstras Algorithmus
  - Übertrage Pfade in Originalgraph und berechne dabei Distanzen

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Johnson

- **Erzeuge Graph G_s (neuer „Startknoten“)**

  $O(|V| + |E|)$

- **Berechne $\delta(s, v)$ für alle Knoten $v \in V$ mit Bellman-Ford**

- **Breche ab, wenn Bellman-Ford negativen Zyklus erkennt**
- **Sonst**
  - Berechne **weight$_h$** mithilfe von $h(v) = \delta(s, v)$
  - Berechne kürzeste Pfade für jeden Knoten mit Dijkstras Algorithmus
  - Übertrage Pfade in Originalgraph und berechne dabei Distanzen

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Johnson

- **Erzeuge Graph G_s (neuer „Startknoten“)**

  $O(|V| + |E|)$

- **Berechne $\delta(s, v)$ für alle Knoten $v \in V$ mit Bellman-Ford**

  $O(|V| \cdot |E|)$

- **Breche ab, wenn Bellman-Ford negativen Zyklus erkennt**
- **Sonst**
  - Berechne **weight$_h$** mithilfe von $h(v) = \delta(s, v)$
  - Berechne kürzeste Pfade für jeden Knoten mit Dijkstras Algorithmus
  - Übertrage Pfade in Originalgraph und berechne dabei Distanzen

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Johnson

- **Erzeuge Graph G_s (neuer „Startknoten“)**

  $O(|V| + |E|)$

- **Berechne $\delta(s, v)$ für alle Knoten $v \in V$ mit Bellman-Ford**

  $O(|V| \cdot |E|)$

- **Breche ab, wenn Bellman-Ford negativen Zyklus erkennt**
- **Sonst**
  - Berechne **weight$_h$** mithilfe von $h(v) = \delta(s, v)$

    $O(|V| + |E|)$

  - Berechne kürzeste Pfade für jeden Knoten mit Dijkstras Algorithmus
  - Übertrage Pfade in Originalgraph und berechne dabei Distanzen

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Johnson

- **Erzeuge Graph G_s (neuer „Startknoten“)**

  $O(|V| + |E|)$

- **Berechne $\delta(s, v)$ für alle Knoten $v \in V$ mit Bellman-Ford**

  $O(|V| \cdot |E|)$

- **Breche ab, wenn Bellman-Ford negativen Zyklus erkennt**
- **Sonst**
  - Berechne **weight$_h$** mithilfe von $h(v) = \delta(s, v)$

    $O(|V| + |E|)$

  - Berechne kürzeste Pfade für jeden Knoten mit Dijkstras Algorithmus

    $O(|V| \cdot (|V| \cdot \log|V| + |E|))$

  - Übertrage Pfade in Originalgraph und berechne dabei Distanzen

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Johnson

- **Erzeuge Graph G_s (neuer „Startknoten“)**

  $O(|V| + |E|)$

- **Berechne $\delta(s, v)$ für alle Knoten $v \in V$ mit Bellman-Ford**

  $O(|V| \cdot |E|)$

- **Breche ab, wenn Bellman-Ford negativen Zyklus erkennt**
- **Sonst**
  - Berechne **weight$_h$** mithilfe von $h(v) = \delta(s, v)$

    $O(|V| + |E|)$

  - Berechne kürzeste Pfade für jeden Knoten mit Dijkstras Algorithmus

    $O(|V| \cdot (|V| \cdot \log|V| + |E|))$

  - Übertrage Pfade in Originalgraph und berechne dabei Distanzen

    $O(|V|^2)$

© 2006-2024 Jörg Striegnitz

Der Algorithmus von Johnson

- **Erzeuge Graph G_s (neuer „Startknoten“)**

  $O(|V| + |E|)$

- **Berechne $\delta(s, v)$ für alle Knoten $v \in V$ mit Bellman-Ford**

  $O(|V| \cdot |E|)$

- **Breche ab, wenn Bellman-Ford negativen Zyklus erkennt**
- **Sonst**
  - Berechne **weight$_h$** mithilfe von $h(v) = \delta(s, v)$

    $O(|V| + |E|)$

  - Berechne kürzeste Pfade für jeden Knoten mit Dijkstras Algorithmus

    $O(|V| \cdot (|V| \cdot \log|V| + |E|))$

  - Übertrage Pfade in Originalgraph und berechne dabei Distanzen

    $O(|V|^2)$

Insgesamt: $O(|V|^2 \cdot \log|V| + |V| \cdot |E|)$

-> besser als Floyd-Warshall

© 2006-2024 Jörg Striegnitz

VII. Graphen

7. Graphen

7.1. Einführung und Begriffsbestimmungen
7.2. Traversierung
7.3. Topologisches Sortieren
7.4. Kürzeste Wege
7.4.1. Einführung
7.4.2. *Single-Source Best Path* Algorithmen
7.4.3. *All-Pairs Best Path* Algorithmen
7.5. Maximaler Fluss
7.6. Minimaler Spannbaum

© 2006-2023 Jörg Striegnitz

Fluss-Netzwerke

- **Interpretiere gewichteten Graph als Fluss-Netzwerk**
  - Material-Fluss auf Fließbändern bestimmter Kapazität
  - Pipelines unterschiedlicher Dicke
  - Kantengewicht entspricht maximaler Kapazität
- **Quelle und Senke**
- **Fragestellung**

Maximal möglicher Fluss von Quelle zu Senke

© 2006-2024 Jörg Striegnitz

Beispiel: Flussgraph

B *Flussgraph*

Produktionsstrasse

- Warenanlieferung bei s
- Auslieferung bei d
- Verabeitungsstationen v_1,...,v_4
- Kapazität der Förderbänder

© 2006-2024 Jörg Striegnitz

Beispiel: Flussgraph

B *Flussgraph*

Produktionsstrasse

- Warenanlieferung bei s
- Auslieferung bei d
- Verabeitungsstationen v_1,...,v_4
- Kapazität der Förderbänder

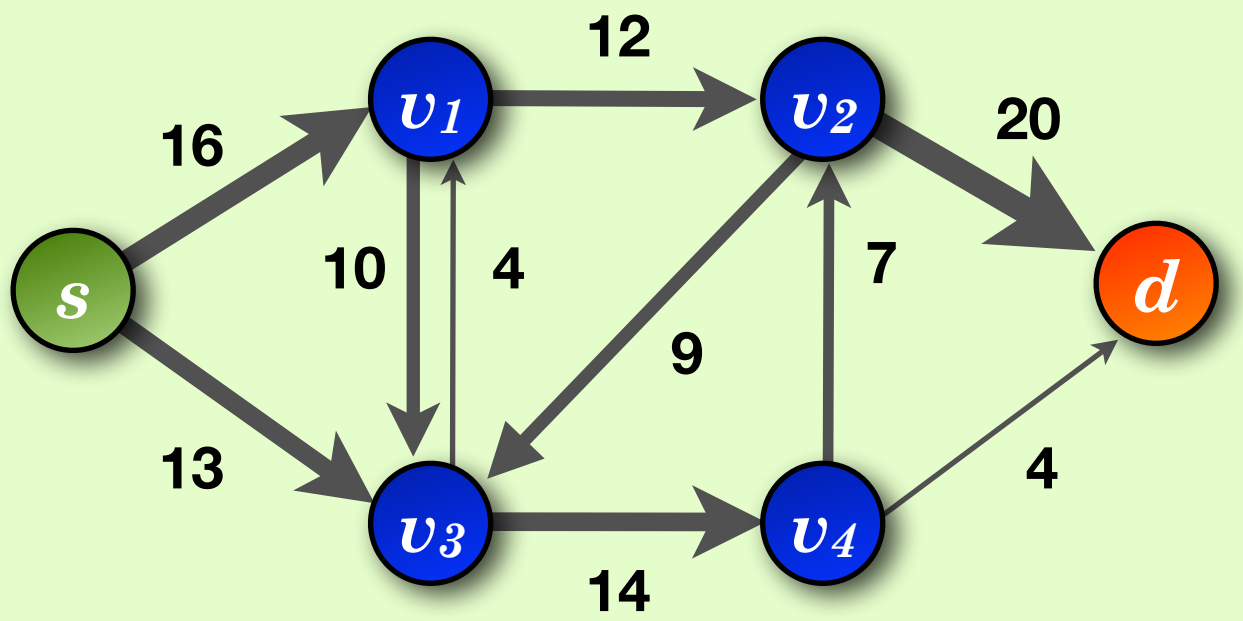
© 2006-2024 Jörg Striegnitz

Beispiel: Flussgraph

B *Flussgraph*

Produktionsstrasse

- Warenanlieferung bei s
- Auslieferung bei d
- Verabeitungsstationen v_1,...,v_4
- Kapazität der Förderbänder



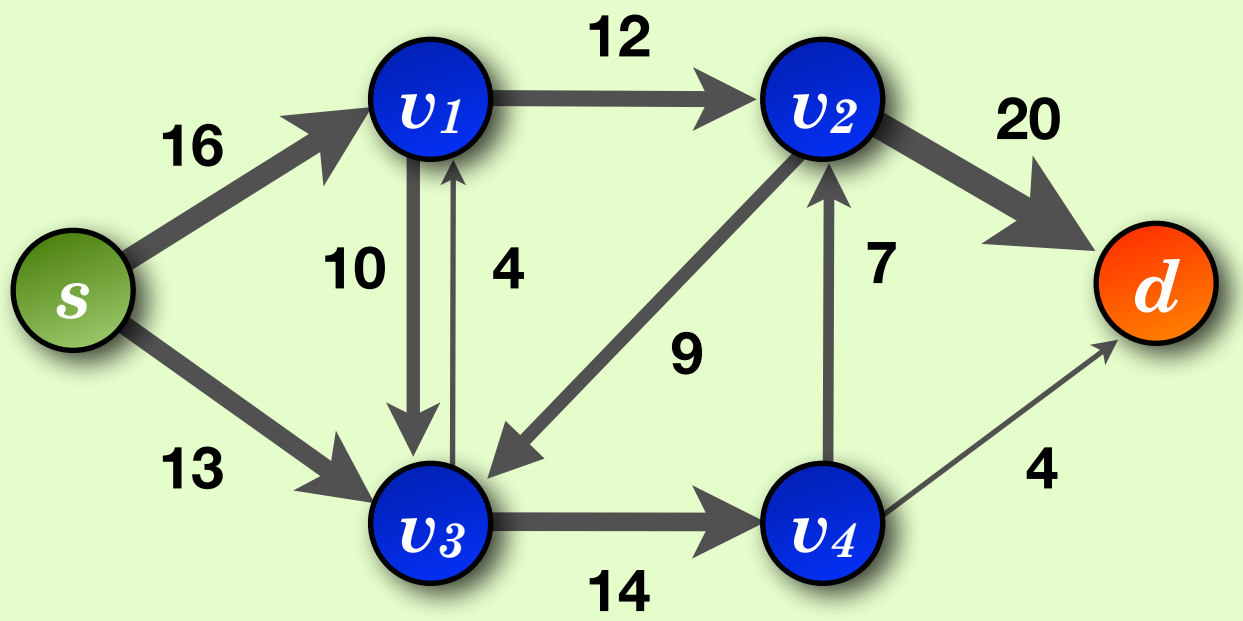
© 2006-2024 Jörg Striegnitz

Beispiel: Flussgraph

B *Flussgraph*

Produktionsstrasse

- Warenanlieferung bei s
- Auslieferung bei d
- Verabeitungsstationen v_1,...,v_4
- Kapazität der Förderbänder



Wie viel Ware darf maximal angenommen werden?

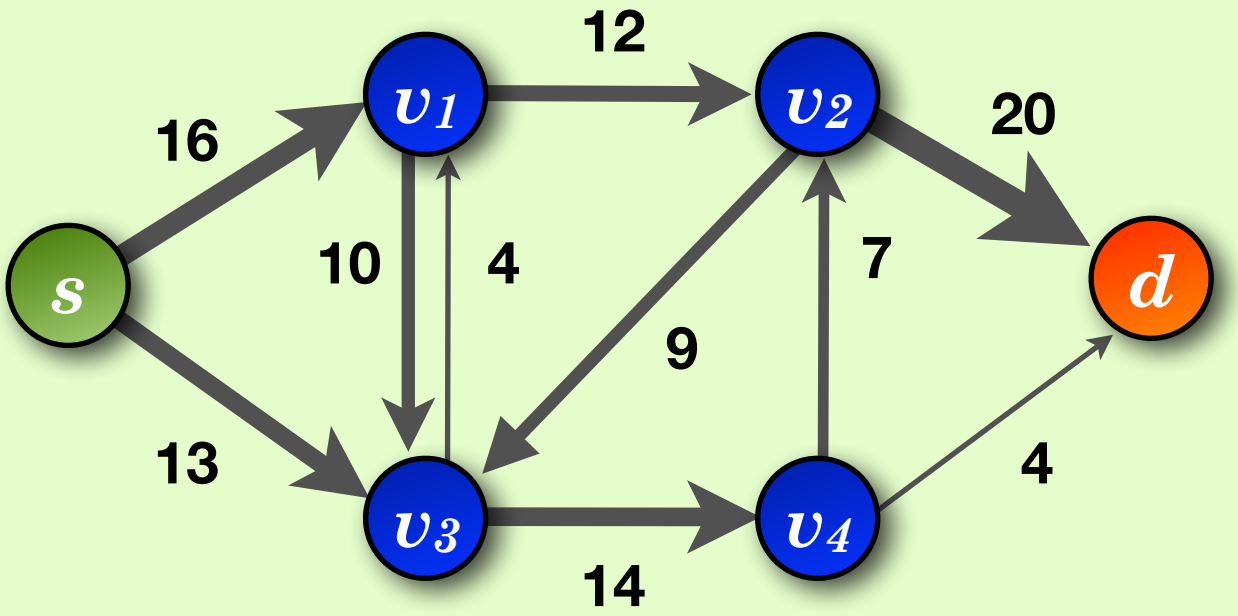
© 2006-2024 Jörg Striegnitz

Beispiel: Flussgraph

B *Flussgraph und Fluss*

Produktionsstrasse

- Warenanlieferung bei s
- Auslieferung bei d
- Verabeitungsstationen v_1,...,v_4
- Kapazität der Förderbänder



© 2006-2024 Jörg Striegnitz

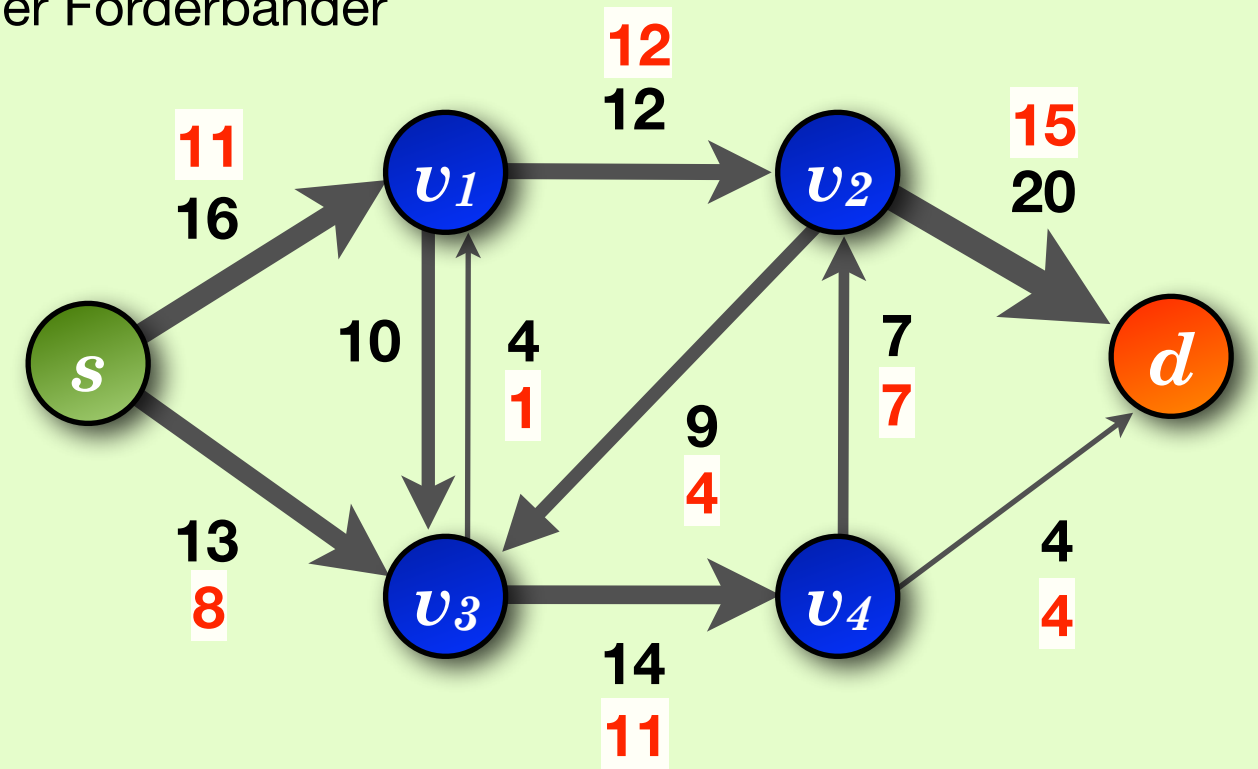
Beispiel: Flussgraph

B *Flussgraph und Fluss*

Produktionsstrasse

- Warenanlieferung bei s
- Auslieferung bei d
- Verabeitungsstationen $v_1,\ldots,v_4$
- Kapazität der Förderbänder

19 Einheiten ??



© 2006-2024 Jörg Striegnitz

D Flussnetzwerk

Ein **Flussnetzwerk** ist ein Quadrupel $F = (G, c, s, d)$ aus einem gerichteten Graphen $G = (V, E)$ zusammen mit

- einem Quellknoten $s \in V$
- einem Zielknoten $d \in V$
- und einer Kapazitätsfunktion $c : V \times V \to \mathbb{R}$, für die gilt:

$$\begin{array}{ll} c(u,v) \geq 0 & \text{falls } (u,v) \in E \\ c(u,v) = 0 & \text{sonst} \end{array}$$

© 2006-2024 Jörg Striegnitz

D Flussnetzwerk

Ein **Flussnetzwerk** ist ein Quadrupel $F = (G, c, s, d)$ aus einem gerichteten Graphen $G = (V, E)$ zusammen mit

- einem Quellknoten $s \in V$
- einem Zielknoten $d \in V$
- und einer Kapazitätsfunktion $c : V \times V \to \mathbb{R}$, für die gilt:

$$\begin{array}{ll} c(u,v) \geq 0 & \text{falls } (u,v) \in E \\ c(u,v) = 0 & \text{sonst} \end{array}$$

c ist totale Funktion!

© 2006-2024 Jörg Striegnitz

D Flussnetzwerk

Ein **Flussnetzwerk** ist ein Quadrupel $F = (G, c, s, d)$ aus einem gerichteten Graphen $G = (V, E)$ zusammen mit

- einem Quellknoten $s \in V$
- einem Zielknoten $d \in V$
- und einer Kapazitätsfunktion $c : V \times V \to \mathbb{R}$, für die gilt:

$$\begin{array}{ll} c(u,v) \geq 0 & \text{falls } (u,v) \in E \\ c(u,v) = 0 & \text{sonst} \end{array}$$

c ist totale Funktion!

- **Vereinfachend gehen wir davon aus, dass**
  - jeder Knoten vom Quellknoten aus erreichbar ist
  - der Zielknoten von jedem Knoten erreichbar ist

© 2006-2024 Jörg Striegnitz

Theoretische Modellierung - Fluss

D Fluss

Sei $F = (G, c, s, d)$ ein Flussnetzwerk. Ein **Fluss** ist eine Funktion $f : V \times V \to \mathbb{R}$, welche die folgenden Eigenschaften erfüllt

1. **Einhaltung der Kapazität**: $\forall (u, v) \in V \times V : f(u, v) \leq c(u, v)$
2. **Flusserhaltung**: $\forall u \in V - \{s, d\} : \sum_{v \in V} f(u, v) = 0$
3. **Schiefsymmetrisch**: $f(u, v) = -f(v, u)$

$f(u, v)$ bezeichnet dann den Fluss vom Knoten u zum Knoten v und der Wert eines Flusses $|f|$ ist definiert durch

$$|f| = \sum_{v \in V} f(s, v)$$

($|f|$ ist demnach der Gesamtfluss, ausgehend von der Quelle s).

© 2006-2024 Jörg Striegnitz

Theoretische Modellierung - Fluss

D Fluss

Sei $F = (G, c, s, d)$ ein Flussnetzwerk. Ein **Fluss** ist eine Funktion $f : V \times V \to \mathbb{R}$, welche die folgenden Eigenschaften erfüllt

1. **Einhaltung der Kapazität**: $\forall (u, v) \in V \times V : f(u, v) \leq c(u, v)$
2. **Flusserhaltung**: $\forall u \in V - \{s, d\} : \sum_{v \in V} f(u, v) = 0$
3. **Schiefsymmetrisch**: $f(u, v) = -f(v, u)$

$f(u, v)$ bezeichnet dann den Fluss vom Knoten u zum Knoten v und der Wert eines Flusses $|f|$ ist definiert durch

$$|f| = \sum_{v \in V} f(s, v)$$

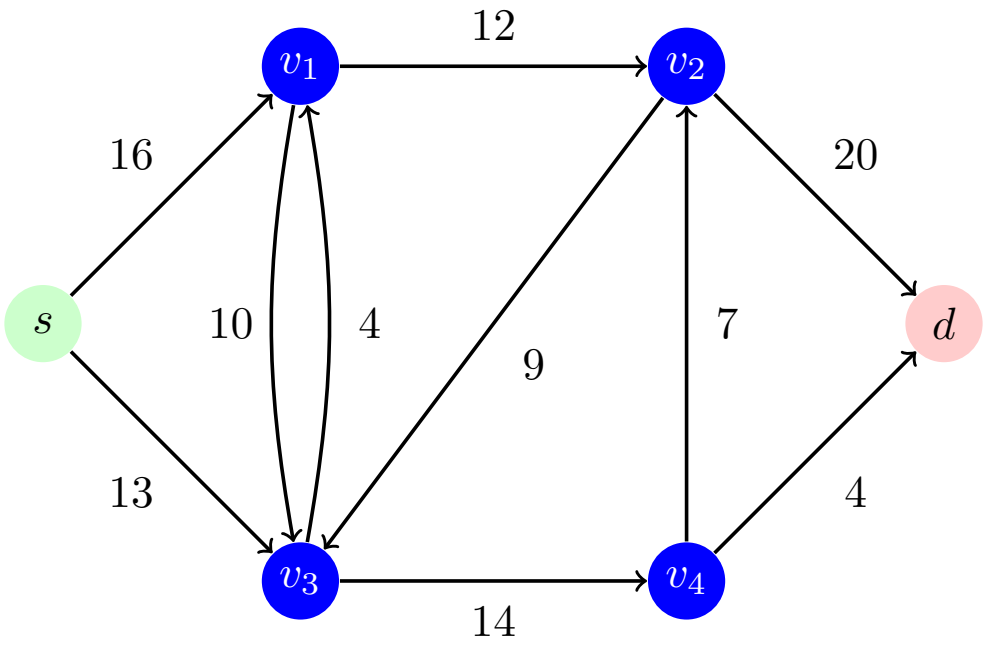
($|f|$ ist demnach der Gesamtfluss, ausgehend von der Quelle s).

- **Problem des maximalen Flusses**
  - finde Fluss maximalen Betrags

© 2006-2024 Jörg Striegnitz

Ford-Fulkerson: Kapazitätsfunktion

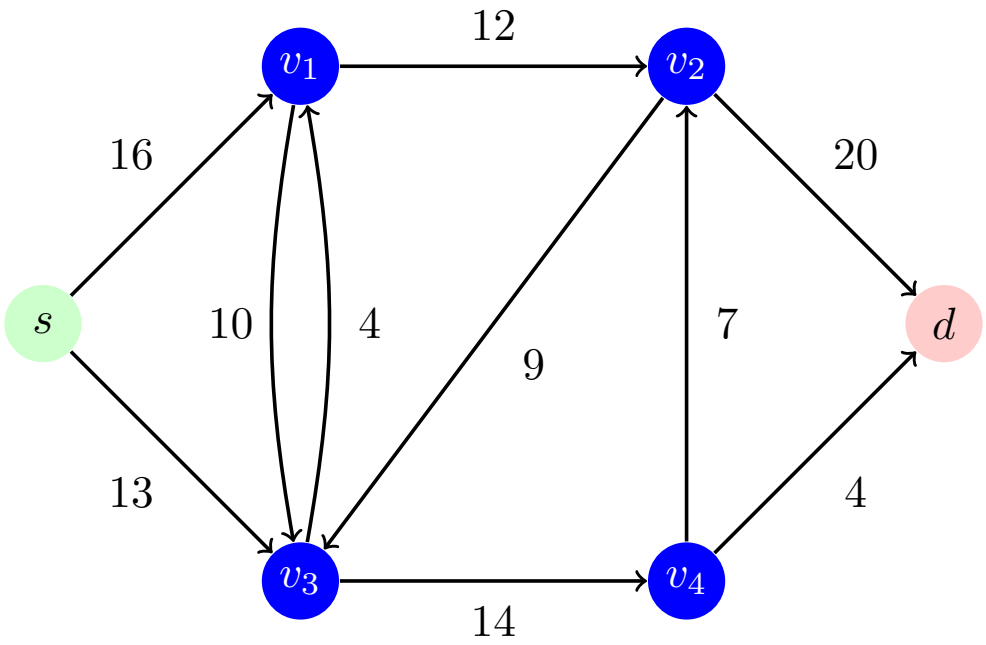
Ausgangsgraph



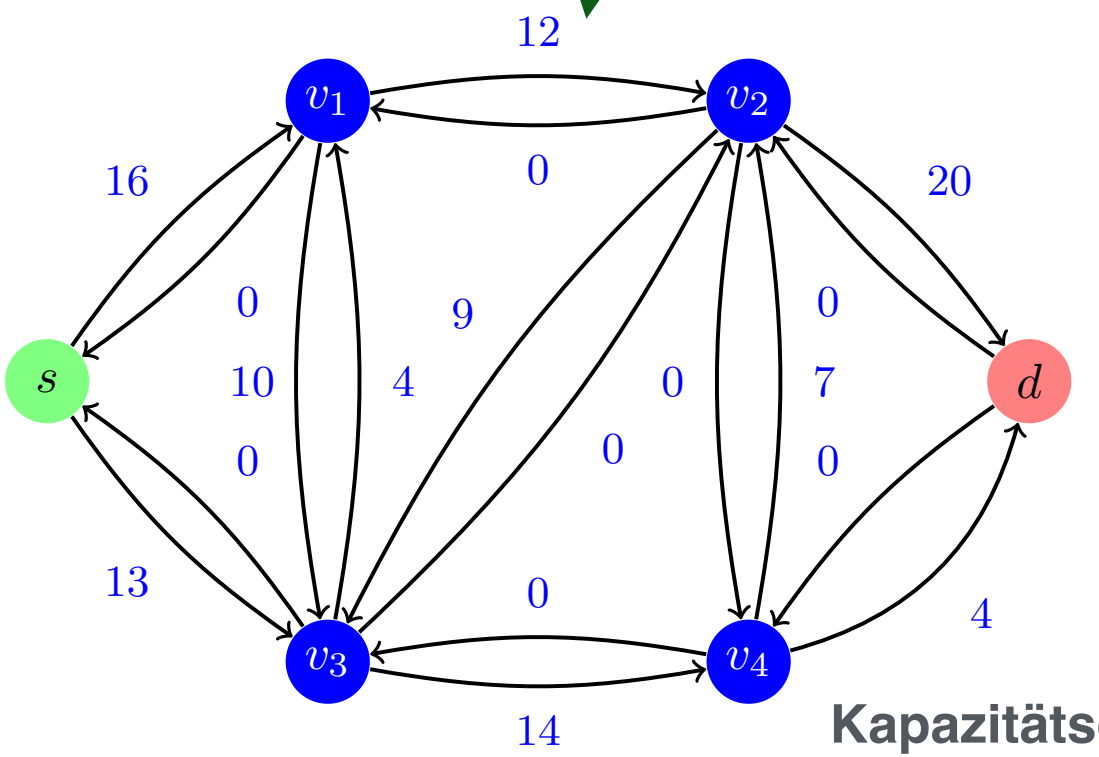
© 2006-2024 Jörg Striegnitz

Ford-Fulkerson: Kapazitätsfunktion

Ausgangsgraph



erweitere w zu $\boldsymbol{c}$
(Schleifen wurden ausgelassen)

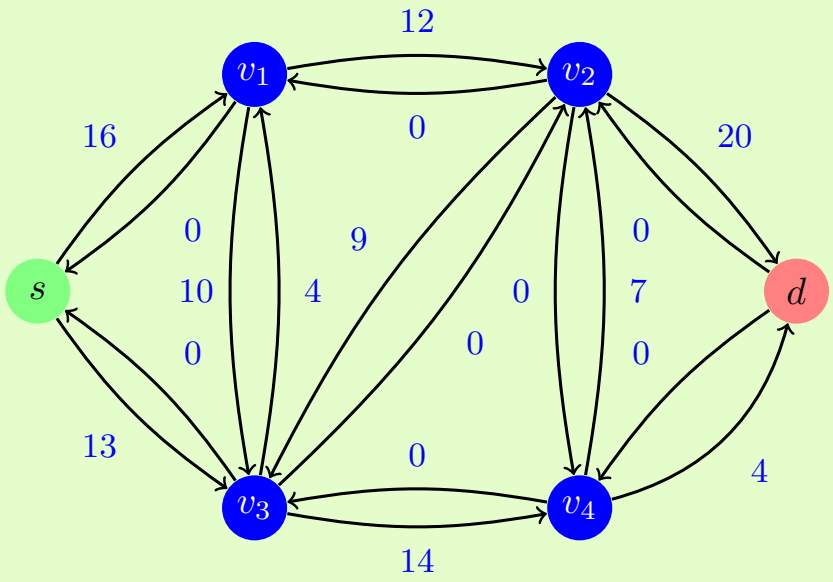


Kapazitätsgraph

© 2006-2024 Jörg Striegnitz

Ford-Fulkerson: Das Residualnetzwerk

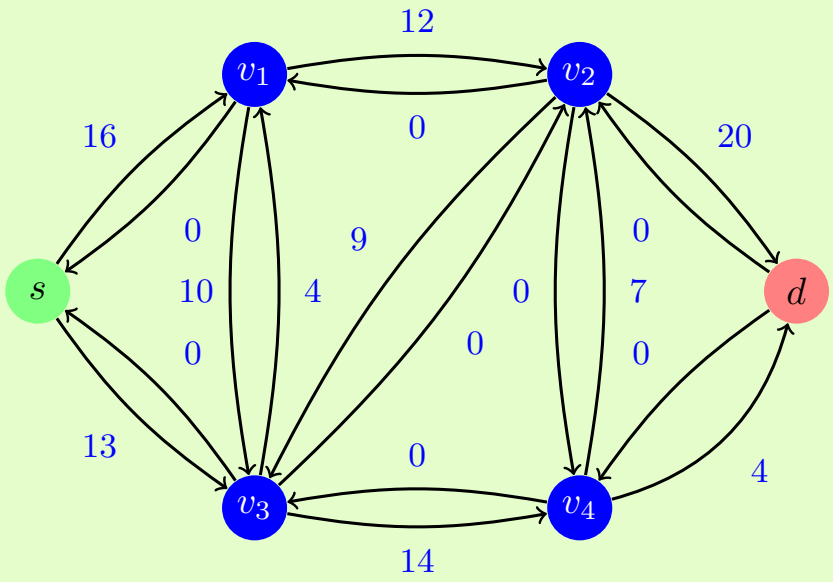
Kapazität c



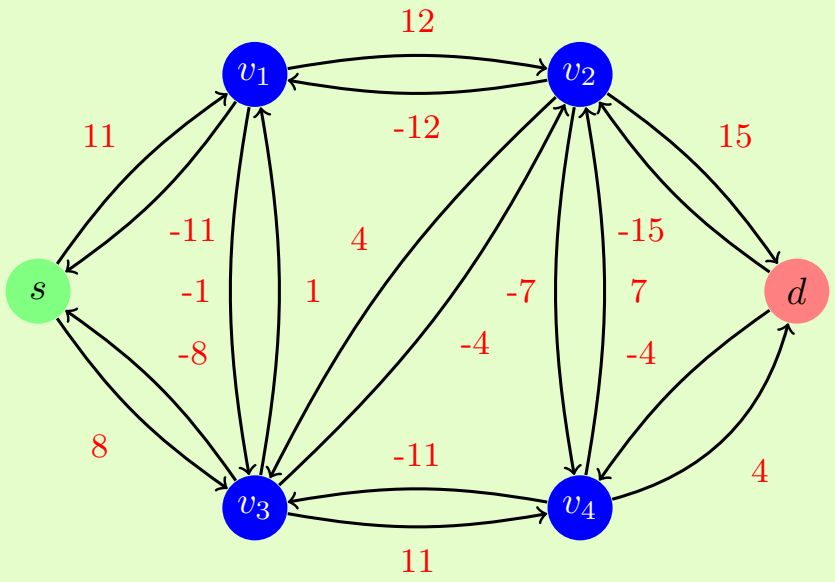
© 2006-2024 Jörg Striegnitz

Ford-Fulkerson: Das Residualnetzwerk

Kapazität c



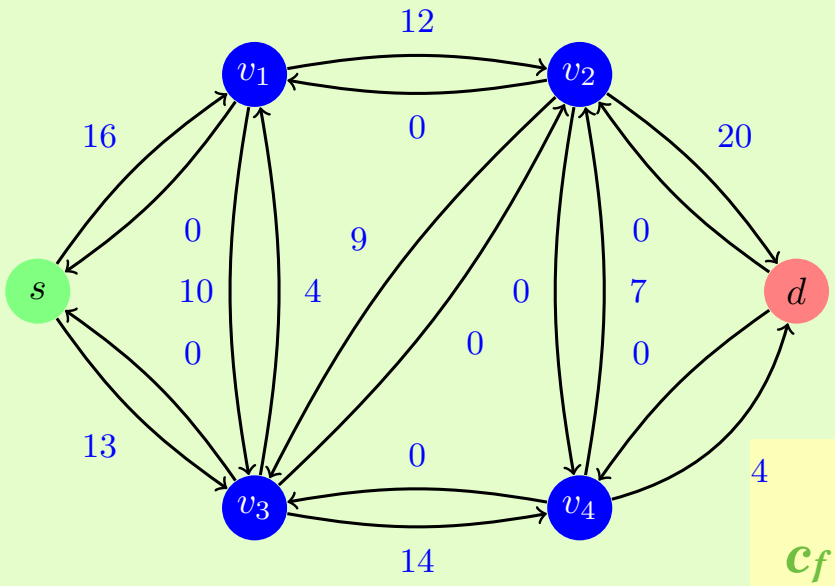
Fluss f



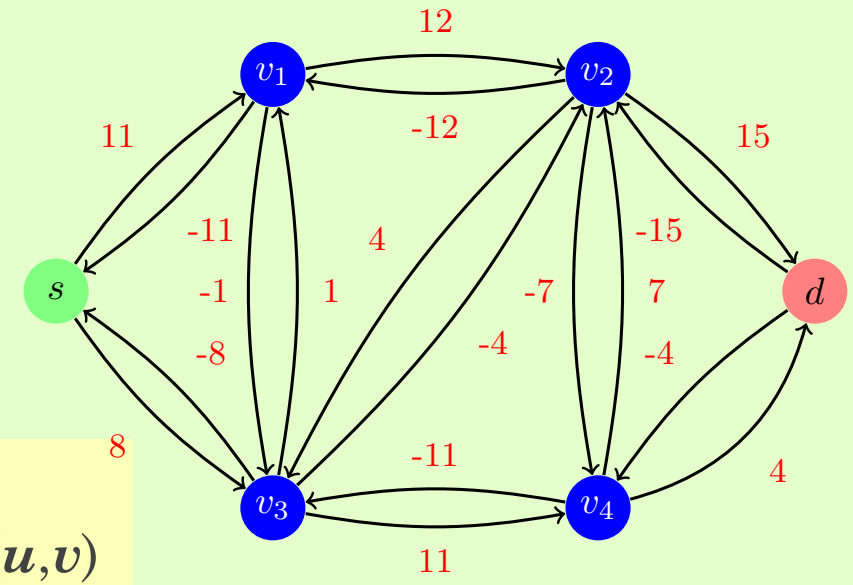
© 2006-2024 Jörg Striegnitz

Ford-Fulkerson: Das Residualnetzwerk

Kapazität c

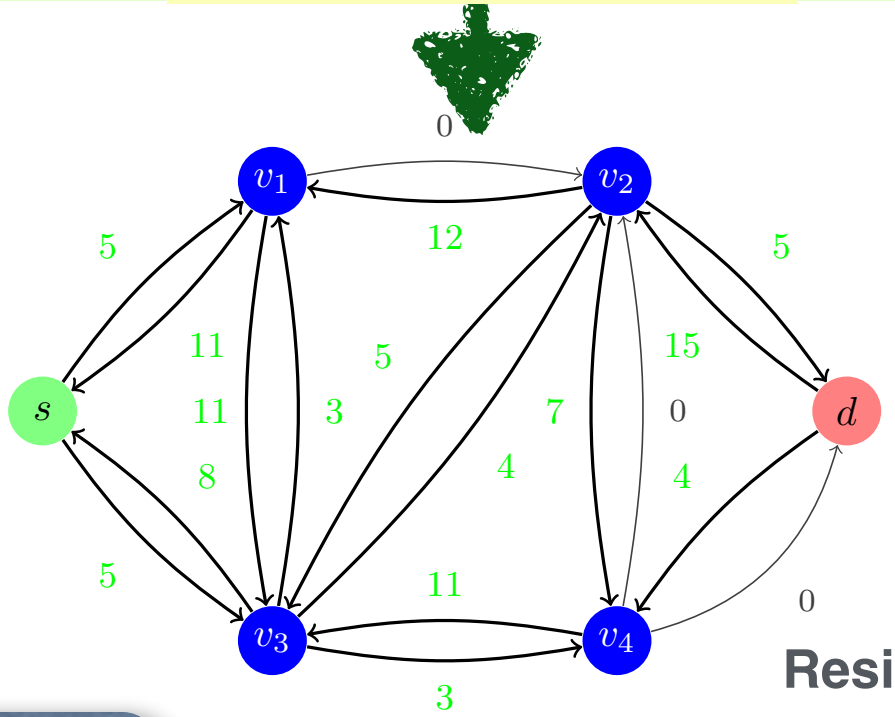


Fluss f



Restkapazität

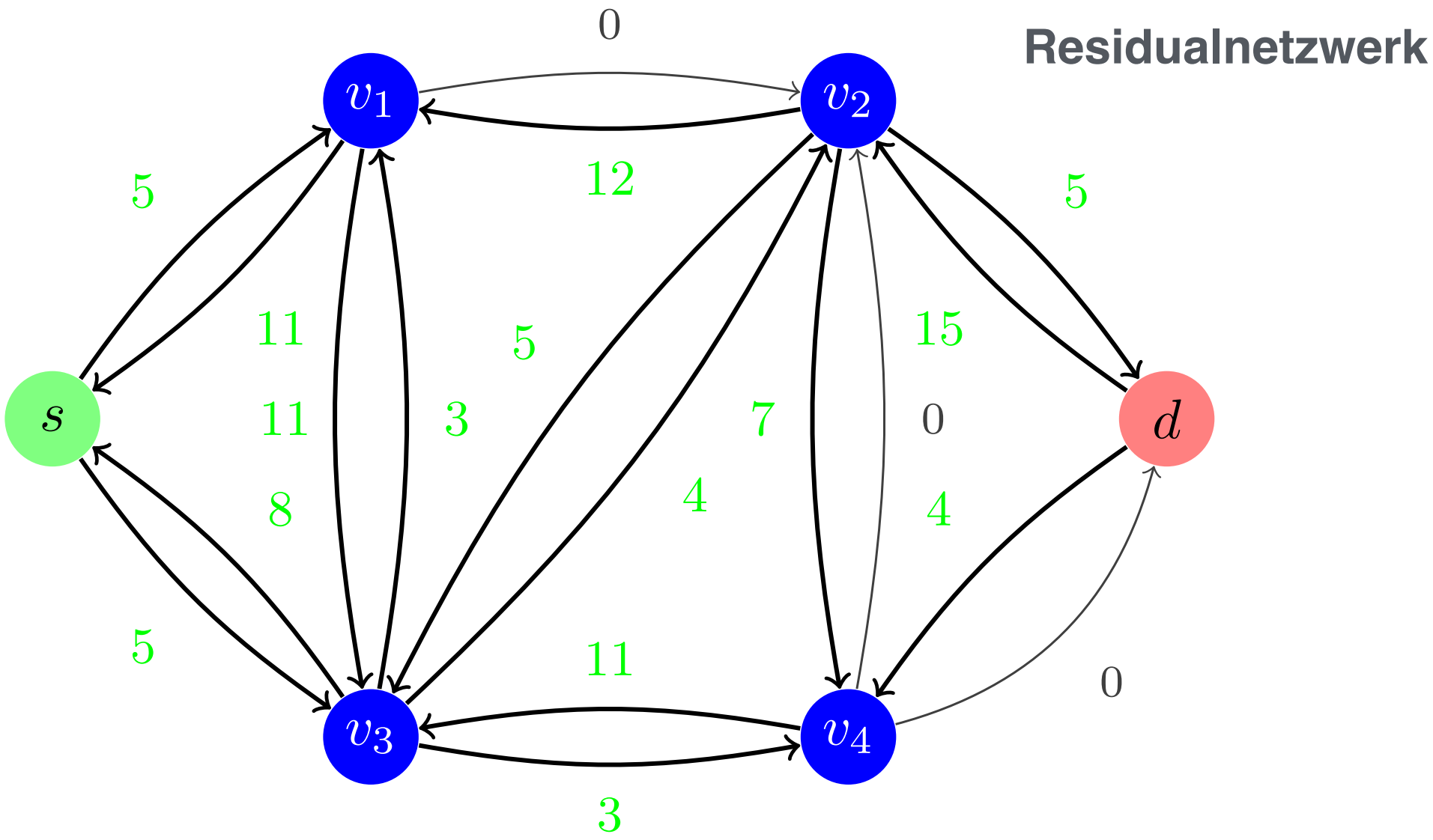
$$c_f(u,v) = c(u,v) - f(u,v)$$



Residualnetzwerk

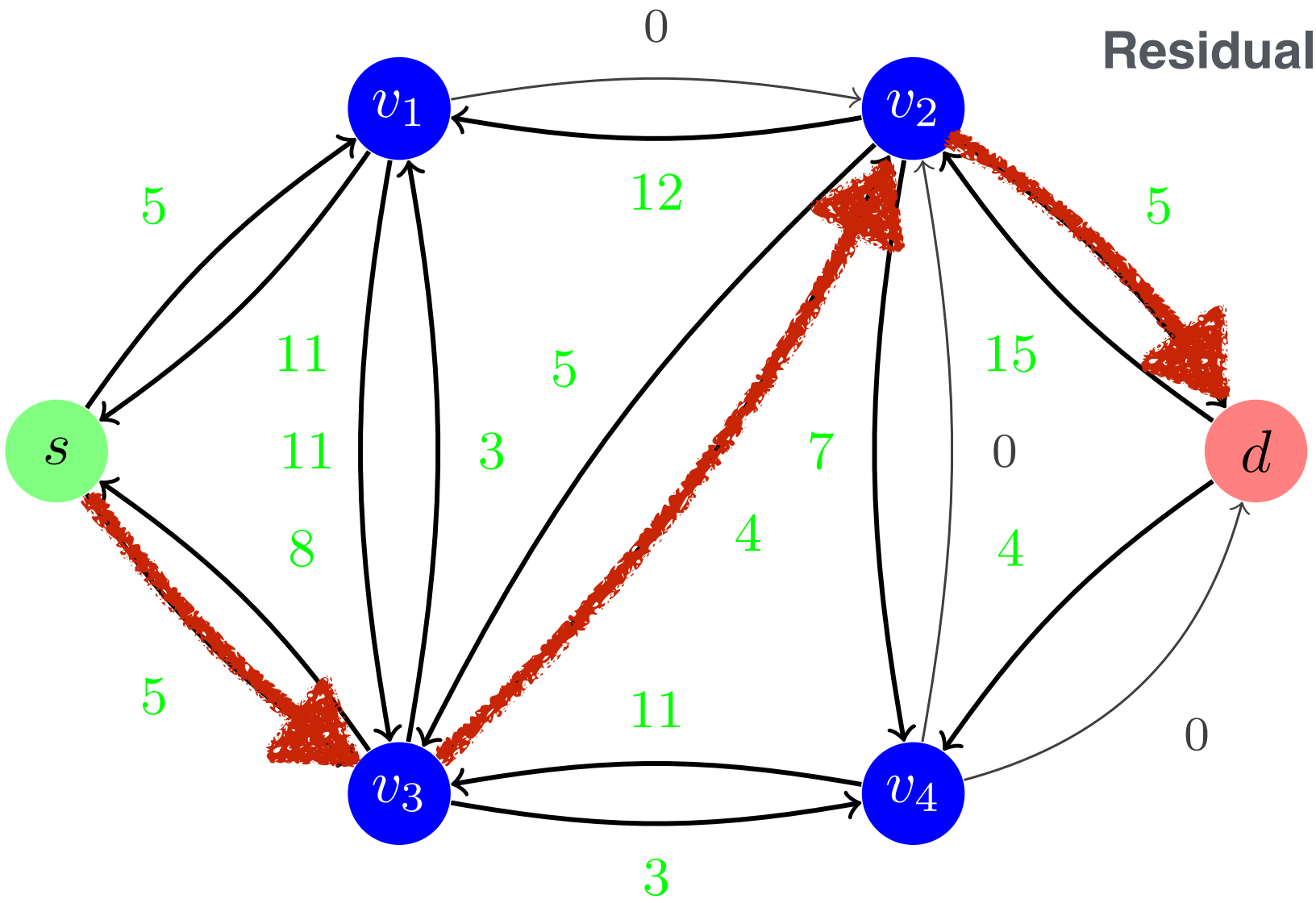
© 2006-2024 Jörg Striegnitz

Ford-Fulkerson: Pfad im Residualnetzwerk



© 2006-2024 Jörg Striegnitz

Ford-Fulkerson: Pfad im Residualnetzwerk

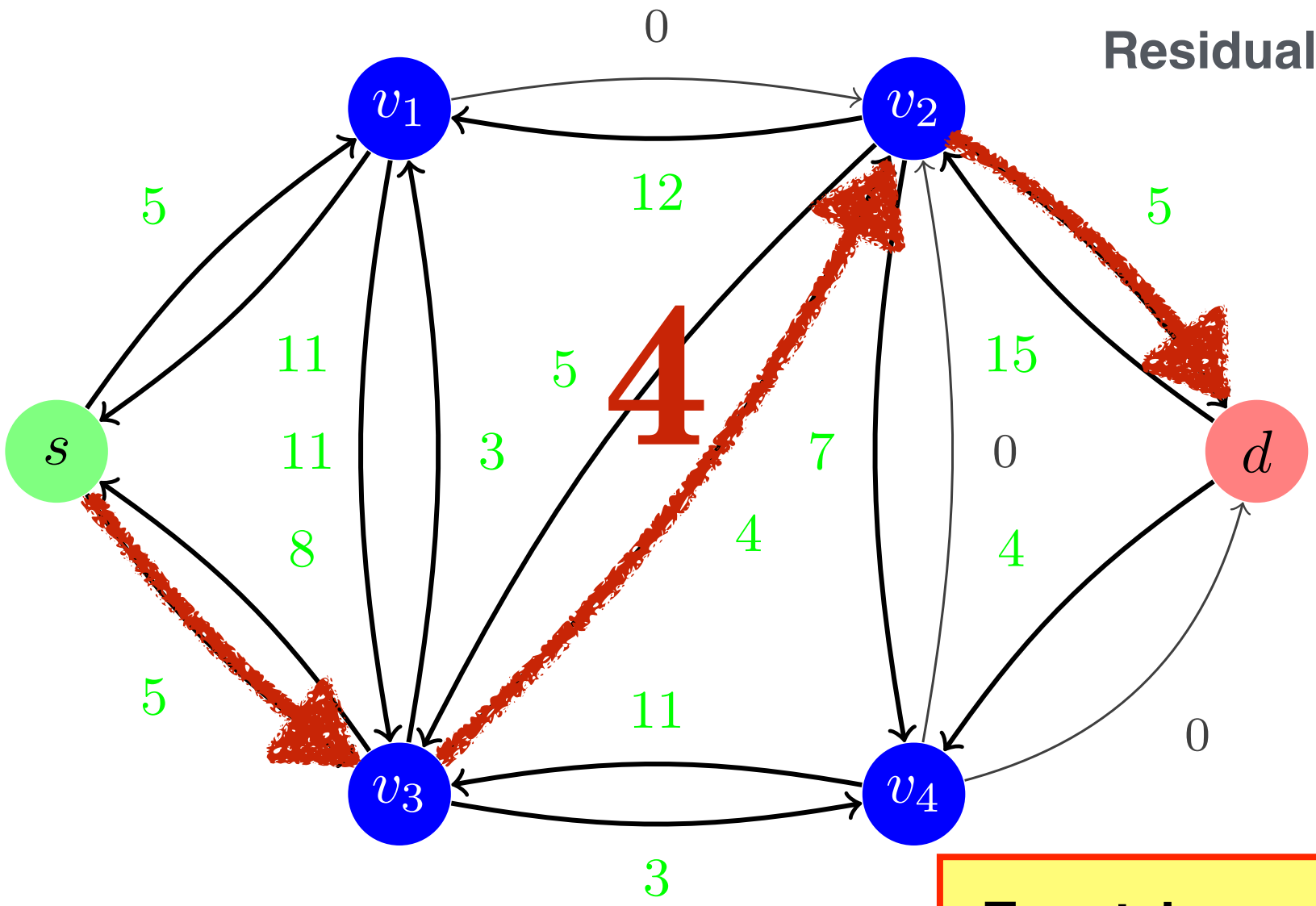


Finde Pfad von s nach d

Kanten mit Gewicht ≤ 0 werden ignoriert

© 2006-2024 Jörg Striegnitz

Ford-Fulkerson: Pfad im Residualnetzwerk



Finde Pfad von s nach d

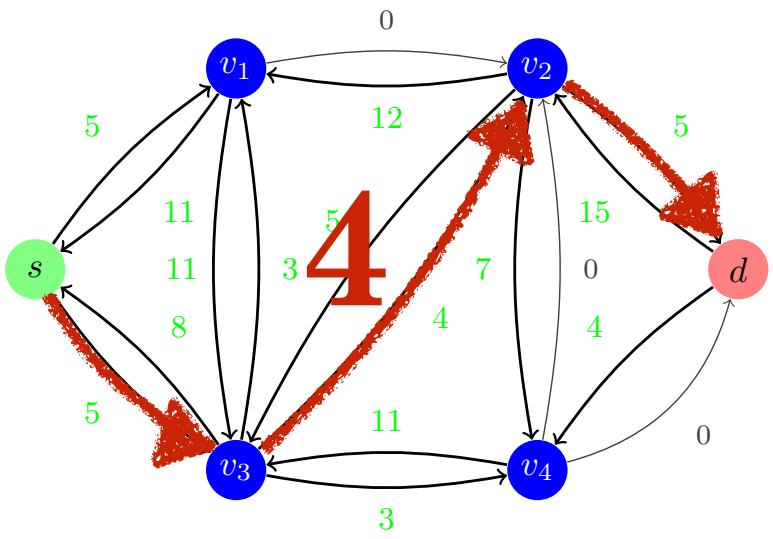
Kanten mit Gewicht ≤ 0 werden ignoriert

Zusatzkapazität für 4 Einheiten

© 2006-2024 Jörg Striegnitz

Ford-Fulkerson: Fluss aktualisieren

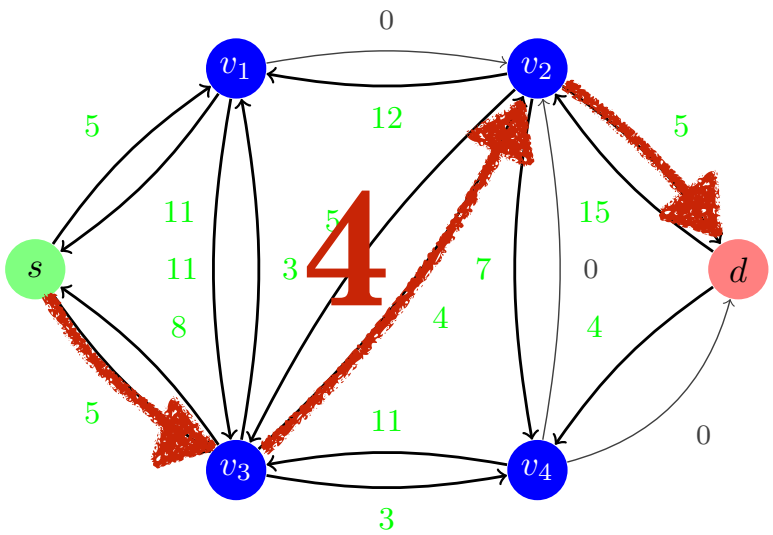
Residualnetzwerk

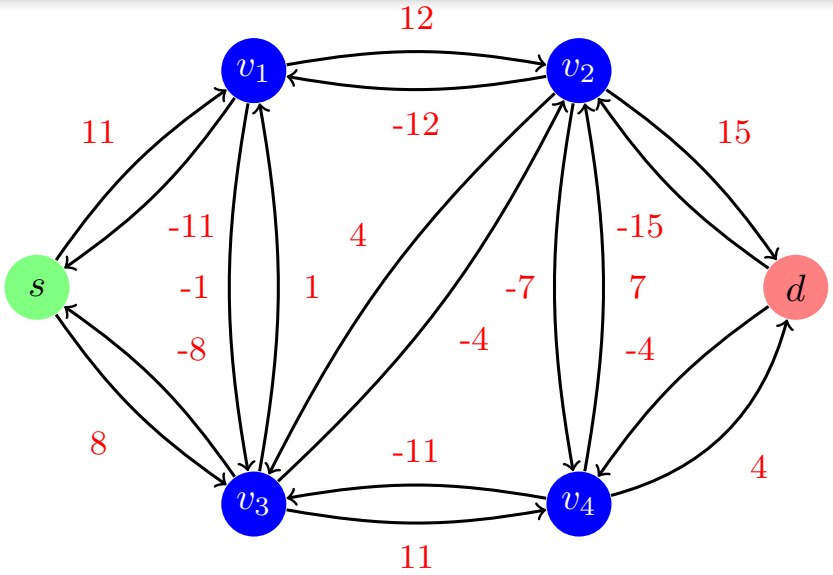


© 2006-2024 Jörg Striegnitz

Ford-Fulkerson: Fluss aktualisieren

Residualnetzwerk

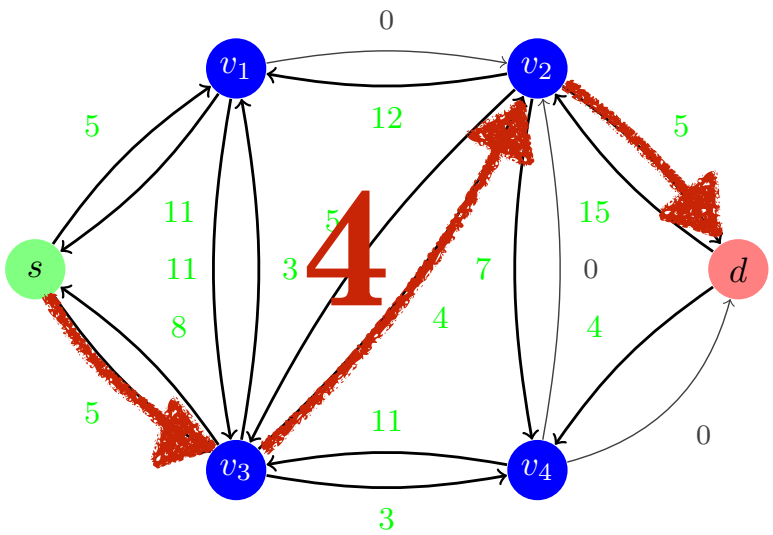


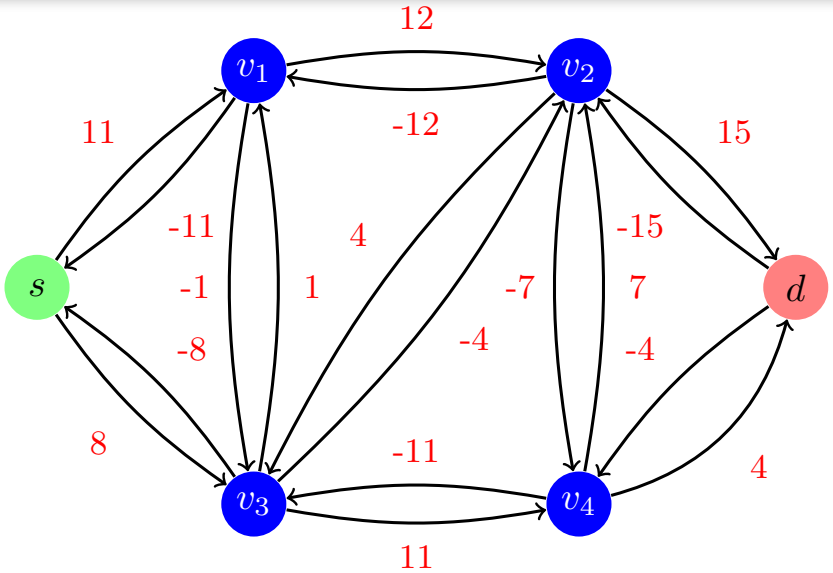


© 2006-2024 Jörg Striegnitz

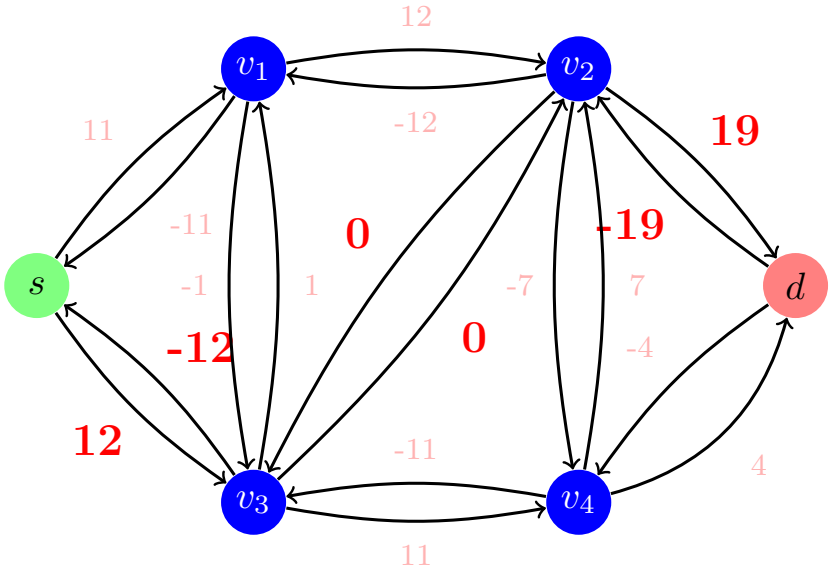
Ford-Fulkerson: Fluss aktualisieren

Residualnetzwerk



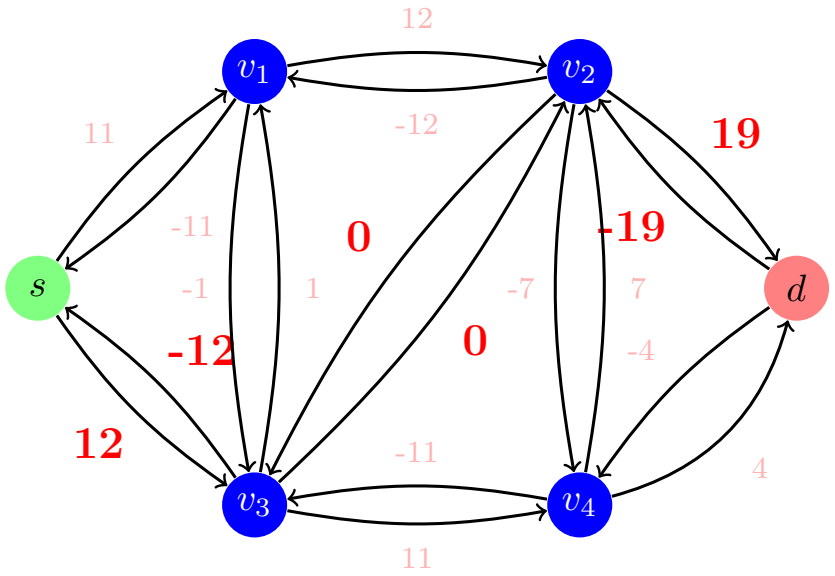


Aktualisiere Fluss



© 2006-2024 Jörg Striegnitz

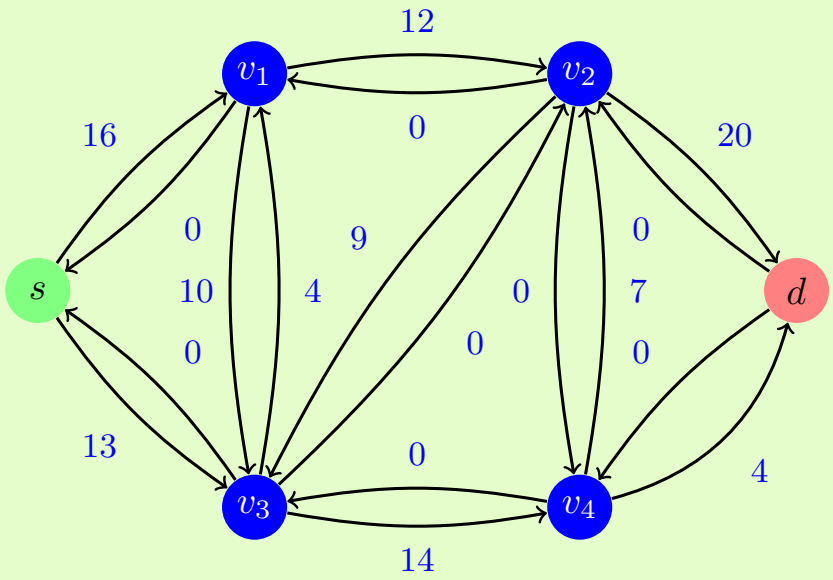
Ford-Fulkerson: Residualnetzwerk aktualisieren



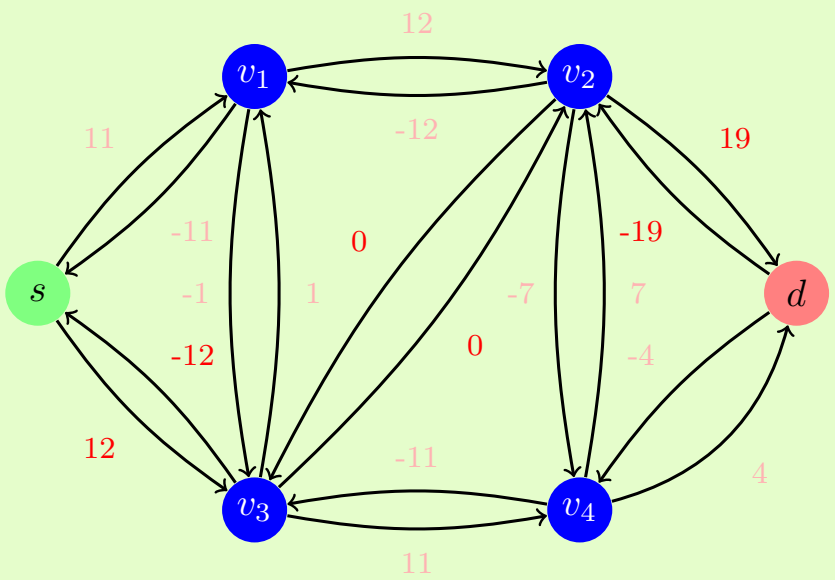
© 2006-2024 Jörg Striegnitz

Ford-Fulkerson: Residualnetzwerk aktualisieren

Kapazität c



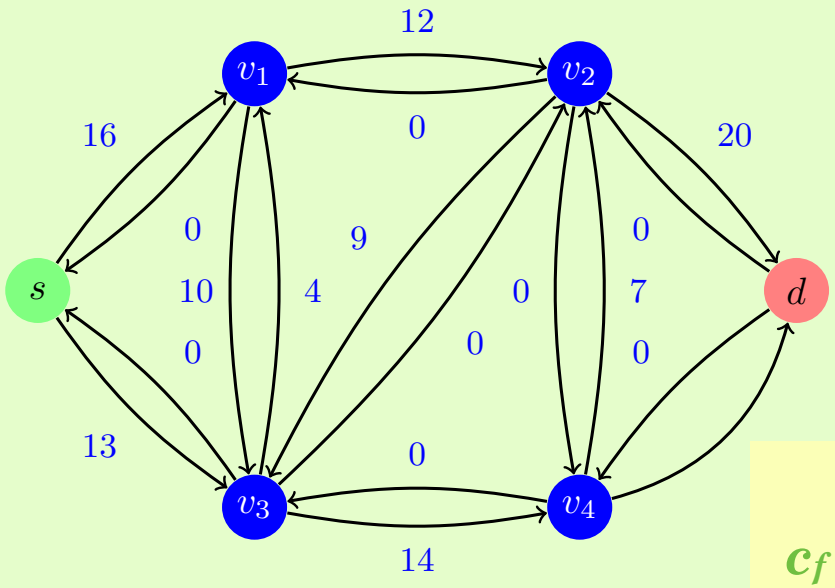
Fluss f



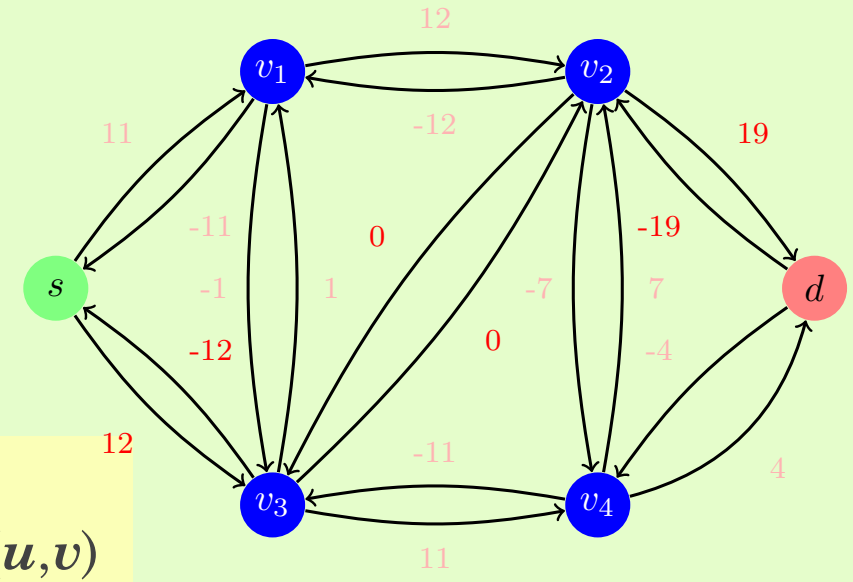
© 2006-2024 Jörg Striegnitz

Ford-Fulkerson: Residualnetzwerk aktualisieren

Kapazität c

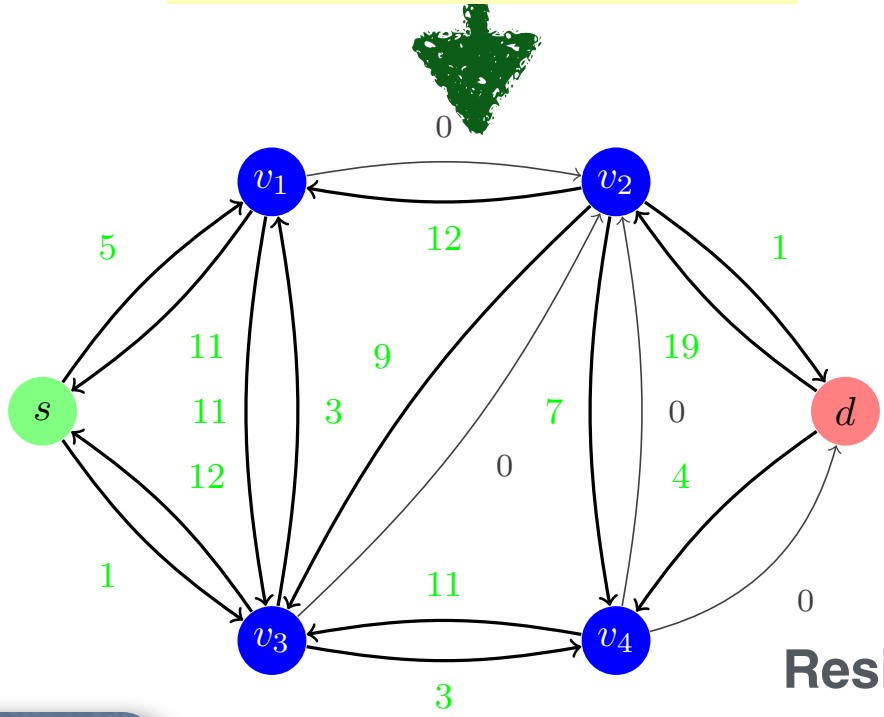


Fluss f



Restkapazität

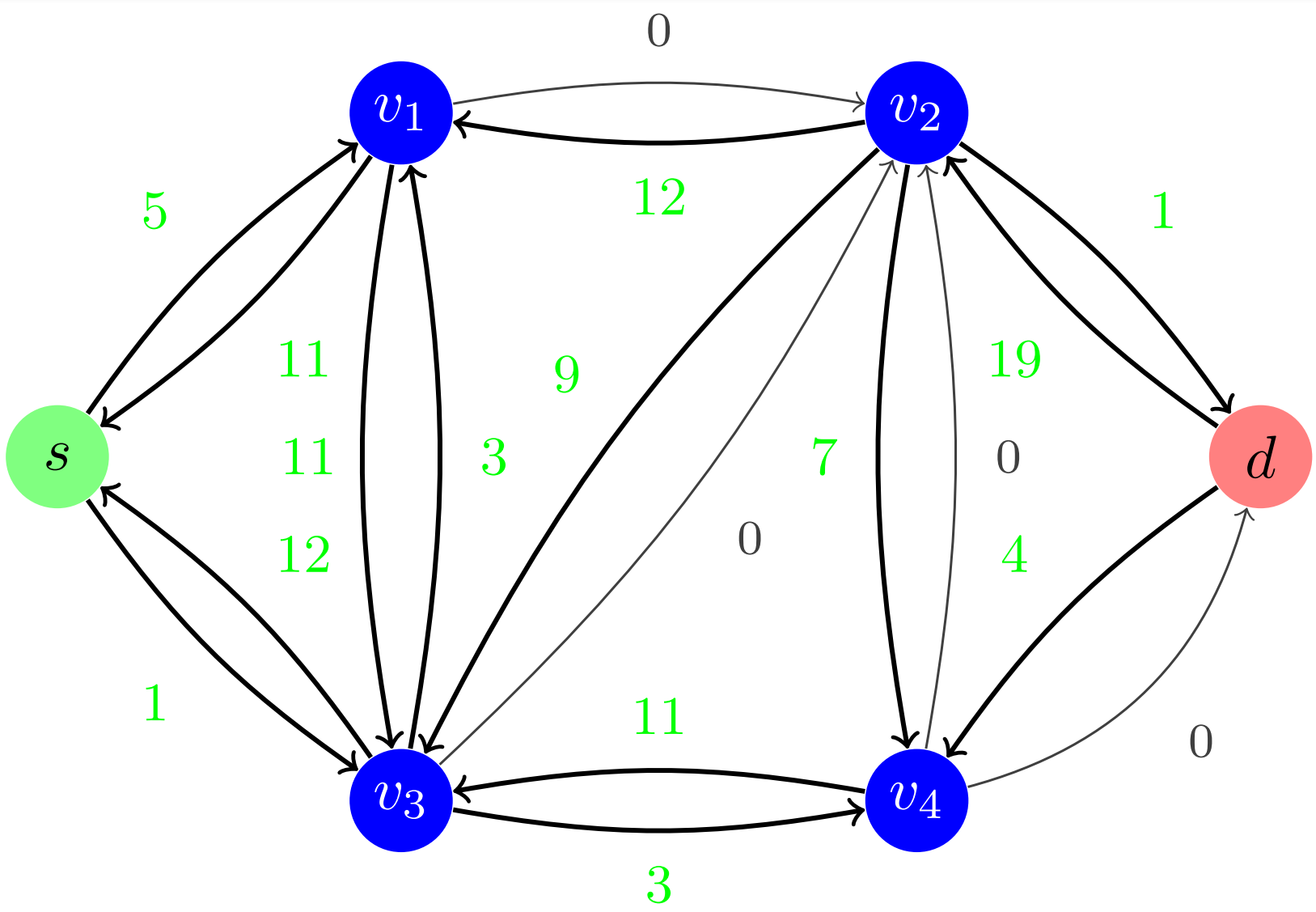
$c_f(u,v) = c(u,v) - f(u,v)$



Residualnetzwerk

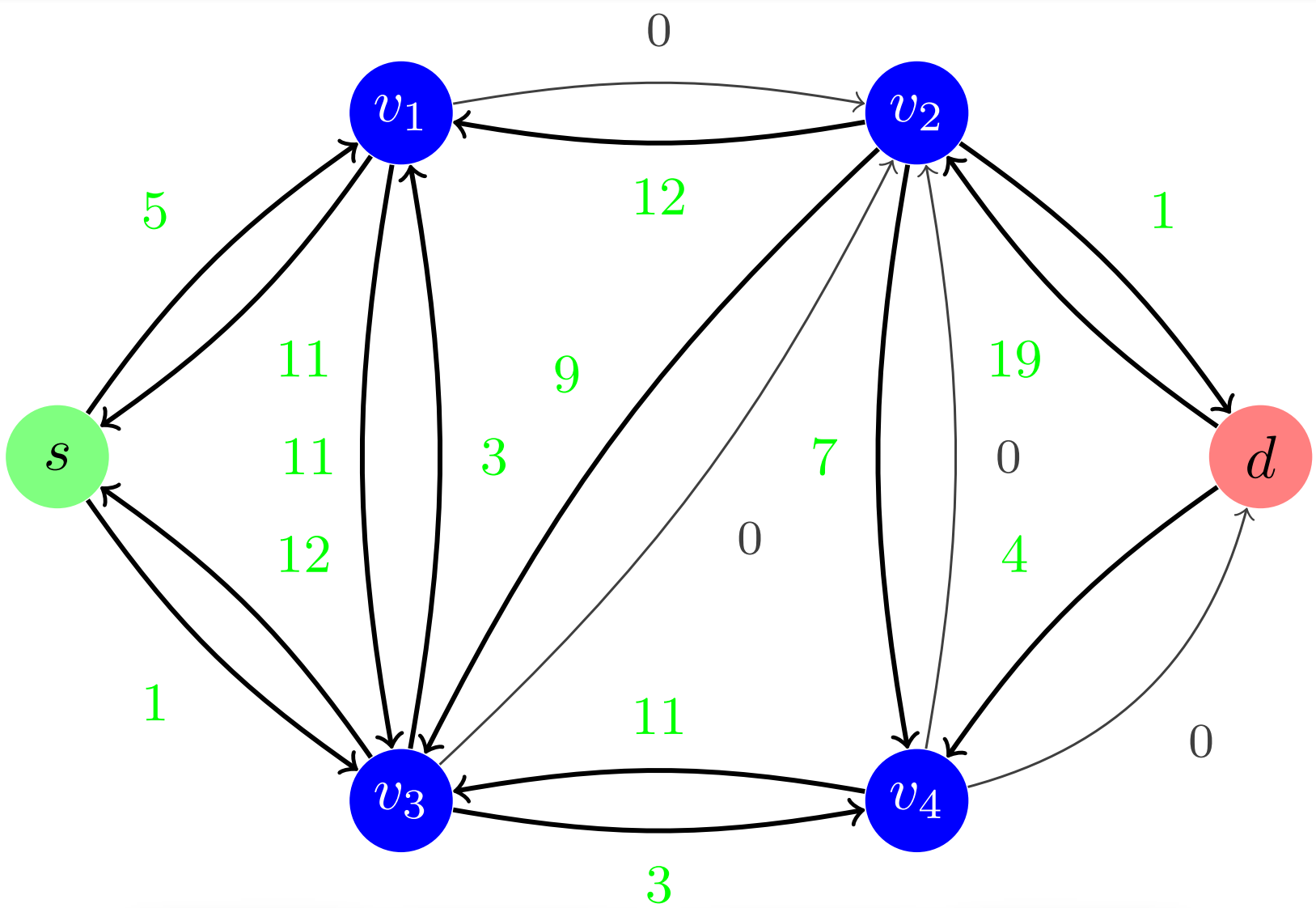
© 2006-2024 Jörg Striegnitz

Ford-Fulkerson: Residualnetzwerk aktualisieren



© 2006-2024 Jörg Striegnitz

Ford-Fulkerson: Residualnetzwerk aktualisieren



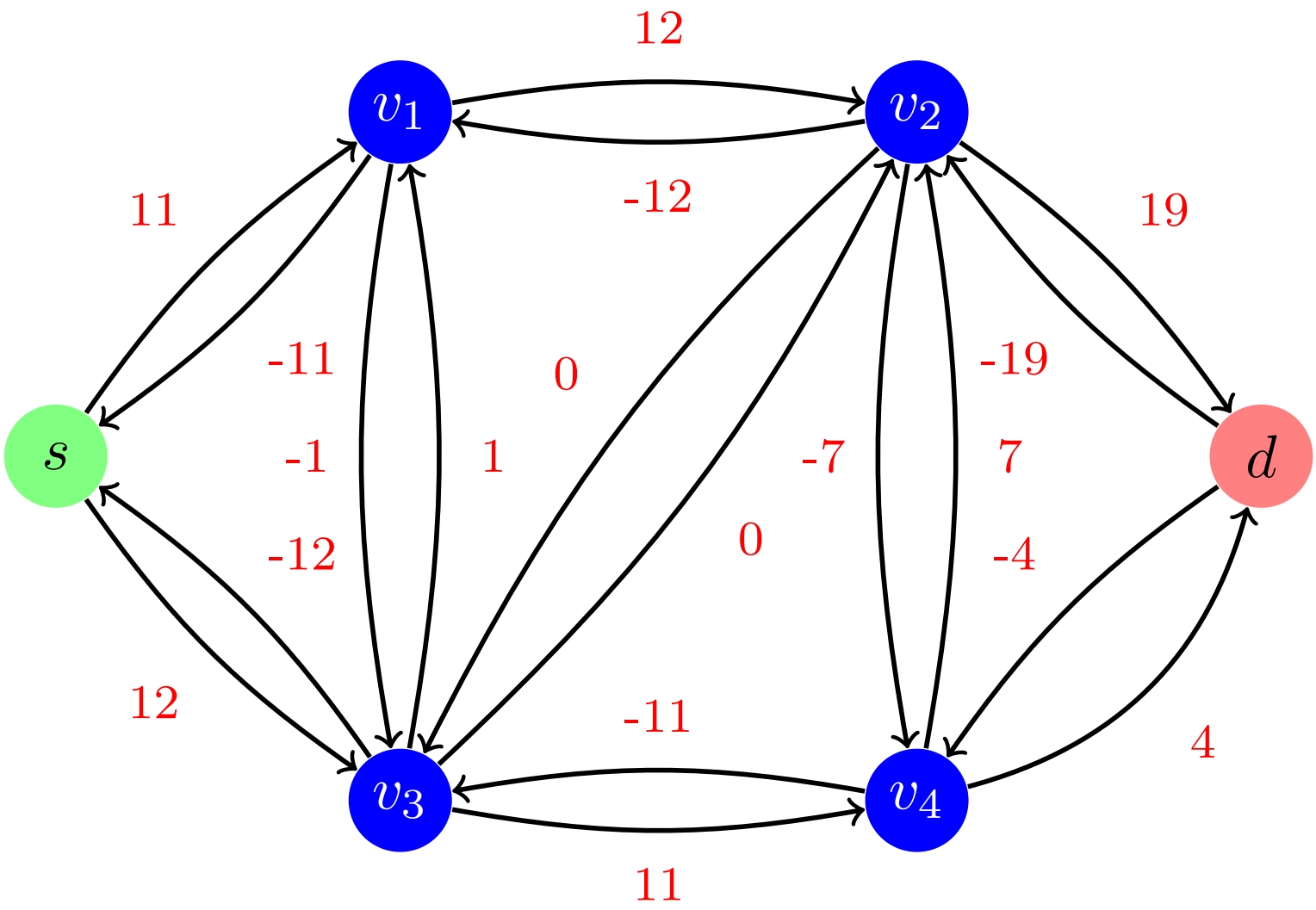
Kein Pfad von s nach d

Kanten mit Gewicht ≤ 0 werden ignoriert

© 2006-2024 Jörg Striegnitz

Ford-Fulkerson: Residualnetzwerk aktualisieren

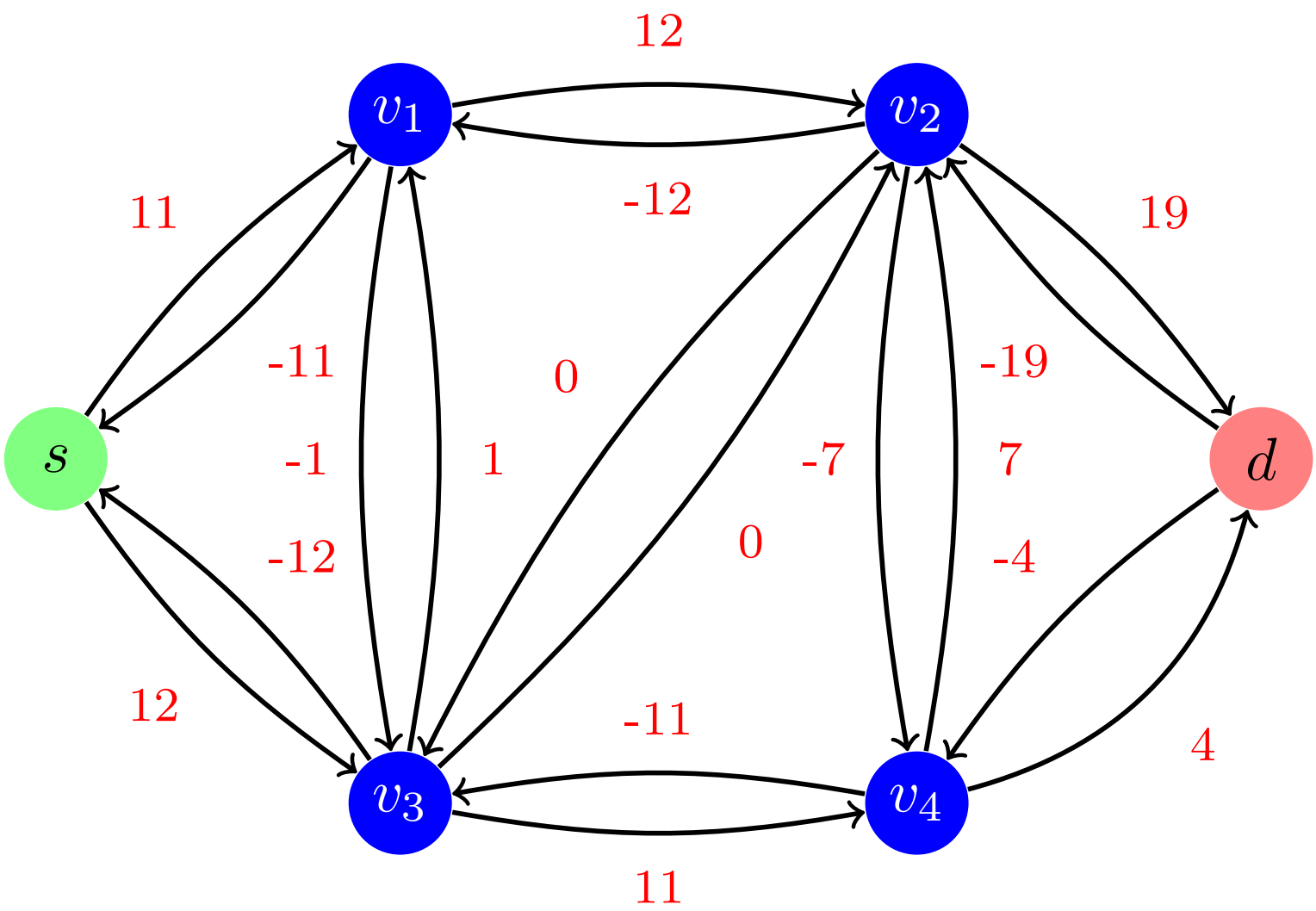
Fluss f



© 2006-2024 Jörg Striegnitz

Ford-Fulkerson: Residualnetzwerk aktualisieren

Fluss f



Maximaler Fluss ist 23

© 2006-2024 Jörg Striegnitz

Algorithmus von Ford-Fulkerson

```
Ford-Fulkerson(F=( G=(V,E),c,s,d))
```

for each edge (u,v) **in** E

$f[u,v] = f[v,u] = 0$

solange ein Pfad p **von** s **nach** d **in** G_f **existiert:**

$c_f(p) \leftarrow$ **min**$(c_f(u,v) : (u,v)$ **in** $p)$

for each edge (u,v) **in** p

$f[u,v] \leftarrow f[u,v] + c_f(p)$

$f[v,u] \leftarrow -f[u,v]$

© 2006-2024 Jörg Striegnitz

Algorithmus von Ford-Fulkerson

```
Ford-Fulkerson(F=( G=(V,E),c,s,d))
```

for each edge (u,v) **in** E

$f[u,v] = f[v,u] = 0$

solange ein Pfad p **von** s **nach** d **in** G_f **existiert:**

$c_f(p) \leftarrow$ **min**$(c_f(u,v) : (u,v)$ **in** $p)$

for each edge (u,v) **in** p

$f[u,v] \leftarrow f[u,v] + c_f(p)$

$f[v,u] \leftarrow -f[u,v]$

Laufzeit: $O(|f^*|)$

wobei f^* der maximale Fluss ist

© 2006-2024 Jörg Striegnitz

Maximales Matching

© 2006-2024 Jörg Striegnitz

Maximales Matching

- **Weitere Anwendung für Algorithmen zur Berechnung des maximalen Flusses: maximales Matching**

© 2006-2024 Jörg Striegnitz

Maximales Matching

- **Weitere Anwendung für Algorithmen zur Berechnung des maximalen Flusses: maximales Matching**

- **Ausgangspunkt: Bipartiter Graph**

D Bipartiter Graph

Sei $G = (V, E)$ ein ungerichteter Graph. G heißt **bipartit**, falls sich eine disjiunkte Zerlegung von V in V_1 und V_2 finden lässt, so dass

$$\forall (u, v) \in E : (u \in V_1 \wedge v \in V_2) \ \vee (u \in V_2 \wedge v_\epsilon V_1)$$

© 2006-2024 Jörg Striegnitz

Maximales Matching

- **Weitere Anwendung für Algorithmen zur Berechnung des maximalen Flusses: maximales Matching**

- **Ausgangspunkt: Bipartiter Graph**

D Bipartiter Graph

Sei $G = (V, E)$ ein ungerichteter Graph. G heißt **bipartit**, falls sich eine disjiunkte Zerlegung von V in V_1 und V_2 finden lässt, so dass

$$\forall (u, v) \in E : (u \in V_1 \wedge v \in V_2) \ \vee (u \in V_2 \wedge v_\epsilon V_1)$$

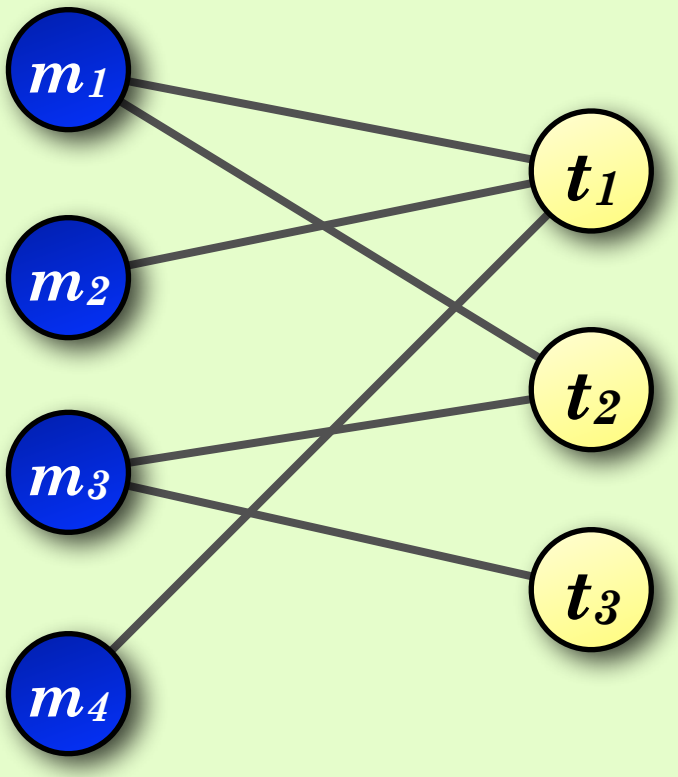
- **Vereinfachend:**
  - Knoten lassen sich in zwei Gruppen aufteilen, so dass keine Kante zwei Knoten derselben Gruppe verbindet

© 2006-2024 Jörg Striegnitz

Anwendungsbeispiel

B Modellierung von Produktionsprozess

- Ein Unternehmen hat 4 Maschinen $m_1,\ldots,m_4$.
- Drei Aufgaben $t_1,\ldots,t_3$ seien aktuell zu erledigen
- Jede Maschine kann nur bestimmte Aufgaben übernehmen:

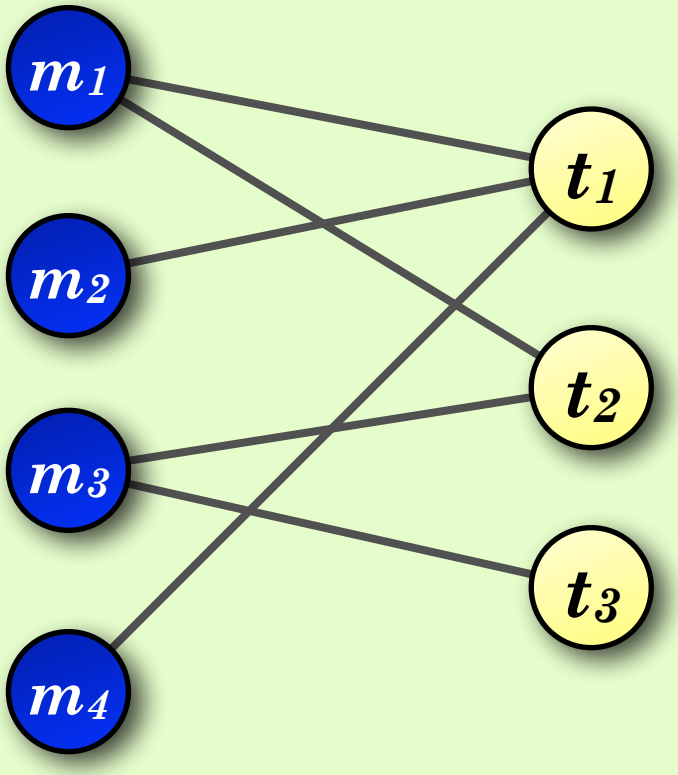


© 2006-2024 Jörg Striegnitz

Anwendungsbeispiel

B *Modellierung von Produktionsprozess*

- Ein Unternehmen hat 4 Maschinen $m_1,\ldots,m_4$.
- Drei Aufgaben $t_1,\ldots,t_3$ seien aktuell zu erledigen
- Jede Maschine kann nur bestimmte Aufgaben übernehmen:



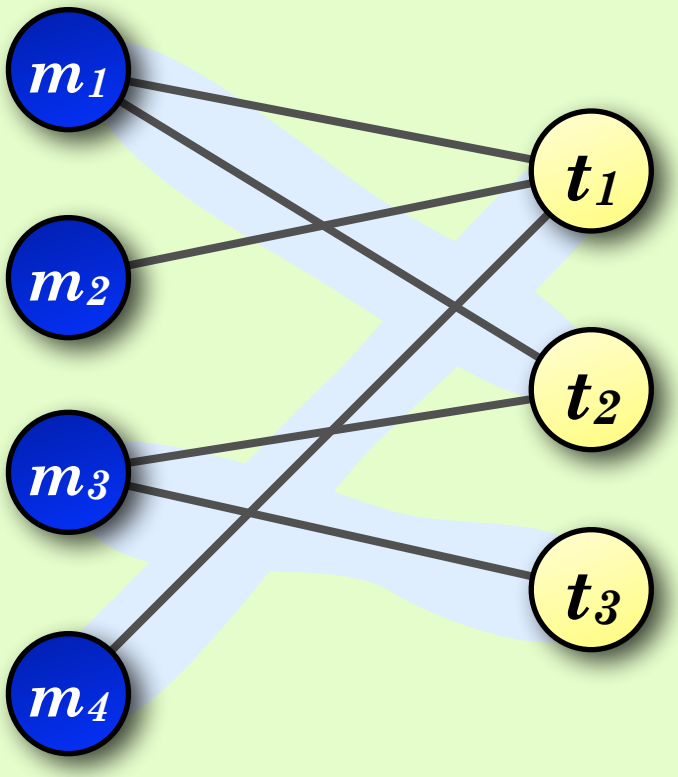
Wie viele Aufgaben können maximal parallel abgearbeitet werden

© 2006-2024 Jörg Striegnitz

Anwendungsbeispiel

B *Modellierung von Produktionsprozess*

- Ein Unternehmen hat 4 Maschinen $m_1,\ldots,m_4$.
- Drei Aufgaben $t_1,\ldots,t_3$ seien aktuell zu erledigen
- Jede Maschine kann nur bestimmte Aufgaben übernehmen:



Wie viele Aufgaben können maximal parallel abgearbeitet werden

© 2006-2024 Jörg Striegnitz

D Matching / maximales Matching

Seien $G = (V, E)$ ein ungerichteter Graph und adjacents : $V \times \mathcal{P}(E) \to \mathcal{P}(V)$ eine Funktion mit

$$\text{adjacents}(v, M) = \{u \in V - \{v\} \mid (v, u) \in M \vee (u, v) \in M\}$$

$M \subseteq E$ heißt **Matching**, falls

$$\forall v \in V : |\text{adjacents}(v, M)| \leq 1$$

Die Menge aller Matchings in G bezeichnen wir mit $\mathcal{M}(G)$ und $M \in \mathcal{M}(G)$ heißt **maximales Matching**, genau dann, wenn

$$\forall M' \in \mathcal{M}(G) : |M'| \leq |M|$$

- **Algorithmus zum Auffinden eines maximalen Matching beruht auf Ford-Fulkersons-Methode**

© 2006-2024 Jörg Striegnitz

Bestimmung des maximalen Matchings

- **Sei G=(V,E) ein bipartiter Graph mit V=L∪R**

- **Erweitere G zu einem Flussnetzwerk F=(G‘,c,s,d):**
  - G‘=(V ∪ {s,d}, E‘) mit
    E‘ = {(s,u) : u∈L} ∪ **/* verbinde Quelle s mit Knoten aus L */**
    {(u,v) : u∈L, v∈R ∧ (u,v)∈E} ∪ **/* Kanten verlaufen von L nach R */**
    {(v,d) : v∈R } **/* verbinde Knoten aus R mit Ziel s */**
  - ∀(u,v)∈E‘ : c(u,v)=1

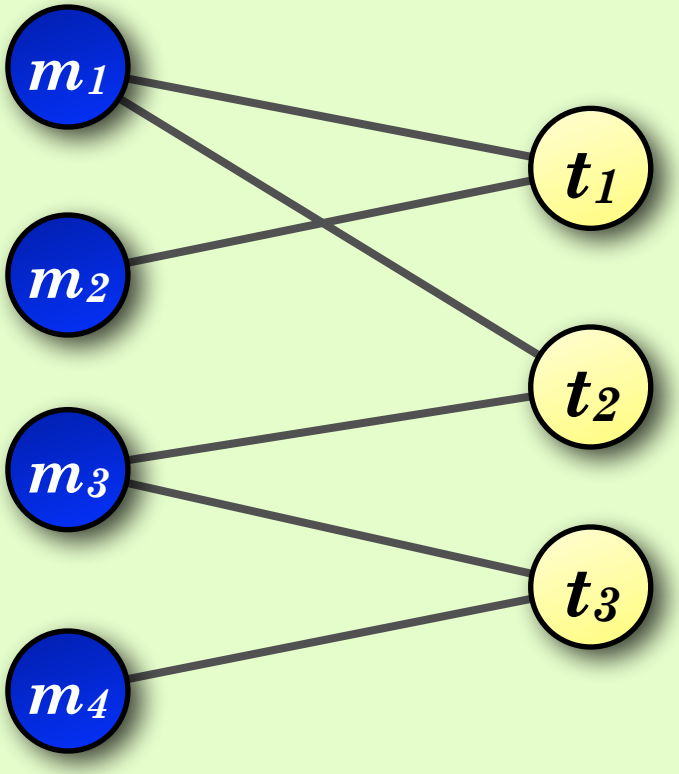
- **Wende Ford-Fulkerson-Methode auf F an**
  - Matching wird dann aus Kanten gebildet, die L und R verbinden und zum maximalen Fluss beitragen

© 2006-2024 Jörg Striegnitz

Anwendungsbeispiel

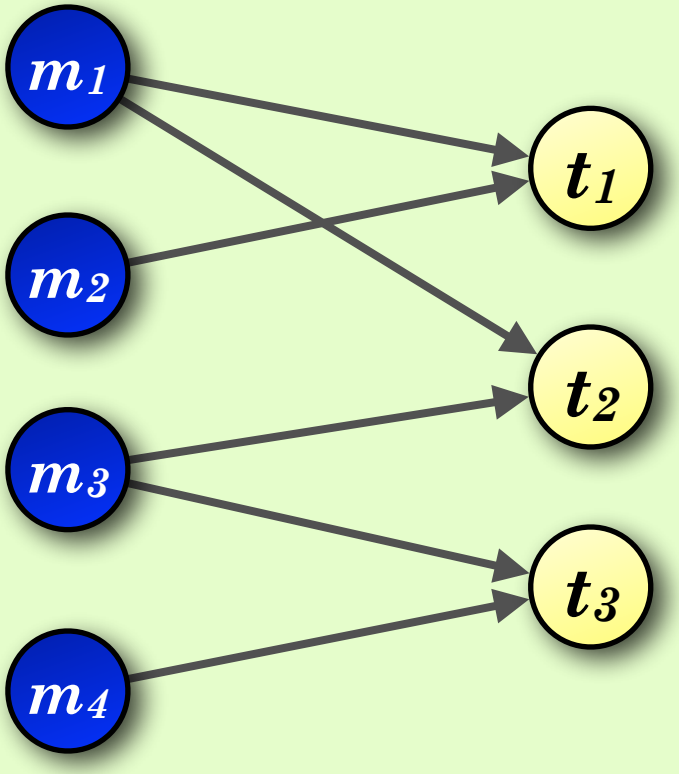
B Berechnung eines maximalen Matchings

- Gegeben sei folgender bipartiter Graph:



© 2006-2024 Jörg Striegnitz

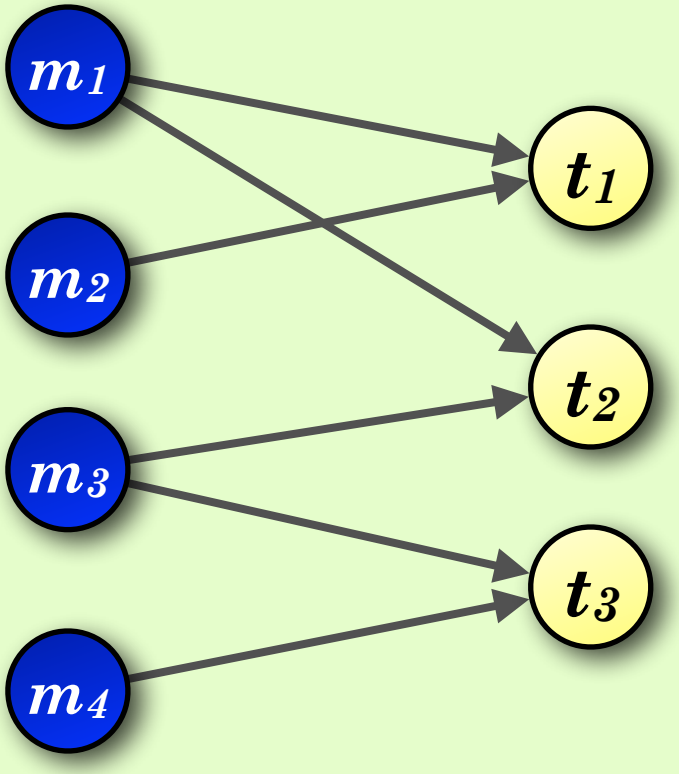
B Berechnung eines maximalen Matchings



© 2006-2024 Jörg Striegnitz

B Berechnung eines maximalen Matchings

- Richte Kanten aus

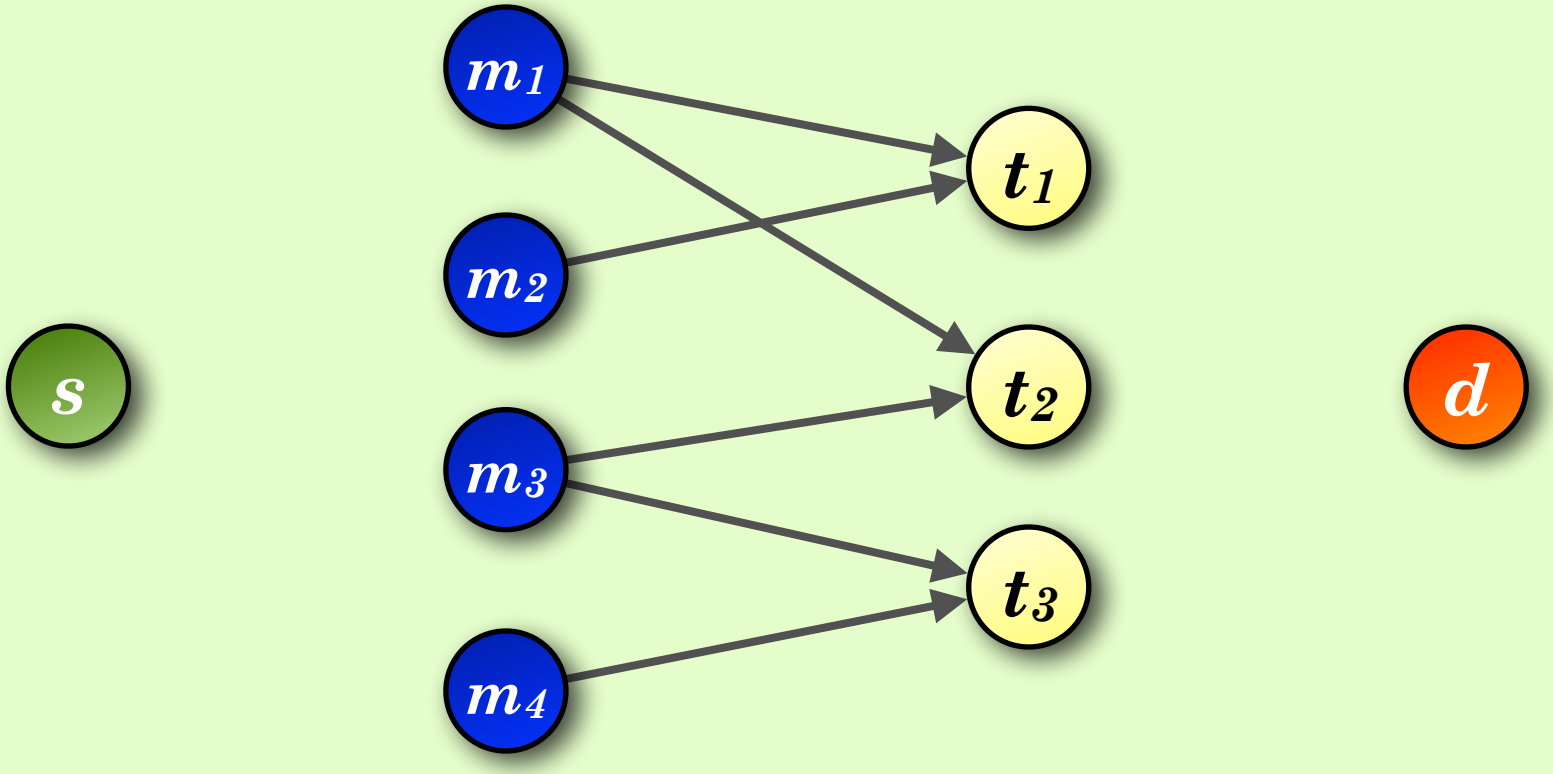


© 2006-2024 Jörg Striegnitz

Anwendungsbeispiel

B *Berechnung eines maximalen Matchings*

- Richte Kanten aus
- Führe Quell- und Zielknoten ein

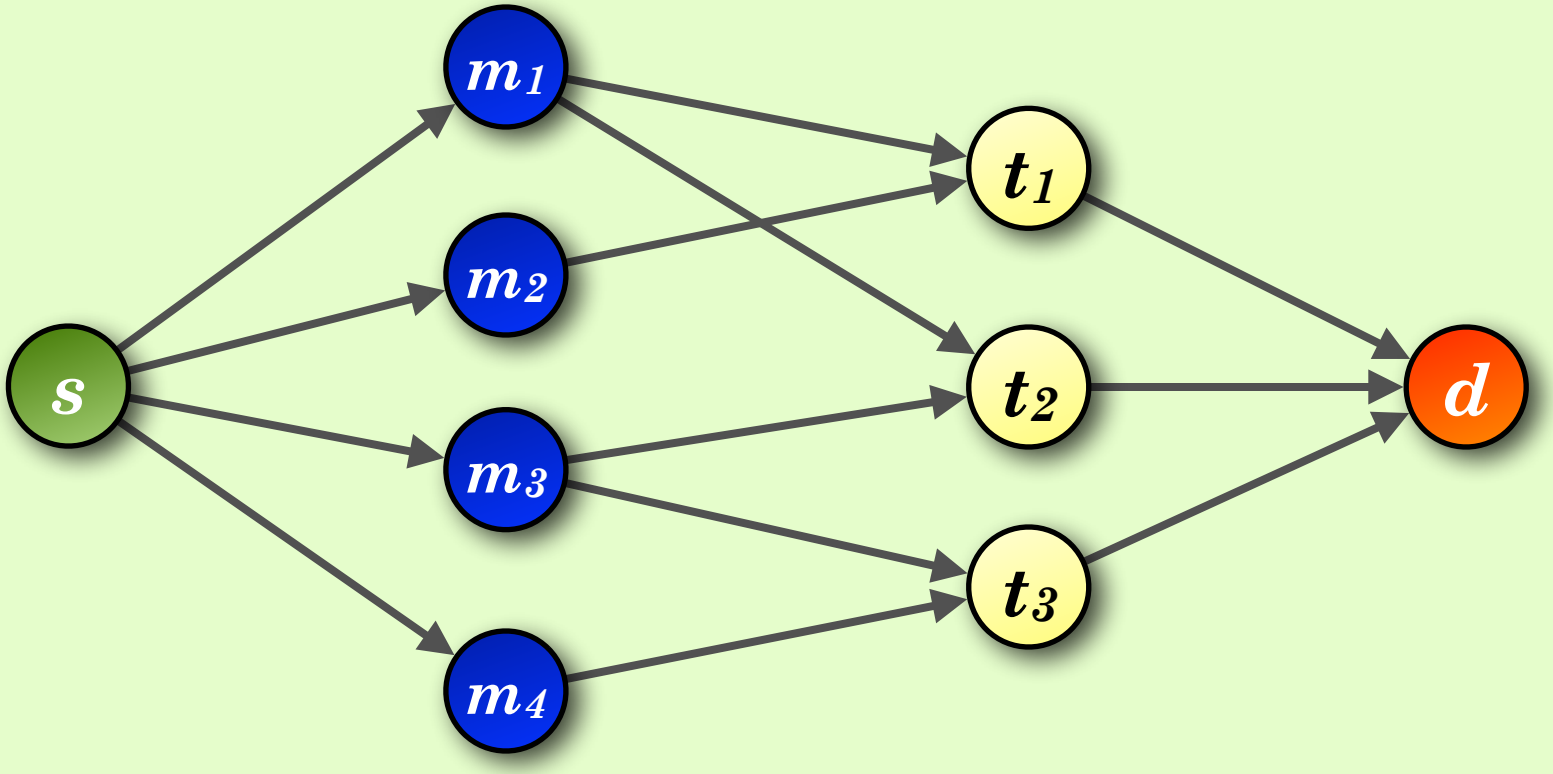


© 2006-2024 Jörg Striegnitz

Anwendungsbeispiel

B *Berechnung eines maximalen Matchings*

- Richte Kanten aus
- Führe Quell- und Zielknoten ein
- Verbinde Quell- und Zielknoten mit Graph

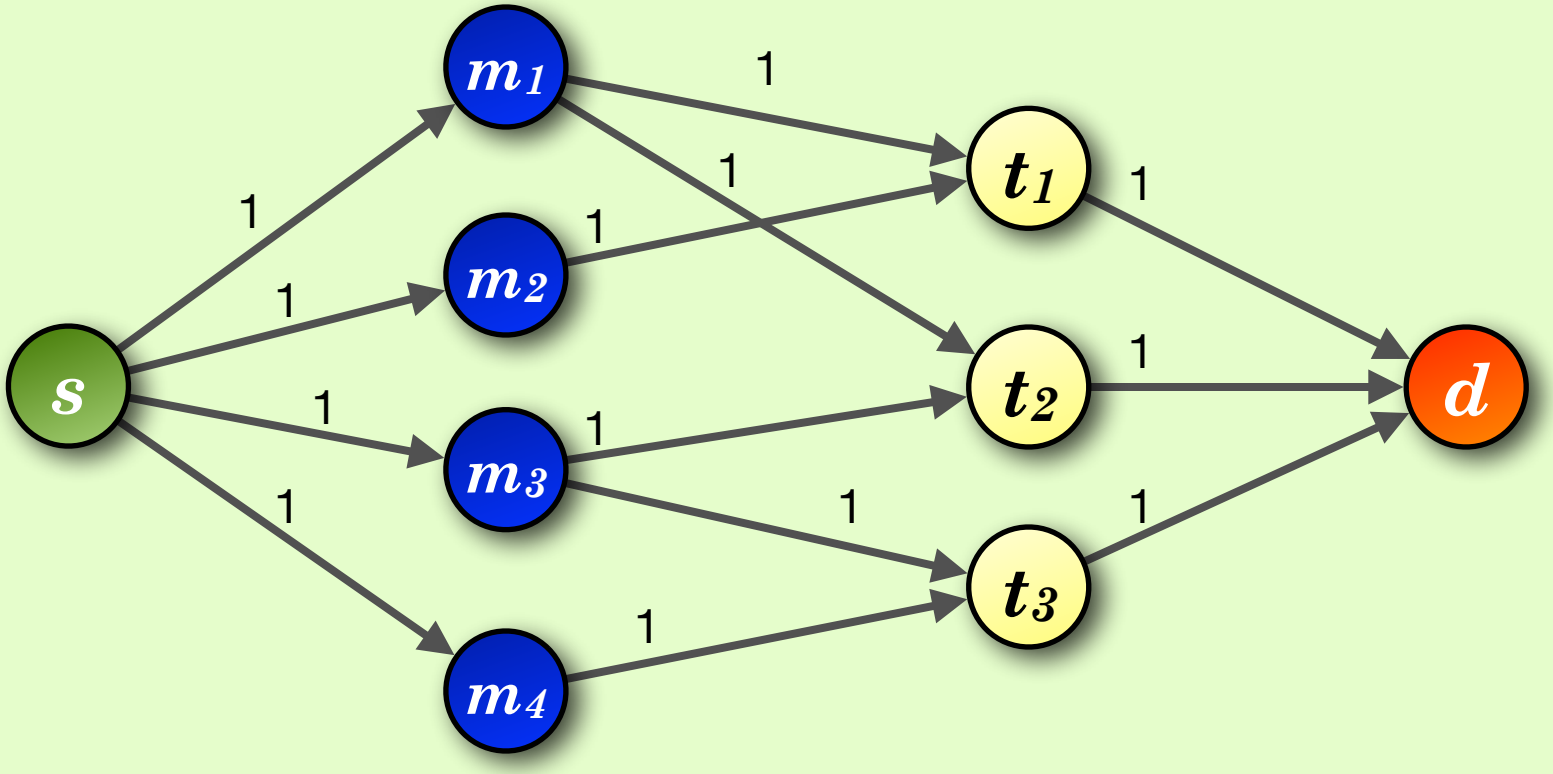


© 2006-2024 Jörg Striegnitz

Anwendungsbeispiel

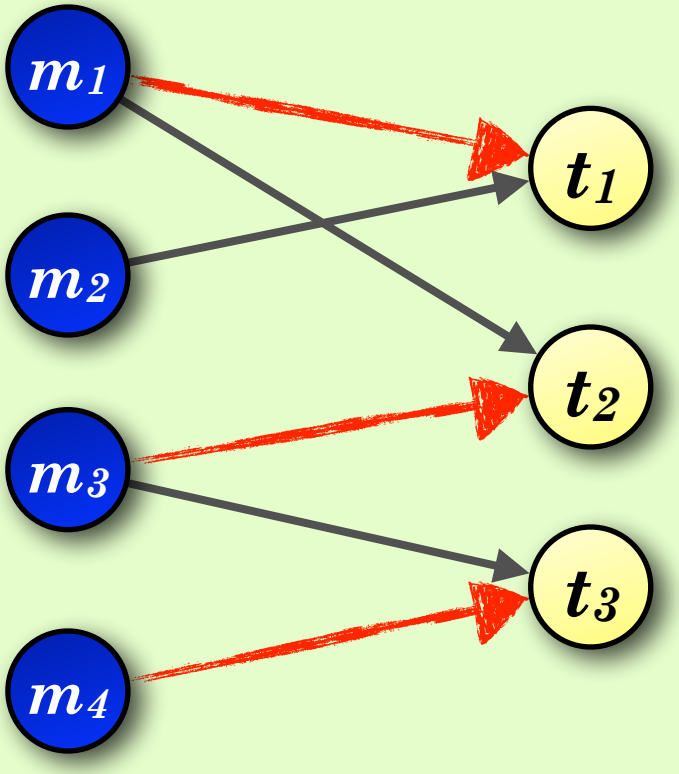
B Berechnung eines maximalen Matchings

- Richte Kanten aus
- Führe Quell- und Zielknoten ein
- Verbinde Quell- und Zielknoten mit Graph
- Setze Kapazitäten auf 1



© 2006-2024 Jörg Striegnitz

B Berechnung eines maximalen Matchings

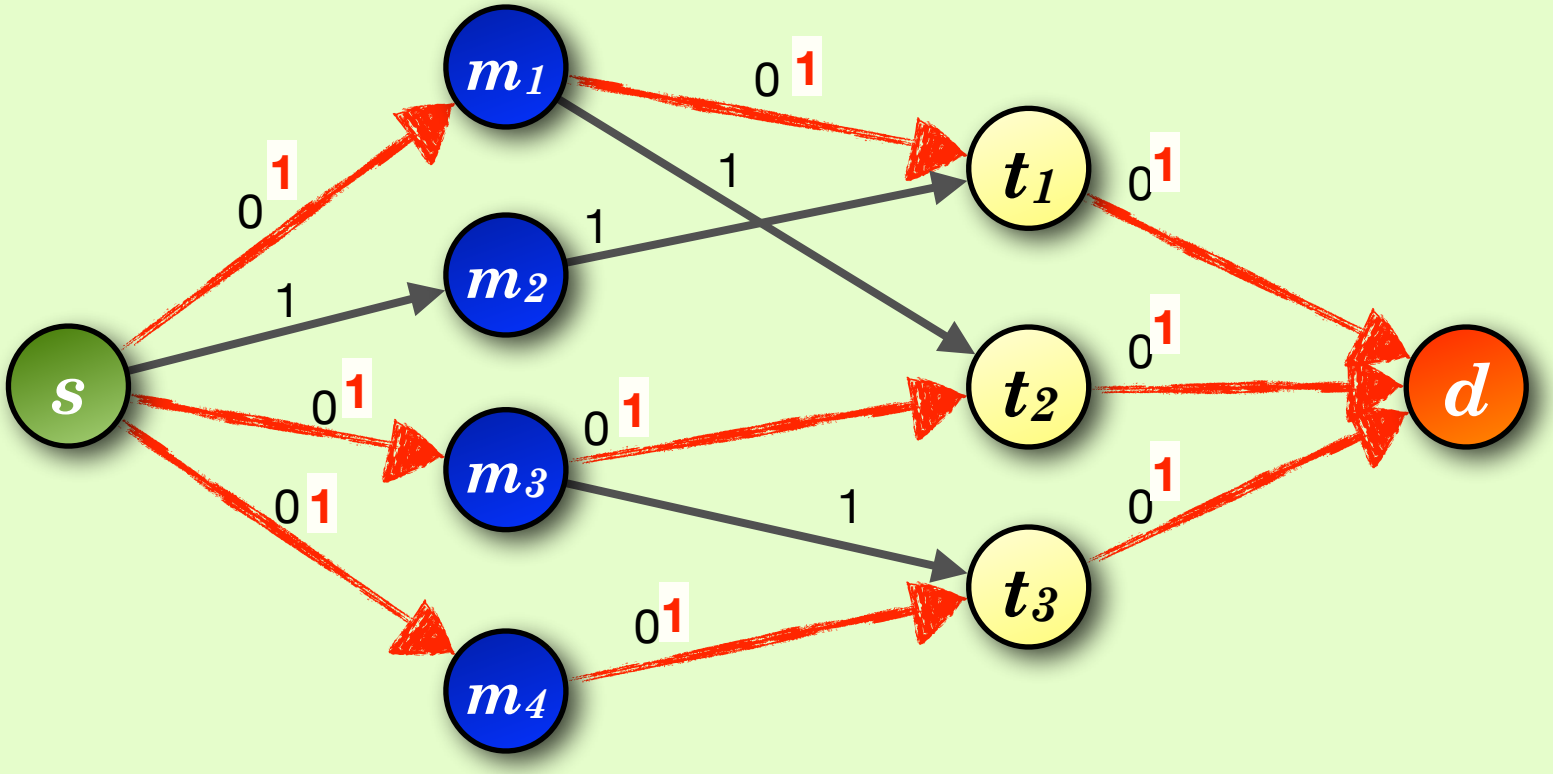


© 2006-2024 Jörg Striegnitz

Anwendungsbeispiel

B Berechnung eines maximalen Matchings

- Berechne maximalen Fluss

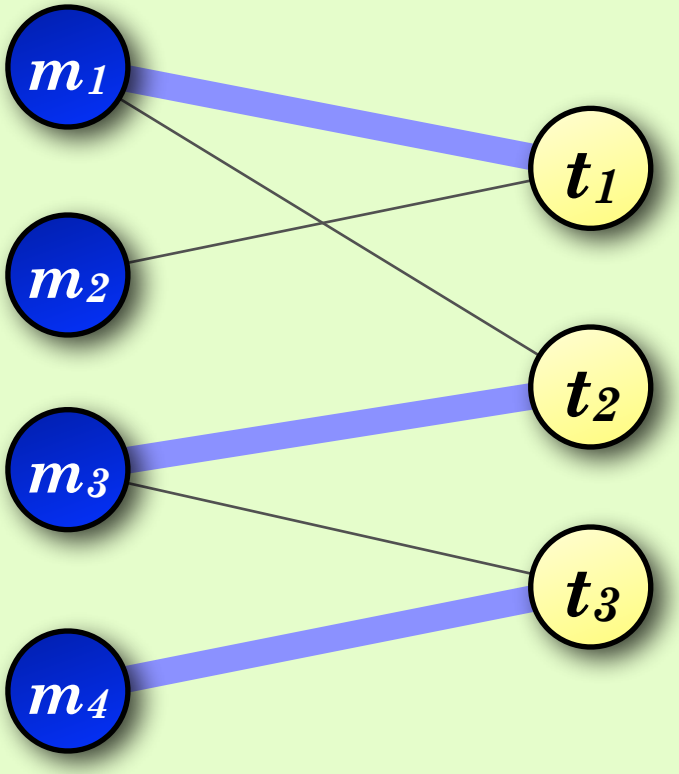


© 2006-2024 Jörg Striegnitz

Anwendungsbeispiel

B *Berechnung eines maximalen Matchings*

- Erhalte maximales Matching:



© 2006-2024 Jörg Striegnitz

VII. Graphen

7. Graphen

7.1. Einführung und Begriffsbestimmungen
7.2. Traversierung
7.3. Topologisches Sortieren
7.4. Kürzeste Wege
7.4.1. Einführung
7.4.2. *Single-Source Best Path* Algorithmen
7.4.3. *All-Pairs Best Path* Algorithmen
7.5. Maximaler Fluss
7.6. Minimaler Spannbaum

© 2006-2023 Jörg Striegnitz

D Baum / Spannbaum / minimaler Spannbaum

Sei $G = (V, E)$ ein beliebiger Graph.

- Ist G frei von Zyklen und zusammenhängend, so heißt G **Baum**. Für einen Knoten $v \in V$ nennt man (G, v) **gewurzelten Baum**.
- Ein **Spannbaum** $G_T = (V_T, E_T)$ von G ist ein Baum[a], der alle Knoten von G umfasst ($V_T = V$) und nur Kanten aus G enthält ($E_T \subseteq E$). Die Menge aller Spannbäume von G heißt $\mathcal{ST}(G)$.
- Ein **minimaler Spannbaum** $G_{T_{min}}$ von G ist ein Spannbaum mit minimalem Gewicht:

$$G_{T_{min}} \in \left\{ G_T \;\middle|\; \text{weight}(G_T) = \min_{G' \in \mathcal{ST}(G)} \text{weight}(G') \right\}$$

[a] somit ein zusammenhängender Graph

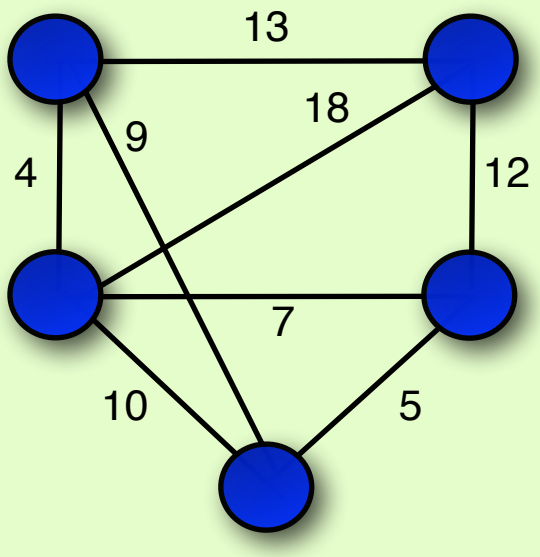
- **Englisch: *Minimal Spanning Tree* (MST)**

Siehe auch: zusammenhängender Graph gewichteter Graph

© 2006-2024 Jörg Striegnitz

MST - Beispiel 1

B Spannbaum / Minimaler Spannbaum

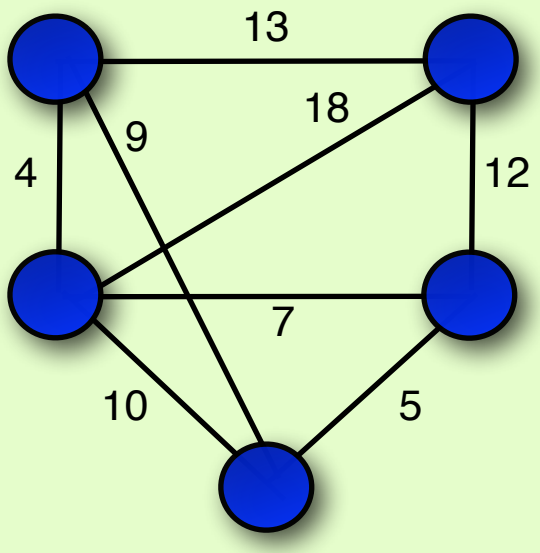


Graph G

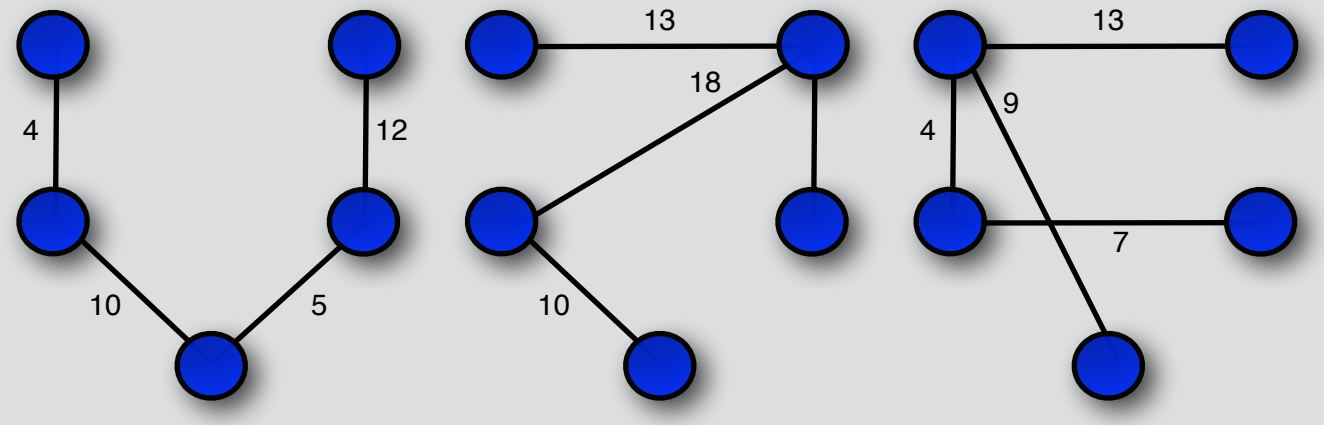
© 2006-2024 Jörg Striegnitz

MST - Beispiel 1

B Spannbaum / Minimaler Spannbaum



Graph G

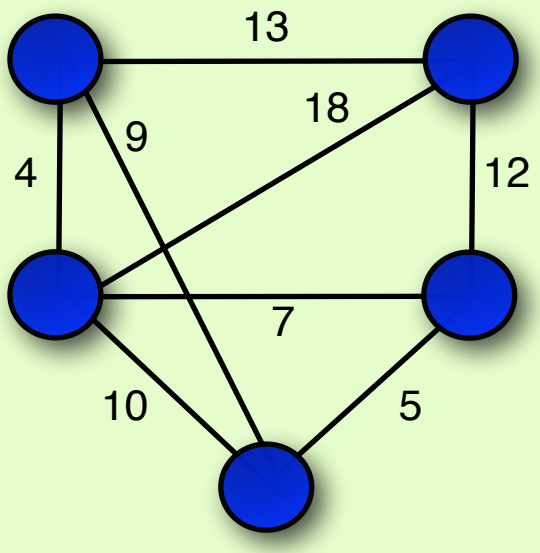


Beispiele für Spannbäume zu G

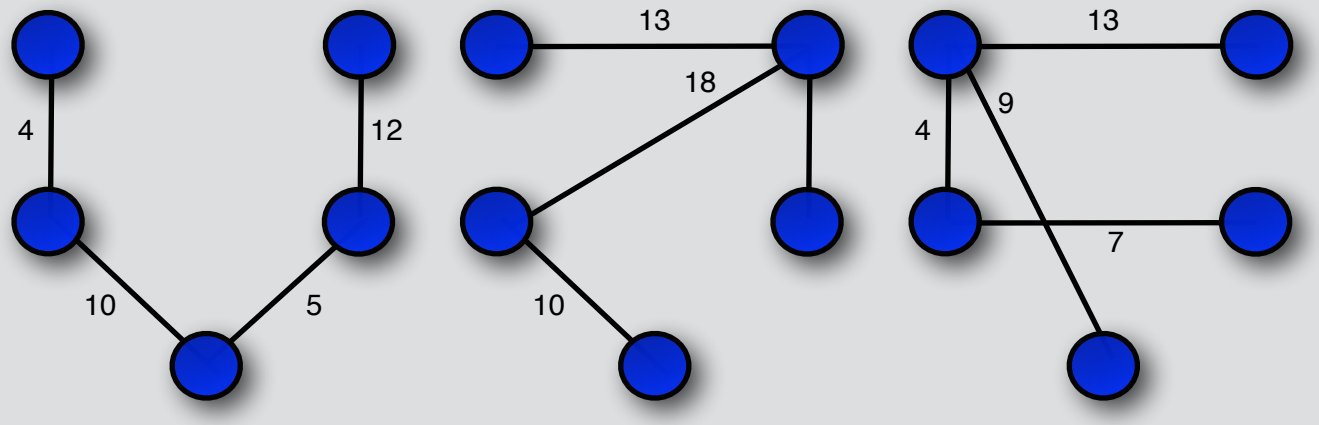
© 2006-2024 Jörg Striegnitz

MST - Beispiel 1

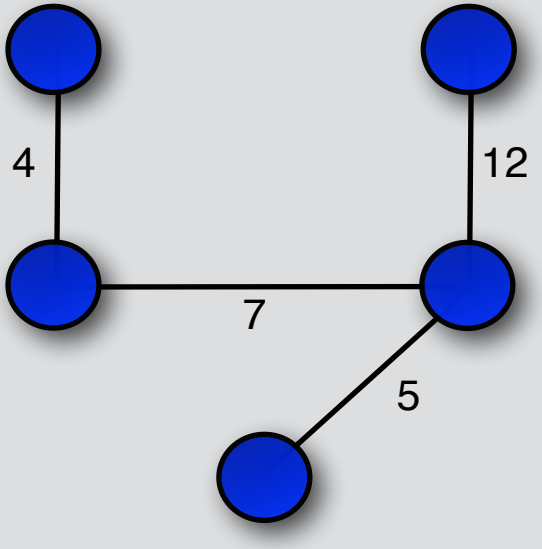
B Spannbaum / Minimaler Spannbaum



Graph G



Beispiele für Spannbäume zu G



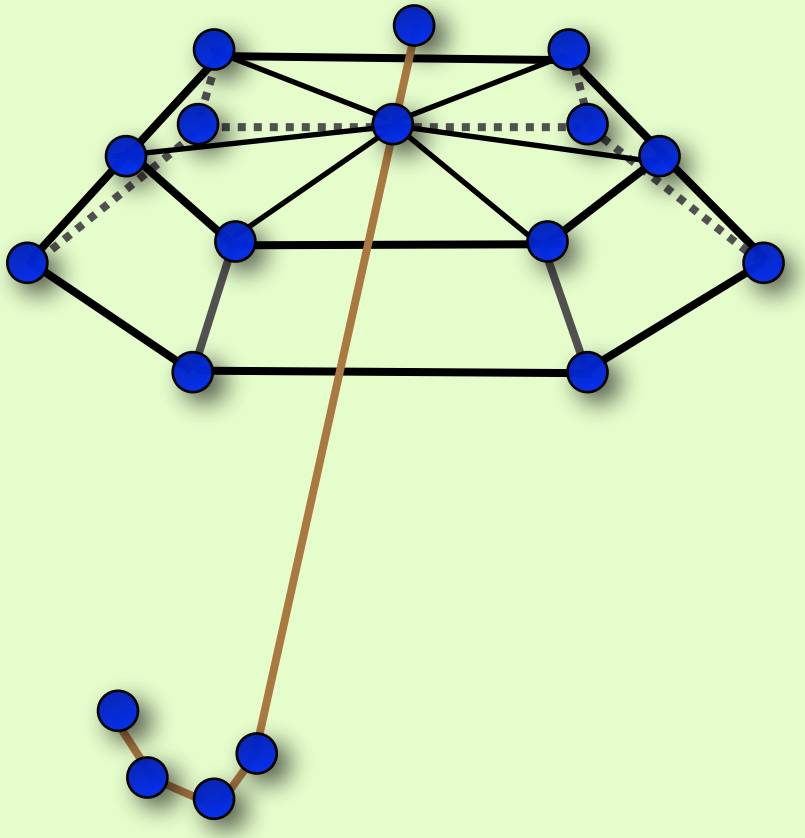
minimaler Spannbaum zu G

mit Gewicht 28

© 2006-2024 Jörg Striegnitz

MST - Beispiel 2

B Spannbaum

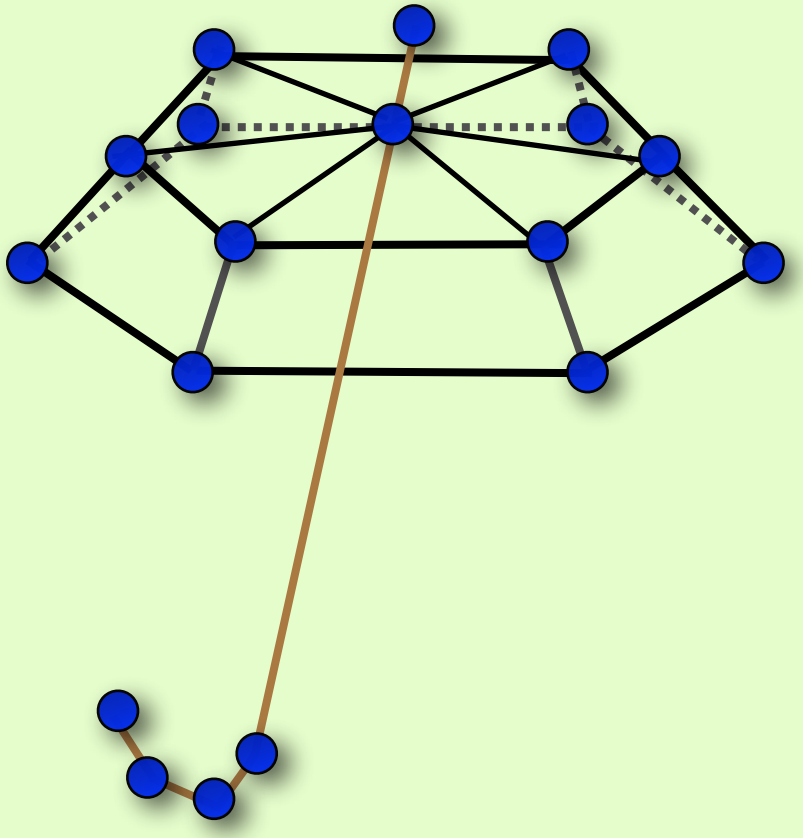


Graph G

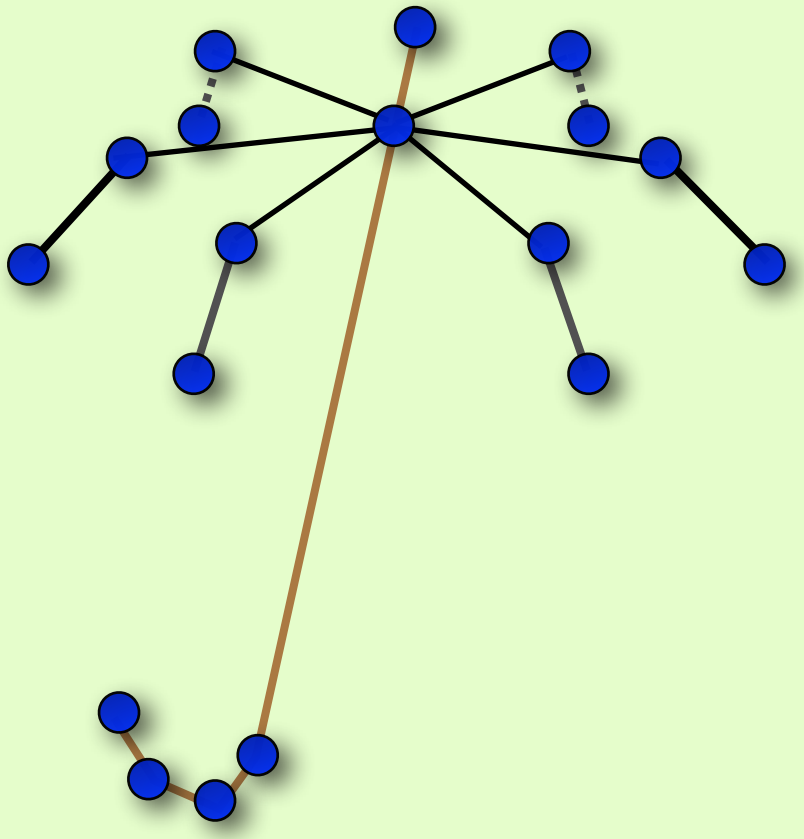
© 2006-2024 Jörg Striegnitz

MST - Beispiel 2

B Spannbaum



Graph G

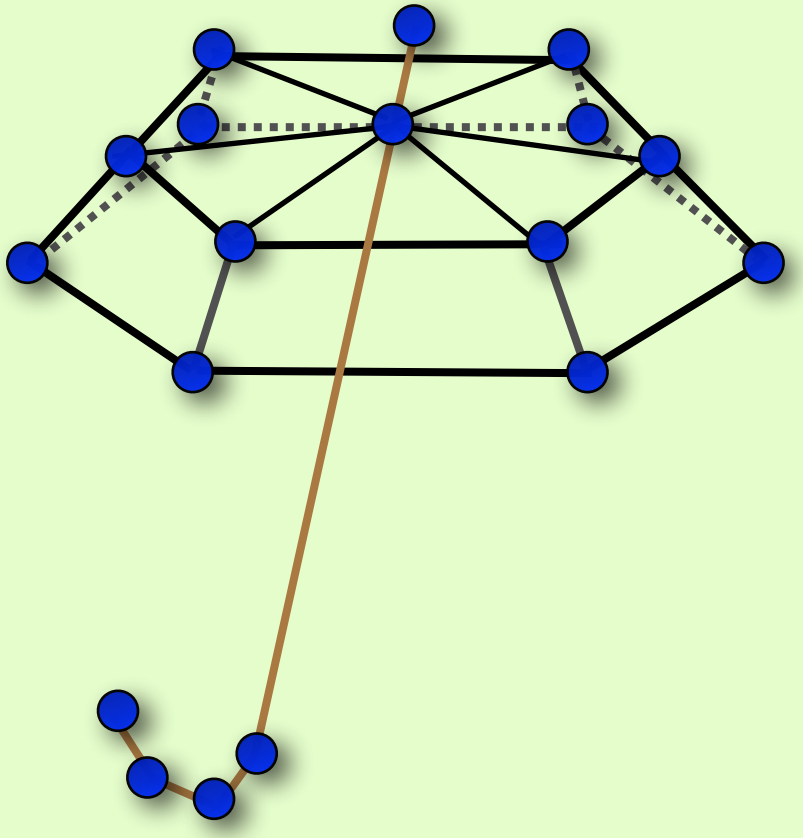


Ein Spannbaum zu G

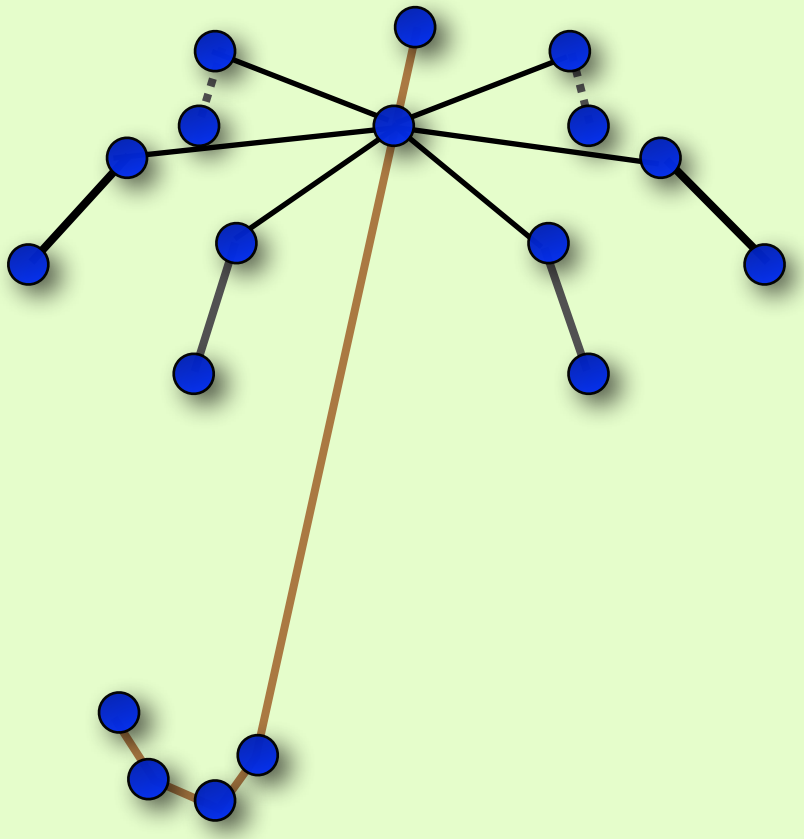
© 2006-2024 Jörg Striegnitz

MST - Beispiel 2

B **Spannbaum**



Graph G



Ein Spannbaum zu G

- **Zahlreiche Anwendungen;** z.B.:
  - elektrische Leitungen (minimiere Kabellänge)
  - Wasserversorgung
  - Grundlage für Approximationsalgorithmen (später)

© 2006-2024 Jörg Striegnitz

Brute Force-Methode scheidet aus

- ***Brute-Force*-Algorithmus:** Teste alle Spannbäume
- **Wie viele aufspannende Bäume hat ein Graph?**
  Die ist ein schweres Problem, aber das folgende Ergebnis für vollständige Graphen gibt einen Hinweis, auf was man sich einläßt:

© 2006-2024 Jörg Striegnitz

Brute Force-Methode scheidet aus

- ***Brute-Force*-Algorithmus:** Teste alle Spannbäume
- **Wie viele aufspannende Bäume hat ein Graph?**
  Die ist ein schweres Problem, aber das folgende Ergebnis für vollständige Graphen gibt einen Hinweis, auf was man sich einläßt:

S Satz von Caley

Sei $G = (V, E)$ ein vollständiger Graph mit $|V| = n$. Dann gibt es in G

$$n^{n-2}$$

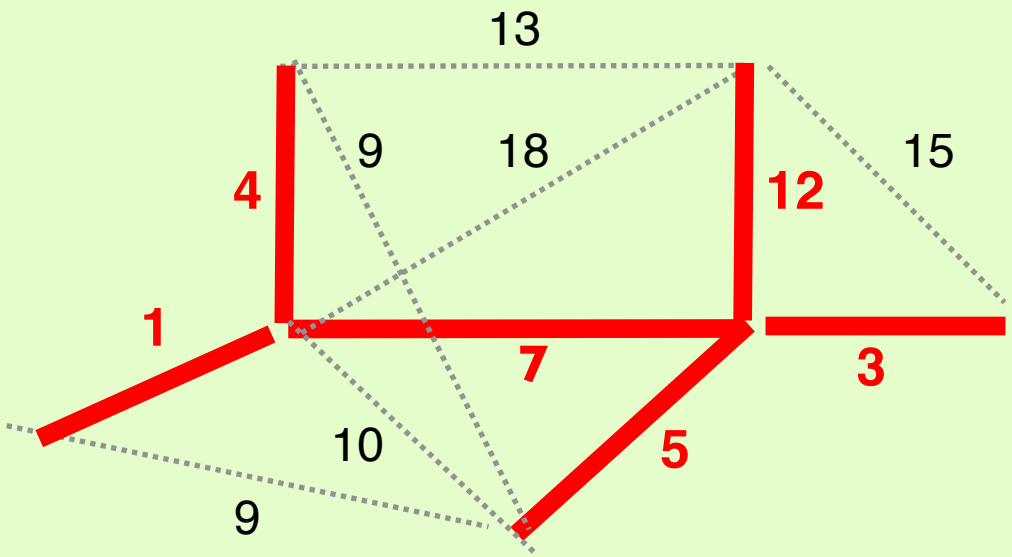
aufspannende Bäume.

Ohne Beweis …

© 2006-2024 Jörg Striegnitz

Charakterisierung der optimalen Lösung 1

B **Minimaler Spannbaum -** Zerlegung der optimalen Lösung

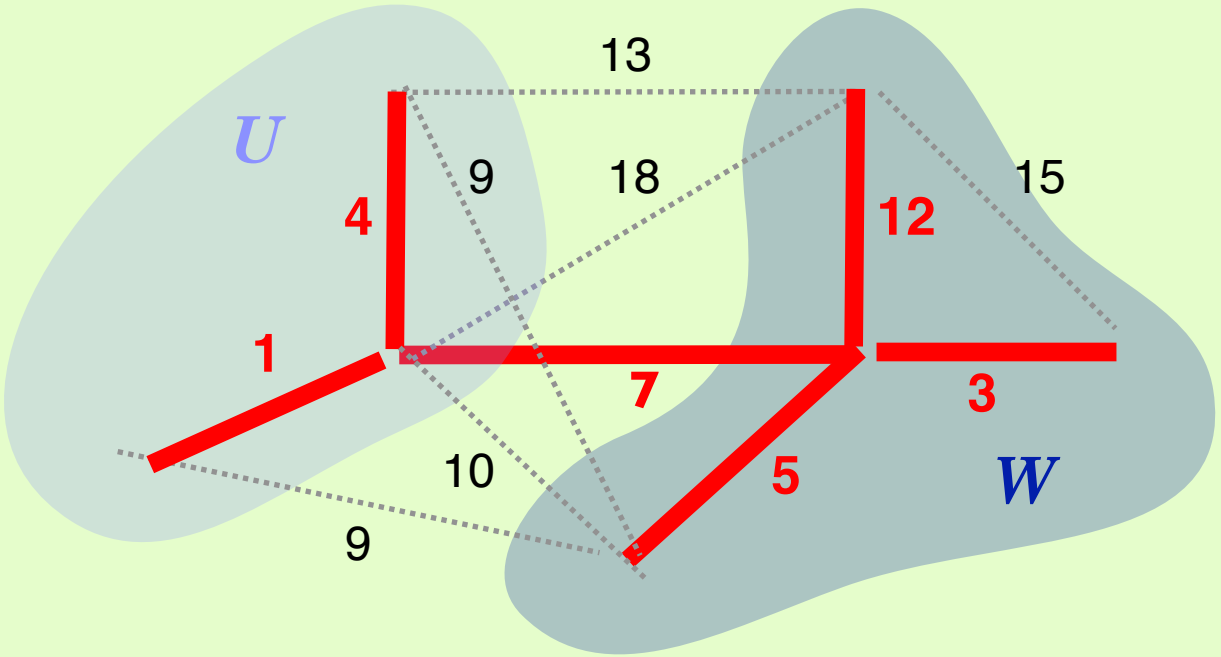


MST G_T hat Gewicht **32**

© 2006-2024 Jörg Striegnitz

Charakterisierung der optimalen Lösung 2

B **Minimaler Spannbaum -** Zerlegung der optimalen Lösung

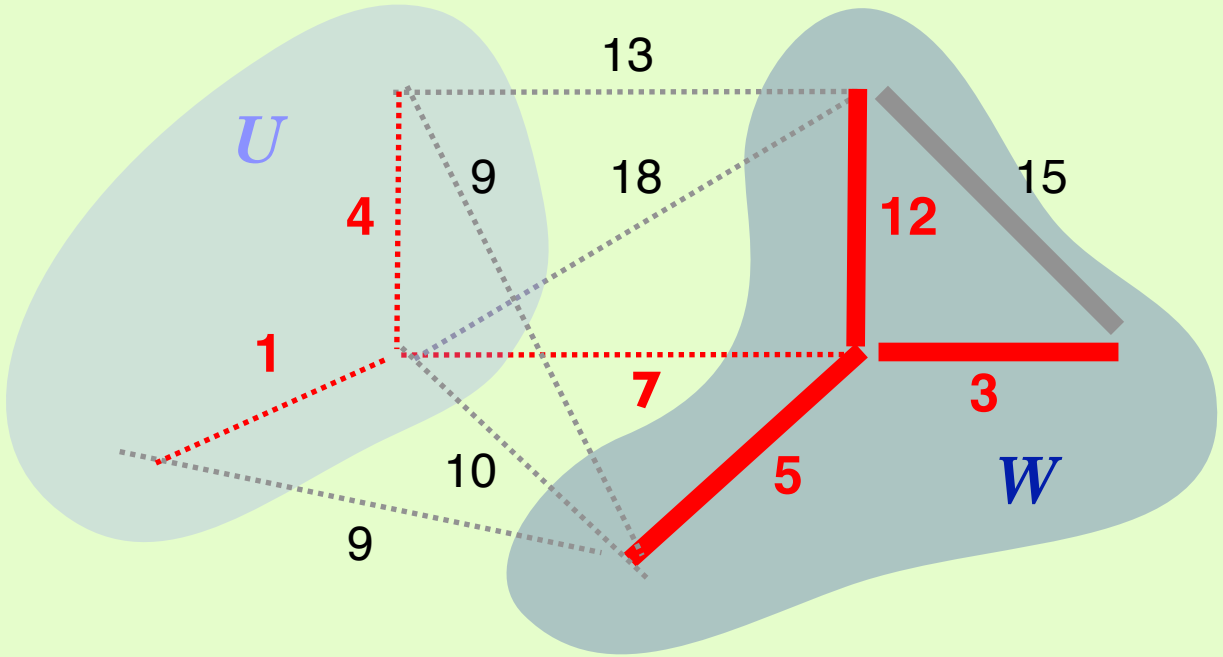


Zerteile Knoten in disjunkte Mengen U und W

© 2006-2024 Jörg Striegnitz

Charakterisierung der optimalen Lösung 3

B **Minimaler Spannbaum -** Zerlegung der optimalen Lösung

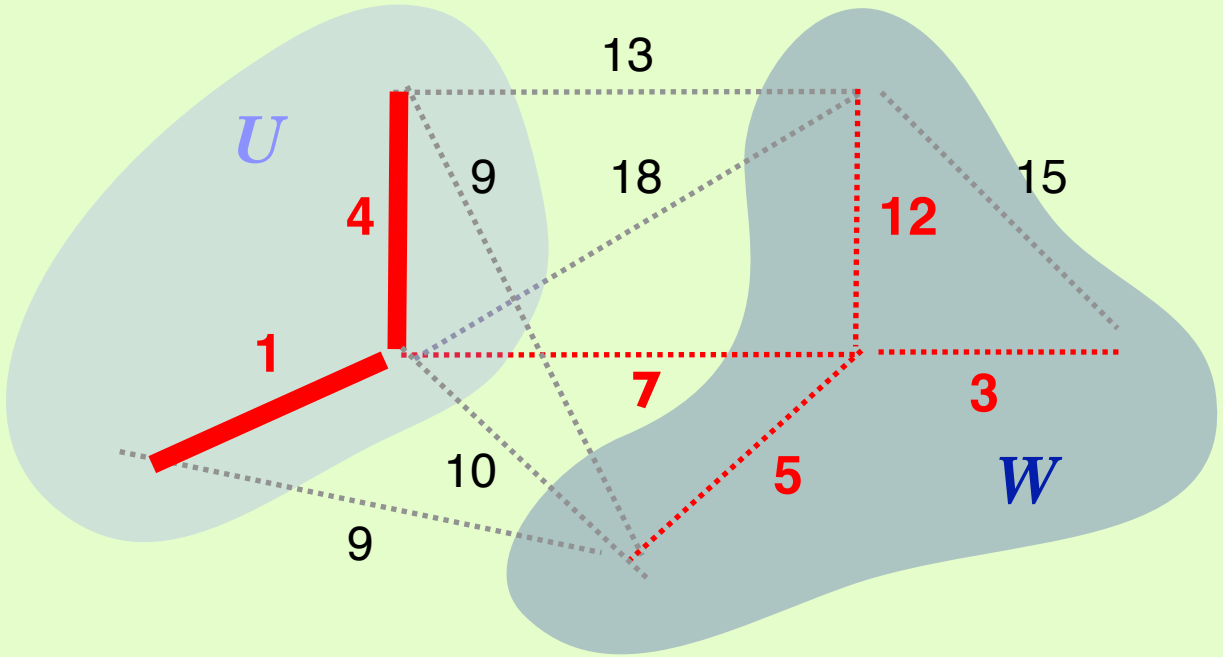


Betrachte $G[W]$: dort ist $G_T[W]$ MST!

© 2006-2024 Jörg Striegnitz

Charakterisierung der optimalen Lösung 4

B **Minimaler Spannbaum -** Zerlegung der optimalen Lösung

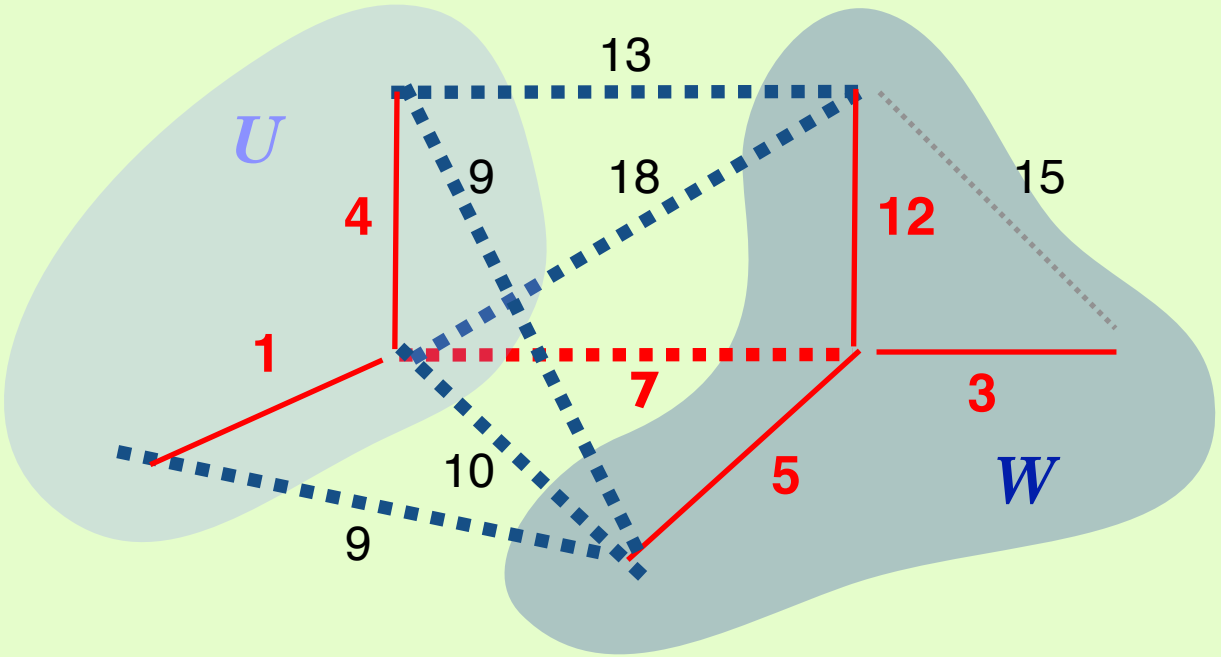


Betrachte $G[U]$: dort ist $G_T[U]$ MST!

© 2006-2024 Jörg Striegnitz

Charakterisierung der optimalen Lösung 5

B **Minimaler Spannbaum -** Zerlegung der optimalen Lösung



G_T entsteht $G_T[U]$ und $G_T[W]$
durch Wahl der günstigsten Verbindungskante

© 2006-2024 Jörg Striegnitz

Charakterisierung der optimalen Lösung 1

S Charakteristik minimaler Spannbäume

Sei $G = (V, E)$ ein Graph und $G_T = (V, E_T)$ ein minimaler Spannbaum zu G. Eine Kante $e = (u_0, w_0) \in E_T$ teilt die Knotenmenge V wie folgt in zwei disjunkte Mengen U und W auf:

$$\begin{aligned} U &= \{u_n \mid u_0, \cdots, u_n \text{ ist ein Pfad in } G_T \text{ mit } (u_i, u_{i+1}) \in E_T \setminus \{e\}\} \\ W &= \{w_n \mid w_0, \cdots, w_n \text{ ist ein Pfad in } G_T \text{ mit } (w_i, w_{i+1}) \in E_T \setminus \{e\}\} \end{aligned}$$

mit jeweils $0 \leq i < n$. Dann ist die Kante e die günstigste Kante, welche die Knotenmengen U und W verbindet:

$$\text{weight}(e) = \min_{u \in U, w \in W} \text{weight}(u, w)$$

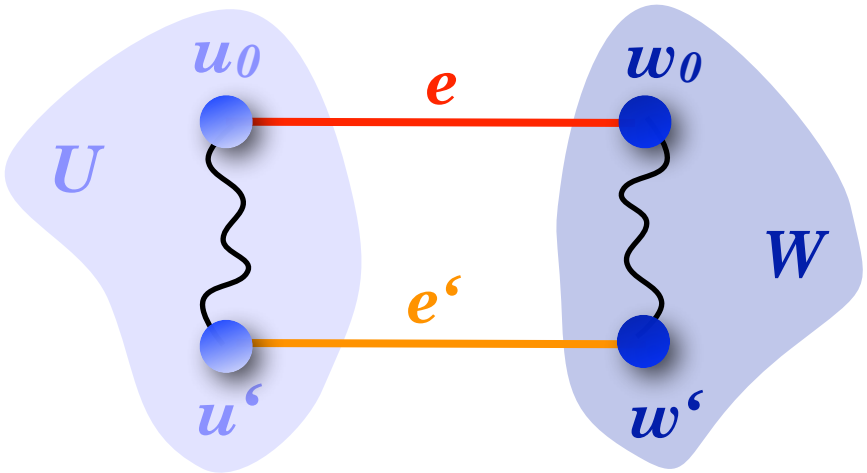
wobei $\text{weight}(u, w) = \infty$, falls $(u, w) \notin E$.

- **Unter Vernachlässigung der Kante e**
  - enthält U alle von u_0 erreichbaren Knoten
  - enthält W alle von v_0 erreichbaren Knoten

© 2006-2024 Jörg Striegnitz

Beweis: Charakteristik minimaler Spannbäume

Beweis: Sei $G_T = (V, \overline{E_T})$ ein minimaler Spannbaum zu G, der e nicht enthält. Dann gibt es in G_T mindestens eine Kante, vielleicht aber auch mehrere Kanten, die U mit W verbinden.
Wenn wir zu G_T die Kante $e = (u_0, w_0)$ hinzufügen, entsteht ein Zyklus, der über e läuft - es muss daher noch einen anderen Weg von u zu w geben, der eine Kante e' enthält, welche U und W verbindet:



Löscht man nun e' aus G_T und fügt stattdessen e ein, so entsteht ein MST, dessen Kosten höchstens so hoch sind, wie die von G_T, da e minimale Kosten hat.

© 2006-2024 Jörg Striegnitz

Rekursionsgleichung

Seien $G = (V, E)$ ein Graph und $V = \{v_1, \ldots, v_n\} = U \cup W$ mit $U \cap W = \emptyset$ so wie $X = ((U \times V) \cup (V \times U)) \cap E$. Dann liefert die Funktion **minEdge**(U, W) mit

$$\textbf{minEdge}(U, W) = \min_{(a,b) \in X} \text{weight}(a, b)$$

das geringste Gewicht einer Kante, die Knoten aus U und W verbindet.

© 2006-2024 Jörg Striegnitz

Rekursionsgleichung

Seien $G = (V, E)$ ein Graph und $V = \{v_1, \ldots, v_n\} = U \cup W$ mit $U \cap W = \emptyset$ so wie $X = ((U \times V) \cup (V \times U)) \cap E$. Dann liefert die Funktion **minEdge**(U, W) mit

$$\textbf{minEdge}(U, W) = \min_{(a,b) \in X} \text{weight}(a, b)$$

das geringste Gewicht einer Kante, die Knoten aus U und W verbindet.

Ferner ist

$$\mathcal{G}_0 = \{G_1 = G[\{v_1\}], \cdots, G_n = G[\{v_n\}]\}$$

ein Wald von minimalen Spannbäumen zu Teilgraphen von G: G_i ist minimaler Spannbaum zu $G[\{v_i\}]$ $(1 \leq i \leq n)$.

© 2006-2024 Jörg Striegnitz

Rekursionsgleichung

Seien $G = (V, E)$ ein Graph und $V = \{v_1, \ldots, v_n\} = U \cup W$ mit $U \cap W = \emptyset$ so wie $X = ((U \times V) \cup (V \times U)) \cap E$. Dann liefert die Funktion **minEdge**(U, W) mit

$$\textbf{minEdge}(U, W) = \min_{(a,b) \in X} \text{weight}(a, b)$$

das geringste Gewicht einer Kante, die Knoten aus U und W verbindet.

Ferner ist

$$\mathcal{G}_0 = \{G_1 = G[\{v_1\}], \cdots, G_n = G[\{v_n\}]\}$$

ein Wald von minimalen Spannbäumen zu Teilgraphen von G: G_i ist minimaler Spannbaum zu $G[\{v_i\}]$ $(1 \leq i \leq n)$.

Betrachte

$$\mathcal{G}_m = \{G_1 = (V_1, E_1), \cdots, G_k = (V_k, E_k)\}$$

wobei G_i minimaler Spannbaum zu $G[V_i]$ ist $(1 \leq i \leq k)$. Wähle eine Kante

$$(v_i, v_j) \in \left\{ e \;\middle|\; \text{weight}(e) = \min_{1 \leq i < j \leq n} \text{minEdge}(V_i, V_j) \right\}$$

Offenbar ist $G_{ij} = (V_i \cup V_j, E_i \cup E_j \cup \{(v_i, v_j)\})$ minimaler Spannbaum zu $G[V_i \cup V_j]$ - wir erhalten:

$$\mathcal{G}_{m+1} = \mathcal{G}_m - \{G_i, G_j\} \cup \{G_{ij}\}$$

Wie man leicht nachvollzieht, enthält $\mathcal{G}_{n-1}$ genau einen Graphen, der MST zu G ist.

© 2006-2024 Jörg Striegnitz

Algorithmus von Kruskal

Initialisiere $\mathcal{G}$ *(* $\mathcal{G}$ ist Wald der kleinsten MSTs *)*

$Q \leftarrow E(G)$ *(* Alle Kanten von G in die Priority-Queue *)*

while $|\mathcal{G}| > 1$

do $(u,w) \leftarrow \min Q$ *(* Kante mit geringsten Gewicht auswählen ... *)*

$Q \leftarrow Q \setminus \{(u,w)\}$ *(* ... und aus Queue entfernen *)*

$G_1 = find(u, \mathcal{G})$ *(* G_1 ist zu u gehöriger MST *)*

$G_2 = find(w, \mathcal{G})$ *(* G_2 ist zu w gehöriger MST *)*

if $(G_1 \neq G_2)$ *(* (u,w) verbindet zwei verschiedene MSTs ? *)*

(G_T ist neuer MST ... *)*

$G_T \leftarrow (\ V(G_1) \cup V(G_2),\ \ E(G_1) \cup E(G_2) \cup \{(u,w)\}\ )$

$\mathcal{G} = (\mathcal{G} - \{G_1, G_2\}) \cup \{G_T\}$

endif

end while

return G_T **wobei** $\mathcal{G} = \{G_T\}$

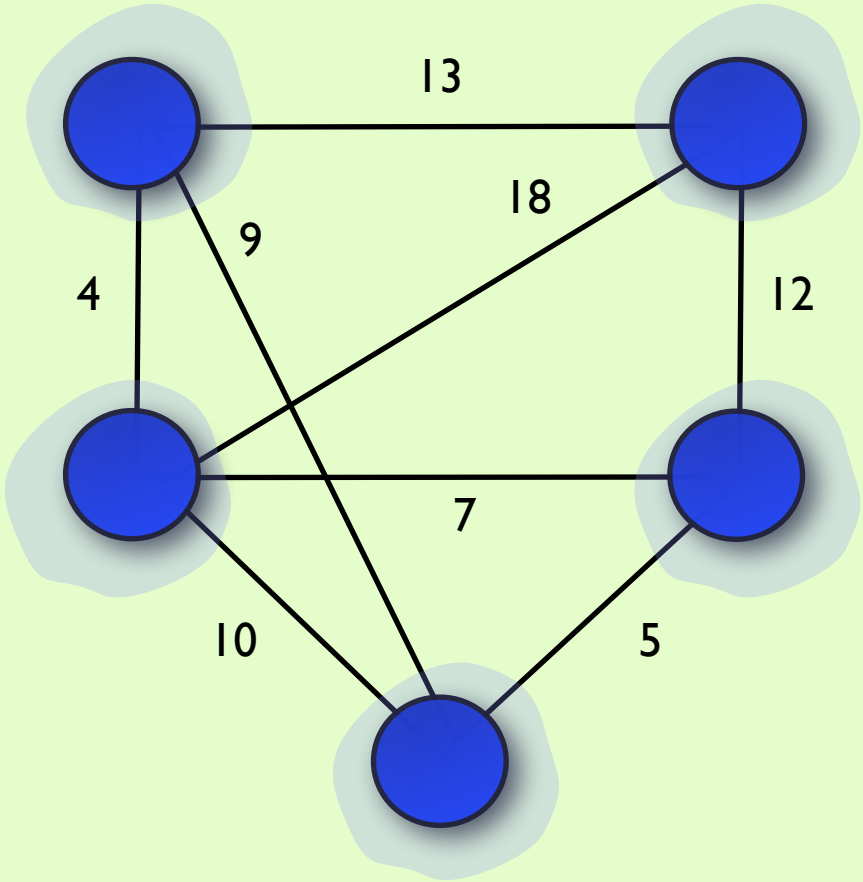
Jospeh Bernard Kruskal

★1929

© 2006-2024 Jörg Striegnitz

MST - Berechnung (Kruskal)

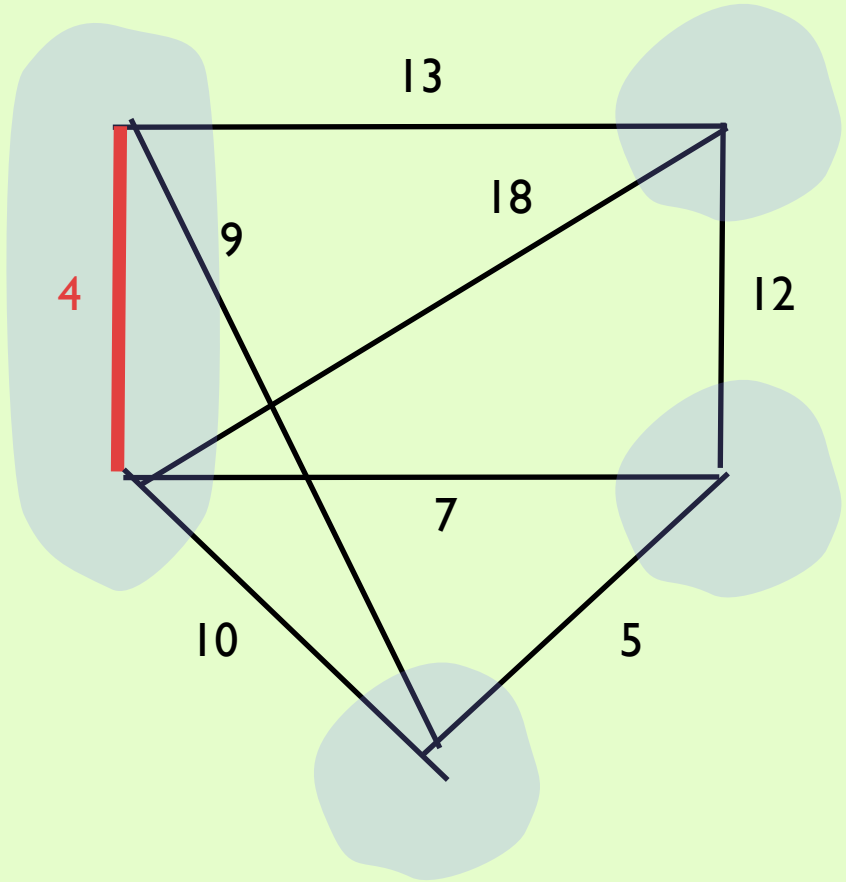
B Minimaler Spannbaum



© 2006-2024 Jörg Striegnitz

MST - Berechnung (Kruskal)

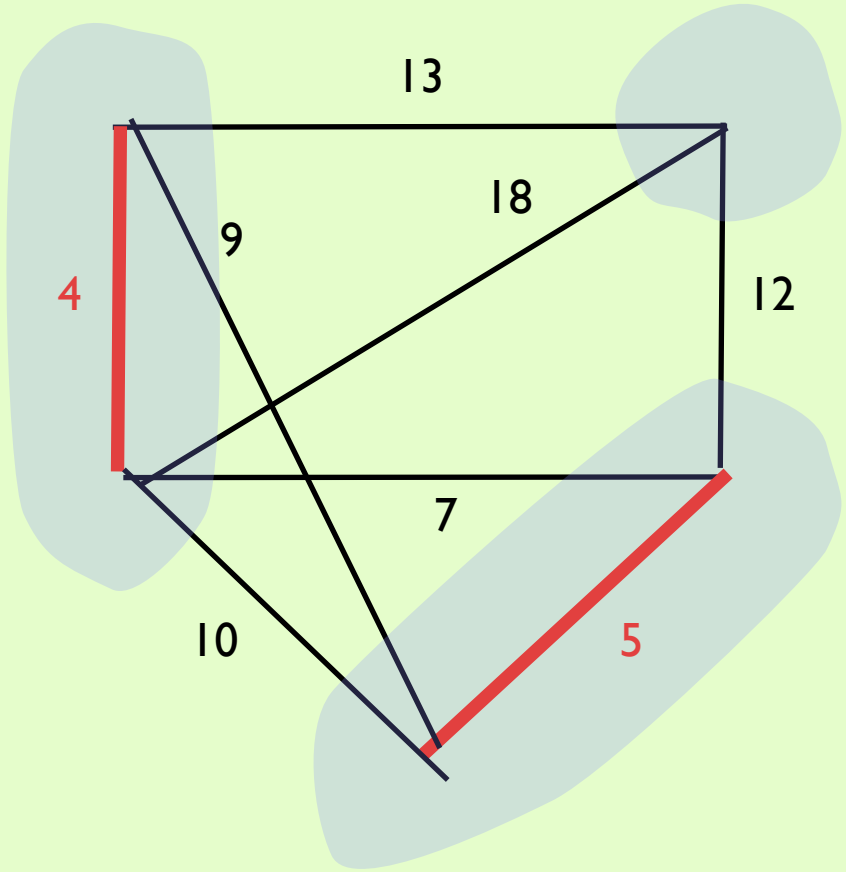
B Minimaler Spannbaum



© 2006-2024 Jörg Striegnitz

MST - Berechnung (Kruskal)

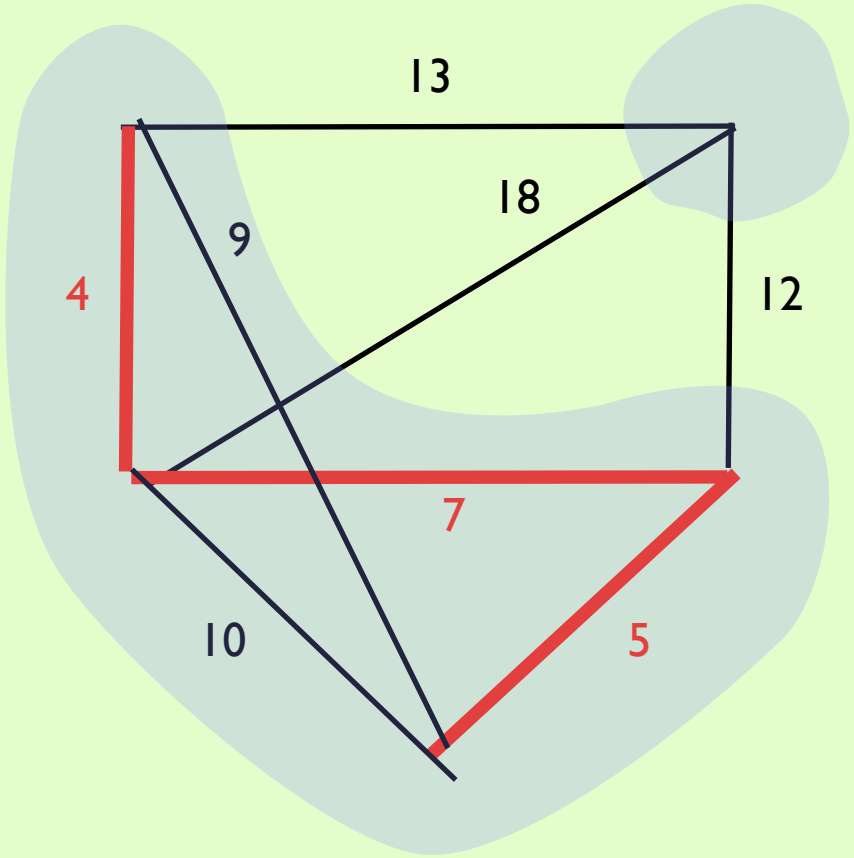
B Minimaler Spannbaum



© 2006-2024 Jörg Striegnitz

MST - Berechnung (Kruskal)

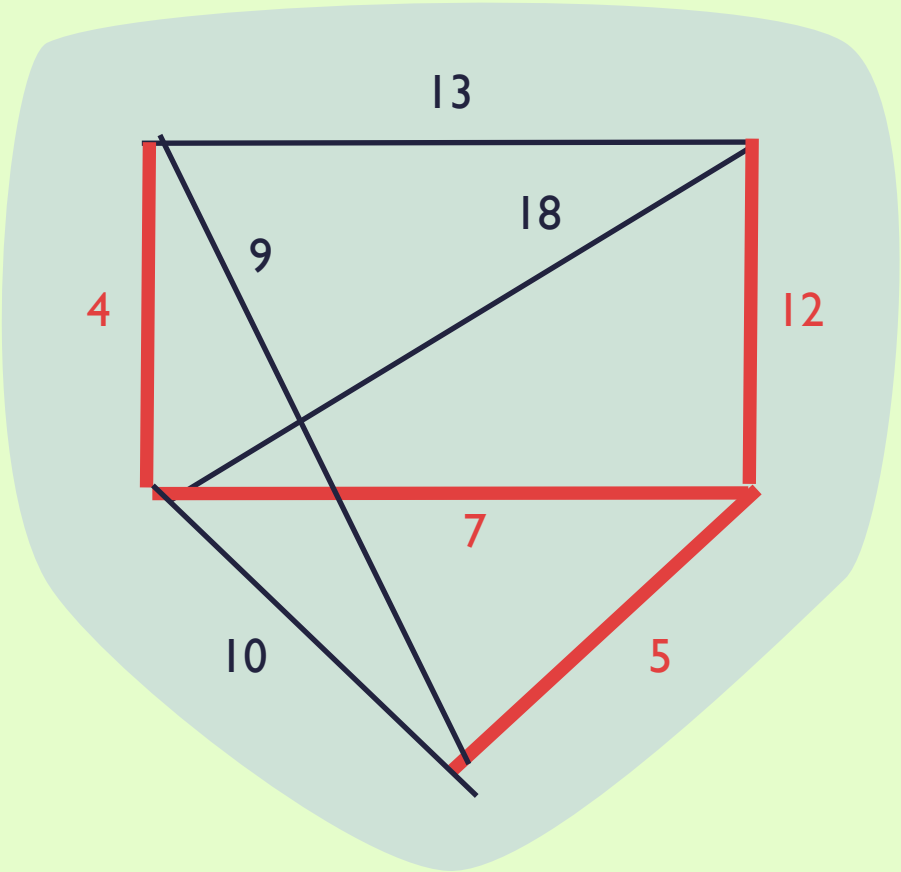
B Minimaler Spannbaum



© 2006-2024 Jörg Striegnitz

MST - Berechnung (Kruskal)

B Minimaler Spannbaum



© 2006-2024 Jörg Striegnitz

Algorithmus von Kruskal

Initialisiere $\mathcal{G}$ *(* $\mathcal{G}$ ist Wald der kleinsten MSTs *)*

$Q \leftarrow E(G)$ *(* Alle Kanten von G in die Priority-Queue *)*

while $|\mathcal{G}| > 1$

do $(u,w) \leftarrow \min Q$ *(* Kante mit geringsten Gewicht auswählen .*

$Q \leftarrow Q \setminus \{(u,w)\}$ *(* ... und aus Queue entfernen*

$G_1 = find(u, \mathcal{G})$ *(* G_1 ist zu u gehöriger MST *)*

$G_2 = find(w, \mathcal{G})$ *(* G_2 ist zu w gehöriger MST *)*

if $(G_1 \neq G_2)$ *(* (u,w) verbindet zwei verschiedene MSTs ? *)*

(G_T ist neuer MST ... *)*

$G_T \leftarrow (\ V(G_1) \cup V(G_2),\ \ E(G_1) \cup E(G_2) \cup \{(u,w)\}\ )$

$\mathcal{G} = (\mathcal{G} - \{G_1, G_2\}) \cup \{G_T\}$

endif

end while

return G_T wobei $\mathcal{G}=\{G_T\}$

Jospeh Bernard Kruskal
★1929

© 2006-2024 Jörg Striegnitz

Algorithmus von Kruskal

Initialisiere $\mathcal{G}$ *(* $\mathcal{G}$ ist Wald der kleinsten MSTs *)*

$Q \leftarrow E(G)$ *(* Alle Kanten von G in die Priority-Queue *)*

while $|\mathcal{G}| > 1$

do $(u,w) \leftarrow \min Q$ *(* Kante mit geringsten Gewicht auswählen ...*

$Q \leftarrow Q \setminus \{(u,w)\}$ *(* ... und aus Queue entfernen*

$G_1 = find(u, \mathcal{G})$ *(* G_1 ist zu u gehöriger MST *)*

$G_2 = find(w, \mathcal{G})$ *(* G_2 ist zu w gehöriger MST *)*

if $(G_1 \neq G_2)$ ~~*(* (u,w) verbindet zwei verschiedene MSTs ? *)*~~

(G_T ist neuer MST ... *)*

$G_T \leftarrow (\ V(G_1) \cup V(G_2),\ \ E(G_1) \cup E(G_2) \cup \{(u,w)\}\ )$

$\mathcal{G} = (\mathcal{G} - \{G_1, G_2\}) \cup \{G_T\}$

endif

end while

return G_T wobei $\mathcal{G} = \{G_T\}$

Jospeh Bernard Kruskal
★1929

Effiziente Implementierung ?

© 2006-2024 Jörg Striegnitz

Vorüberlegung

- Ergebnisgraph G_T gesondert konstruieren - dann reicht es, sich die Knotenmengen der einzelnen Graphen in $\mathcal{G}$ zu merken:

Initialisiere $\mathcal{G}$ (* $\mathcal{G}$ enthält alle Mengen $V_1=\{v_1\}$ bis $V_n=\{v_n\}$ *)

$Q \leftarrow E(G)$ (* Alle Kanten von G in die Priority-Queue *)

$G_T \leftarrow (V,\emptyset)$ (* Ergebnisgraph *)

while $|\mathcal{G}| > 1$

do $(u,w) \leftarrow \min Q$ (* Kante mit geringsten Gewicht auswählen ... *)

$Q \leftarrow Q \setminus \{(u,w)\}$ (* ... und aus Queue entfernen *)

$V_1 = find(u, \mathcal{G})$ (* V_1 ist Knotenmenge des zu u gehöriger MST *)

$V_2 = find(w, \mathcal{G})$ (* V_2 ist Knotenmenge des zu w gehöriger MST *)

if $(V_1 \neq V_2)$ (* (u,w) verbindet zwei verschiedene MSTs ? *)

(* G_T ist neuer MST ... *)

$G_T \leftarrow (V, E(G_T) \cup \{(u,w)\})$

$\mathcal{G} = merge(\mathcal{G}, V_1, V_2)$ (* $= (\mathcal{G} - \{V_1, V_2\}) \cup \{\{V_1 \cup V_2\}\}$ *)

endif

end while

return G_T

$\mathcal{G}$ ist Menge disjunkter Mengen

© 2006-2024 Jörg Striegnitz

Disjunkte Mengen - Operationen

- **Gesucht:** Darstellung einer Menge disjunkter Mengen $\mathcal{G}$, welche die folgenden Operationen möglichst effizient unterstützt:
  - **find(x, $\mathcal{G}$)**: finde die Teilmenge in $\mathcal{G}$, welche das Element x enthält.
  - **join($\mathcal{G}$,x,y)**: Bilde die Vereinigung der Mengen aus $\mathcal{G}$, die x bzw. y enthalten:

$$\text{join}(\mathcal{G},x,y) = \text{find}(x, \mathcal{G}) \cup \text{find}(y, \mathcal{G})$$

  - **merge($\mathcal{G}$,x,y)**: vereinfache $\mathcal{G}$, wie folgt;

$$\text{merge}(\mathcal{G},x,y) = (\mathcal{G} - \text{find}(x, \mathcal{G}) - \text{find}(y, \mathcal{G})) \cup \text{join}(\mathcal{G},x,y)$$

  - **compare(X, Y)**: Test zweier Mengen auf Gleichheit.

Vermeidung komplexer Mengenoperationen (z.B. **compare** oder **merge**):

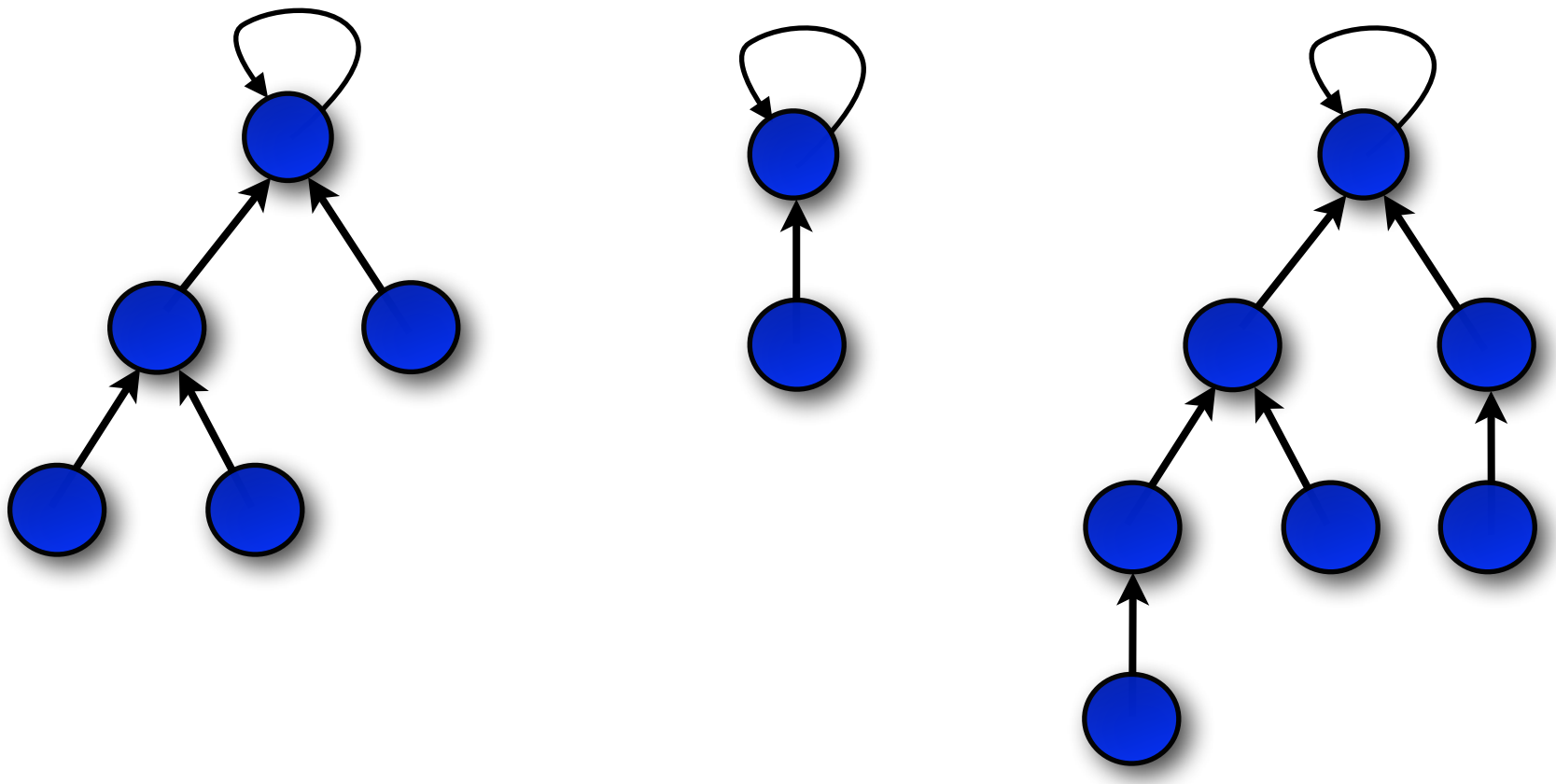
ordne jeder Teilmenge einen eindeutigen Stellvertreter zu

und arbeite wo möglich mit Stellvertreter (z.B. **find** liefert Stellvertreter, keine Menge; **compare** vergleicht Stellvertreter, keine Mengen).

© 2006-2024 Jörg Striegnitz

Darstellung disjunkter Mengen (Idee)

- Darstellung einer Menge als „**umgedrehter Baum**“
- An dessen Wurzel steht der **eindeutige Stellvertreter** der Menge
- Eine Menge von Teilmengen ist dann ein Wald:

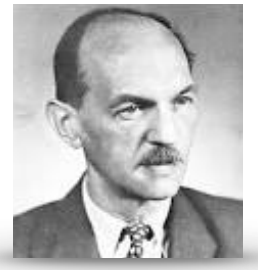


{ {1,2,3,4,5}, {6,7}, {8,9,10,11,12,13,14} }

© 2006-2024 Jörg Striegnitz

Algorithmus von Prim

- **1930 Entdeckt von Vojtěch Jarnik**
  - daher auch Jarnik-Algorithmus genannt
  - von Robert C. Prim 1957 wiederentdeckt



Vojtěch Jarnik

Robert Clay Prim

- **Gemeinsamkeiten mit Dijkstra-Algorithmus**
  - speichere aktuell minimale Kosten, mit denen Knoten an Spannbaum angebunden werden kann, in Feld **k**
  - Knoten in Priority-Queue (sortiert nach Feld **k**)

- **Erzeuge sukzessive grösseren MST**
  - im Vergleich zur Kruskal: **ein** MST (wird sukzessive optimiert), nicht mehrere MSTs, die zu größerem verschmolzen werden

© 2006-2024 Jörg Striegnitz

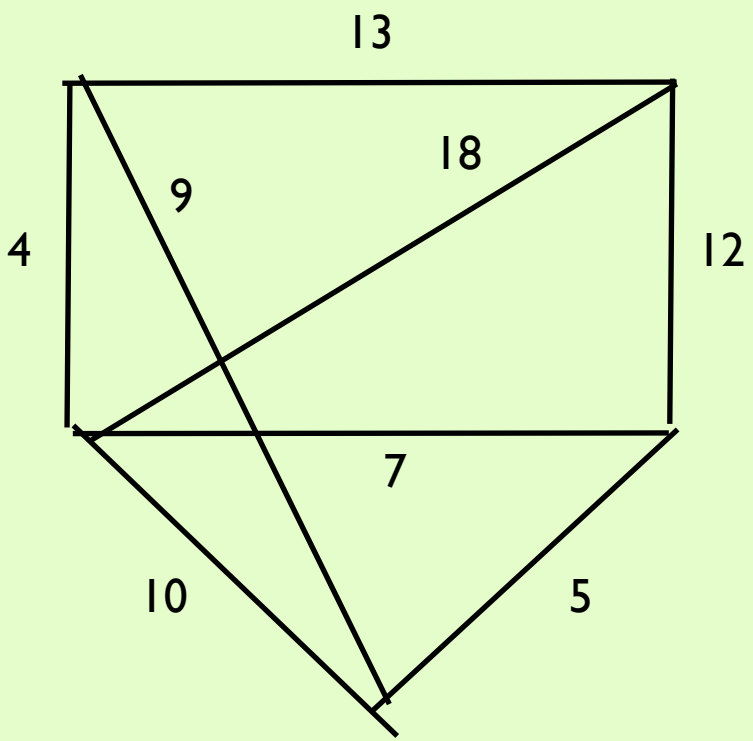
Algorithmus von Prim

for alle $u \in V(G)$
  do k[u] $\leftarrow \infty$ *// kleinstes Kantengewicht aller Kanten, die u mit*
  // irgendeinem Knoten des Baums verbinden
    $\pi[u] \leftarrow nil$ *// ist Vater von u im Baum*
end for
k[r] ← 0 *// r ist Startknoten*
$\boldsymbol{Q} \leftarrow V(G)$
while $Q \neq \emptyset$
  do $u \leftarrow$ min Q *// minimaler Knoten bzgl. k*
    $\boldsymbol{Q} \leftarrow \boldsymbol{Q} \setminus \{u\}$ *// aus Queue entfernen*
    for alle $v \in Adj(u)$ *// alle Knoten, die zu u benachbart sind*
      do if $(v \in Q)$ und $(w(u,v) < \mathrm{k}[v])$
        then
          $\pi[v] \leftarrow u$
          $\mathrm{k}[v] \leftarrow w(u,v)$
        end if
      end for
  end while

© 2006-2024 Jörg Striegnitz

Algorithmus von Prim - 2

B Minimaler Spannbaum



	1	2	3	4	5	Q
k	0	∞	∞	∞	∞	
π	nil	nil	nil	nil	nil	1,2,3,4,5

Kante des aktuellen MST

keine Erweiterung / Verbesserung

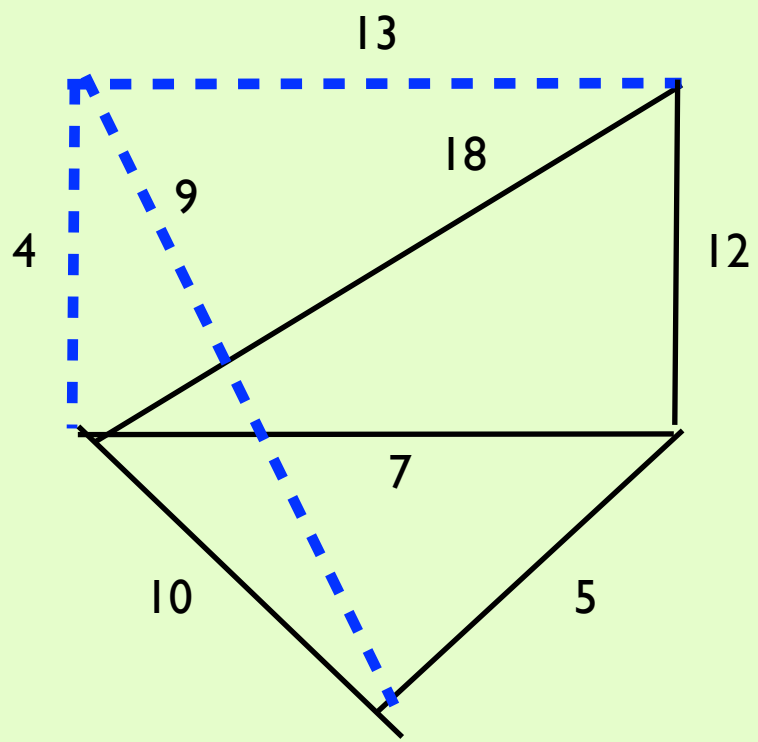
Erweiterung / Verbesserung

Kante wird in dieser Iteration aus MST entfernt

© 2006-2024 Jörg Striegnitz

Algorithmus von Prim - 2

B Minimaler Spannbaum



	1	2	3	4	5	Q
k	0	∞	∞	∞	∞	
π	nil	nil	nil	nil	nil	1,2,3,4,5
	0	**13**	**4**	∞	**9**	3,5,2,4
	nil	**1**	**1**	nil	**1**	

Kante des aktuellen MST

keine Erweiterung / Verbesserung

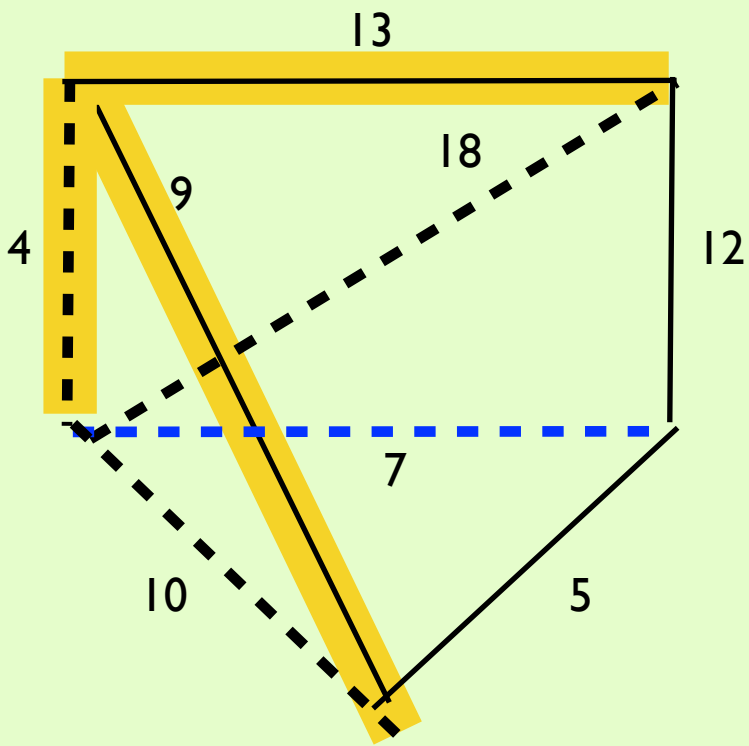
Erweiterung / Verbesserung

Kante wird in dieser Iteration aus MST entfernt

© 2006-2024 Jörg Striegnitz

Algorithmus von Prim - 2

B Minimaler Spannbaum



	1	2	3	4	5	Q
k π	0 nil	∞ nil	∞ nil	∞ nil	∞ nil	1,2,3,4,5
	0 nil	13 1	4 1	∞ nil	9 1	3,5,2,4
	0 nil	13 1	4 1	**7** **3**	9 1	4,5,2

Kante des aktuellen MST

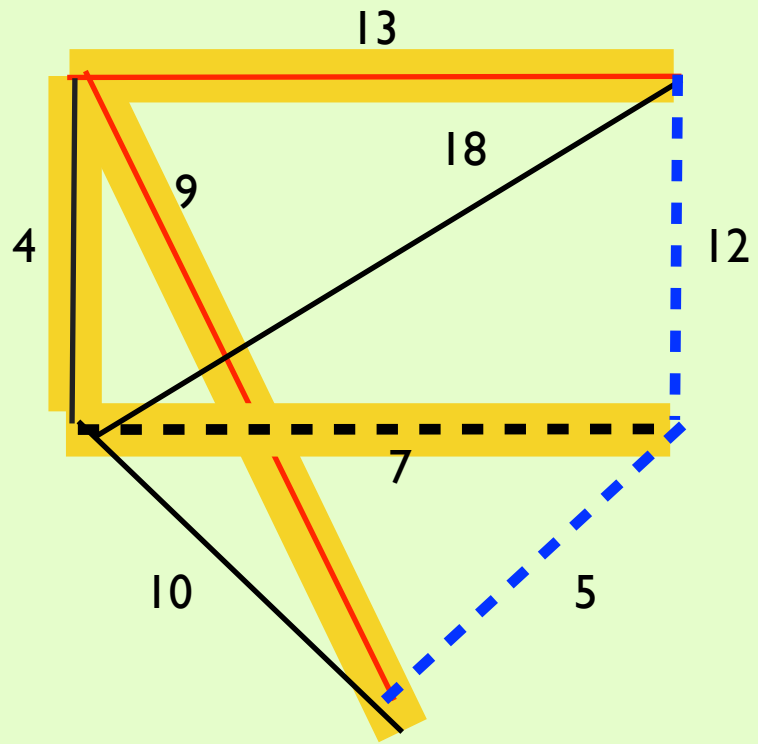
keine Erweiterung / Verbesserung

Erweiterung / Verbesserung

Kante wird in dieser Iteration aus MST entfernt

© 2006-2024 Jörg Striegnitz

B Minimaler Spannbaum



	1	2	3	4	5	Q
k π	0 nil	∞ nil	∞ nil	∞ nil	∞ nil	1,2,3,4,5
	0 nil	13 1	4 1	∞ nil	9 1	3,5,2,4
	0 nil	13 1	4 1	7 3	9 1	4,5,2
	0 nil	**12** **4**	4 1	7 3	**5** **4**	5,2

Kante des aktuellen MST

keine Erweiterung / Verbesserung

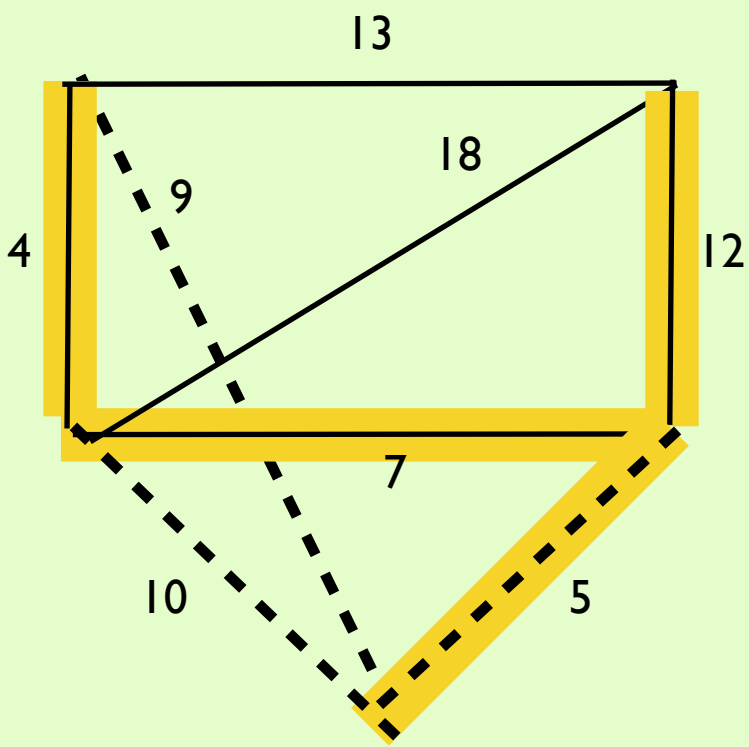
Erweiterung / Verbesserung

Kante wird in dieser Iteration aus MST entfernt

© 2006-2024 Jörg Striegnitz

Algorithmus von Prim - 2

B Minimaler Spannbaum



	1	2	3	4	5	Q
k π	0 nil	∞ nil	∞ nil	∞ nil	∞ nil	1,2,3,4,5
	0 nil	13 1	4 1	∞ nil	9 1	3,5,2,4
	0 nil	13 1	4 1	7 3	9 1	4,5,2
	0 nil	12 4	4 1	7 3	5 4	5,2
	0 nil	12 4	4 1	7 3	5 4	2

Kante des aktuellen MST

keine Erweiterung / Verbesserung

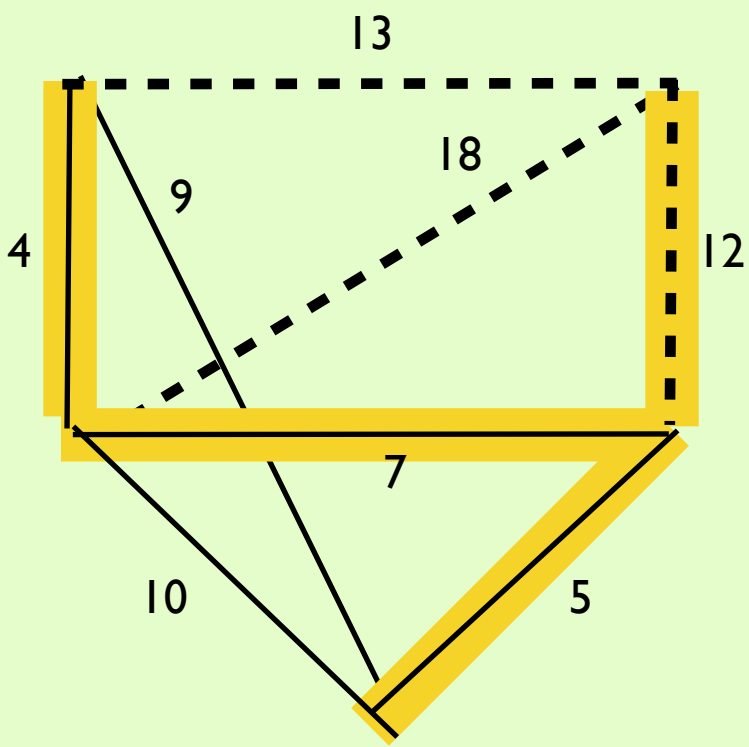
Erweiterung / Verbesserung

Kante wird in dieser Iteration aus MST entfernt

© 2006-2024 Jörg Striegnitz

Algorithmus von Prim - 2

B Minimaler Spannbaum



	1	2	3	4	5	Q
k	0	∞	∞	∞	∞	
π	nil	nil	nil	nil	nil	1,2,3,4,5
	0	13	4	∞	9	
	nil	1	1	nil	1	3,5,2,4
	0	13	4	7	9	
	nil	1	1	3	1	4,5,2
	0	12	4	7	5	
	nil	4	1	3	4	5,2
	0	12	4	7	5	
	nil	4	1	3	4	2
	0	12	4	7	5	
	nil	4	1	3	4	

Kante des aktuellen MST

keine Erweiterung / Verbesserung

Erweiterung / Verbesserung

Kante wird in dieser Iteration aus MST entfernt

© 2006-2024 Jörg Striegnitz

Exkurs: Das Traveling Salesman Problem (TSP)

Minimale Spannbäume in der Anwendung

© 2006-2023 Jörg Striegnitz

Die Faszination des Traveling Salesman Problems (TSP)

(des Handlungsreisendenproblems)

- Das TSP ist einfach und schnell zu beschreiben
- Jeder entwickelt beim Ansehen sofort eine Lösungsstrategie
- Viele, praxisrelevante Probleme lassen sich auf das TSP zurückführen
- Obwohl einfach beschrieben, gehört das TSP zu den hartnäckigsten Problemen überhaupt!

© 2006-2024 Jörg Striegnitz

TSP - seit langem bekannt!

© 2006-2024 Jörg Striegnitz

TSP - seit langem bekannt!

Der

Handlungsreisende

wie er sein soll

und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiß zu sein.

Von

einem alten Commis - Voyageur.

Mit einem Titelkupfer.

Ilmenau 1832,

Druck und Verlag von B. Fr. Voigt.

aber es kann durch eine zweckmäßige Wahl und Eintheilung der Tour, manchmal so viel Zeit gewonnen werden, daß wir es nicht glauben umgehen zu dürfen, auch hierüber einige Vorschriften zu geben.

⋮

worauf der Reisende hauptsächlich zu sehen hat, des Hin- und Herreisens, mit mehr Oekonomie einzurichten. Die Hauptsache besteht immer darin: so viele Orte wie möglich mitzunehmen, ohne den nämlichen Ort zweimal berühren zu müssen.

© 2006-2024 Jörg Striegnitz

TSP - seit langem bekannt!

Der

Handlungsreisende

wie er sein soll

und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiß zu sein.

Von

einem alten Commis - Voyageur.

Mit einem Titelkupfer.

Ilmenau 1832,

Druck und Verlag von B. Fr. Voigt.

aber es kann durch eine zweckmäßige Wahl und Eintheilung der Tour, manchmal so viel Zeit gewonnen werden, daß wir es nicht glauben umgehen zu dürfen, auch hierüber einige Vorschriften zu geben.

⋮

worauf der Reisende hauptsächlich zu sehen hat, des Hin- und Herreisens, mit mehr Oekonomie einzurichten. Die Hauptsache besteht immer darin: so viele Orte wie möglich mitzunehmen, ohne den nämlichen Ort zweimal berühren zu müssen.

© 2006-2024 Jörg Striegnitz

TSP - seit langem bekannt!

Der Handlungsreisende wie er sein soll

und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiß zu sein.

Von
einem alten Commis - Voyageur.

Mit einem Titelkupfer.

Ilmenau 1832,
Druck und Verlag von B. Fr. Voigt.

aber es kann durch eine zweckmäßige Wahl und Eintheilung der Tour, manchmal so viel Zeit gewonnen werden, daß wir es nicht glauben umgehen zu dürfen, auch hierüber einige Vorschriften zu geben.

⋮

worauf der Reisende hauptsächlich zu sehen hat, des Hin- und Herreisens, mit mehr Oekonomie einzurichten. Die Hauptsache besteht immer darin: so viele Orte wie möglich mitzunehmen, ohne den nämlichen Ort zweimal berühren zu müssen.

© 2006-2024 Jörg Striegnitz

TSP - seit langem bekannt!

Der Handlungsreisende wie er sein soll

und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiß zu sein.

Von
einem alten Commis - Voyageur.

Mit einem Titelkupfer.

Ilmenau 1832,
Druck und Verlag von B. Fr. Voigt.

… es kann durch eine *zweckmäßige Wahl und Eintheilung der Tour*, manchmal so viel Zeit gewonnen werden ….

⋮

worauf der Reisende hauptsächlich zu sehen hat, des Hin- und Herreisens, mit mehr Oekonomie einzurichten. Die Hauptsache besteht immer darin: so viele Orte wie möglich mitzunehmen, ohne den nämlichen Ort zweimal berühren zu müssen.

© 2006-2024 Jörg Striegnitz

TSP - seit langem bekannt!

Der Handlungsreisende wie er sein soll

und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiß zu sein.

Von
einem alten Commis - Voyageur.

Mit einem Titelkupfer.

Ilmenau 1832,
Druck und Verlag von W. Fr. Voigt.

… es kann durch eine *zweckmäßige Wahl und Eintheilung der Tour*, manchmal so viel Zeit gewonnen werden ….

… Die Hauptsache besteht immer darin, *so viele Orte wie möglich* mitzunehmen, *ohne den nämlichen Ort zweimal berühren* zu müssen ...

© 2006-2024 Jörg Striegnitz

Das *Traveling Salesman Problem*

Travelling Salesman Problem (TSP) *[informell]*

D Gegeben ist eine **endliche Menge von Städten**, von der eine die **Ausgangsstadt** ist, sowie eine **Entfernungstabelle**. Gesucht ist die kürzeste Tour, die in der Ausgangsstadt startet, alle Städte miteinander verbindet und schließlich in der Ausgangsstadt endet.

© 2006-2024 Jörg Striegnitz

Das *Traveling Salesman Problem*

Travelling Salesman Problem (TSP) *[informell]*

D Gegeben ist eine **endliche Menge von Städten**, von der eine die **Ausgangsstadt** ist, sowie eine **Entfernungstabelle**. Gesucht ist die kürzeste Tour, die in der Ausgangsstadt startet, alle Städte miteinander verbindet und schließlich in der Ausgangsstadt endet.

© 2006-2024 Jörg Striegnitz

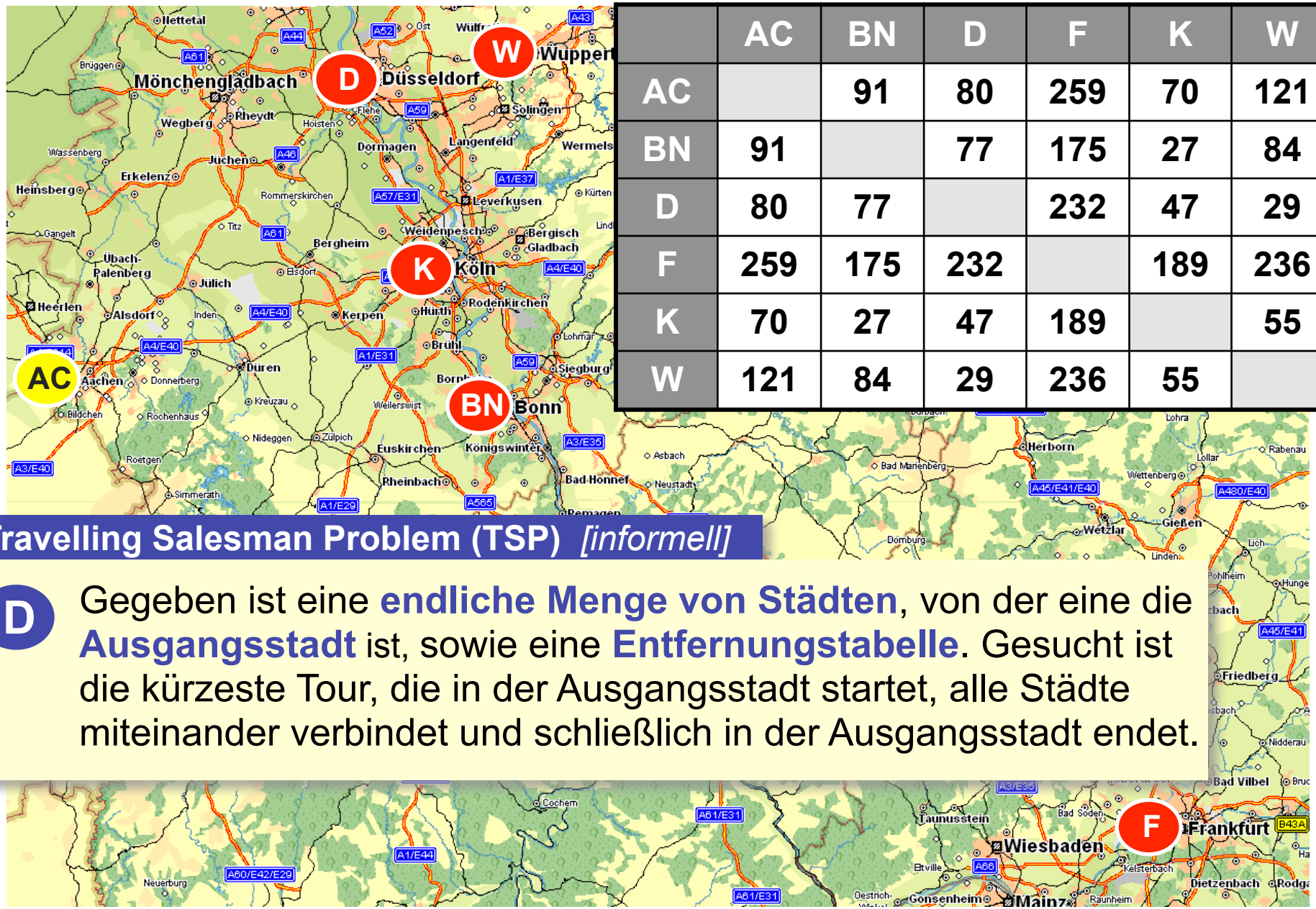
Das *Traveling Salesman Problem*

Travelling Salesman Problem (TSP) *[informell]*

D Gegeben ist eine **endliche Menge von Städten**, von der eine die **Ausgangsstadt** ist, sowie eine **Entfernungstabelle**. Gesucht ist die kürzeste Tour, die in der Ausgangsstadt startet, alle Städte miteinander verbindet und schließlich in der Ausgangsstadt endet.

© 2006-2024 Jörg Striegnitz

Das *Traveling Salesman Problem*



	AC	BN	D	F	K	W
AC		91	80	259	70	121
BN	91		77	175	27	84
D	80	77		232	47	29
F	259	175	232		189	236
K	70	27	47	189		55
W	121	84	29	236	55	

Travelling Salesman Problem (TSP) *[informell]*

D Gegeben ist eine **endliche Menge von Städten**, von der eine die **Ausgangsstadt** ist, sowie eine **Entfernungstabelle**. Gesucht ist die kürzeste Tour, die in der Ausgangsstadt startet, alle Städte miteinander verbindet und schließlich in der Ausgangsstadt endet.

© 2006-2024 Jörg Striegnitz

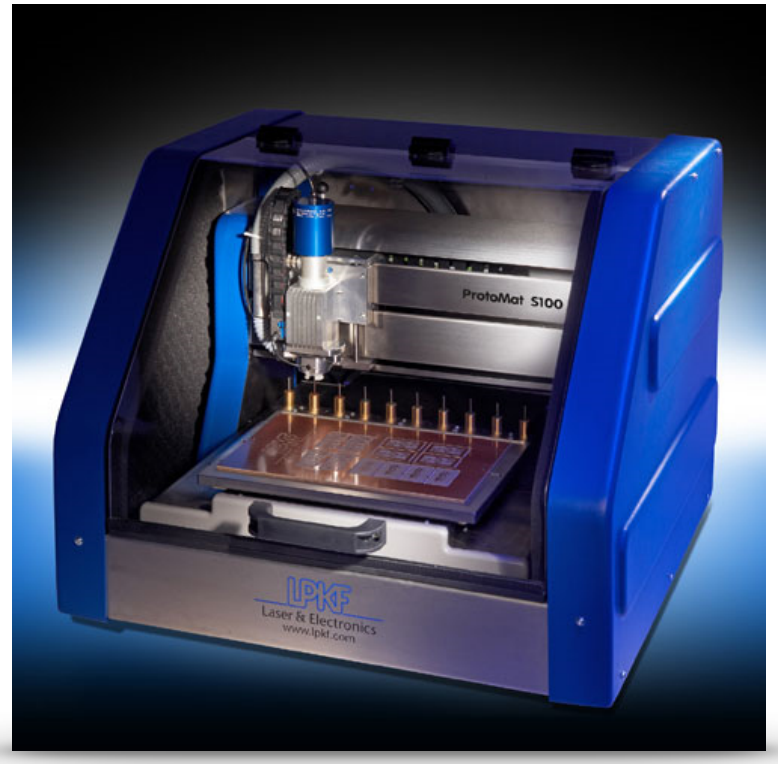
Praktische Anwendungen 1

- **Hochregallager**
  - Auftrag umfasst mehrere Artikel
  - Artikel werden an unterschiedlichen Plätzen gelagert
  - Fahrzeug verlässt Kommissionierungsbereich, stellt Lieferung zusammen und kehr zurück
  - Ziel: zeitminimale Rundreise

© 2006-2024 Jörg Striegnitz

Praktische Anwendungen 2

- **Bohrung von Leiterplatten**
  - Leiterplatte enthält mehrere Bohrungen
  - Bohrungen u.U. unterschiedliche Größe (Werkzeugwechsel!)
- **Bestückung von Platinen**
  - u.U. ebenfalls Werkzeugwechsel
- **Plotten der Belichtungsmaske**
  - Minimierung der Leerlaufzeiten (Phasen, in denen nicht gezeichnet wird)



© 2006-2024 Jörg Striegnitz

Praktische Anwendungen 3

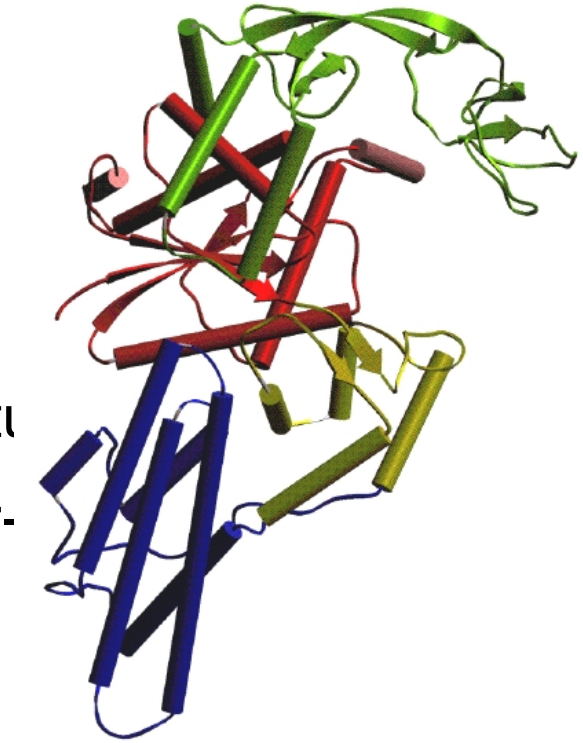
- **Positionierung von Briefkästen**
  - Weg Bürger↔ Briefkasten max. 800 Meter
  - Briefaufkommen abhängig von Einwohnerzahl

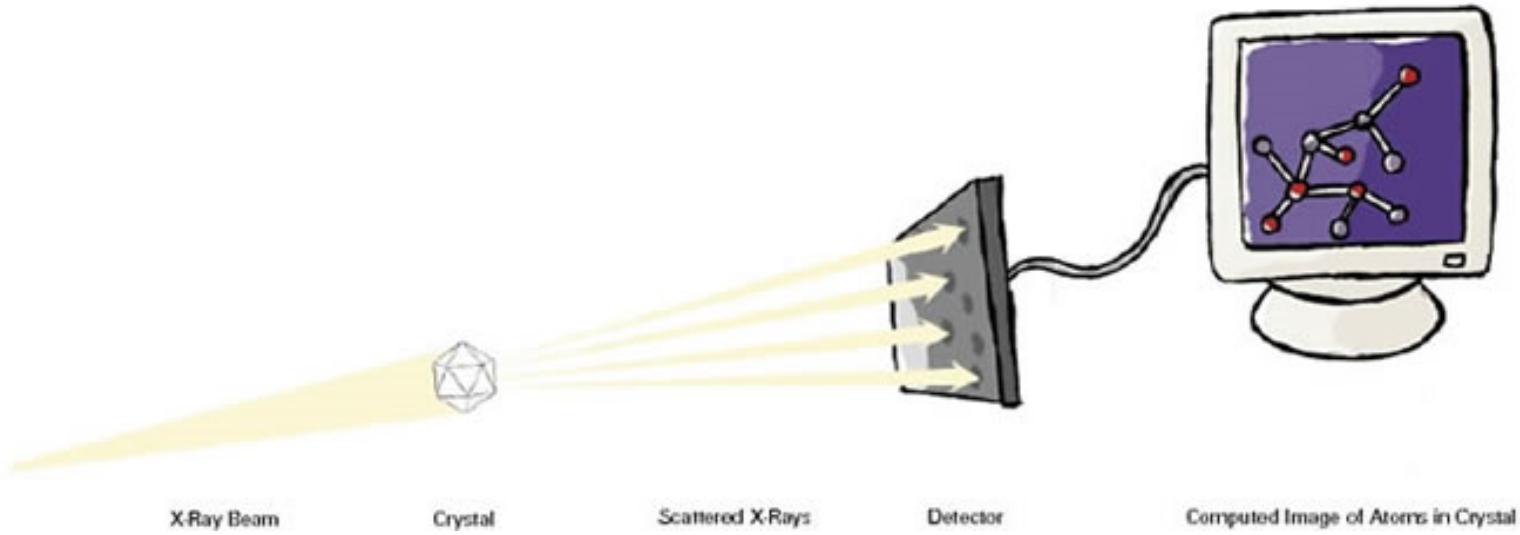
- **Organisation der Leerungen**
  - Leerungsfahrten
  - Zeitfenster zur Leerung ist vorgegeben

© 2006-2024 Jörg Striegnitz

Praktische Anwendungen 4

- **Röntgenkristallographie**
  - Experimentelle Bestimmung der 3D-Struktur von Kristallen (auch: kristallisierte Proteine!)
  - Kristalle beugen Röntgenstrahlen
  - Beugungsmuster erlaubt Rückschlüsse auf 3D-Struktu
  - 5.000 - 30.000 positionsverschiedene Diffraktormeter-messungen erforderlich





© 2006-2024 Jörg Striegnitz

Hamiltonsche Kreise

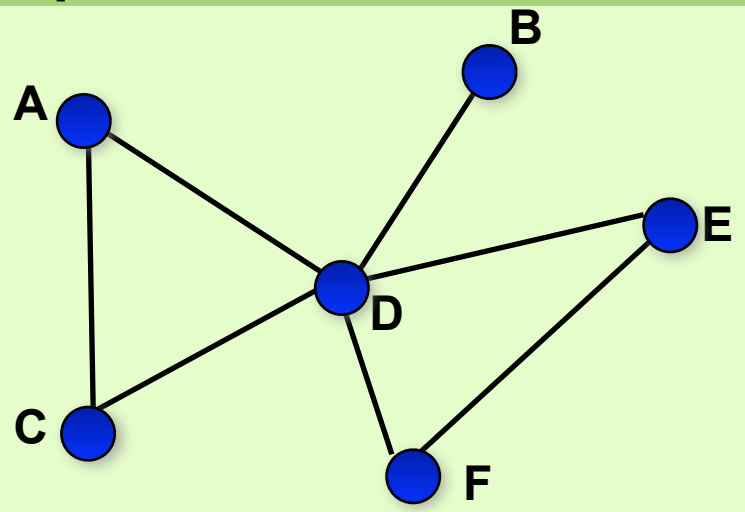
D Kreis / Hamiltonscher Kreis

Sei $G = (V, E)$ ein Graph. Ein Pfad $p = v_1, \cdots, v_n$ heißt **Kreis**, falls $v_1 = v_n$. p heißt **Hamiltonscher Kreis**, falls zusätzlich

$$V = \bigcup_{i=1}^{n} \{v_i\} \quad \wedge \quad n = |V|$$

gilt (jeder Knoten aus V wird genau einmal besucht).

B Graph ohne Hamiltonschen Kreis



- **Es gibt Graphen ohne Hamiltonsche Kreise. Vollständige Graphen enthalten ab $|V|>3$ mehrere Hamiltonsche Kreise**

© 2006-2024 Jörg Striegnitz

Hamiltonsche Kreise

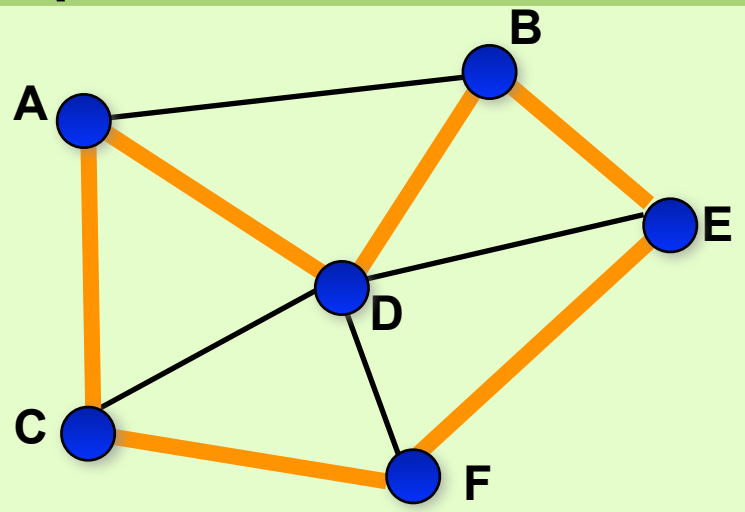
D Kreis / Hamiltonscher Kreis

Sei $G = (V, E)$ ein Graph. Ein Pfad $p = v_1, \cdots, v_n$ heißt **Kreis**, falls $v_1 = v_n$. p heißt **Hamiltonscher Kreis**, falls zusätzlich

$$V = \bigcup_{i=1}^{n} \{v_i\} \quad \wedge \quad n = |V|$$

gilt (jeder Knoten aus V wird genau einmal besucht).

B Graph mit mehreren Hamiltonschen Kreisen



© 2006-2024 Jörg Striegnitz

Hamiltonsche Kreise

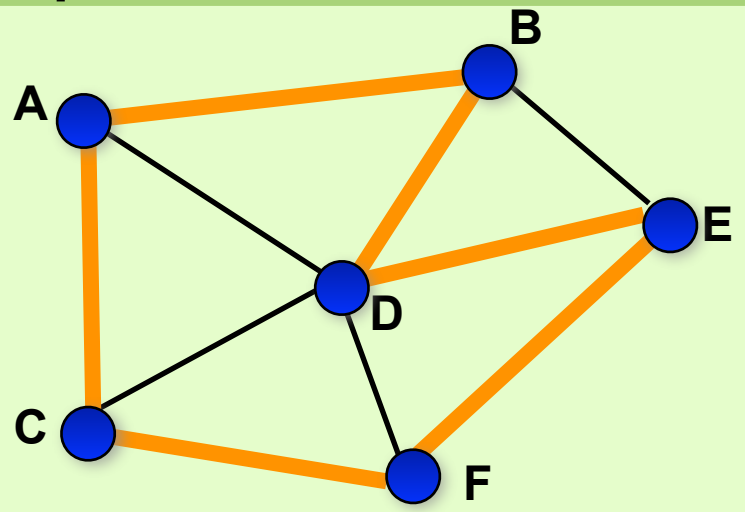
D Kreis / Hamiltonscher Kreis

Sei $G = (V, E)$ ein Graph. Ein Pfad $p = v_1, \cdots, v_n$ heißt **Kreis**, falls $v_1 = v_n$. p heißt **Hamiltonscher Kreis**, falls zusätzlich

$$V = \bigcup_{i=1}^{n} \{v_i\} \quad \wedge \quad n = |V|$$

gilt (jeder Knoten aus V wird genau einmal besucht).

B Graph mit mehreren Hamiltonschen Kreisen



© 2006-2024 Jörg Striegnitz

Hamiltonsche Kreise

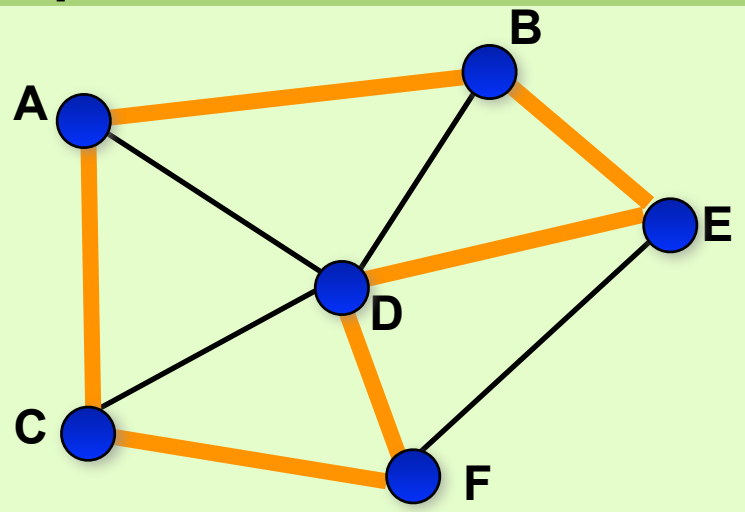
D Kreis / Hamiltonscher Kreis

Sei $G = (V, E)$ ein Graph. Ein Pfad $p = v_1, \cdots, v_n$ heißt **Kreis**, falls $v_1 = v_n$. p heißt **Hamiltonscher Kreis**, falls zusätzlich

$$V = \bigcup_{i=1}^{n} \{v_i\} \quad \wedge \quad n = |V|$$

gilt (jeder Knoten aus V wird genau einmal besucht).

B Graph mit mehreren Hamiltonschen Kreisen



© 2006-2024 Jörg Striegnitz

Hamiltonsche Kreise

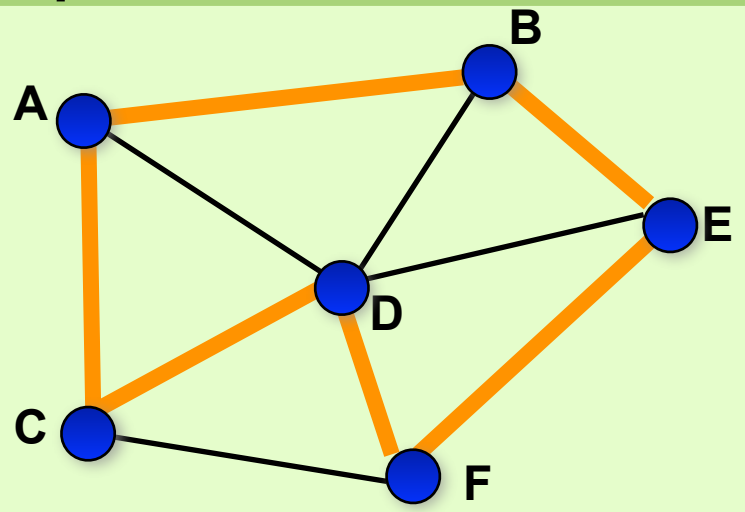
D Kreis / Hamiltonscher Kreis

Sei $G = (V, E)$ ein Graph. Ein Pfad $p = v_1, \cdots, v_n$ heißt **Kreis**, falls $v_1 = v_n$. p heißt **Hamiltonscher Kreis**, falls zusätzlich

$$V = \bigcup_{i=1}^{n} \{v_i\} \quad \wedge \quad n = |V|$$

gilt (jeder Knoten aus V wird genau einmal besucht).

B Graph mit mehreren Hamiltonschen Kreisen



© 2006-2024 Jörg Striegnitz

Hamiltonsche Kreise

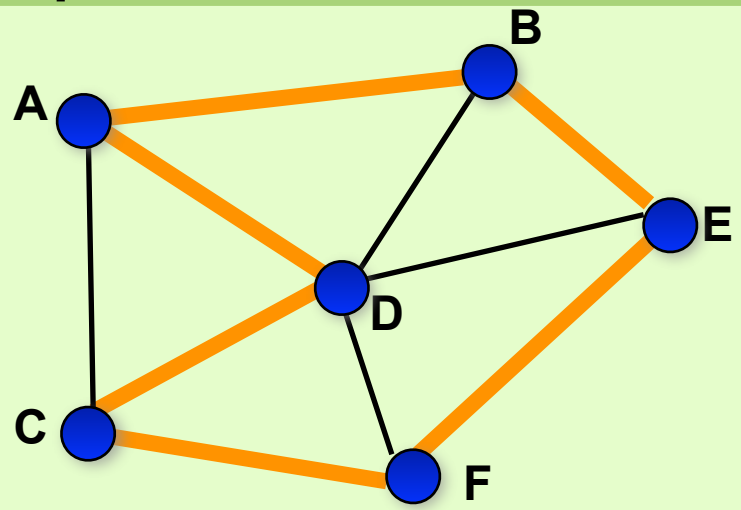
D Kreis / Hamiltonscher Kreis

Sei $G = (V, E)$ ein Graph. Ein Pfad $p = v_1, \cdots, v_n$ heißt **Kreis**, falls $v_1 = v_n$. p heißt **Hamiltonscher Kreis**, falls zusätzlich

$$V = \bigcup_{i=1}^{n} \{v_i\} \quad \wedge \quad n = |V|$$

gilt (jeder Knoten aus V wird genau einmal besucht).

B Graph mit mehreren Hamiltonschen Kreisen



© 2006-2024 Jörg Striegnitz

D Kreis / Hamiltonscher Kreis

Sei $G = (V, E)$ ein gewichteter Graph. Das **Traveling Salesman Problem (TSP)** besteht darin, den Hamiltonschen Kreis in G zu finden, der das geringste Gewicht hat.

Algorithmus zur Bestimmung der *exakten Lösung*

```
Min := MaxInt;
forall Hamiltonschen_Kreise(G) as p do
  if G.w(p) < Min then
    Min := G.w(p);
    Tour := p;
  fi
endfor
```


© 2006-2024 Jörg Striegnitz

Graphentheoretische Betrachtung des TSP

D Kreis / Hamiltonscher Kreis

Sei $G = (V, E)$ ein gewichteter Graph. Das **Traveling Salesman Problem (TSP)** besteht darin, den Hamiltonschen Kreis in G zu finden, der das geringste Gewicht hat.

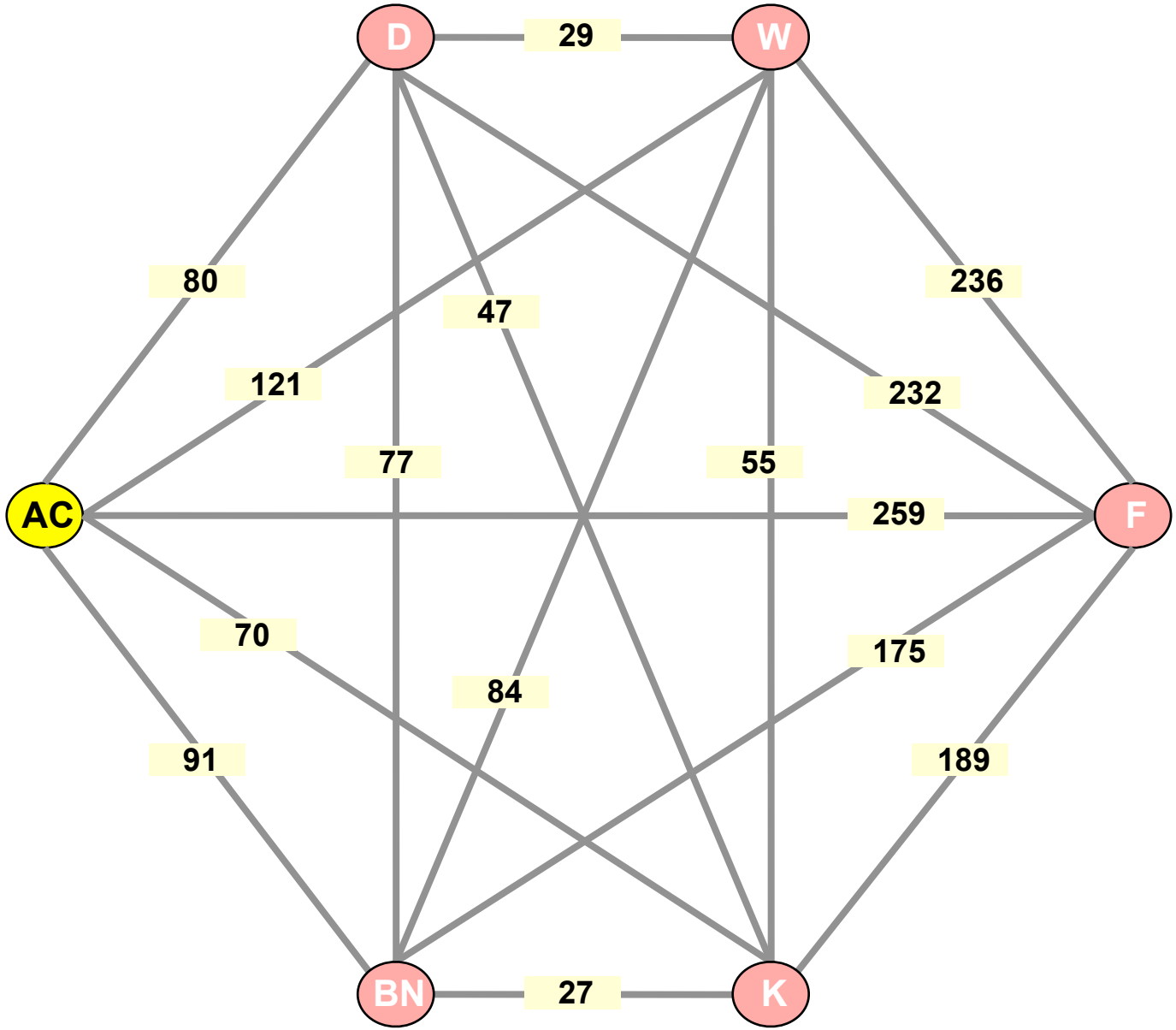
Algorithmus zur Bestimmung der *exakten Lösung*

```
Min := MaxInt;
forall Hamiltonschen_Kreise(G) as p do
  if G.w(p) < Min then
    Min := G.w(p);
    Tour := p;
  fi
endfor
```

Praktikabel?

© 2006-2024 Jörg Striegnitz

1. Schritt abtsrahieren ...



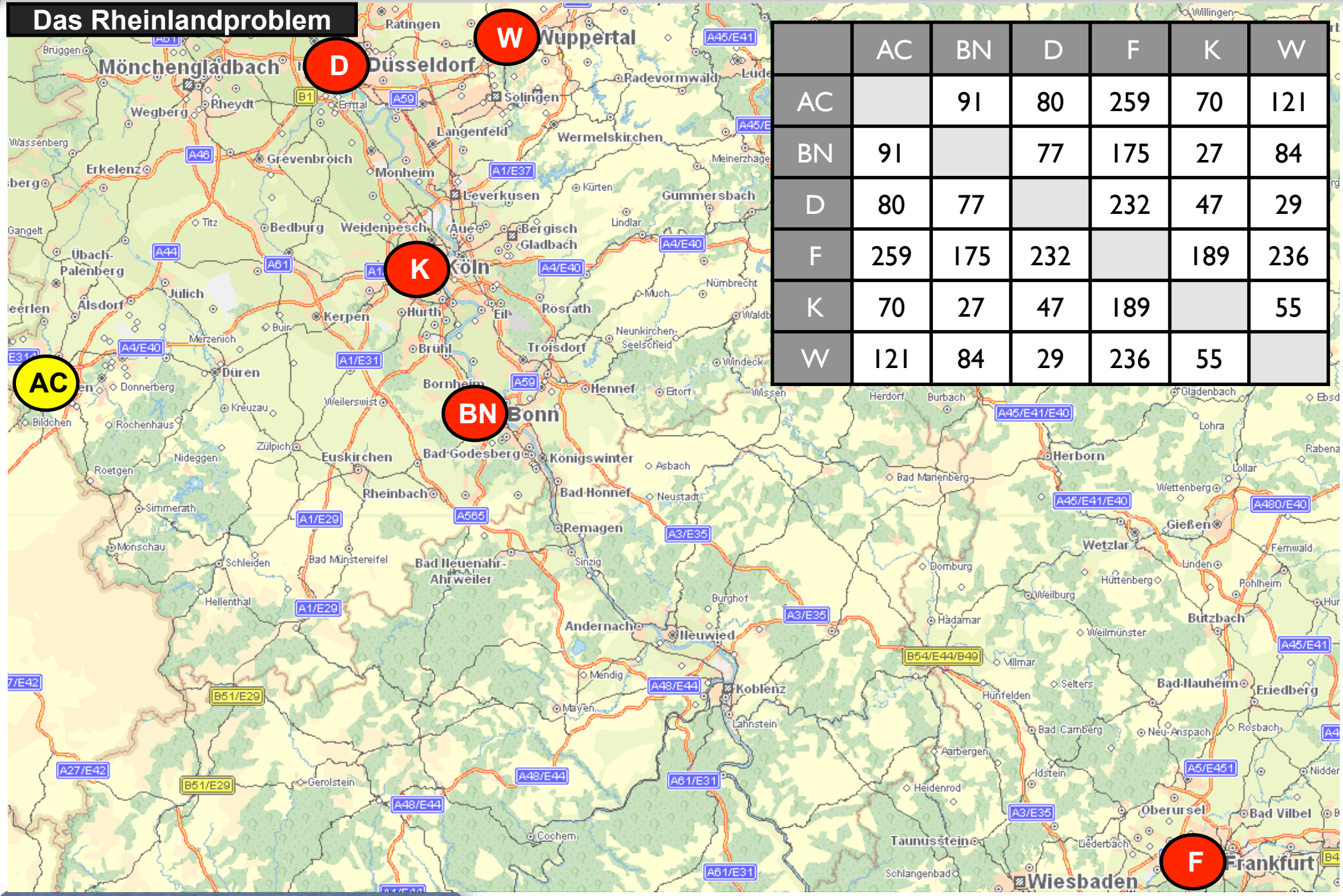
© 2006-2024 Jörg Striegnitz

Anwendung auf das Rheinlandproblem

- Übergang praktisches Problem → formale Darstellung
- Anwendbarkeit des exakten Algorithmus

© 2006-2014 Jörg Striegnitz

Das Rheinlandproblem als Graph I

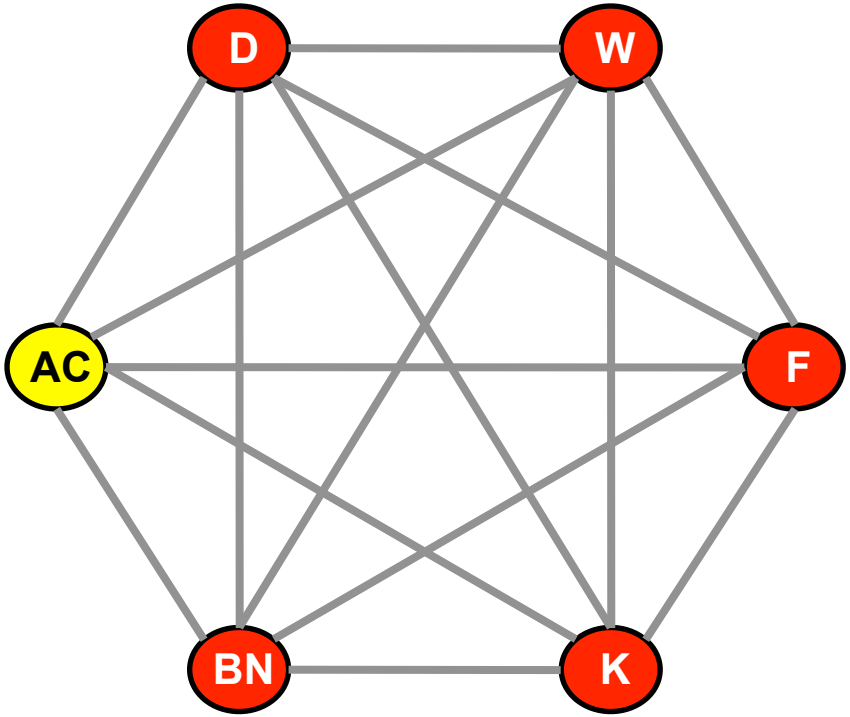


	AC	BN	D	F	K	W
AC		91	80	259	70	121
BN	91		77	175	27	84
D	80	77		232	47	29
F	259	175	232		189	236
K	70	27	47	189		55
W	121	84	29	236	55	

© 2006-2024 Jörg Striegnitz

Das Rheinlandproblem als Graph I

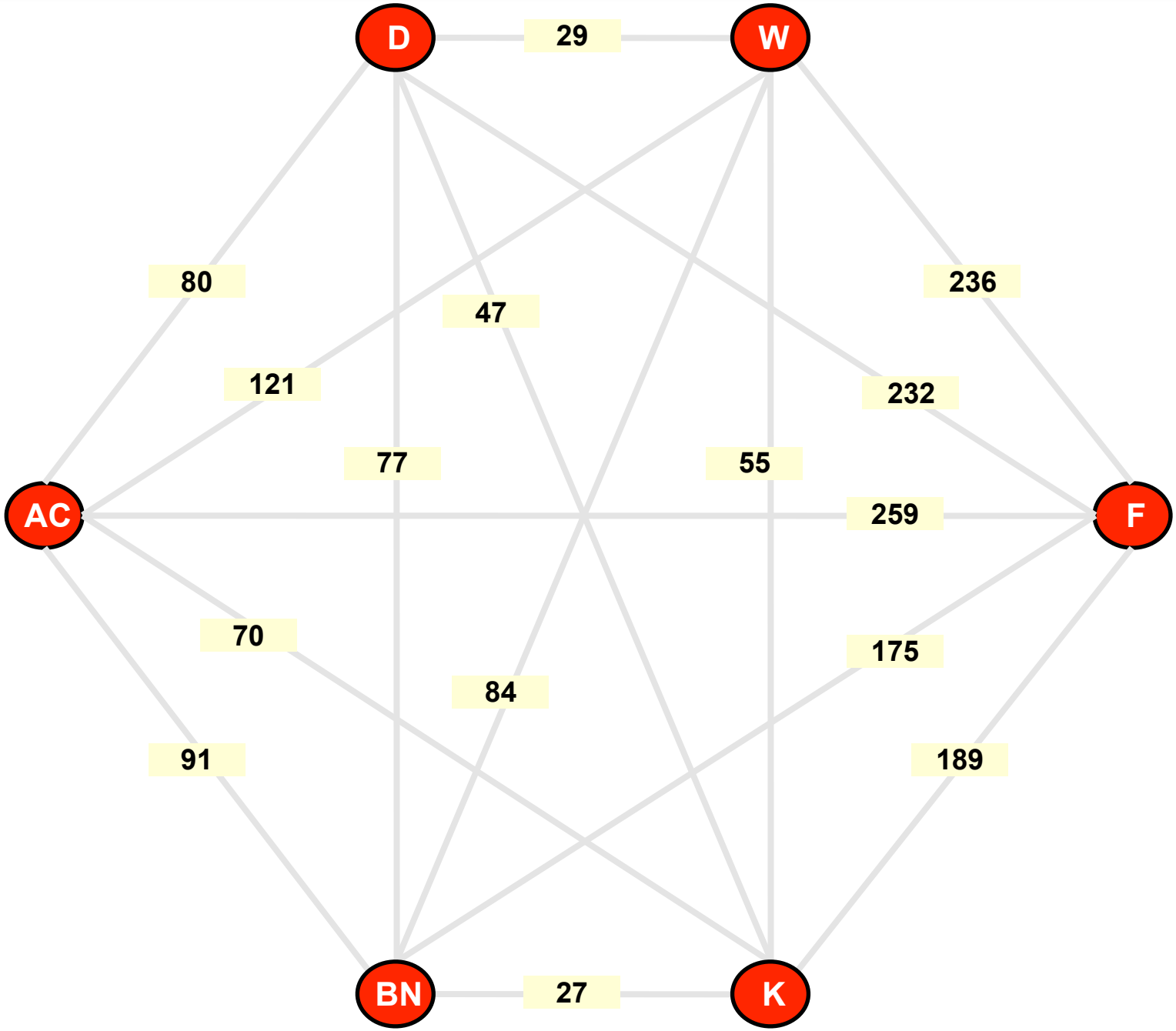
Das Rheinlandproblem



	AC	BN	D	F	K	W
AC		91	80	259	70	121
BN	91		77	175	27	84
D	80	77		232	47	29
F	259	175	232		189	236
K	70	27	47	189		55
W	121	84	29	236	55	

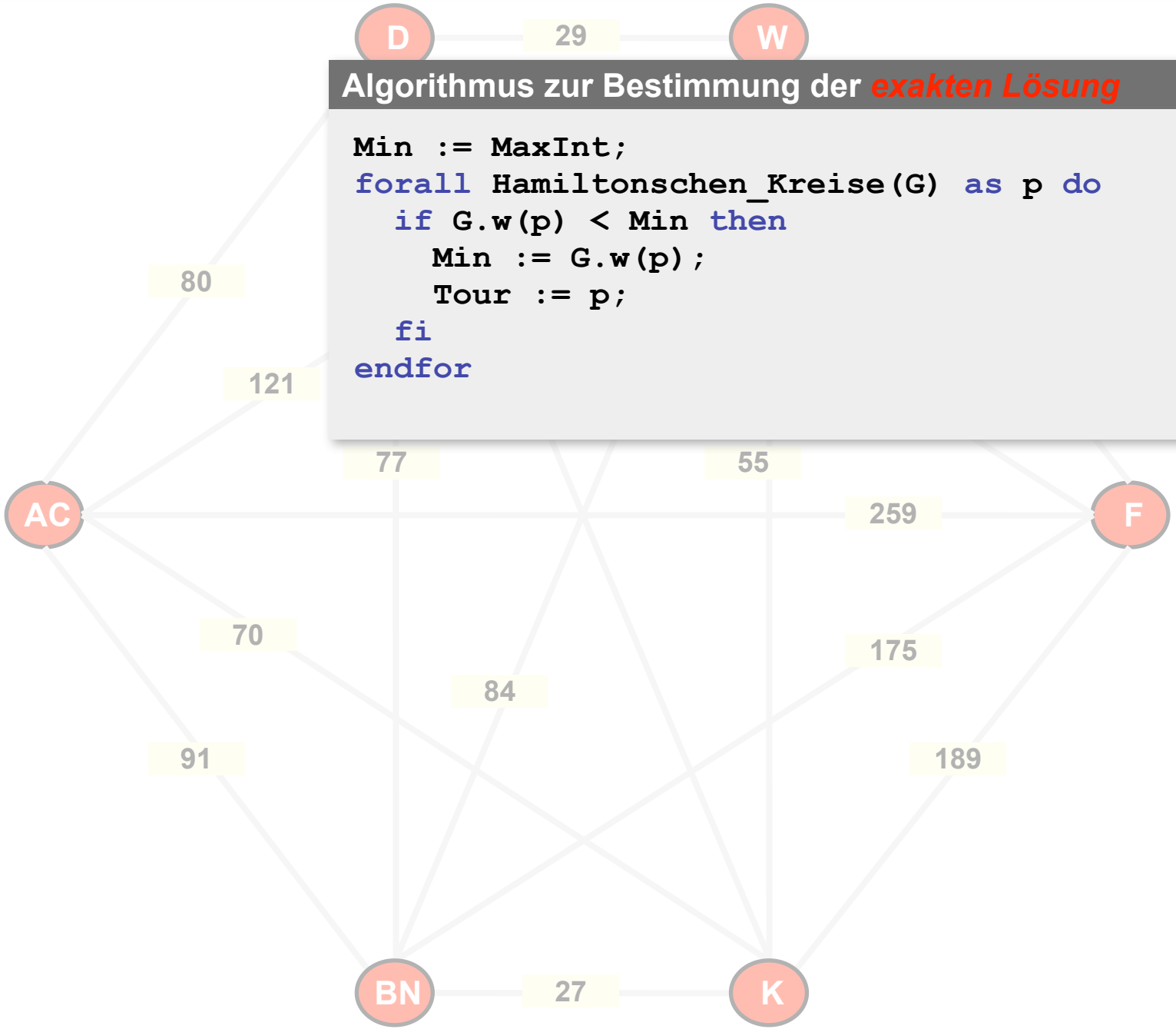
© 2006-2024 Jörg Striegnitz

Das Rheinlandproblem als Graph II



© 2006-2024 Jörg Striegnitz

Das Rheinlandproblem als Graph II

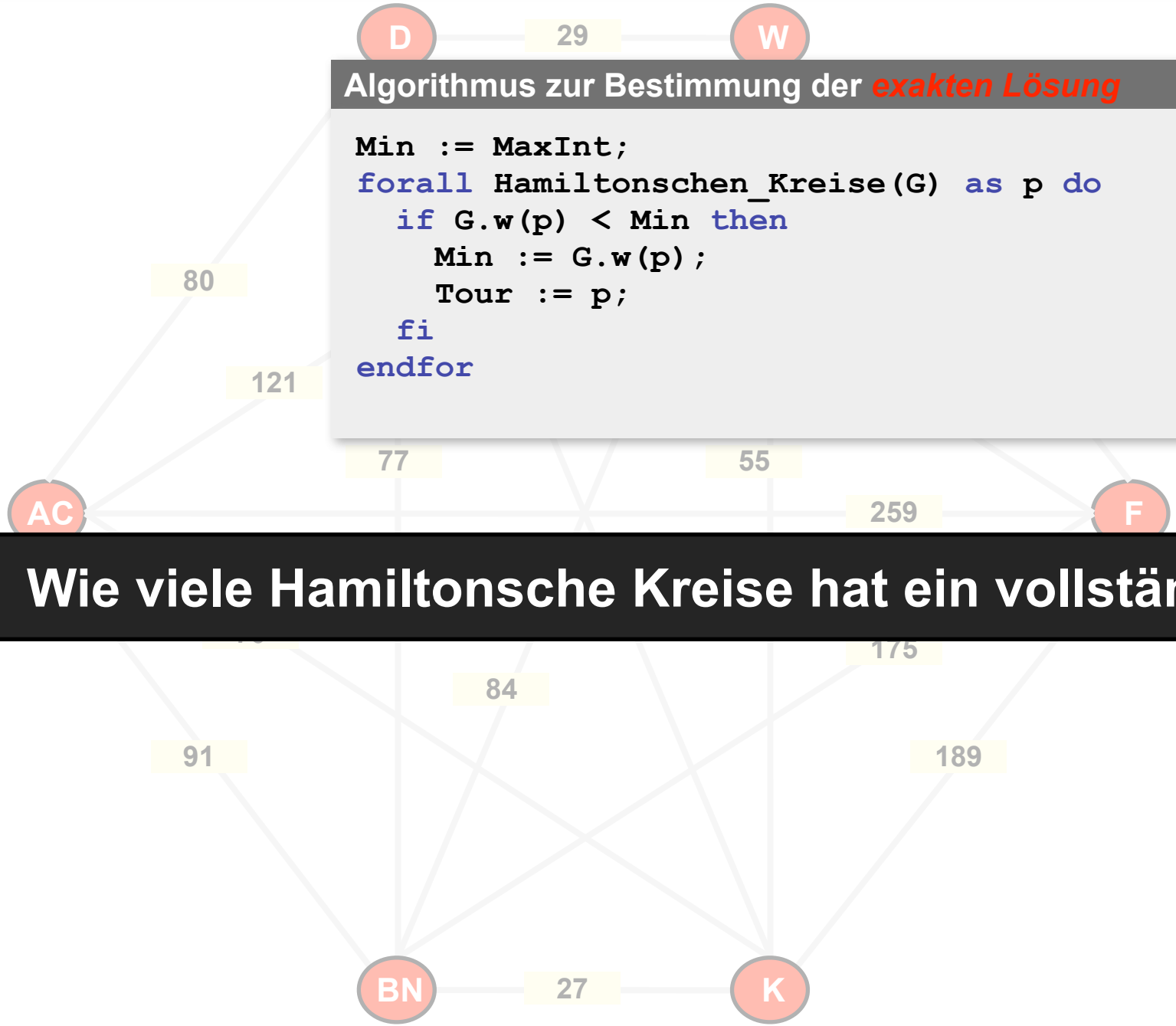


Algorithmus zur Bestimmung der *exakten Lösung*

```
Min := MaxInt;
forall Hamiltonschen_Kreise(G) as p do
  if G.w(p) < Min then
    Min := G.w(p);
    Tour := p;
  fi
endfor
```


© 2006-2024 Jörg Striegnitz

Das Rheinlandproblem als Graph II



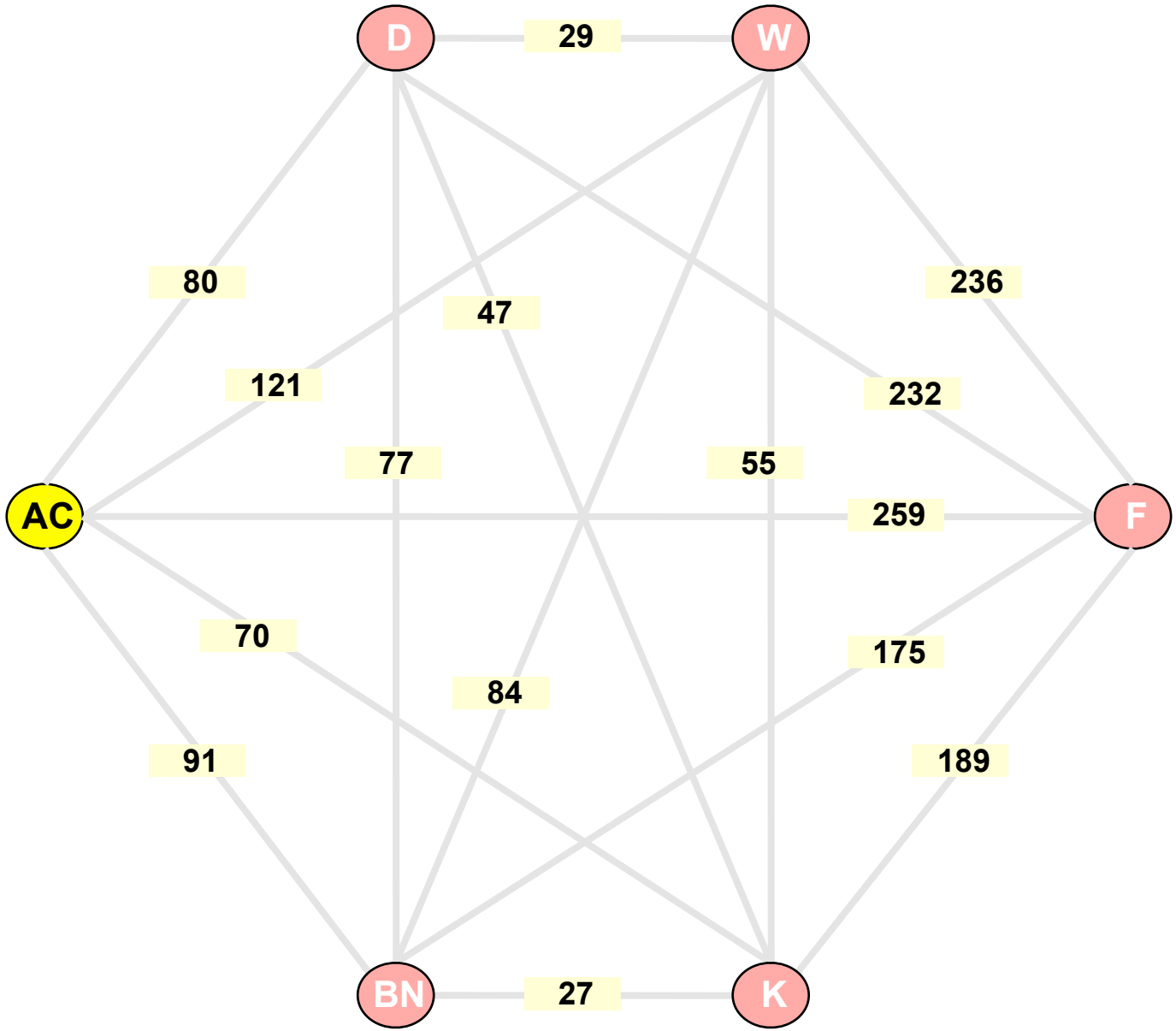
Algorithmus zur Bestimmung der *exakten Lösung*

```
Min := MaxInt;
forall Hamiltonschen_Kreise(G) as p do
  if G.w(p) < Min then
    Min := G.w(p);
    Tour := p;
  fi
endfor
```

Wie viele Hamiltonsche Kreise hat ein vollständiger Graph?

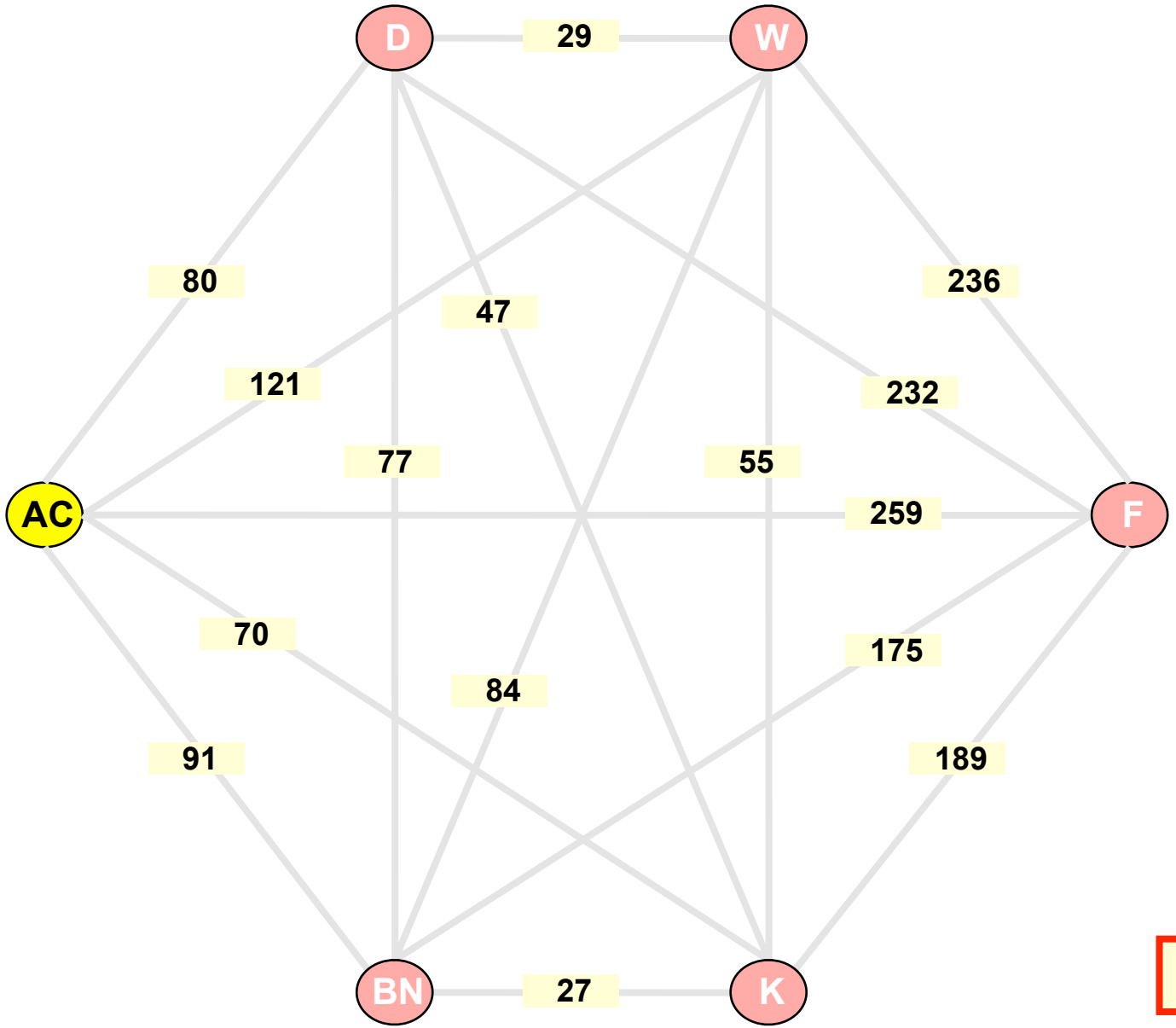
© 2006-2024 Jörg Striegnitz

3. Analysiere Algorithmus ...



© 2006-2024 Jörg Striegnitz

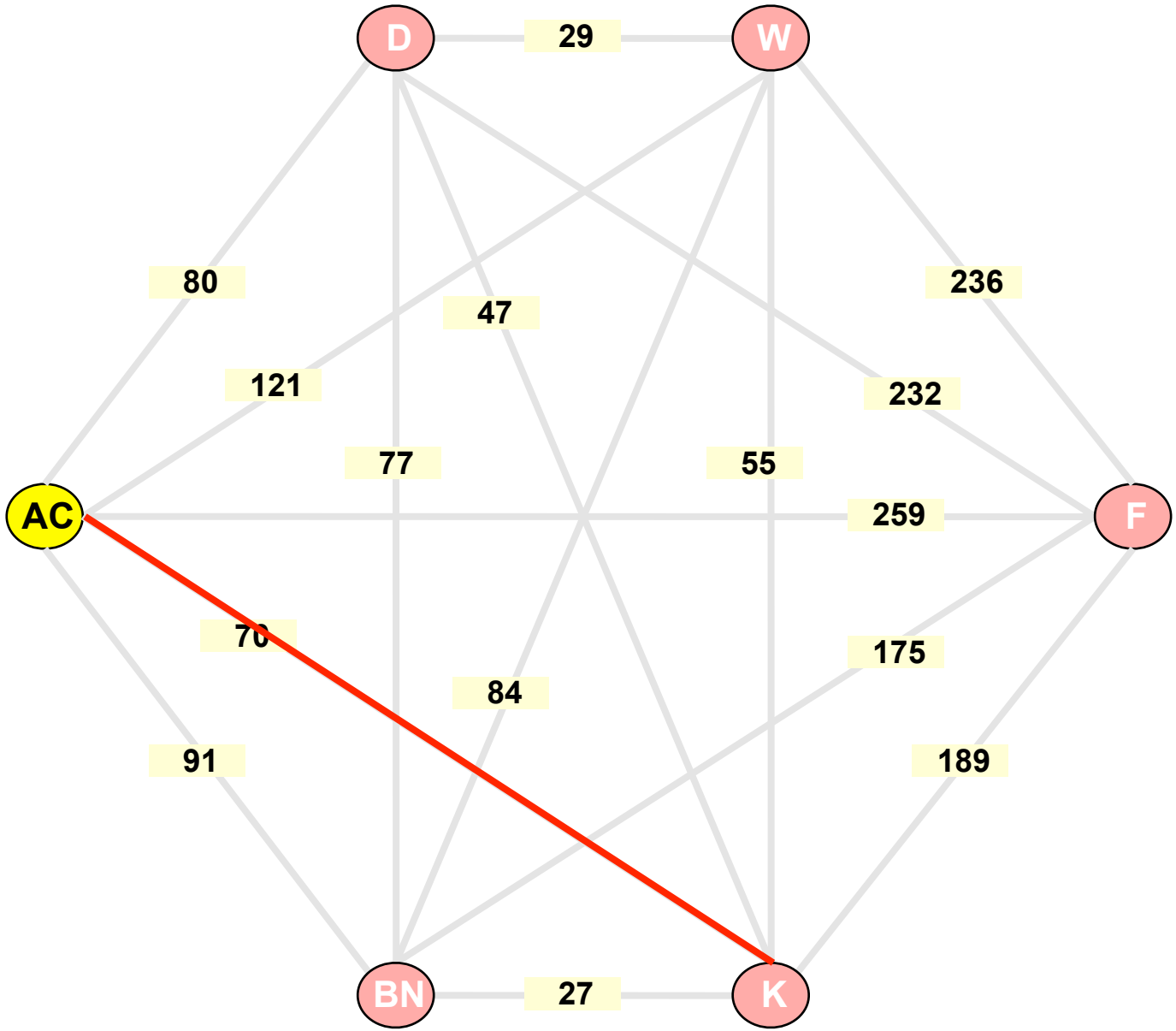
3. Analysiere Algorithmus ...



5 Möglichkeiten

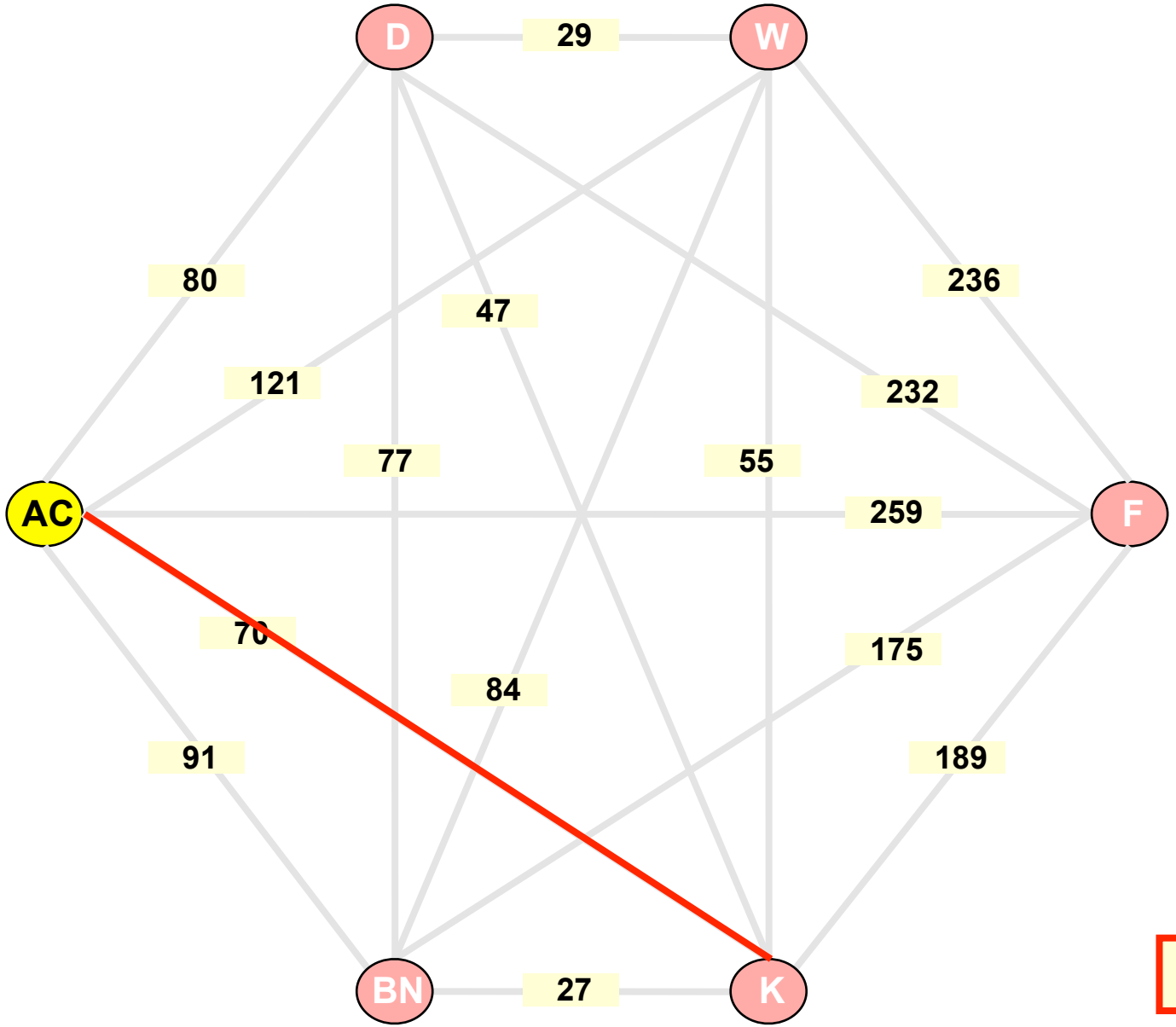
© 2006-2024 Jörg Striegnitz

3. Analysiere Algorithmus ...



© 2006-2024 Jörg Striegnitz

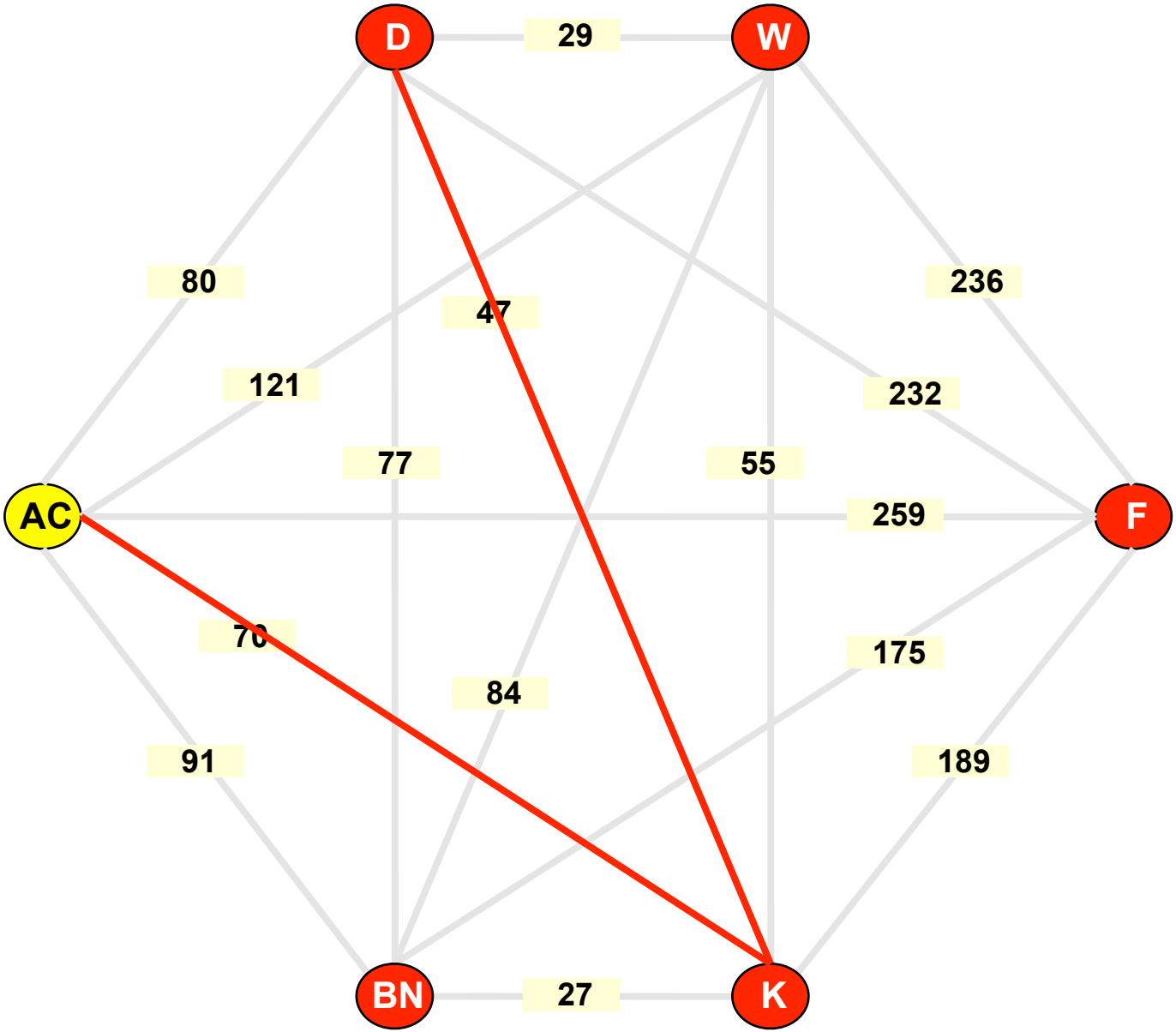
3. Analysiere Algorithmus ...



5*4 Möglichkeiten

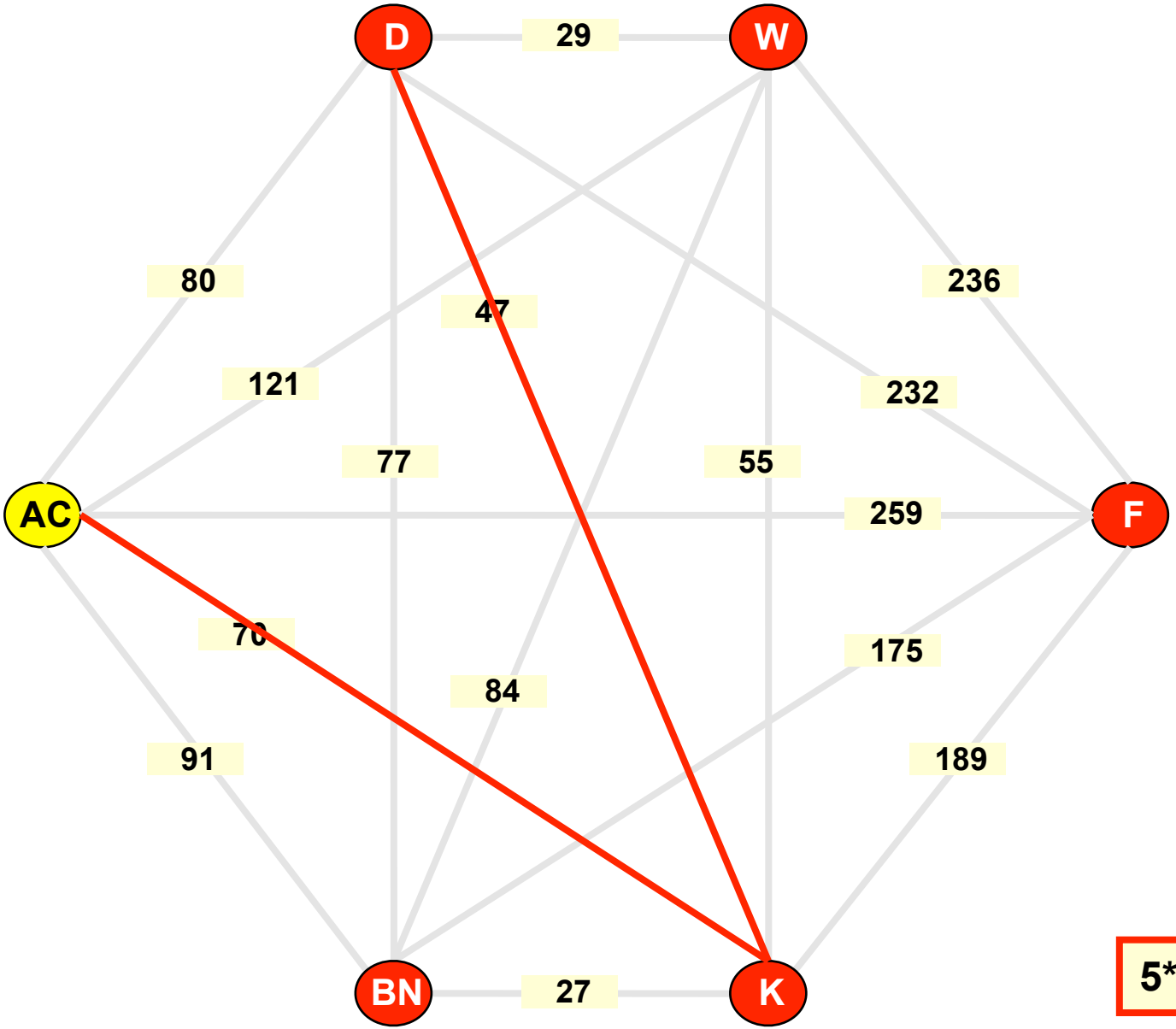
© 2006-2024 Jörg Striegnitz

3. Analysiere Algorithmus ...



© 2006-2024 Jörg Striegnitz

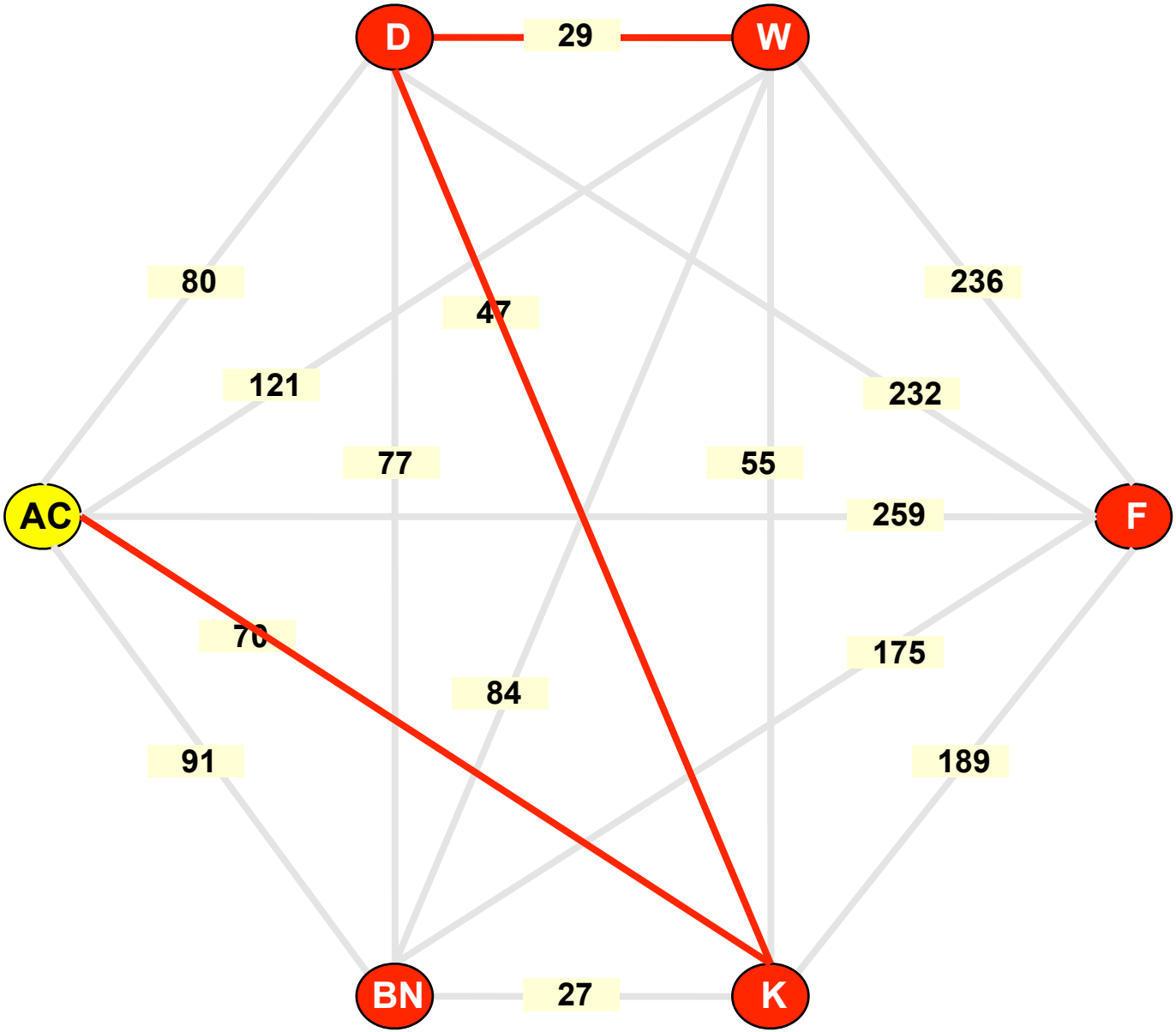
3. Analysiere Algorithmus ...



5*4*3 Möglichkeiten

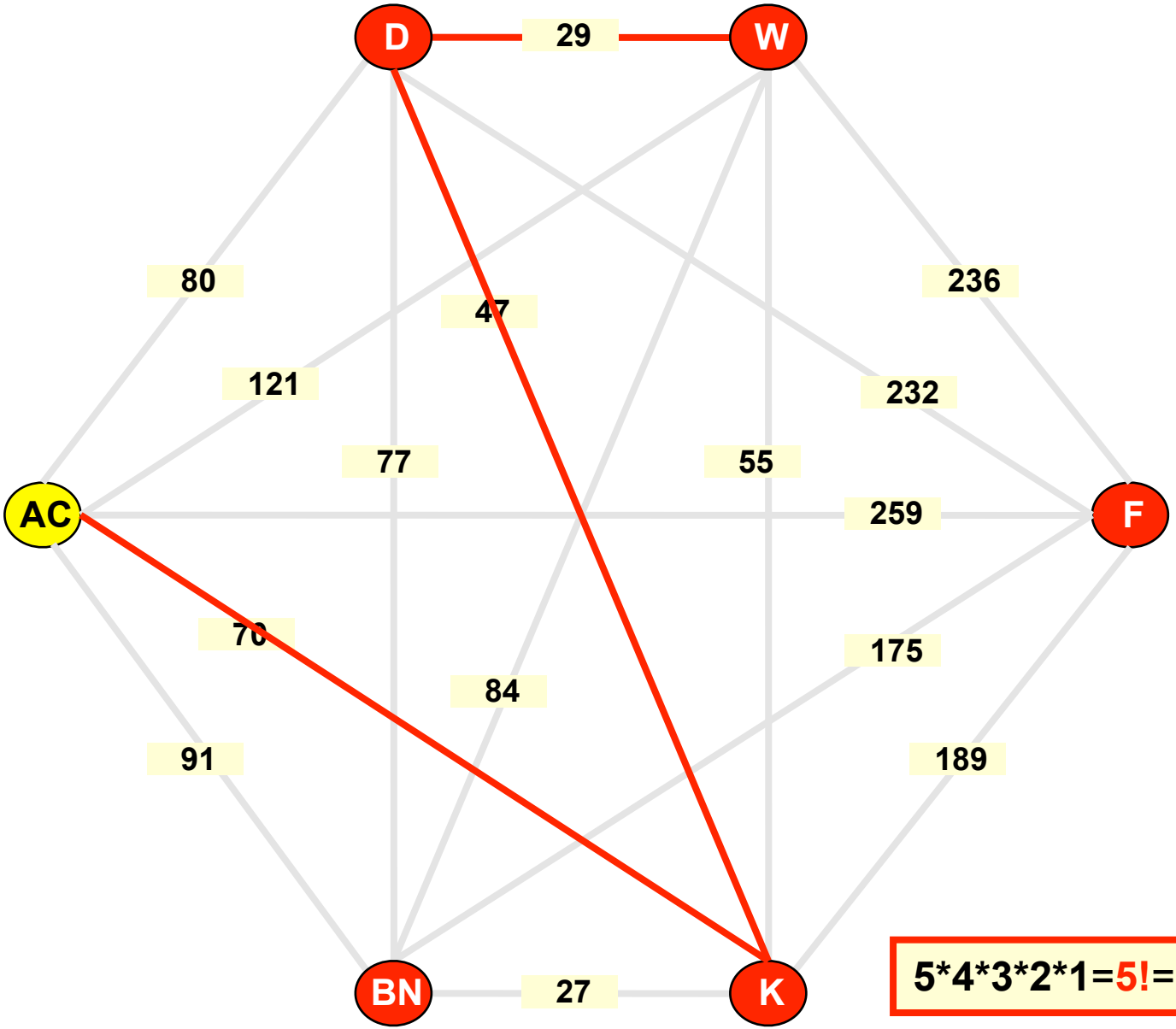
© 2006-2024 Jörg Striegnitz

3. Analysiere Algorithmus ...



© 2006-2024 Jörg Striegnitz

3. Analysiere Algorithmus ...



5*4*3*2*1=5!=120 Möglichkeiten

© 2006-2024 Jörg Striegnitz

Lösung praktikabel?

- In einem vollständigen, schleifenfreien Graphen mit n Knoten gibt es genau $(n\text{-}1)!$ Hamiltonsche Kreise.

© 2006-2024 Jörg Striegnitz

Lösung praktikabel?

- In einem vollständigen, schleifenfreien Graphen mit n Knoten gibt es genau $(n\text{-}1)!$ Hamiltonsche Kreise.
- Da es im Zusammenhang mit dem TSP egal ist, in welcher Richtung ein Hamiltonscher Kreis durchlaufen wird gilt:

Der Aufwand zur *exakten Lösung eines TSP* mit n Städten ist proportional zu

$$\frac{1}{2} \cdot (n-1)!$$

© 2006-2024 Jörg Striegnitz

Lösung praktikabel?

- In einem vollständigen, schleifenfreien Graphen mit n Knoten gibt es genau $(n\text{-}1)!$ Hamiltonsche Kreise.
- Da es im Zusammenhang mit dem TSP egal ist, in welcher Richtung ein Hamiltonscher Kreis durchlaufen wird gilt:

Der Aufwand zur *exakten Lösung eines TSP* mit n Städten ist proportional zu

$$\frac{1}{2} \cdot (n-1)!$$

Angenommen, eine Addition benötigt 10^{-9} Sekunden [1 GFLOP CPU]

© 2006-2024 Jörg Striegnitz

Lösung praktikabel?

- In einem vollständigen, schleifenfreien Graphen mit n Knoten gibt es genau $(n\text{-}1)!$ Hamiltonsche Kreise.
- Da es im Zusammenhang mit dem TSP egal ist, in welcher Richtung ein Hamiltonscher Kreis durchlaufen wird gilt:

Der Aufwand zur *exakten Lösung eines TSP* mit n Städten ist proportional zu

$$\frac{1}{2} \cdot (n-1)!$$

Angenommen, eine Addition benötigt 10^{-9} Sekunden [1 GFLOP CPU]

N	# Touren	Zeit
9	20.160	
10	181.440	
11	1.814.400	
12	19.958.400	0,2 s
13	239.500.800	3 s
14	3.113.510.400	43 s

© 2006-2024 Jörg Striegnitz

Lösung praktikabel?

- In einem vollständigen, schleifenfreien Graphen mit n Knoten gibt es genau $(n\text{-}1)!$ Hamiltonsche Kreise.
- Da es im Zusammenhang mit dem TSP egal ist, in welcher Richtung ein Hamiltonscher Kreis durchlaufen wird gilt:

Der Aufwand zur *exakten Lösung eines TSP* mit n Städten ist proportional zu

$$\frac{1}{2} \cdot (n-1)!$$

Angenommen, eine Addition benötigt 10^{-9} Sekunden [1 GFLOP CPU]

N	# Touren	Zeit
9	20.160	
10	181.440	
11	1.814.400	
12	19.958.400	0,2 s
13	239.500.800	3 s
14	3.113.510.400	43 s

N	# Touren	Zeit
15	43.589.145.600,00	10 min
16	653.837.184.000,00	3 h
17	10.461.394.944.000,00	2 d
18	177.843.714.048.000,00	5 Wochen
19	3.201.186.852.864.000,00	2 Jahre
20	60.822.550.204.416.000,00	38,5 Jahre

© 2006-2024 Jörg Striegnitz

Lösung praktikabel?

- In einem vollständigen, schleifenfreien Graphen mit n Knoten gibt es genau $(n\text{-}1)!$ Hamiltonsche Kreise.
- Da es im Zusammenhang mit dem TSP egal ist, in welcher Richtung ein Hamiltonscher Kreis durchlaufen wird gilt:

Der Aufwand zur *exakten Lösung eines TSP* mit n Städten ist proportional zu

$$\frac{1}{2} \cdot (n-1)!$$

Angenommen, eine Addition benötigt 10^{-9} Sekunden [1 GFLOP CPU]

N	# Touren	Zeit
26	7 Quadrillion (10^{33})	

© 2006-2024 Jörg Striegnitz

Lösung praktikabel?

- In einem vollständigen, schleifenfreien Graphen mit n Knoten gibt es genau $(n\text{-}1)!$ Hamiltonsche Kreise.
- Da es im Zusammenhang mit dem TSP egal ist, in welcher Richtung ein Hamiltonscher Kreis durchlaufen wird gilt:

Der Aufwand zur *exakten Lösung eines TSP* mit n Städten ist proportional zu

$$\frac{1}{2} \cdot (n-1)!$$

Angenommen, eine Addition benötigt 10^{-9} Sekunden [1 GFLOP CPU]

N	# Touren	Zeit
26	7 Quadrillion (10^{33})	6,4 Mrd Jahre

© 2006-2024 Jörg Striegnitz

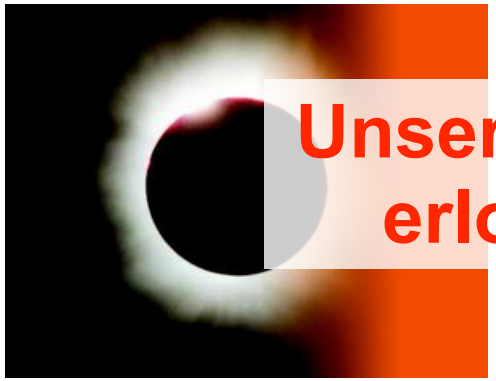
Lösung praktikabel?

- In einem vollständigen, schleifenfreien Graphen mit n Knoten gibt es genau $(n\text{-}1)!$ Hamiltonsche Kreise.
- Da es im Zusammenhang mit dem TSP egal ist, in welcher Richtung ein Hamiltonscher Kreis durchlaufen wird gilt:

Der Aufwand zur *exakten Lösung eines TSP* mit n Städten ist proportional zu

$$\frac{1}{2} \cdot (n-1)!$$

Angenommen, eine Addition benötigt 10^{-9} Sekunden [1 GFLOP CPU]

N	# Touren	Zeit
26	7 Quadrillion (10^{33})	 Unsere Sonne ist erloschen ☹

© 2006-2024 Jörg Striegnitz

Lösung praktikabel?

Algorithmus zur Bestimmung der exakten Lösung ist

***so* nicht praktikabel!**

© 2006-2024 Jörg Striegnitz

Es existieren Lösungen für größere Probleme ...

- **15.112 Städte**
- **Rechenzeit: 22,6 Jahre**
  [Skaliert auf 500Mhz CPU]

© 2006-2024 Jörg Striegnitz

Braucht man immer eine optimale Lösung?

- **Optimale (exakte) Lösung oft theoretischer Natur**
  - Schwankende Bedingungen (z.B. Wetter, Staus, Ausfälle)
  - Ungenaues / idealisiertes Modell (z.B. Anfahrtszeiten vom Motor, Distanzmaße, symmetrische Modelle)
  - Abstand *worst*- und *best case* u.U. relativ klein (163 km beim Rheinlandproblem)

- **Eine hinreichend gute Lösung muss rasch zur Verfügung stehen!**
  - Steuerung von Maschinen (Echtzeitbedingungen)

- **Daher: Entwickle Heuristik (Lösungsstrategie), welche schnell eine „relativ“ gute Lösung liefert**

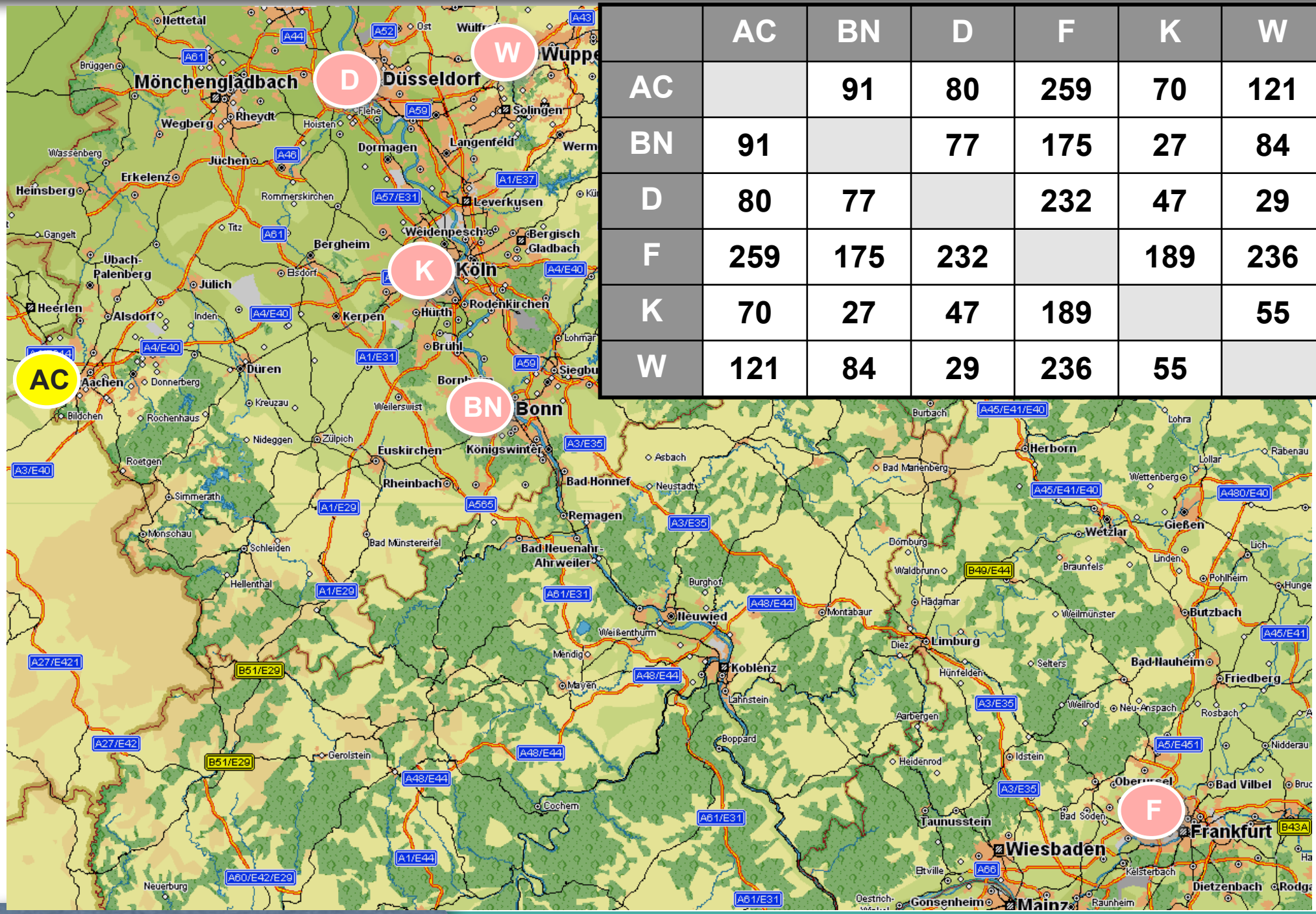
© 2006-2024 Jörg Striegnitz

Heuristiken und approximative Verfahren

- Wir stellen eine Heuristik zur Lösung des TSP vor und untersuchen sie
- Wir stellen eine Heuristik vor, die für ein ähnlich erscheinendes Problem garantiert die optimale Lösung findet
- Wir konstruieren aus dieser ein approximatives Verfahren (eine Heuristik **mit Gütegarantie**) für das TSP

© 2006-2024 Jörg Striegnitz

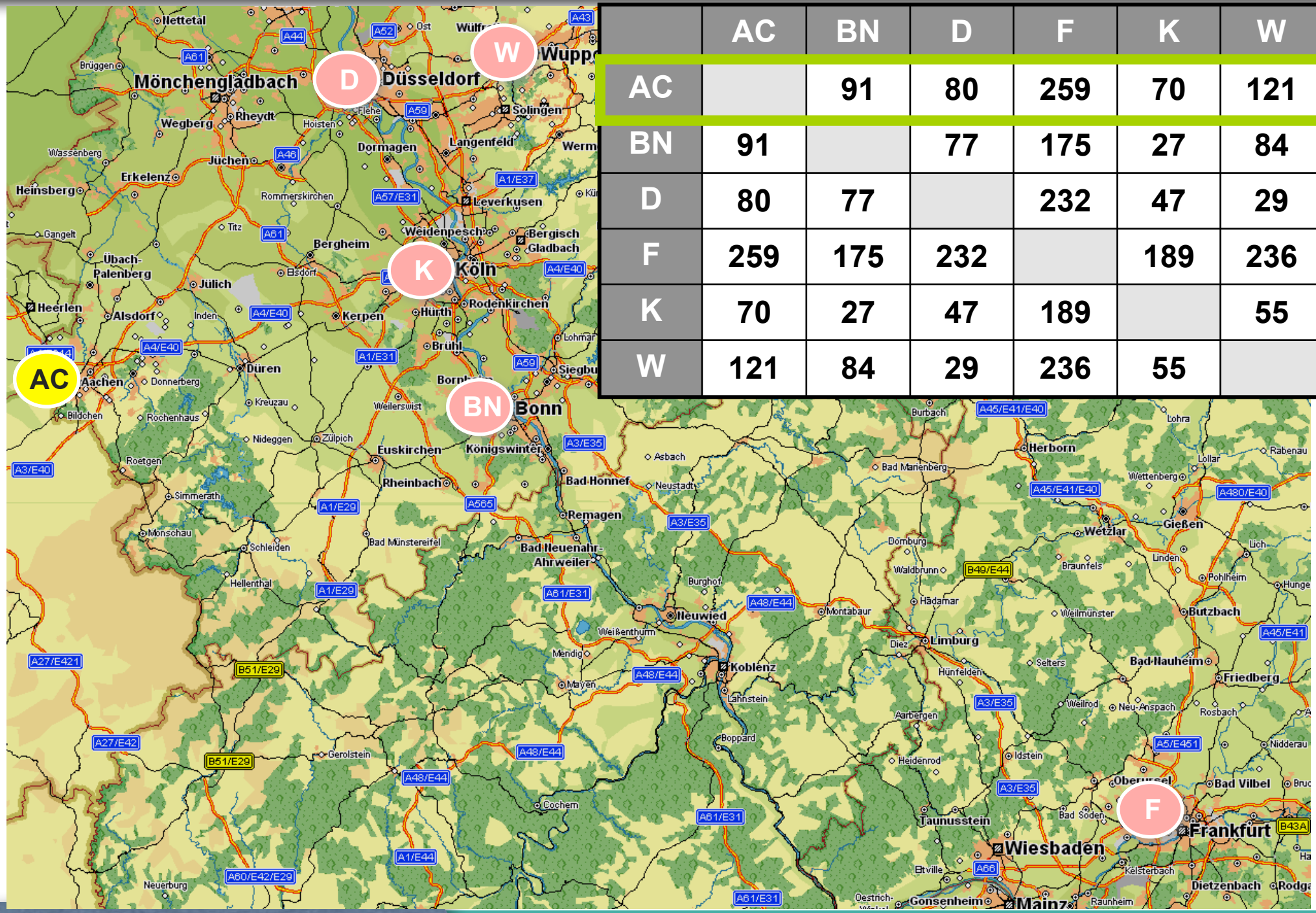
Eine Lösungsstrategie: Die *Nearest Neighbor* Heuristik



	AC	BN	D	F	K	W
AC		91	80	259	70	121
BN	91		77	175	27	84
D	80	77		232	47	29
F	259	175	232		189	236
K	70	27	47	189		55
W	121	84	29	236	55	

© 2006-2024 Jörg Striegnitz

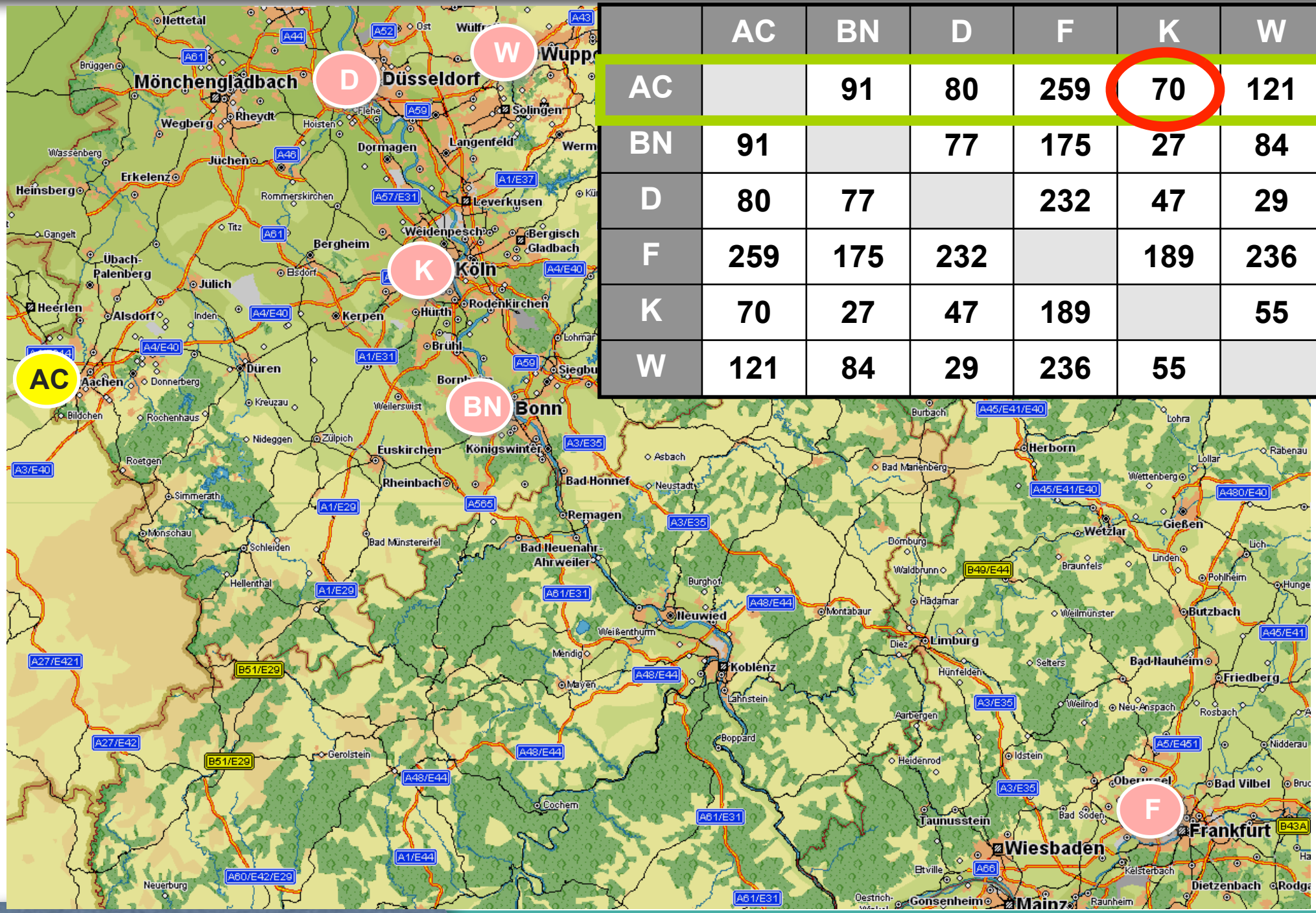
Eine Lösungsstrategie: Die *Nearest Neighbor* Heuristik



	AC	BN	D	F	K	W
AC		91	80	259	70	121
BN	91		77	175	27	84
D	80	77		232	47	29
F	259	175	232		189	236
K	70	27	47	189		55
W	121	84	29	236	55	

© 2006-2024 Jörg Striegnitz

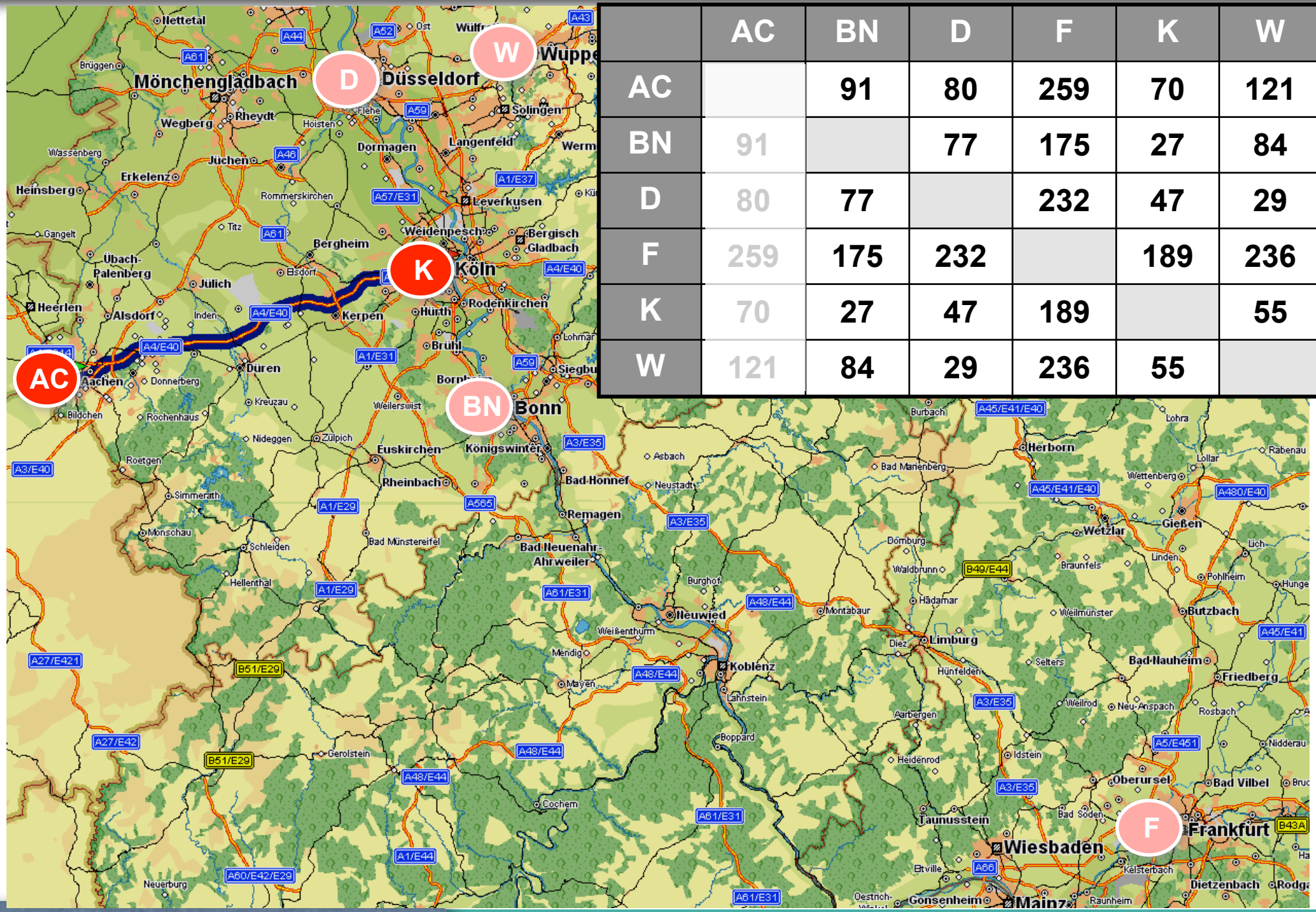
Eine Lösungsstrategie: Die *Nearest Neighbor* Heuristik



	AC	BN	D	F	K	W
AC		91	80	259	70	121
BN	91		77	175	27	84
D	80	77		232	47	29
F	259	175	232		189	236
K	70	27	47	189		55
W	121	84	29	236	55	

© 2006-2024 Jörg Striegnitz

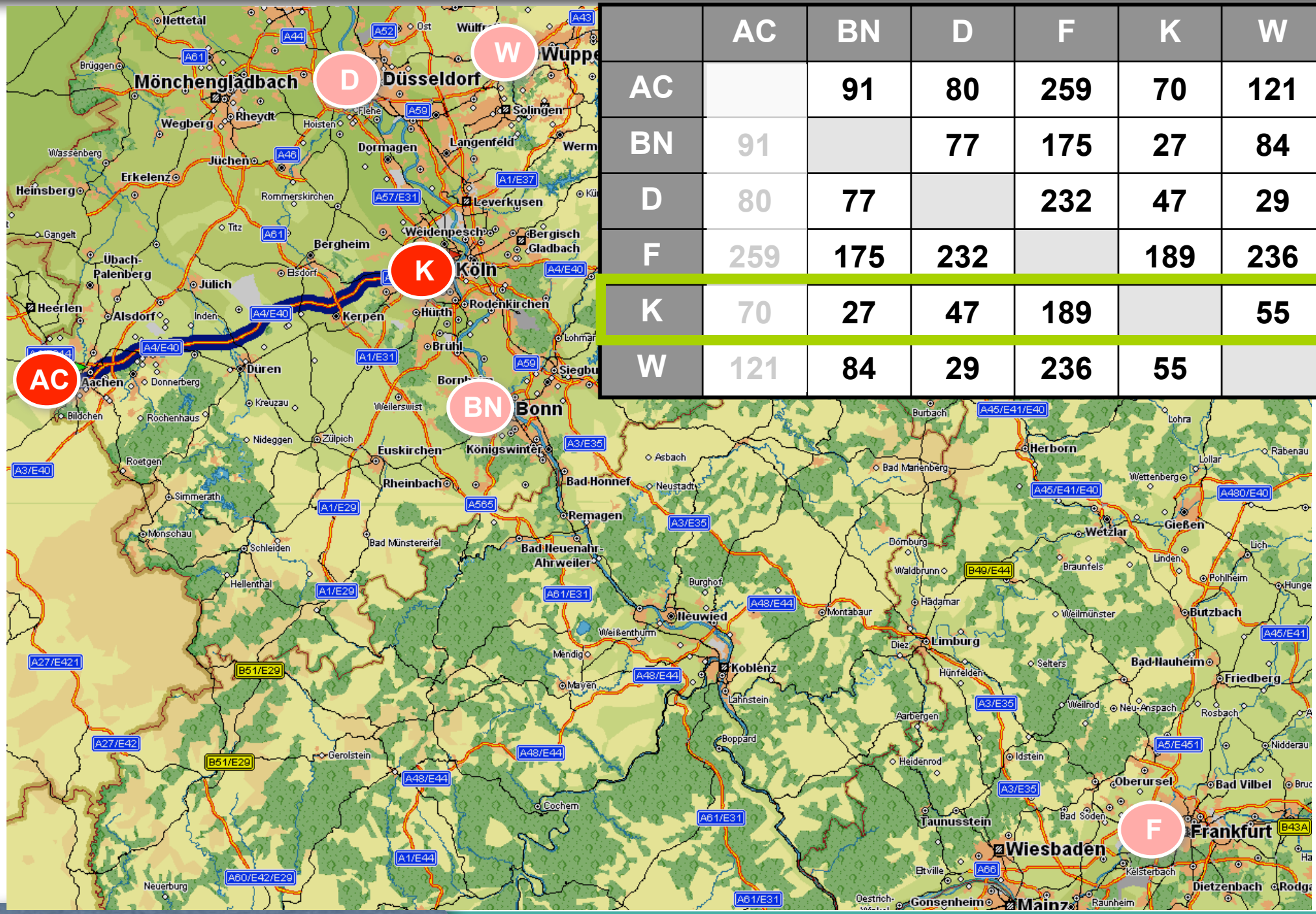
Eine Lösungsstrategie: Die *Nearest Neighbor* Heuristik



	AC	BN	D	F	K	W
AC		91	80	259	70	121
BN	91		77	175	27	84
D	80	77		232	47	29
F	259	175	232		189	236
K	70	27	47	189		55
W	121	84	29	236	55	

© 2006-2024 Jörg Striegnitz

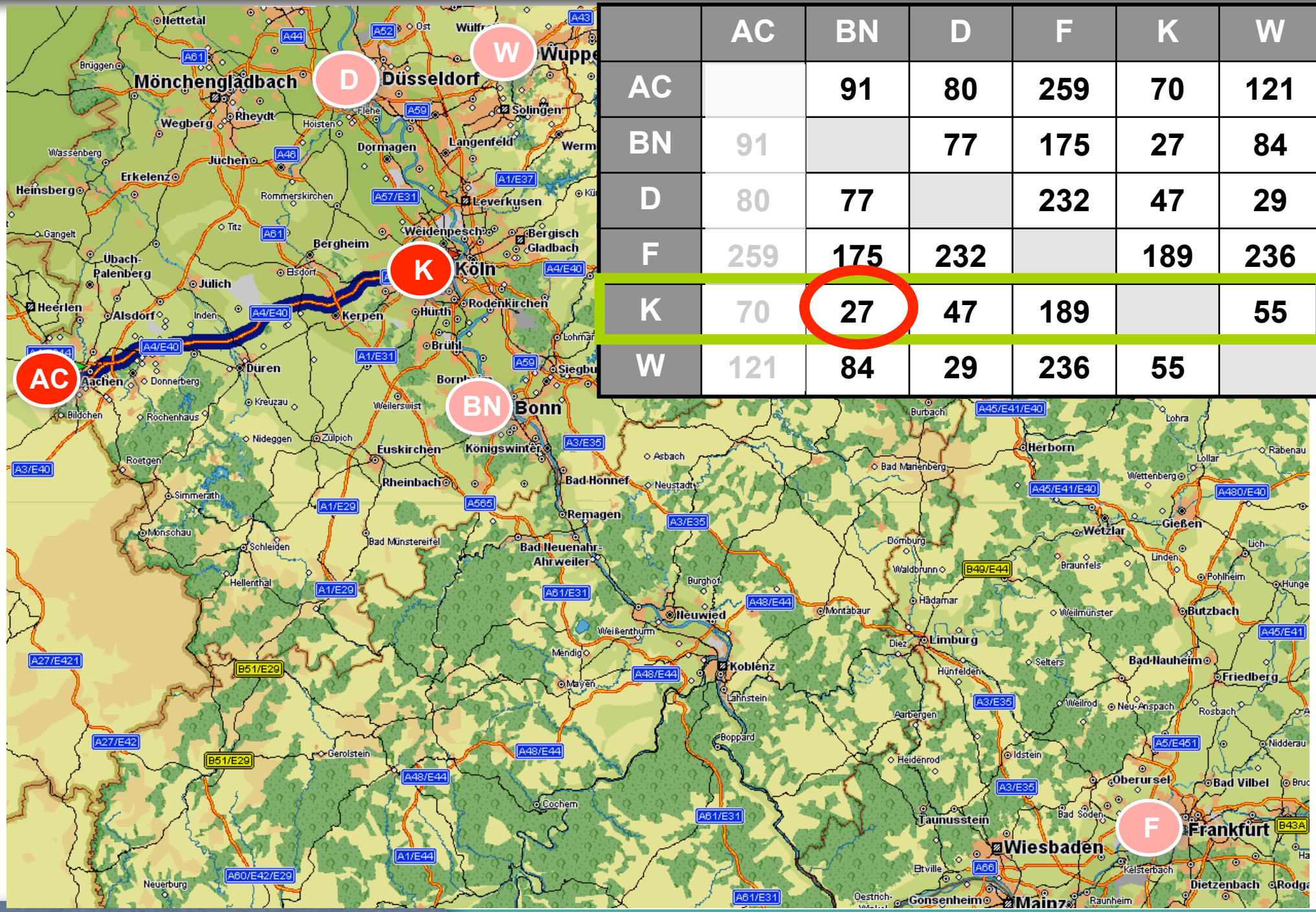
Eine Lösungsstrategie: Die *Nearest Neighbor* Heuristik



	AC	BN	D	F	K	W
AC		91	80	259	70	121
BN	91		77	175	27	84
D	80	77		232	47	29
F	259	175	232		189	236
K	70	27	47	189		55
W	121	84	29	236	55	

© 2006-2024 Jörg Striegnitz

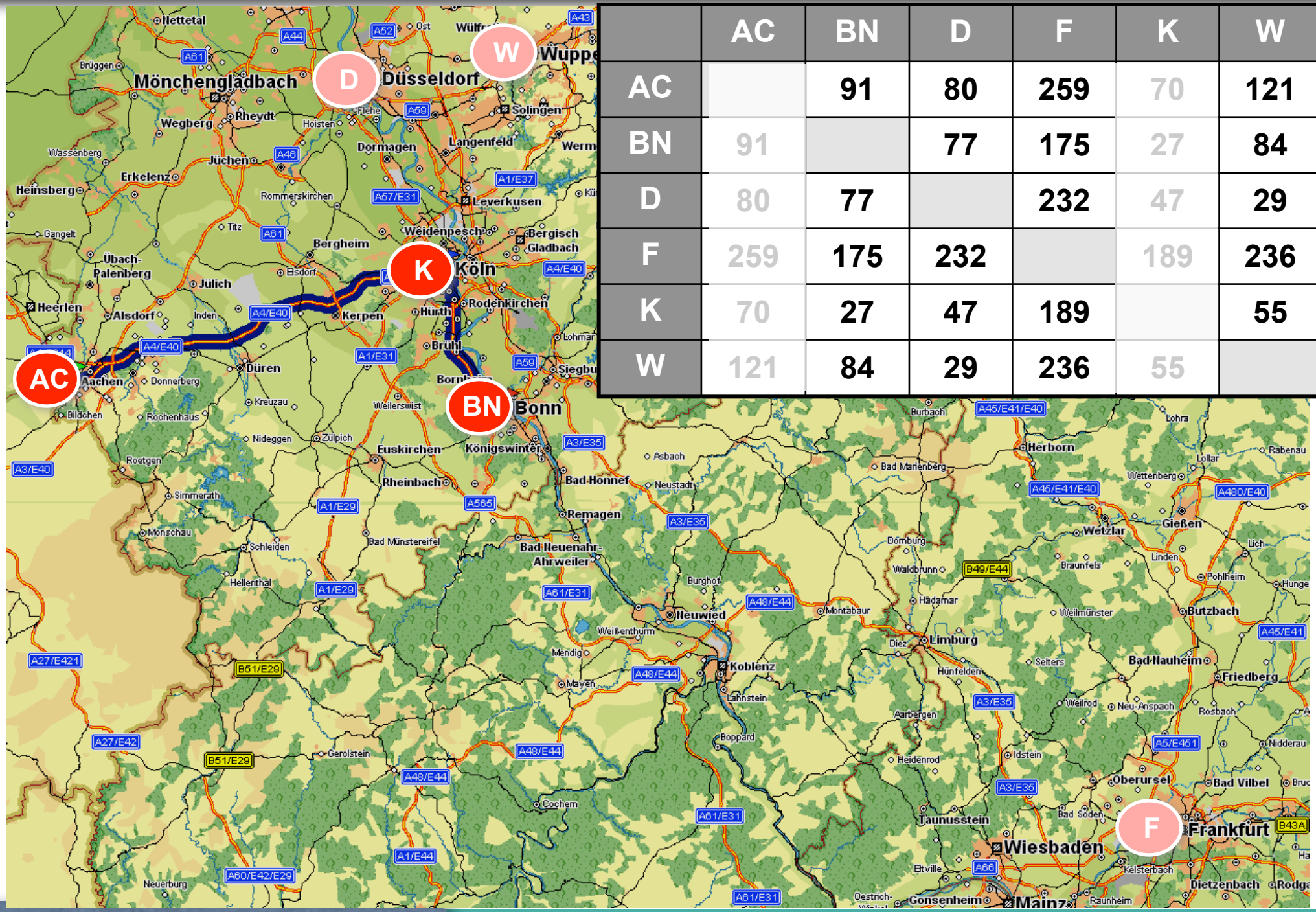
Eine Lösungsstrategie: Die *Nearest Neighbor* Heuristik



	AC	BN	D	F	K	W
AC		91	80	259	70	121
BN	91		77	175	27	84
D	80	77		232	47	29
F	259	175	232		189	236
K	70	27	47	189		55
W	121	84	29	236	55	

© 2006-2024 Jörg Striegnitz

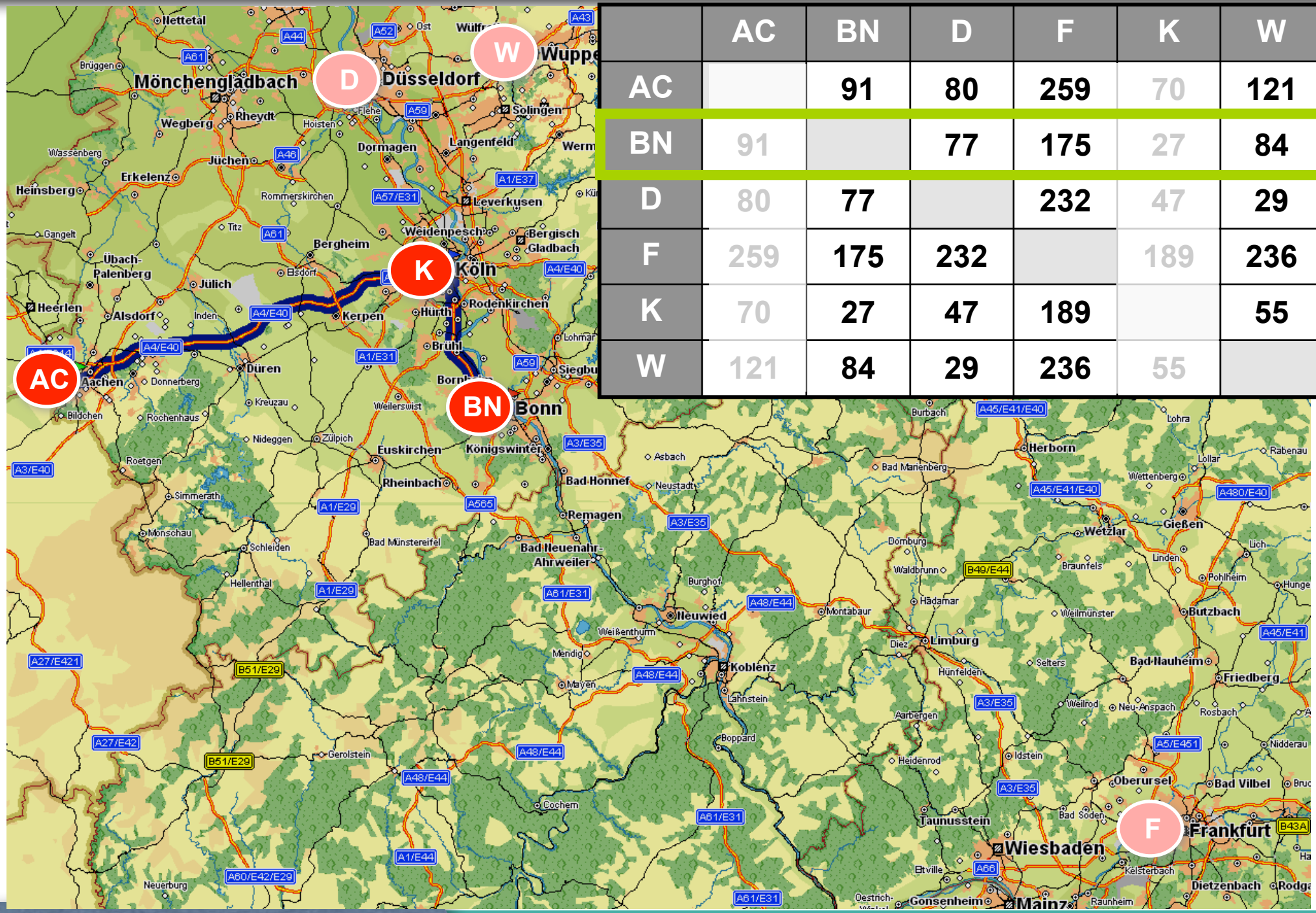
Eine Lösungsstrategie: Die *Nearest Neighbor* Heuristik



	AC	BN	D	F	K	W
AC		91	80	259	70	121
BN	91		77	175	27	84
D	80	77		232	47	29
F	259	175	232		189	236
K	70	27	47	189		55
W	121	84	29	236	55	

© 2006-2024 Jörg Striegnitz

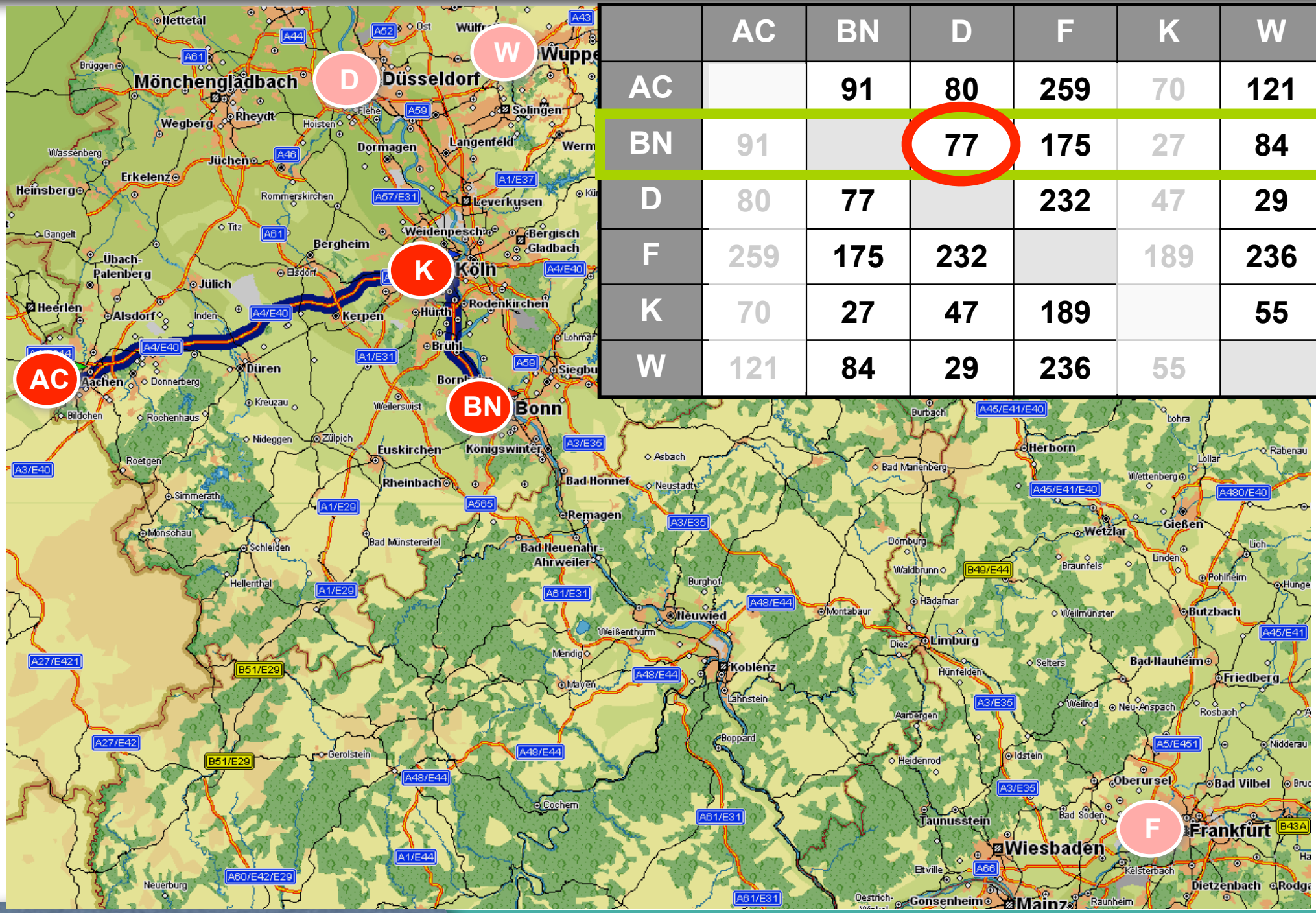
Eine Lösungsstrategie: Die *Nearest Neighbor* Heuristik



	AC	BN	D	F	K	W
AC		91	80	259	70	121
BN	91		77	175	27	84
D	80	77		232	47	29
F	259	175	232		189	236
K	70	27	47	189		55
W	121	84	29	236	55	

© 2006-2024 Jörg Striegnitz

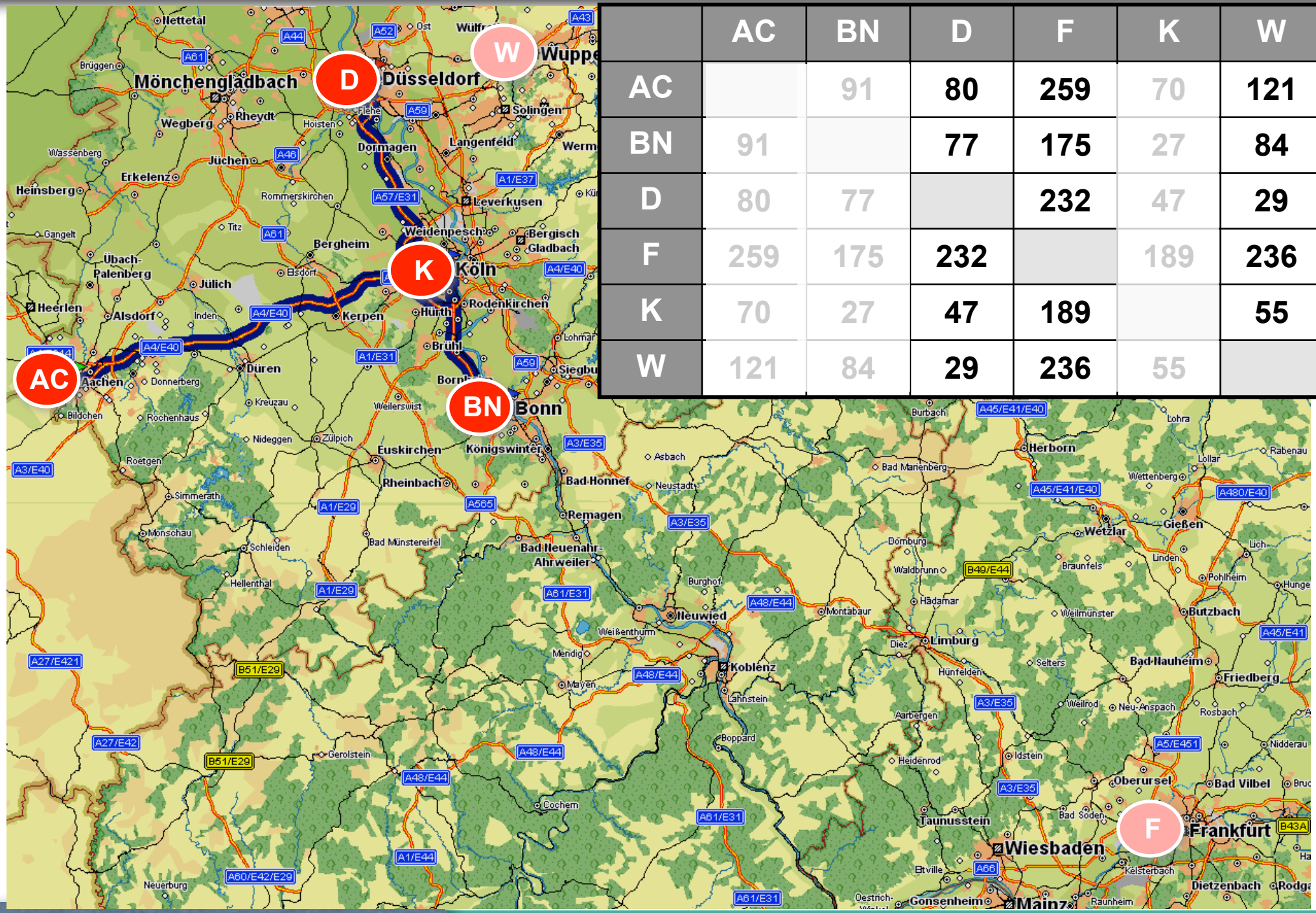
Eine Lösungsstrategie: Die *Nearest Neighbor* Heuristik



	AC	BN	D	F	K	W
AC		91	80	259	70	121
BN	91		77	175	27	84
D	80	77		232	47	29
F	259	175	232		189	236
K	70	27	47	189		55
W	121	84	29	236	55	

© 2006-2024 Jörg Striegnitz

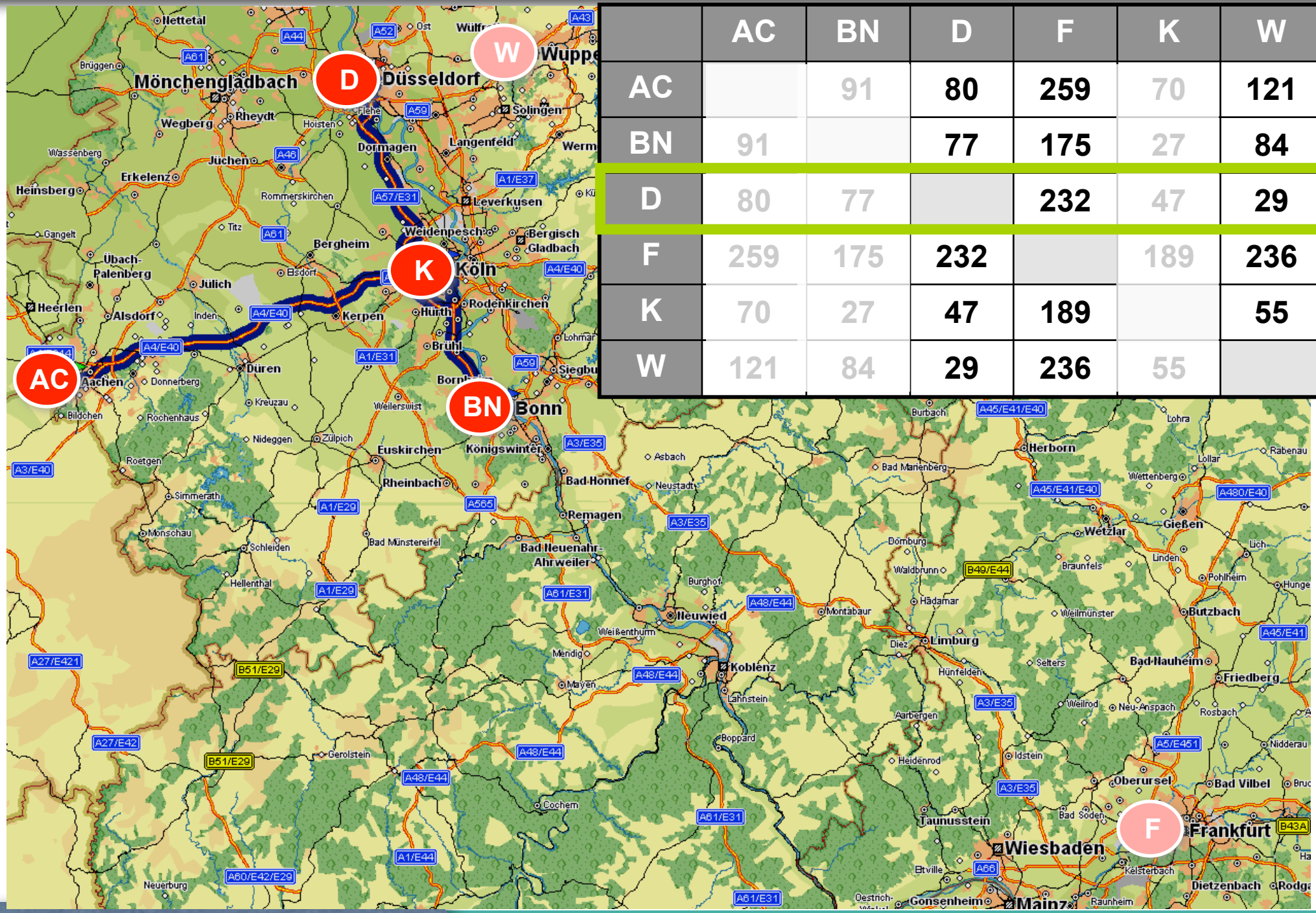
Eine Lösungsstrategie: Die *Nearest Neighbor* Heuristik



	AC	BN	D	F	K	W
AC		91	80	259	70	121
BN	91		77	175	27	84
D	80	77		232	47	29
F	259	175	232		189	236
K	70	27	47	189		55
W	121	84	29	236	55	

© 2006-2024 Jörg Striegnitz

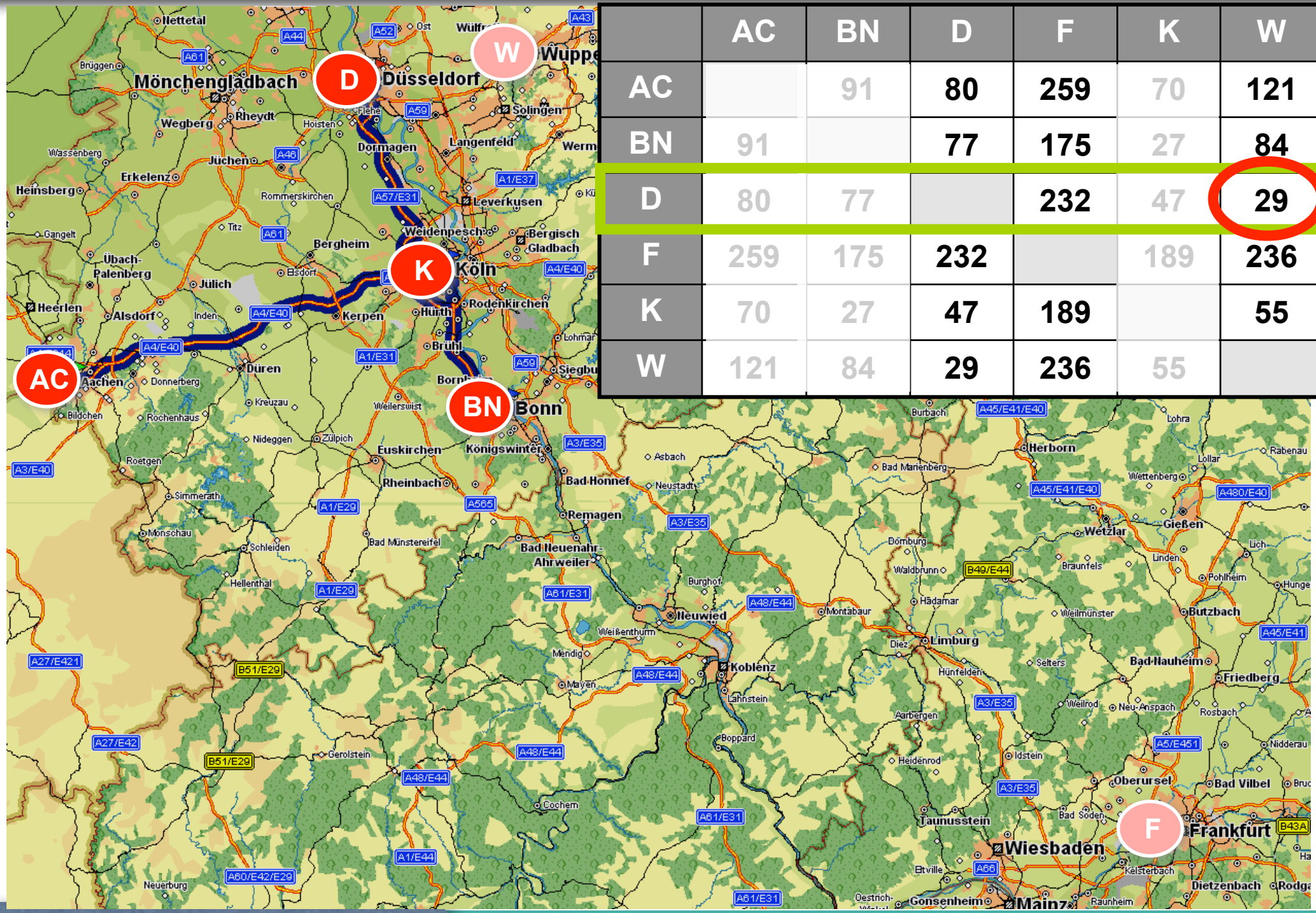
Eine Lösungsstrategie: Die *Nearest Neighbor* Heuristik



	AC	BN	D	F	K	W
AC		91	80	259	70	121
BN	91		77	175	27	84
D	80	77		232	47	29
F	259	175	232		189	236
K	70	27	47	189		55
W	121	84	29	236	55	

© 2006-2024 Jörg Striegnitz

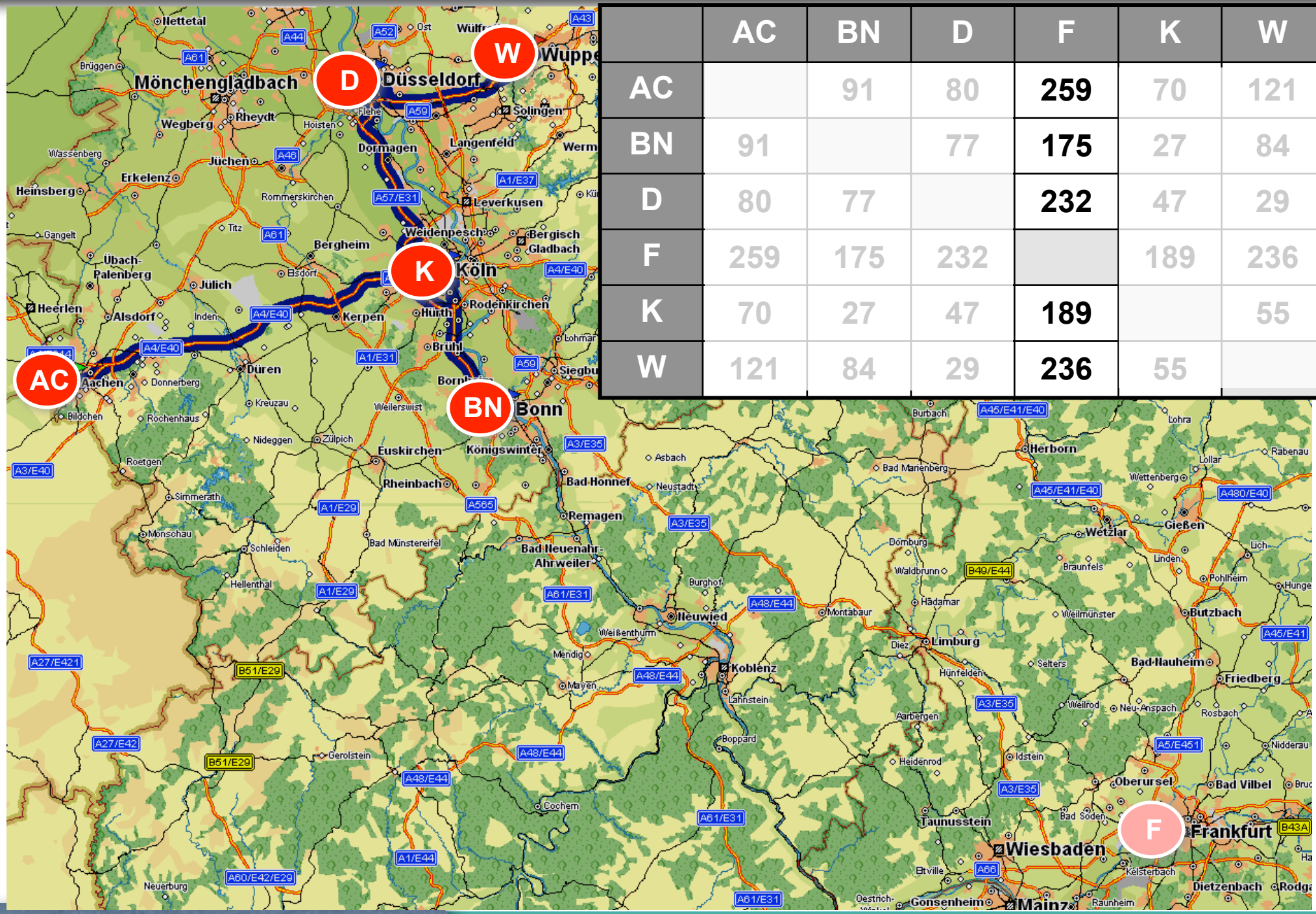
Eine Lösungsstrategie: Die *Nearest Neighbor* Heuristik



	AC	BN	D	F	K	W
AC		91	80	259	70	121
BN	91		77	175	27	84
D	80	77		232	47	29
F	259	175	232		189	236
K	70	27	47	189		55
W	121	84	29	236	55	

© 2006-2024 Jörg Striegnitz

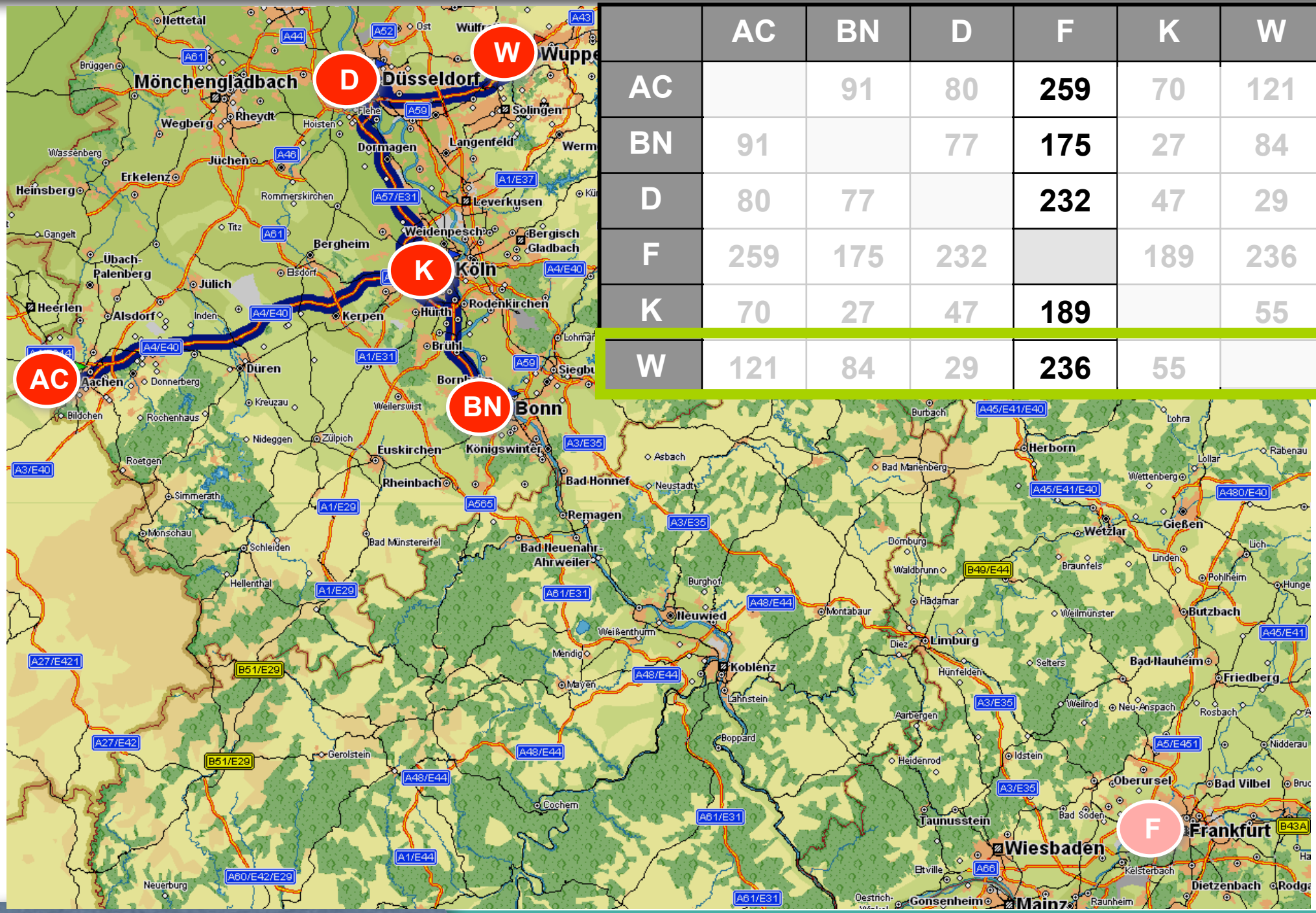
Eine Lösungsstrategie: Die *Nearest Neighbor* Heuristik



	AC	BN	D	F	K	W
AC		91	80	259	70	121
BN	91		77	175	27	84
D	80	77		232	47	29
F	259	175	232		189	236
K	70	27	47	189		55
W	121	84	29	236	55	

© 2006-2024 Jörg Striegnitz

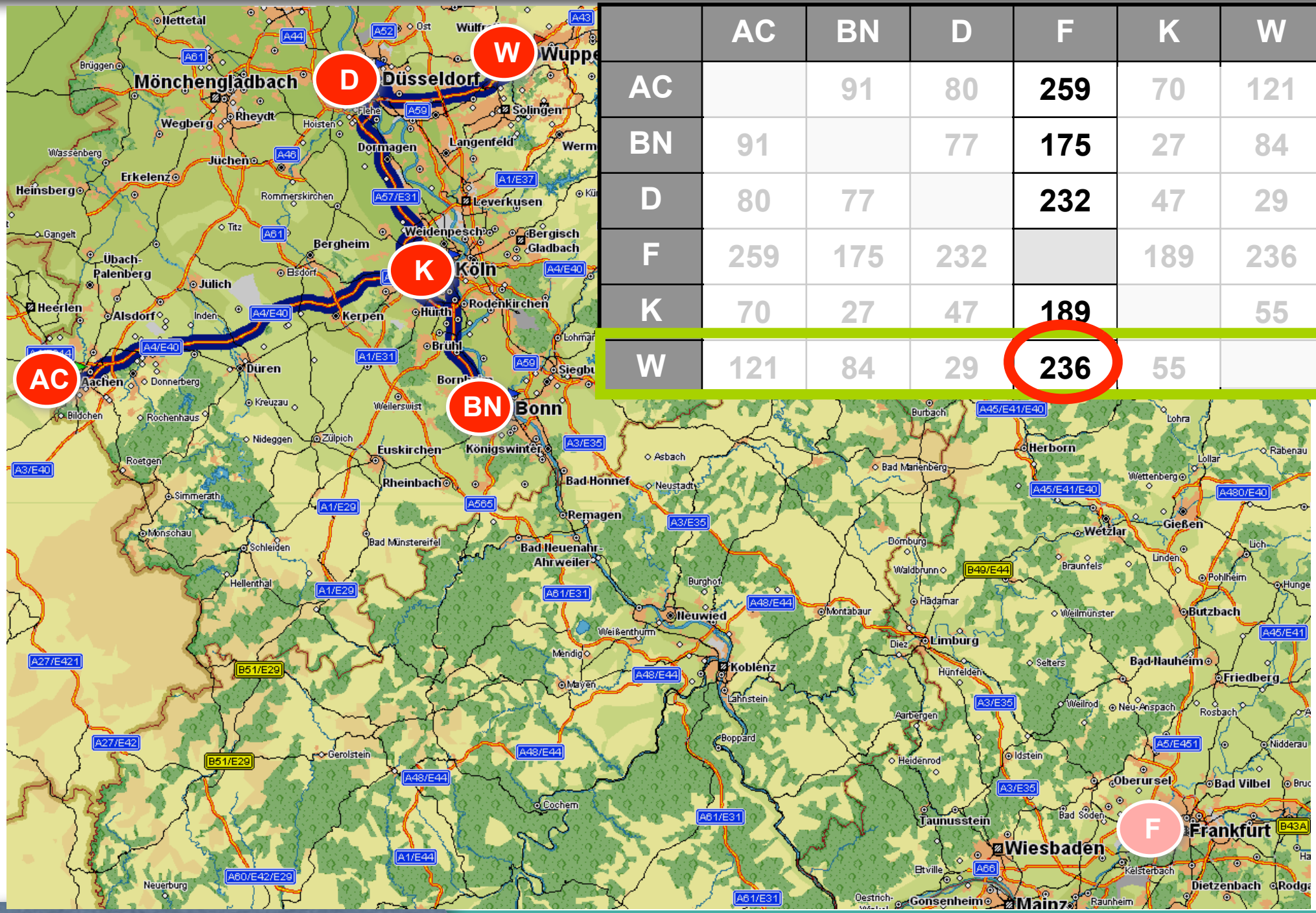
Eine Lösungsstrategie: Die *Nearest Neighbor* Heuristik



	AC	BN	D	F	K	W
AC		91	80	259	70	121
BN	91		77	175	27	84
D	80	77		232	47	29
F	259	175	232		189	236
K	70	27	47	189		55
W	121	84	29	236	55	

© 2006-2024 Jörg Striegnitz

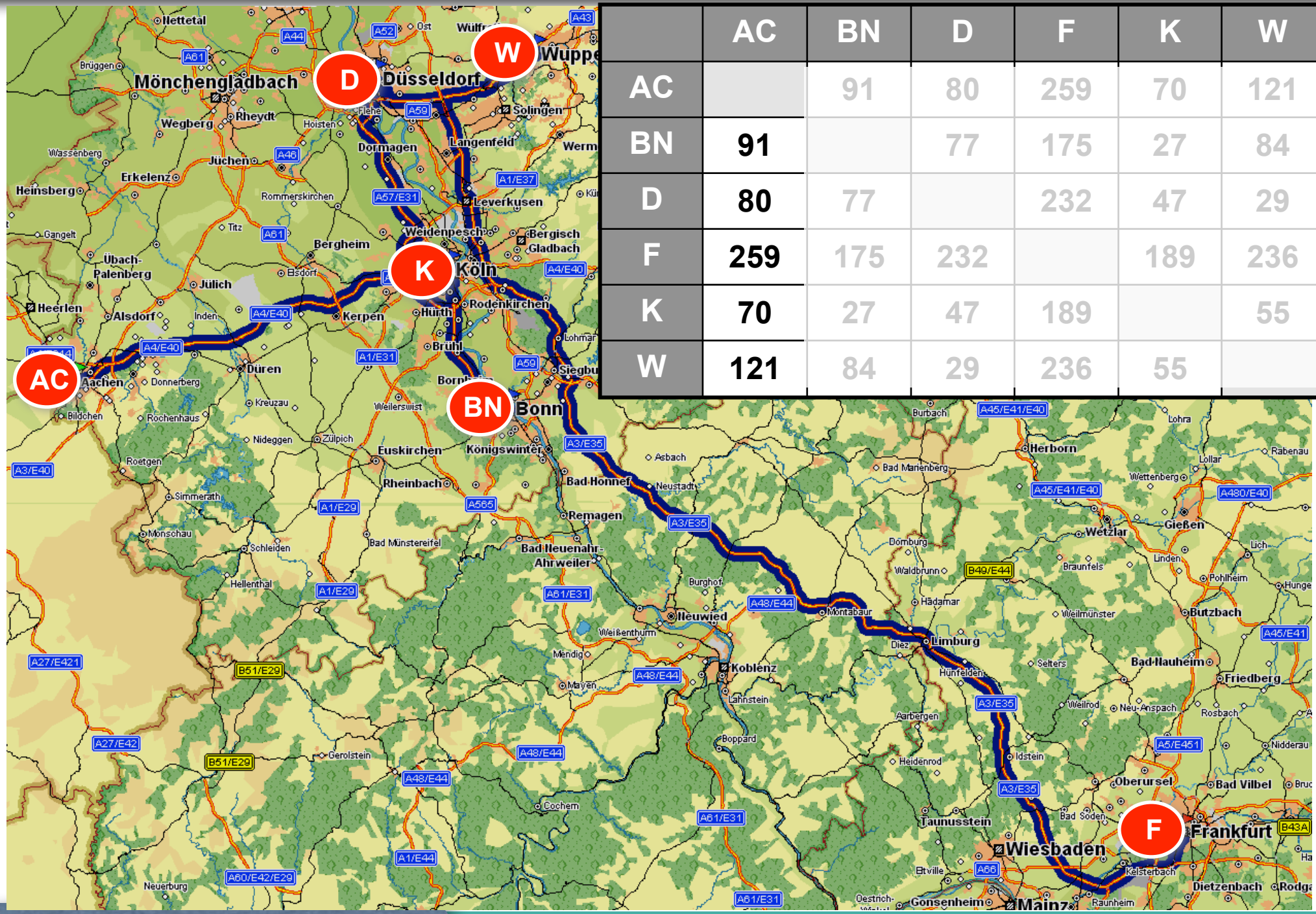
Eine Lösungsstrategie: Die *Nearest Neighbor* Heuristik



	AC	BN	D	F	K	W
AC		91	80	259	70	121
BN	91		77	175	27	84
D	80	77		232	47	29
F	259	175	232		189	236
K	70	27	47	189		55
W	121	84	29	236	55	

© 2006-2024 Jörg Striegnitz

Eine Lösungsstrategie: Die *Nearest Neighbor* Heuristik



	AC	BN	D	F	K	W
AC		91	80	259	70	121
BN	91		77	175	27	84
D	80	77		232	47	29
F	259	175	232		189	236
K	70	27	47	189		55
W	121	84	29	236	55	

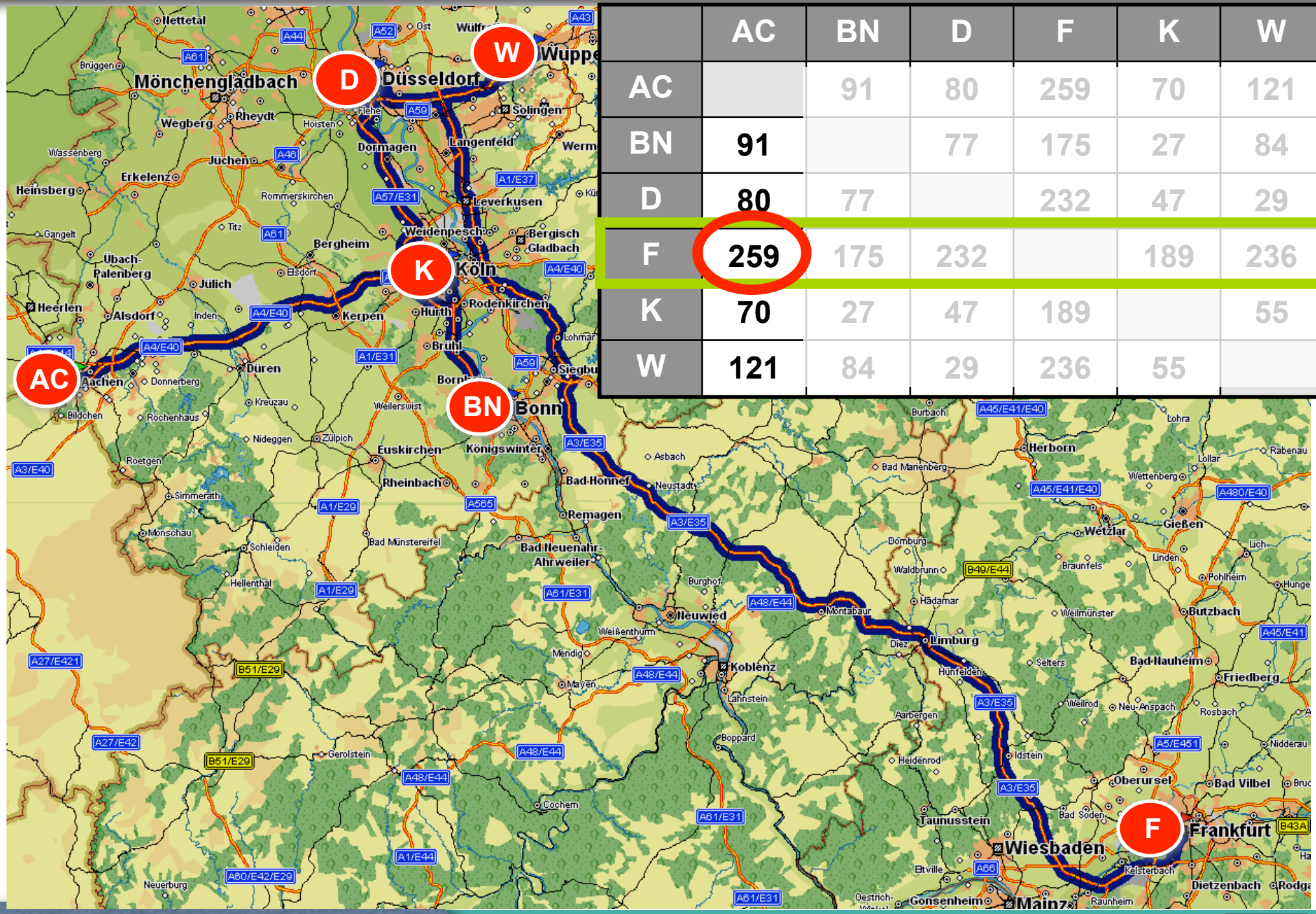
© 2006-2024 Jörg Striegnitz

Eine Lösungsstrategie: Die *Nearest Neighbor* Heuristik

	AC	BN	D	F	K	W
AC		91	80	259	70	121
BN	91		77	175	27	84
D	80	77		232	47	29
F	259	175	232		189	236
K	70	27	47	189		55
W	121	84	29	236	55	

© 2006-2024 Jörg Striegnitz

Eine Lösungsstrategie: Die *Nearest Neighbor* Heuristik



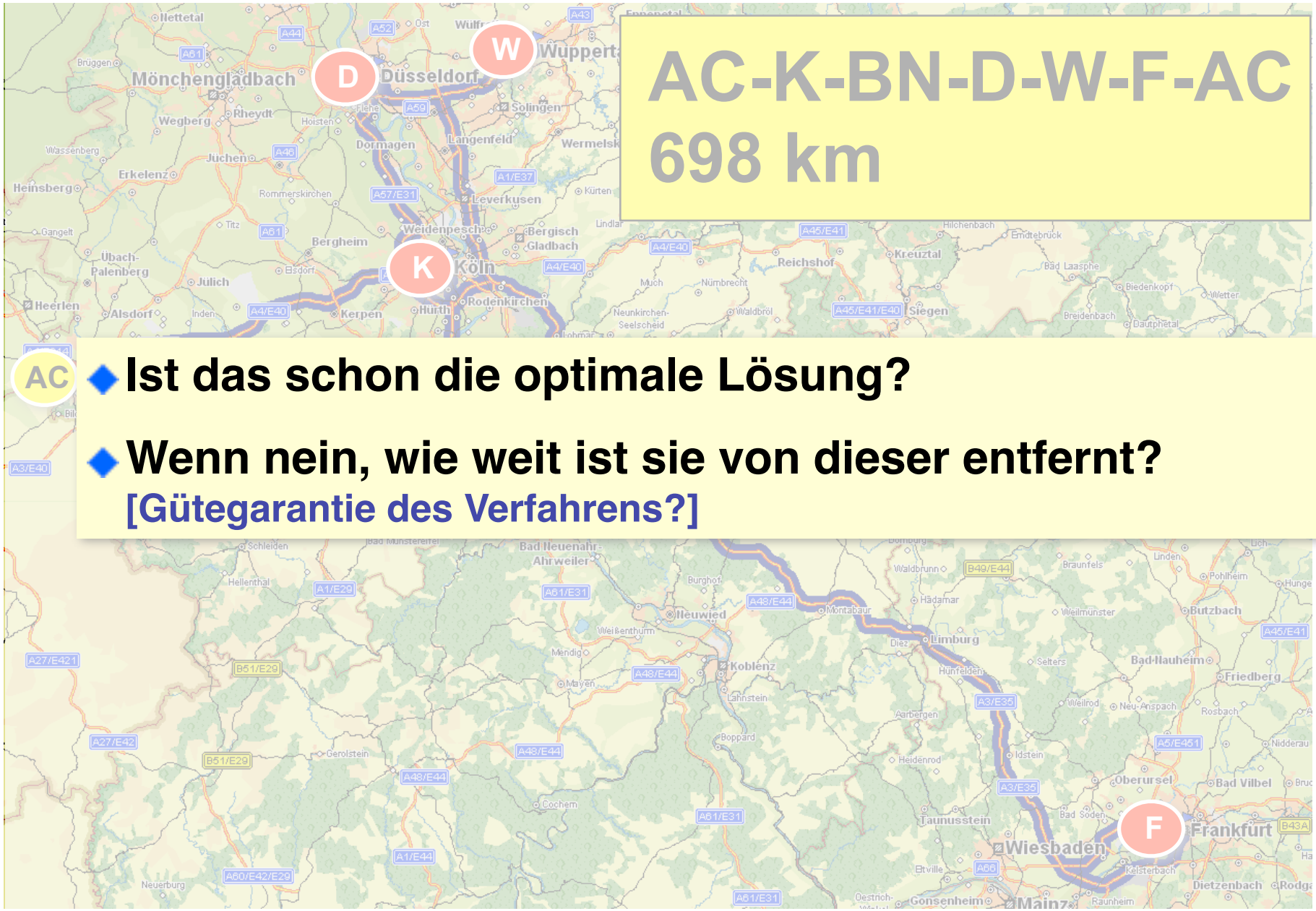
	AC	BN	D	F	K	W
AC		91	80	259	70	121
BN	91		77	175	27	84
D	80	77		232	47	29
F	259	175	232		189	236
K	70	27	47	189		55
W	121	84	29	236	55	

© 2006-2024 Jörg Striegnitz

Ergebnis *Nearest Neighbor* Heuristik

AC-K-BN-D-W-F-AC
698 km
AC
K
BN
D
W
F
Mönchengladbach
Düsseldorf
Wuppertal
Solingen
Langenfeld
Dormagen
Leverkusen
Weidenpesch
Bergisch Gladbach
Köln
Rodenkirchen
Hürth
Brühl
Kerpen
Aachen
Düren
Bonn
Siegburg
Königswinter
Bad Honnef
Remagen
Bad Neuenahr-Ahrweiler
Euskirchen
Rheinbach
Neuwied
Koblenz
Limburg
Montabaur
Wiesbaden
Frankfurt
Mainz
Gießen
Wetzlar
Marburg
Siegen
Herborn
Bad Nauheim
Friedberg
Butzbach
Oberursel
Bad Vilbel
Taunusstein
Idstein
Heidenrod
Aarbergen
Lahnstein
Boppard
Cochem
Gerolstein
Mayen
Bad Münstereifel
Monschau
Heerlen
Alsdorf
Jülich
Übach-Palenberg
Erkelenz
Heinsberg
Wegberg
Rheydt
Jüchen
Bergheim
Rommerskirchen
Kreuzau
Nideggen
Zülpich
Roetgen
Simmerath
Schleiden
Hellenthal
Neuerburg
Dietzenbach
Gonsenheim
Kelsterbach
Raunheim
Etzville
Oestrich-Winkel
Bad Soden
Hochheim
Eschborn
Bad Homburg
Friedrichsdorf
Grävenwiesbach
Usingen
Weilrod
Neu-Anspach
Rosbach
Nidderau
Hanau
Bad Laasphe
Biedenkopf
Dautphetal
Breidenbach
Ockershausen
Gladenbach
Lohra
Lollar
Rabenau
Wetzlar
Braunfels
Weilmünster
Selters
Hünfelden
Diez
Hadamar
Waldbrunn
Burbach
Neunkirchen
Wilnsdorf
Eschenburg
Betzdorf
Wissen
Windeck
Waldbröl
Nümbrecht
Much
Neunkirchen-Seelscheid
Lohmar
Eitorf
Asbach
Neustadt
Bad Manenberg
Burghof
Weißenthurm
Mendig
Dreis
Kreuztal
Reichshof
Hilchenbach
Emdtebrück
Lindlar
Kürten
Hoisten
Flehe
Ost
Wülfrath
Wermelskirchen
Titz
Inden
Donnerberg
Bildchen
Rochenhaus
Weilerswist
Wassenberg
Gangelt
Brüggen
Elsdorf
Pulheim
Porz
Troisdorf
Bad Münstereifel
Bad Neuenahr-Ahrweiler
Sinzig
Bad Breisig
Andernach
Neuwied
Weißenthurm
Koblenz
Lahnstein
Boppard
Bad Ems
Nassau
Diez
Limburg
Hadamar
Montabaur
Bad Marienberg
Altenkirchen
Betzdorf
Siegen
Dillenburg
Haiger
Herborn
Dillenburg
Wetzlar
Gießen
Lich
Pohlheim
Butzbach
Friedberg
Bad Nauheim
Bad Homburg
Oberursel
Bad Vilbel
Hanau
Frankfurt
Wiesbaden
Mainz
Gonsenheim
Raunheim
Kelsterbach
Dietzenbach
Rodgau
Langen
Neu-Isenburg
Offenbach
Hanau
Aschaffenburg
Miltenberg
Wertheim
Tauberbischofsheim
Bad Mergentheim
Weinsberg
Heilbronn
Bad Wimpfen
Eberbach
Mosbach
Buchen
Walldürn
Miltenberg
Aschaffenburg
Alzenau
Gelnhausen
Bad Orb
Schlüchtern
Fulda
Lauterbach
Alsfeld
Marburg
Biedenkopf
Dautphetal
Breidenbach
Bad Laasphe
Hilchenbach
Kreuztal
Siegen
Netphen
Freudenberg
Kirchen
Betzdorf
Wissen
Windeck
Eitorf
Lohmar
Much
Nümbrecht
Waldbröl
Neunkirchen-Seelscheid
Bad Honnef
Königswinter
Siegburg
Troisdorf
Bad Godesberg
Bonn
Brühl
Hürth
Rodenkirchen
Köln
Porz
Leverkusen
Langenfeld
Solingen
Wuppertal
Düsseldorf
Mönchengladbach
Rheydt
Wegberg
Erkelenz
Heinsberg
Übach-Palenberg
Heerlen
Alsdorf
Jülich
Aachen
Düren
Euskirchen
Rheinbach
Bad Münstereifel
Monschau
Schleiden
Hellenthal
Simmerath
Roetgen
Nideggen
Zülpich
Kreuzau
Rochenhaus
Bildchen
Donnerberg
Inden
Titz
Rommerskirchen
Bergheim
Elsdorf
Kerpen
Hürth
Brühl
Bonn
Siegburg
Königswinter
Bad Honnef
Remagen
Bad Neuenahr-Ahrweiler
Sinzig
Bad Breisig
Andernach
Neuwied
Weißenthurm
Mendig

Ergebnis *Nearest Neighbor* Heuristik



- **Ist das schon die optimale Lösung?**
- **Wenn nein, wie weit ist sie von dieser entfernt?**
  [Gütegarantie des Verfahrens?]

© 2006-2024 Jörg Striegnitz

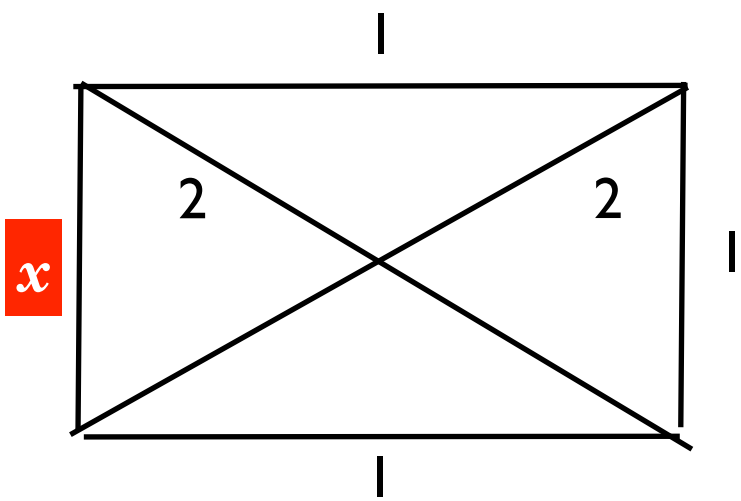
Verfahrensgüte

D Güte eines Hamiltonschen Kreises

Sei $G = (V, E)$ ein gewichteter Graph und p_{Opt} ein Hamiltonscher Kreis in G von minimalem Gewicht. Dann ist die **Güte eines Hamiltonschen Kreises** p in G definiert durch

$$Q(p) = \frac{\text{Weight}(p)}{\text{Weight}(p_{Opt})}$$

Die *Nearest Neighbor* Heuristik liefet beliebig schlechte Lösungen!



- Sei $x>3$

© 2006-2024 Jörg Striegnitz

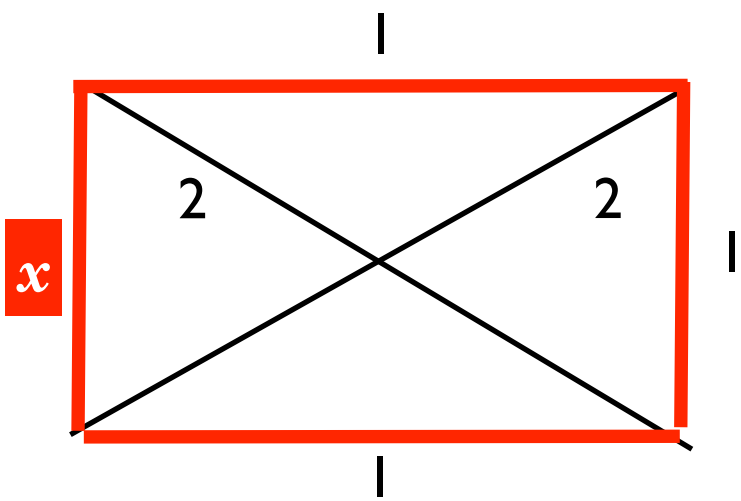
Verfahrensgüte

D Güte eines Hamiltonschen Kreises

Sei $G = (V, E)$ ein gewichteter Graph und p_{Opt} ein Hamiltonscher Kreis in G von minimalem Gewicht. Dann ist die **Güte eines Hamiltonschen Kreises** p in G definiert durch

$$Q(p) = \frac{\text{Weight}(p)}{\text{Weight}(p_{Opt})}$$

Die *Nearest Neighbor* Heuristik liefet beliebig schlechte Lösungen!



- Sei $x>3$
- Die *Neareast Neighbor* Heuristik wählt immer den **roten Pfad** mit Gewicht $3+x$

© 2006-2024 Jörg Striegnitz

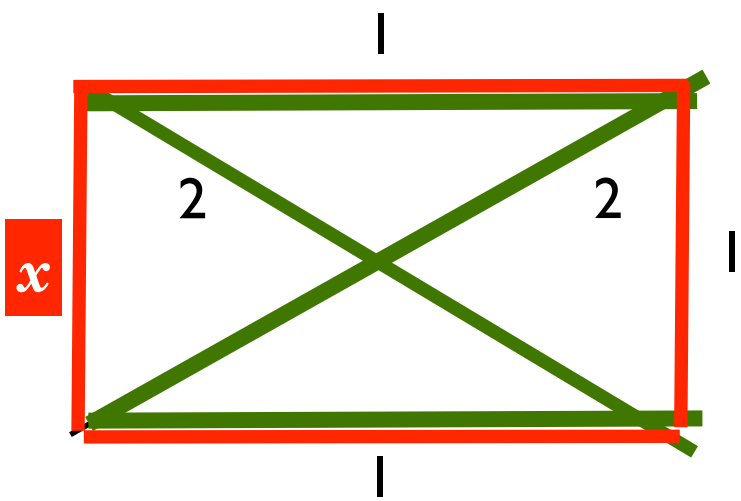
Verfahrensgüte

D Güte eines Hamiltonschen Kreises

Sei $G = (V, E)$ ein gewichteter Graph und p_{Opt} ein Hamiltonscher Kreis in G von minimalem Gewicht. Dann ist die **Güte eines Hamiltonschen Kreises** p in G definiert durch

$$Q(p) = \frac{\text{Weight}(p)}{\text{Weight}(p_{Opt})}$$

Die *Nearest Neighbor* Heuristik liefet beliebig schlechte Lösungen!



- Sei $x>3$
- Die *Neareast Neighbor* Heuristik wählt immer den **roten Pfad** mit Gewicht $3+x$
- Optimal ist grüner Pfad mit Gewicht 6

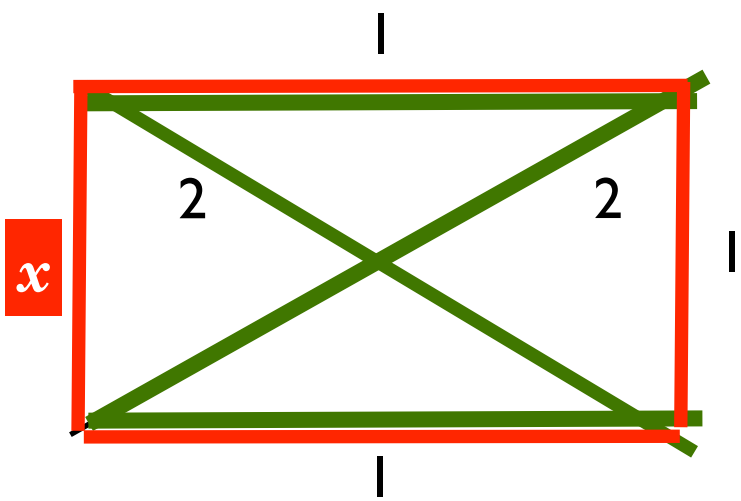
© 2006-2024 Jörg Striegnitz

D Güte eines Hamiltonschen Kreises

Sei $G = (V, E)$ ein gewichteter Graph und p_{Opt} ein Hamiltonscher Kreis in G von minimalem Gewicht. Dann ist die **Güte eines Hamiltonschen Kreises** p in G definiert durch

$$Q(p) = \frac{\text{Weight}(p)}{\text{Weight}(p_{Opt})}$$

Die *Nearest Neighbor* Heuristik liefet beliebig schlechte Lösungen!



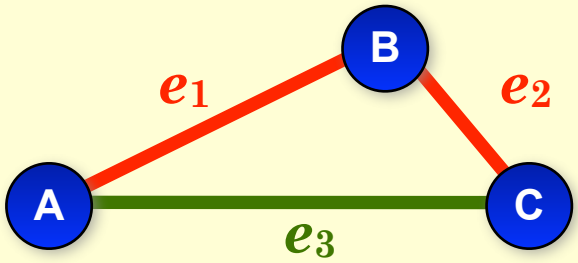
- Sei $x>3$
- Die *Neareast Neighbor* Heuristik wählt immer den **roten Pfad** mit Gewicht $3+x$
- Optimal ist grüner Pfad mit Gewicht 6
- Also gilt:

$$Q(p) = \frac{3+x}{6}$$

© 2006-2024 Jörg Striegnitz

Metrische *Traveling Salesman* Probleme

- TSP, so wie bis jetzt definiert, oft zu allgemein
- In der Praxis gilt oft die **Dreiecksungleichung**:



$$\text{weight}(e_3) \leq \text{weight}(e_1) + \text{weight}(e_2)$$

Der direkte Weg ist immer der kürzeste!

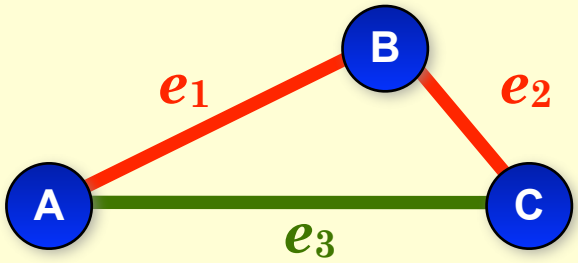
- TSPs, in denen Dreiecksungleichung gilt heißen **Metrische TSPs**
- Für metrische TSPs liefert die *Nearest Neighbor* Heuristik:

$$Q(p) \leq \frac{1 + \lceil \mathrm{ld}(n) \rceil}{2}$$

© 2006-2024 Jörg Striegnitz

Metrische *Traveling Salesman* Probleme

- TSP, so wie bis jetzt definiert, oft zu allgemein
- In der Praxis gilt oft die **Dreiecksungleichung**:



$$\text{weight}(e_3) \leq \text{weight}(e_1) + \text{weight}(e_2)$$

Der direkte Weg ist immer der kürzeste!

- TSPs, in denen Dreiecksungleichung gilt heißen **Metrische TSPs**
- Für metrische TSPs liefert die *Nearest Neighbor* Heuristik:

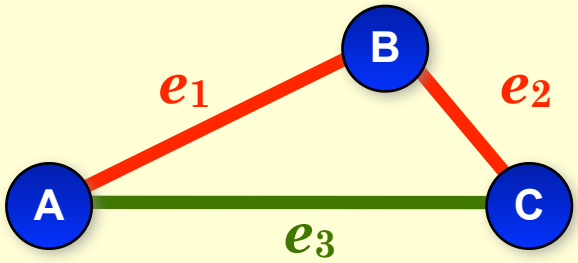
$$Q(p) \leq \frac{1 + \lceil \mathrm{ld}(n) \rceil}{2}$$

n	Q(p)
4	1,5
8	2
32	3
128	4

© 2006-2024 Jörg Striegnitz

Metrische *Traveling Salesman* Probleme

- TSP, so wie bis jetzt definiert, oft zu allgemein
- In der Praxis gilt oft die **Dreiecksungleichung**:



$$\text{weight}(e_3) \leq \text{weight}(e_1) + \text{weight}(e_2)$$

Der direkte Weg ist immer der kürzeste!

- TSPs, in denen Dreiecksungleichung gilt heißen **Metrische TSPs**
- Für metrische TSPs liefert die *Nearest Neighbor* Heuristik:

$$Q(p) \leq \frac{1 + \lceil \mathrm{ld}(n) \rceil}{2}$$

n	Q(p)
4	1,5
8	2
32	3
128	4

☹ **Güte bleibt abhängig vom Input !**

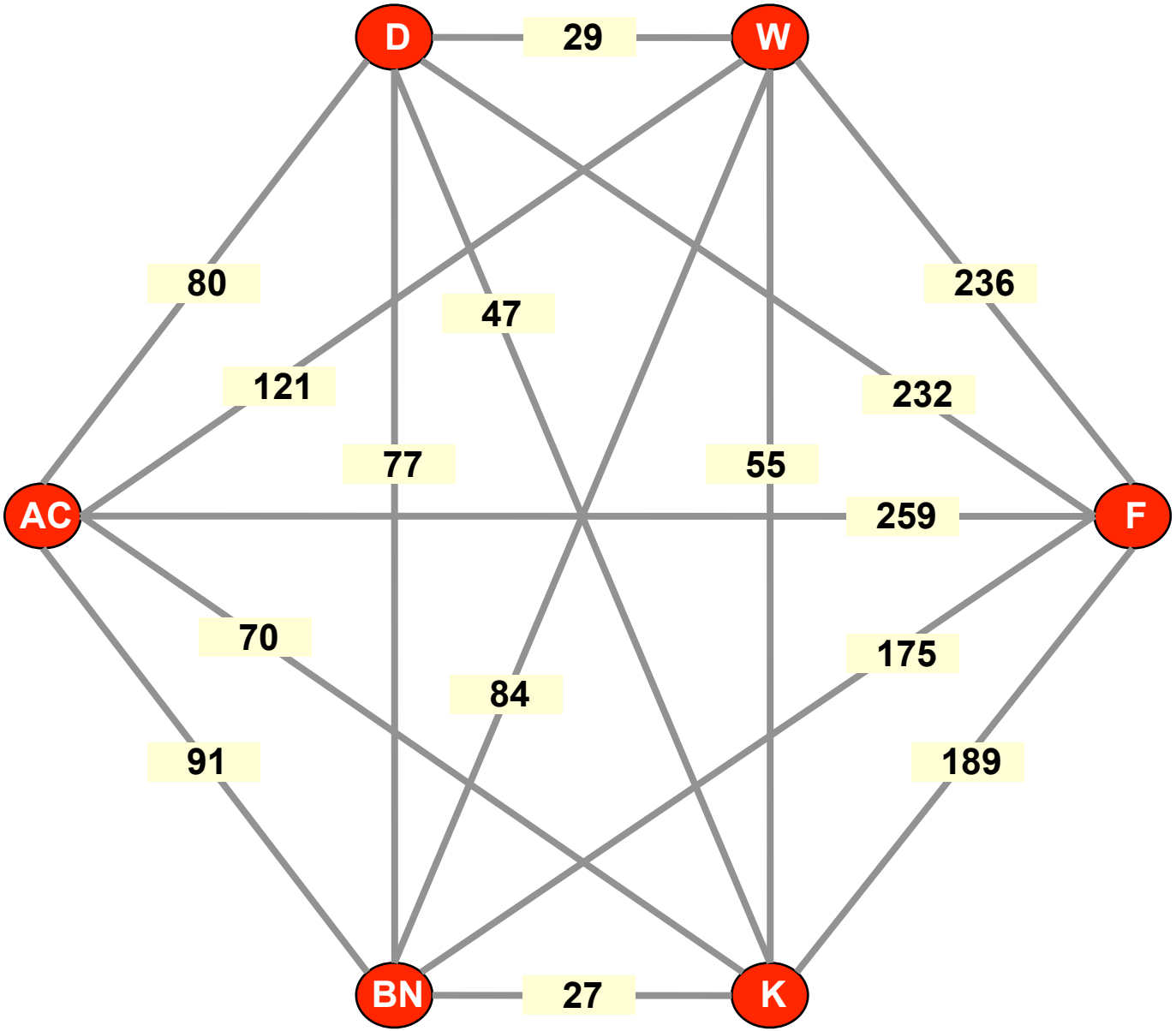
© 2006-2024 Jörg Striegnitz

MST-Heuristik

- **Nur für metrische TSPs anwendbar!**
- **Heuristik, die auf der Konstruktion eines MST beruht**
  - Suche MST
  - Konstruiere aus MST gerichteten, gewichteten und zyklischen Graphen
    aus jeder ungerichteten Kante werden zwei gewichtsgleiche gerichtete
  - Traversiere Graph im Sinne eines Eulerschen Pfades
    jede Kante wird genau einmal durchlaufen
    - Durchlaufene Kanten werden in den Ergebnispfad aufgenommen
    - Trifft man dabei auf einen bereits besuchten Knoten, so werden diese übersprungen und man fügt zum Ergebnispfad die zum nächsten Knoten nicht besuchten Knoten hinzu

© 2006-2024 Jörg Striegnitz

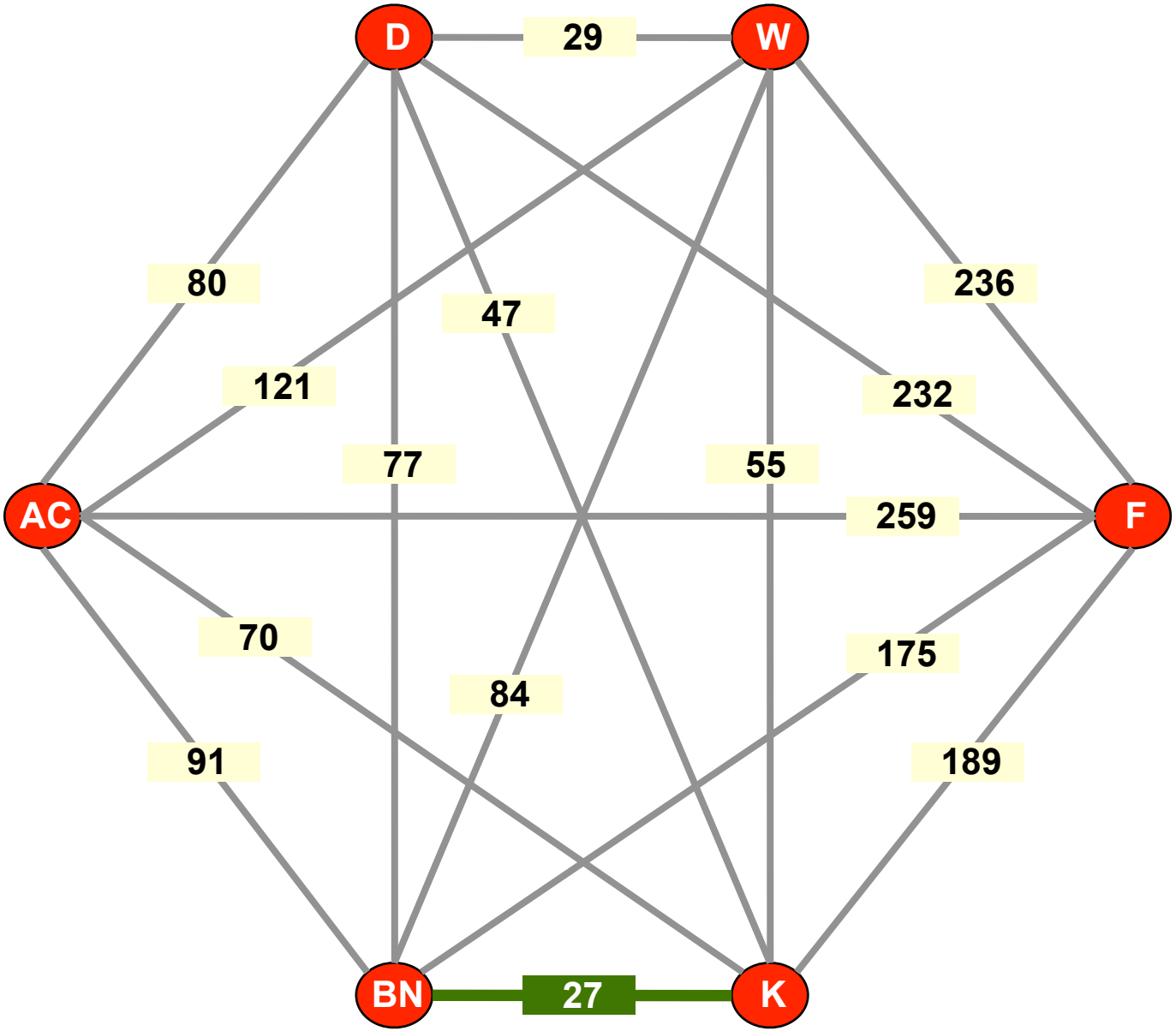
Anwendung auf das Rheinland-Problem:



1. MST z.B. mit Kruskal-Algorithmus ...

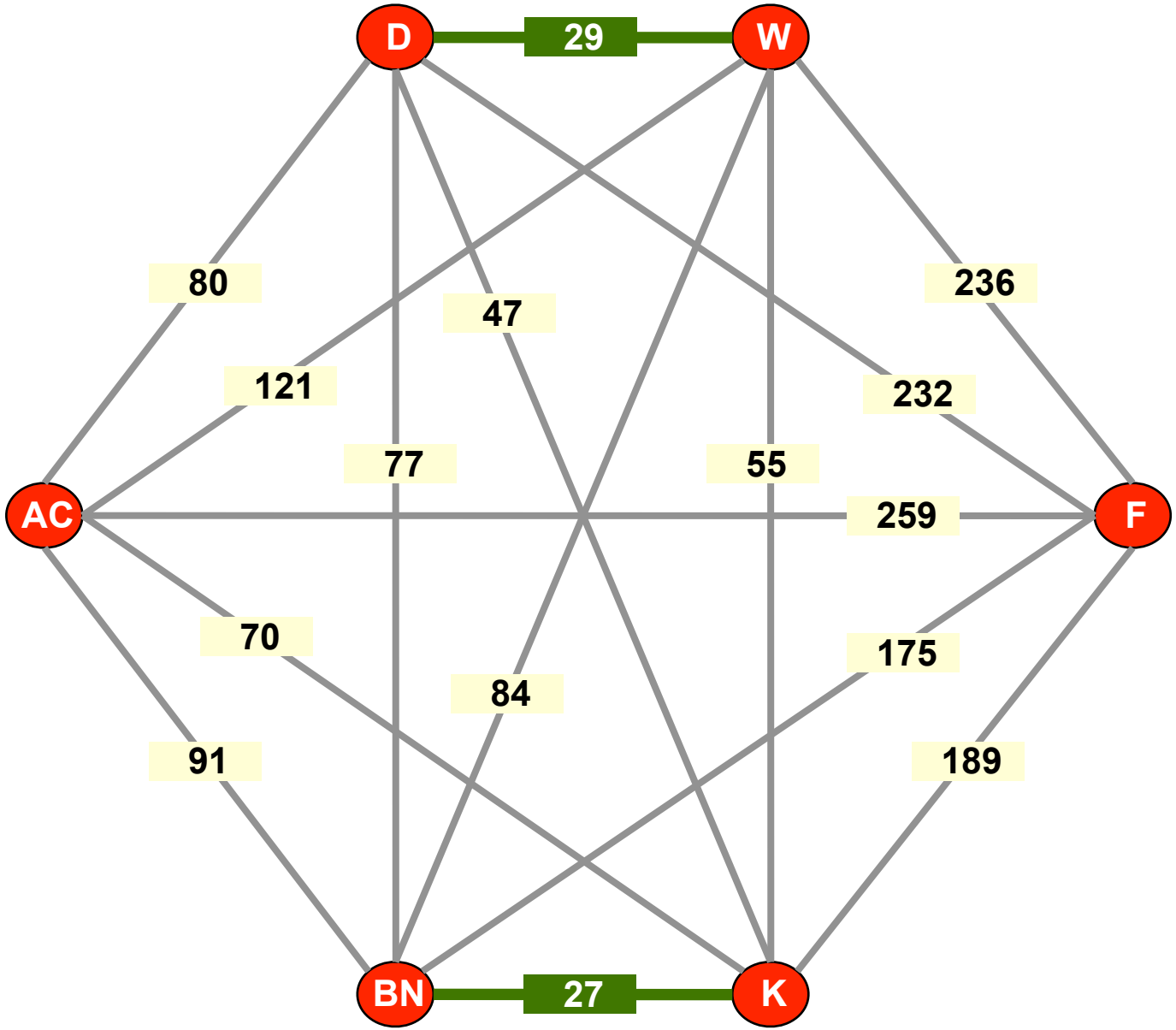
© 2006-2024 Jörg Striegnitz

Anwendung auf das Rheinland-Problem:



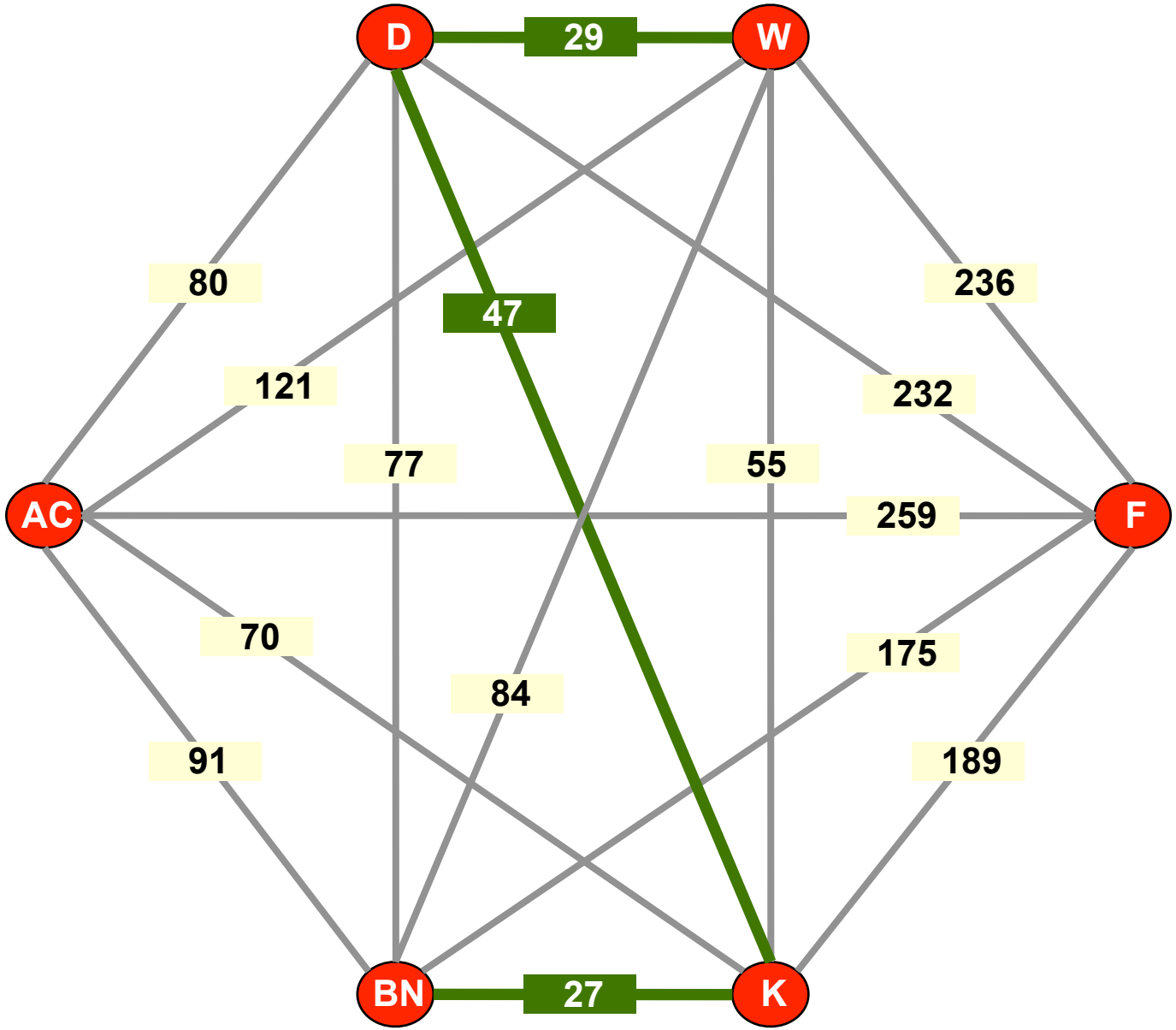
© 2006-2024 Jörg Striegnitz

Anwendung auf das Rheinland-Problem:



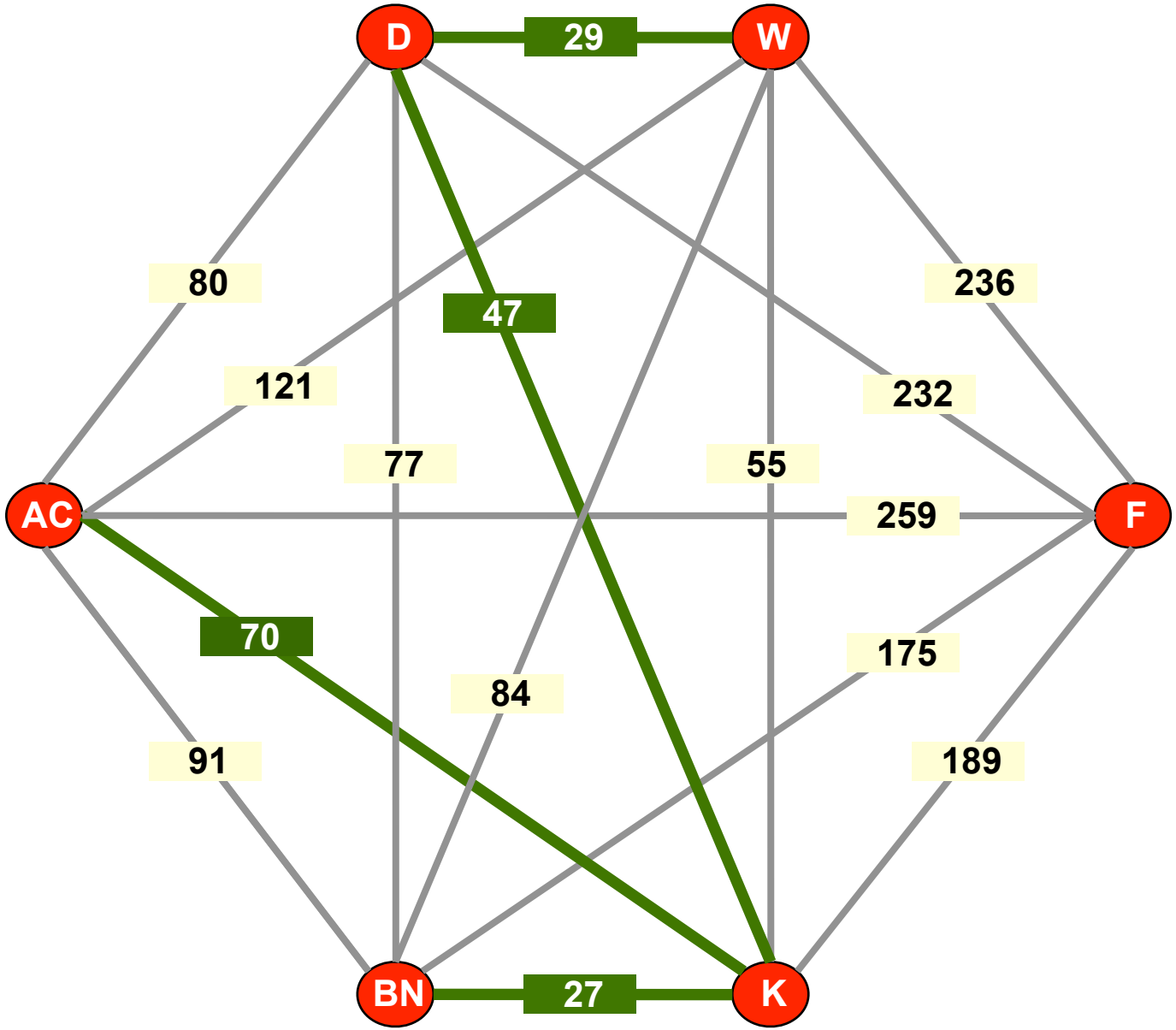
© 2006-2024 Jörg Striegnitz

Anwendung auf das Rheinland-Problem:



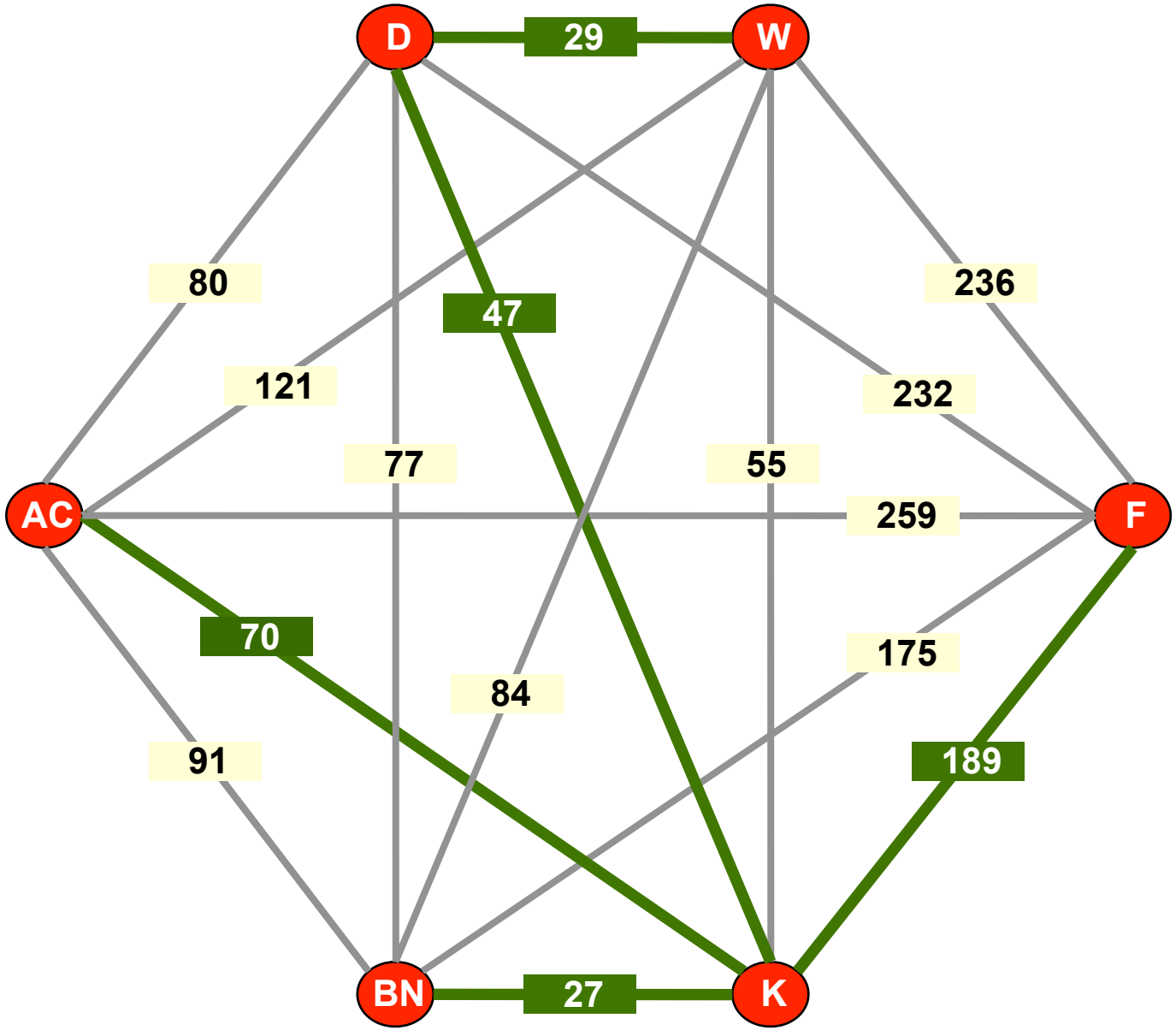
© 2006-2024 Jörg Striegnitz

Anwendung auf das Rheinland-Problem:



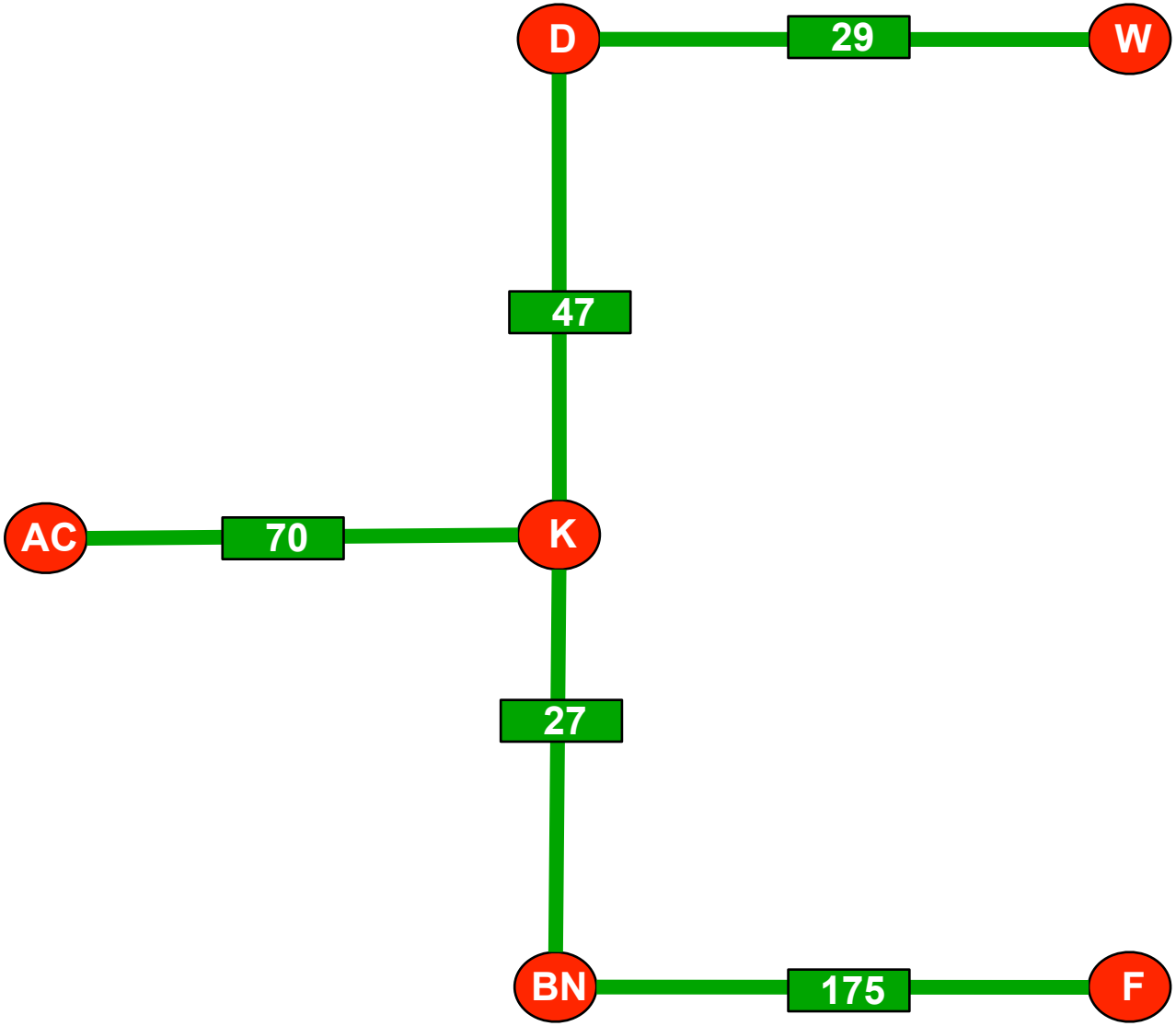
© 2006-2024 Jörg Striegnitz

Anwendung auf das Rheinland-Problem:



© 2006-2024 Jörg Striegnitz

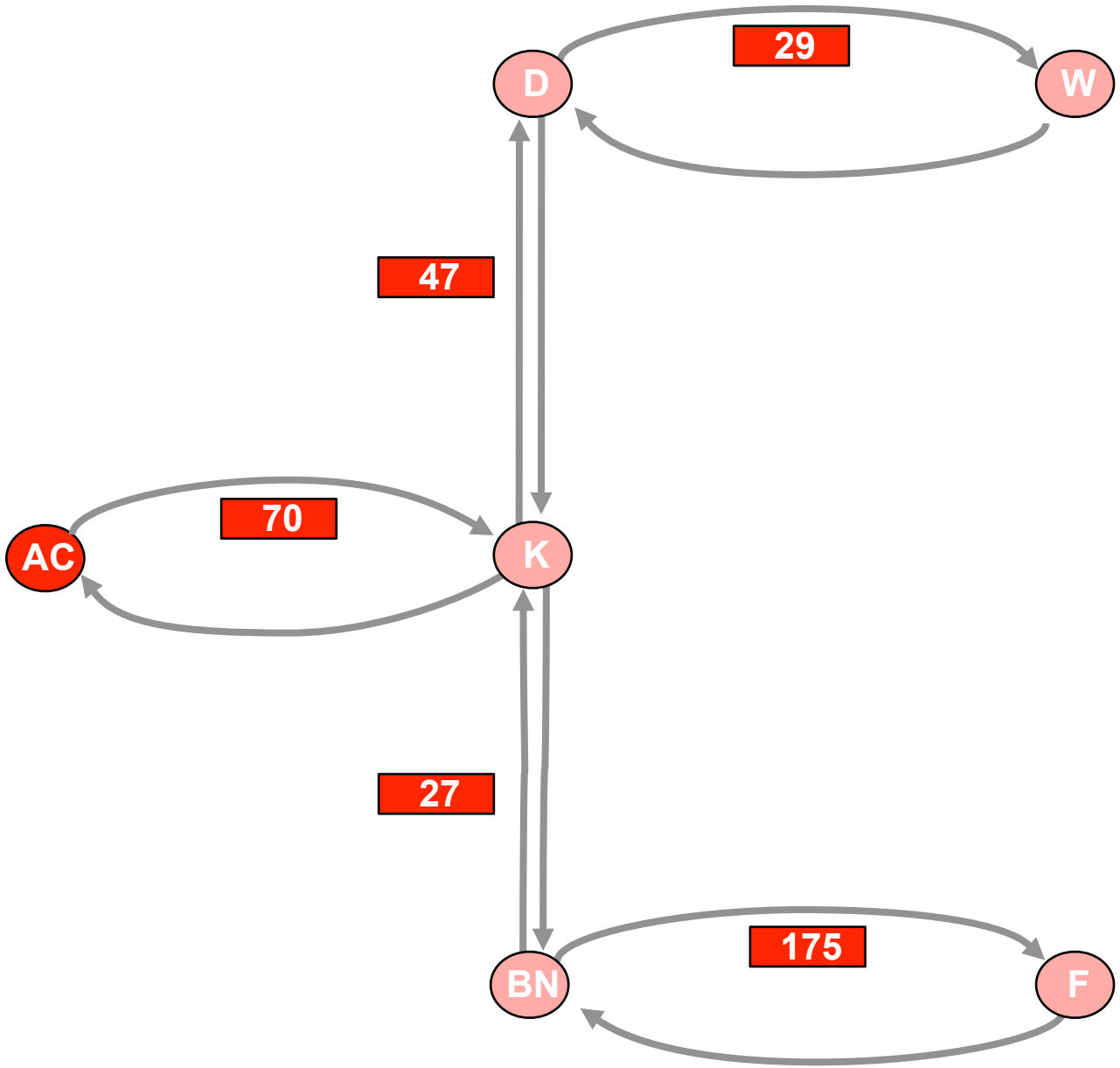
MST zum Rheinlandproblem:



2. Kanten „verdoppeln“ ...

© 2006-2024 Jörg Striegnitz

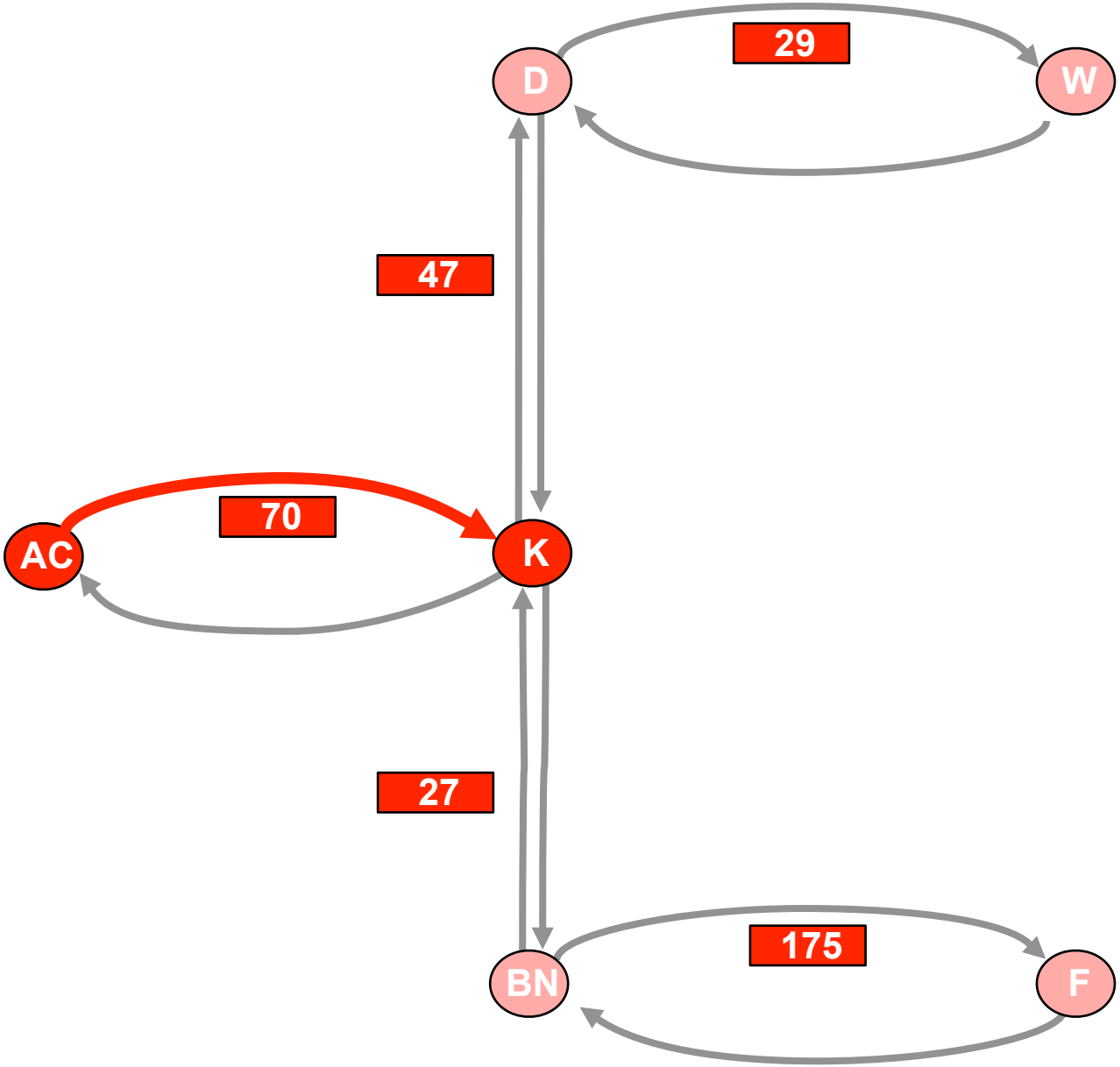
Gerichteter Graph (alternativ: Multigraph)



3. Konstruiere Euler-Pfad ...

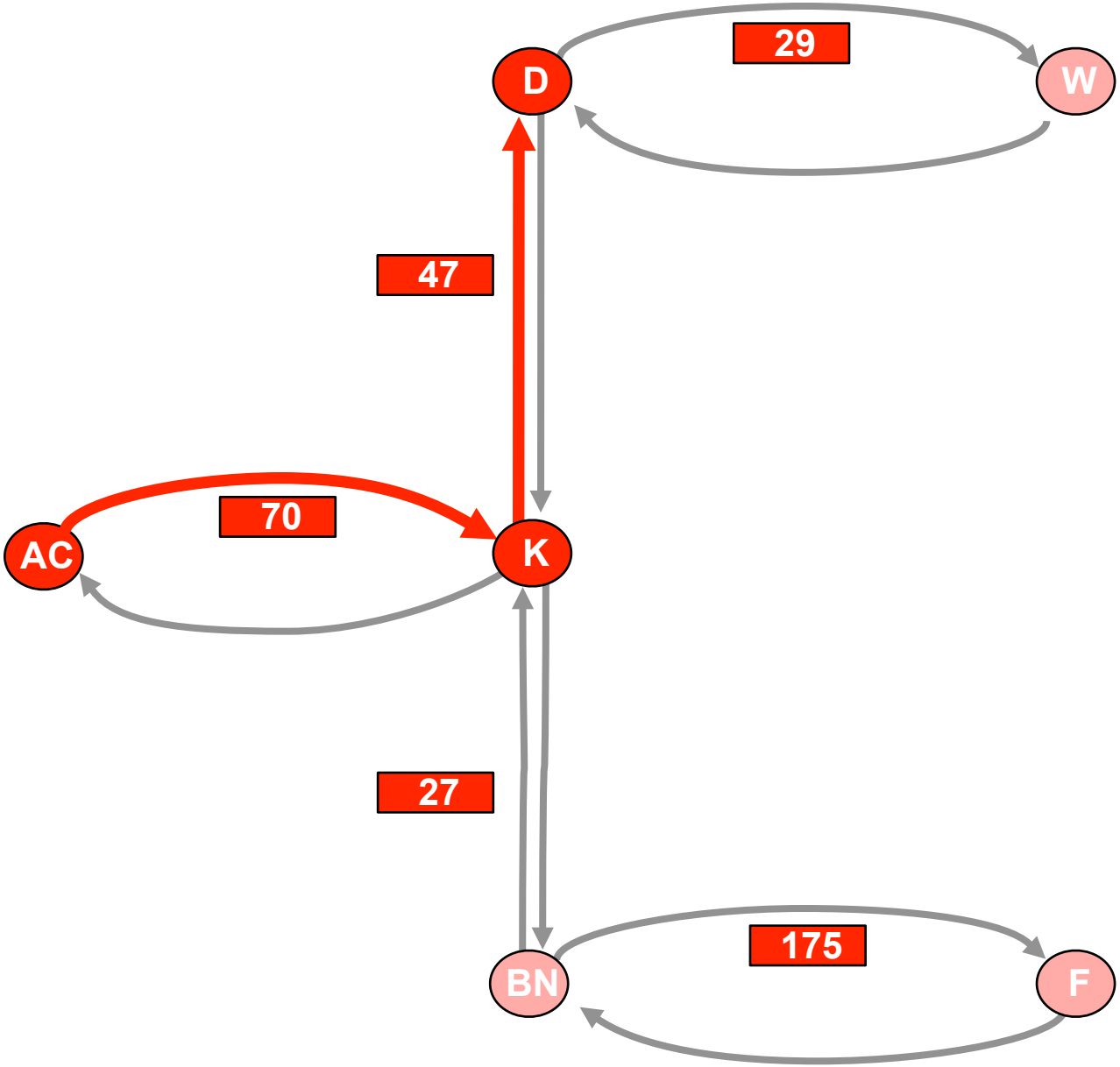
© 2006-2024 Jörg Striegnitz

Gerichteter Graph



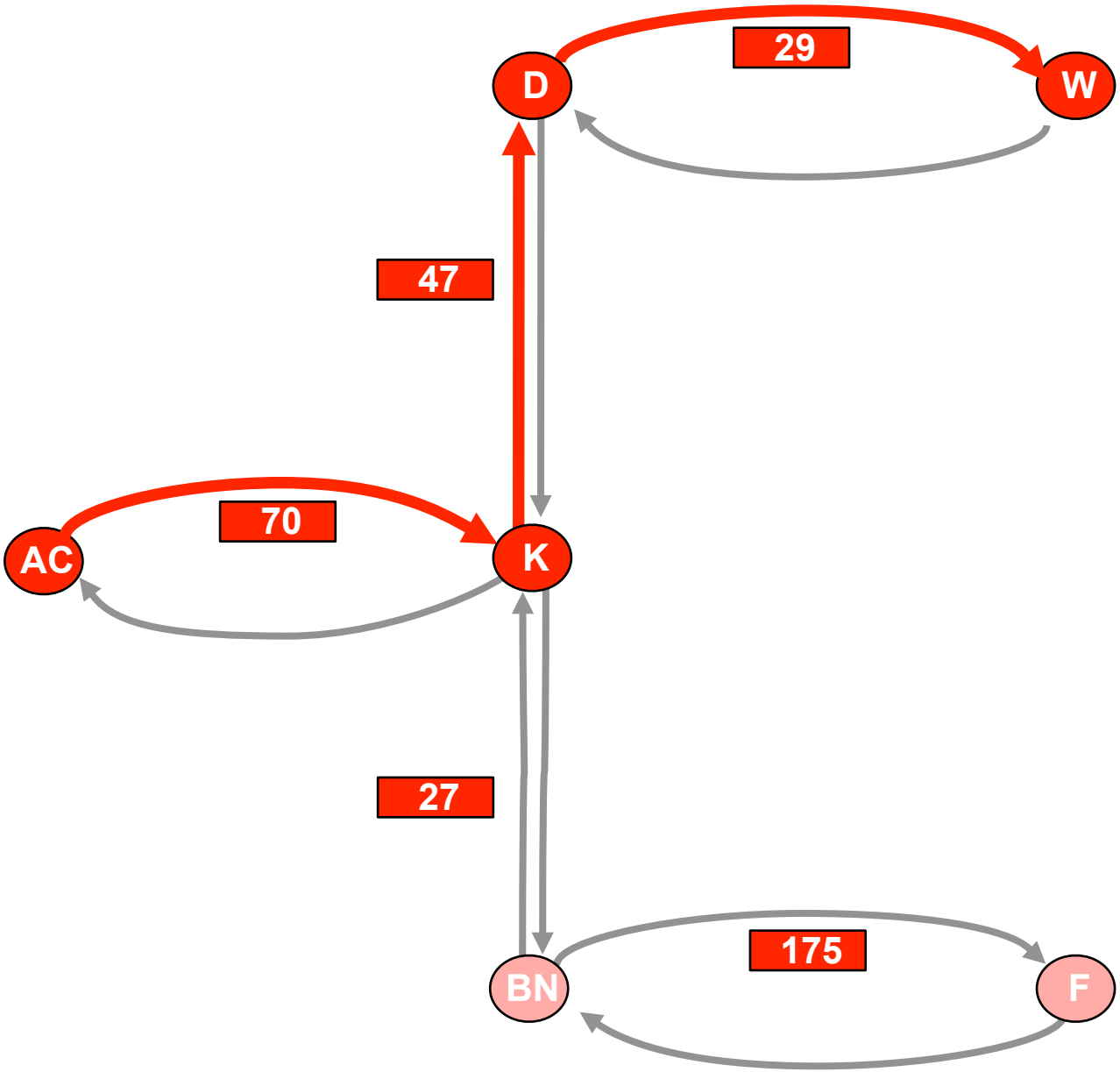
© 2006-2024 Jörg Striegnitz

Gerichteter Graph



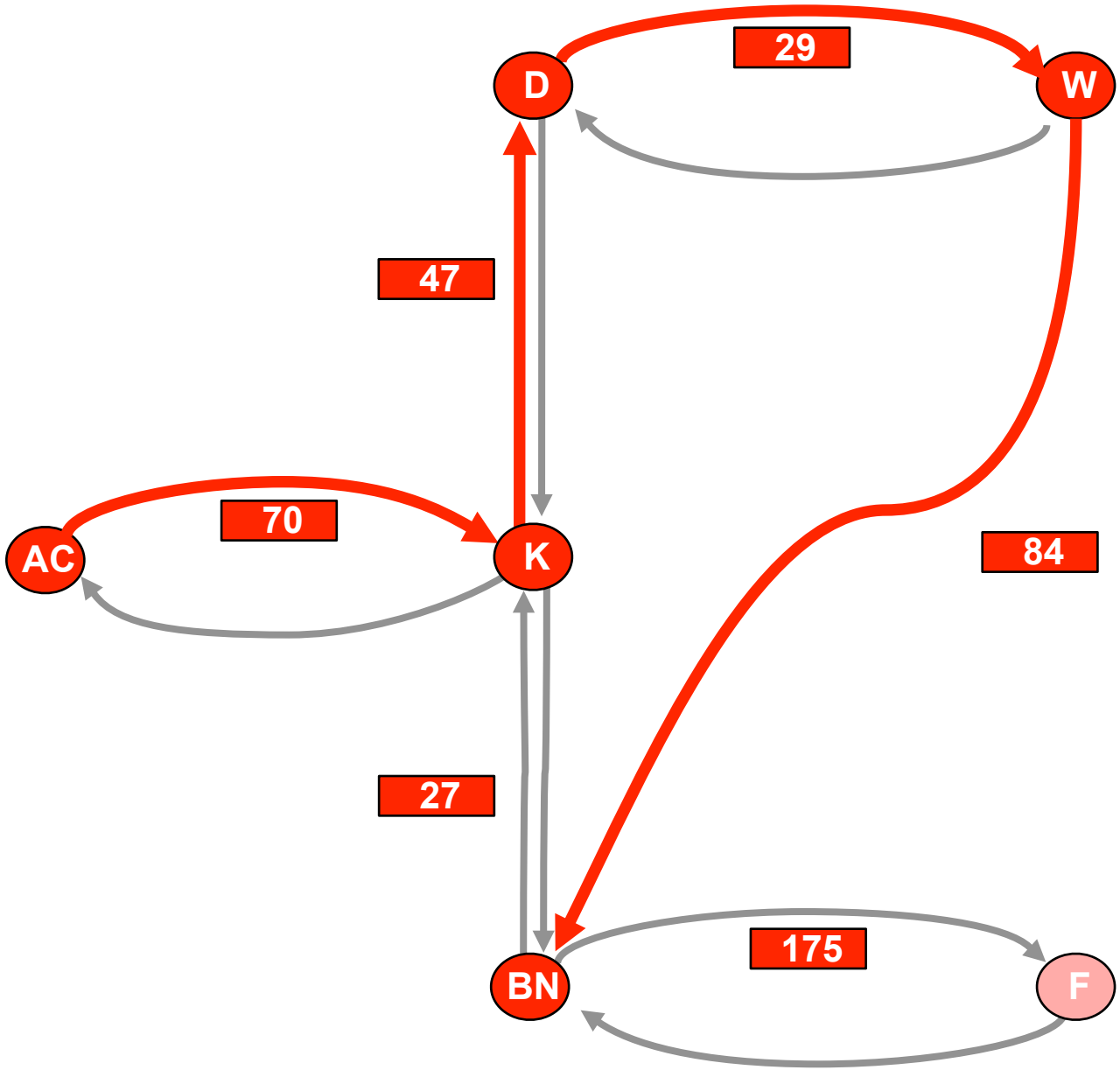
© 2006-2024 Jörg Striegnitz

Gerichteter Graph



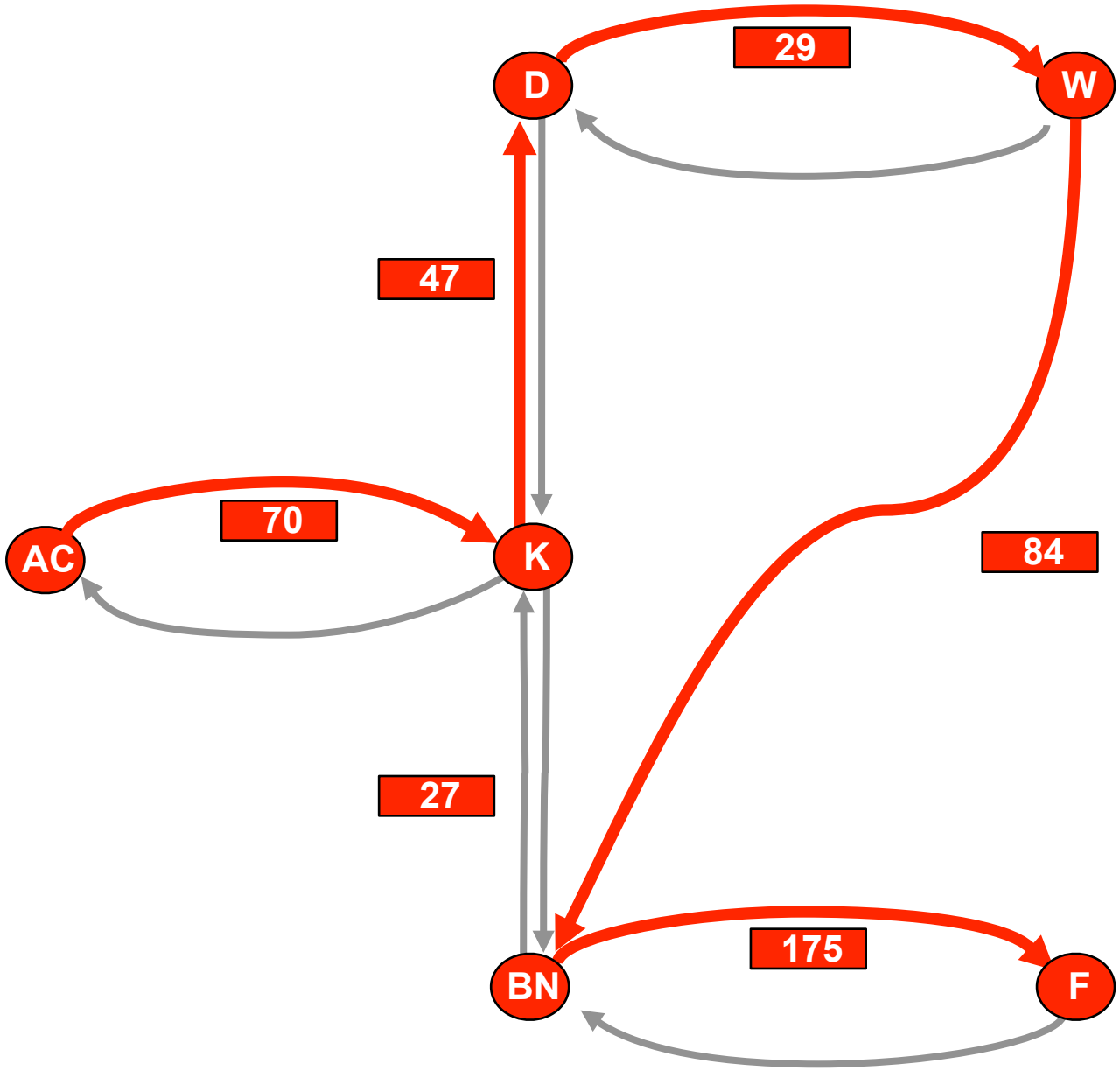
© 2006-2024 Jörg Striegnitz

Gerichteter Graph



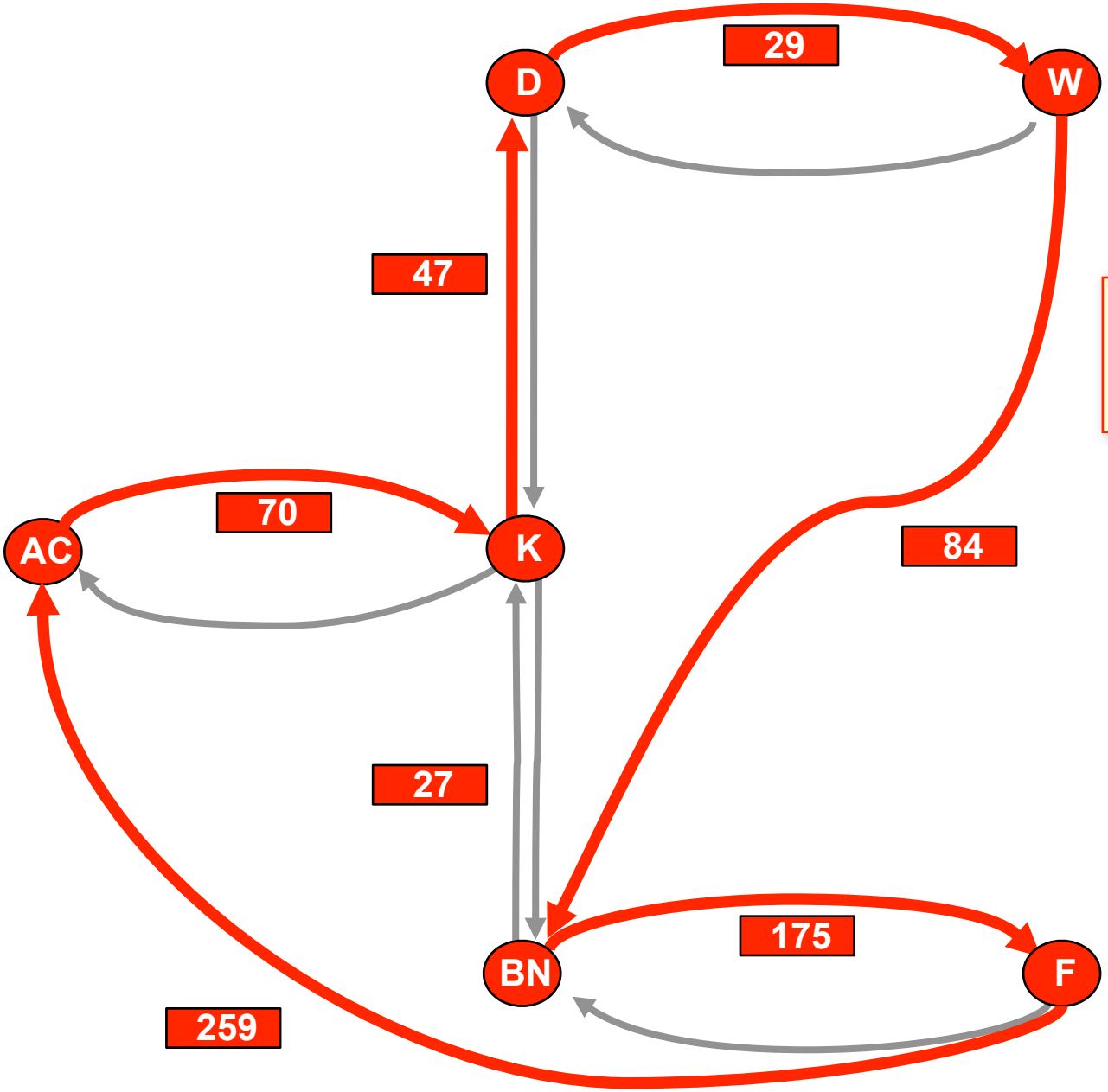
© 2006-2024 Jörg Striegnitz

Gerichteter Graph



© 2006-2024 Jörg Striegnitz

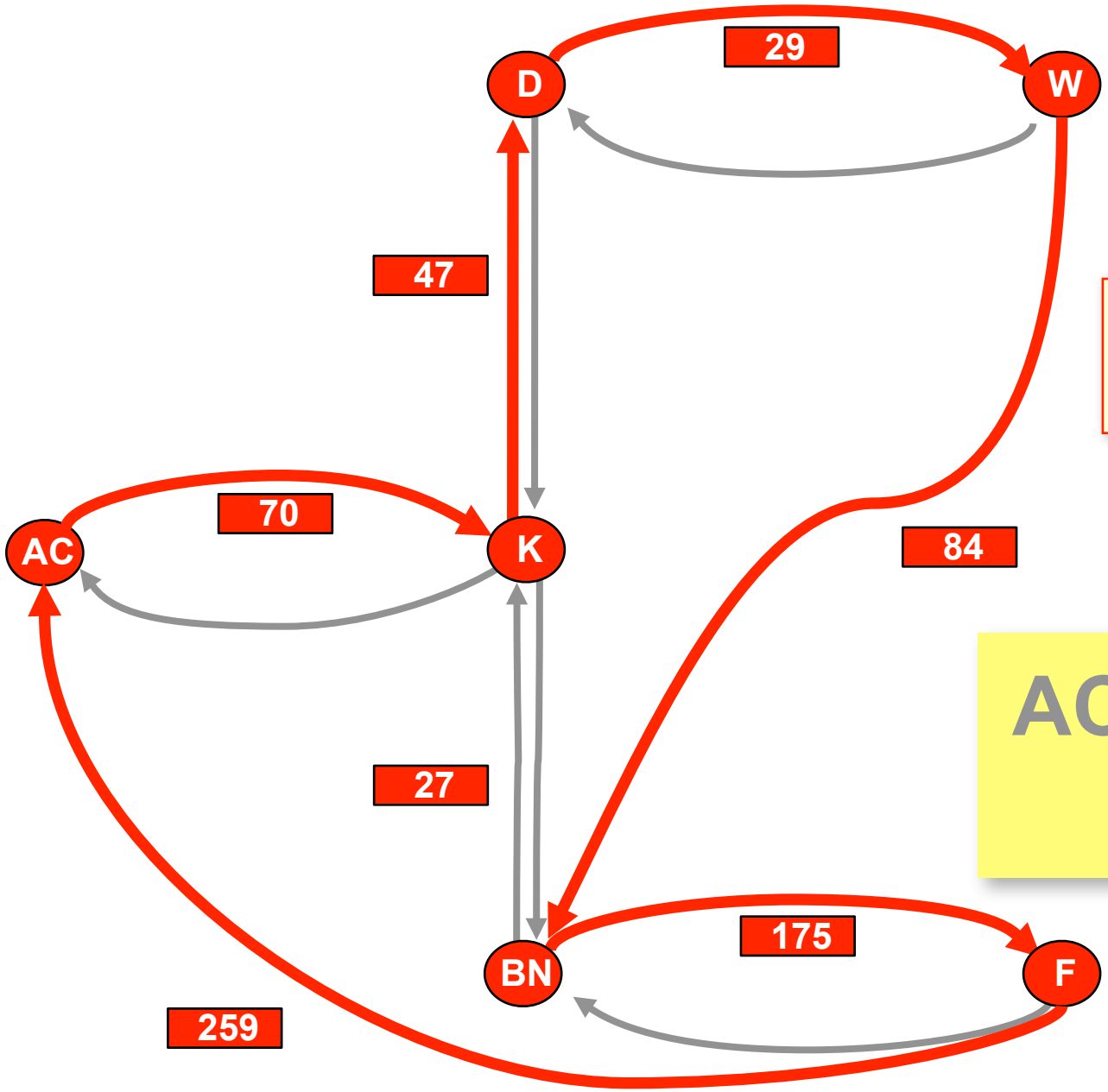
Gerichteter Graph



Alternativ:
Einfache Tiefensuche

© 2006-2024 Jörg Striegnitz

Gerichteter Graph



Alternativ:
Einfache Tiefensuche

AC-K-D-W-B-F-AC
664 km

© 2006-2024 Jörg Striegnitz

S Güte der MST-Heuristik

Sei $G = (V, E)$ ein metrischer, gewichteter Graph und p_{MST} ein Hamiltonscher Kreis, der mit der MST-Heuristik berechnet wurde. Dann gilt:

$$Q(p_{MST}) \leq 2$$

© 2006-2024 Jörg Striegnitz

S Güte der MST-Heuristik

Sei $G = (V, E)$ ein metrischer, gewichteter Graph und p_{MST} ein Hamiltonscher Kreis, der mit der MST-Heuristik berechnet wurde. Dann gilt:

$$Q(p_{MST}) \leq 2$$

Unabhängig vom Input!

© 2006-2024 Jörg Striegnitz

S Güte der MST-Heuristik

Sei $G = (V, E)$ ein metrischer, gewichteter Graph und p_{MST} ein Hamiltonscher Kreis, der mit der MST-Heuristik berechnet wurde. Dann gilt:

$$Q(p_{MST}) \leq 2$$

Unabhängig vom Input!

Beweis: Für einen minimalen Spannbaum $G_T = (V, E_T)$ von G gilt

$$\sum_{e \in E_t} \text{weight}(e) \leq \text{Weight}(p_{Opt})$$

Denn G_T enthält nur die kleinsten Kanten und der Hamiltonsche Kreis hat mindestens eine Kante mehr.
Für einen Eulerpfad p_{Euler} im "verdoppelten" Graph gilt

$$\text{weight}(p_{Euler}) = 2 \cdot \sum_{e \in E_T} \text{weight}(e) \leq 2 \cdot \text{Weight}(p_{Opt})$$

Wegen der Dreiecksungleichung, kann das Überspringen eines Knotens das Pfadgewicht nicht erhöhen und wir erhalten daher:

$$\text{Weight}(p_{MST}) \leq 2 \cdot \text{Weight}(p_{Opt})$$

© 2006-2024 Jörg Striegnitz

Optimierungspotential bei MST-Heuristik

- **Allgemeine Strategie der MST-Heuristik:**
  - Ausgangspunkt ist Graph G

1. **Konstruiere MST T**
2. **Erweitere T zu einem Multigraph T^***
   und zwar so, dass T^* **Euler-Tour** enthält
3. **„Verkürze“ die Euler-Tour, um hamiltonschen Kreis in G zu erhalten**

© 2006-2024 Jörg Striegnitz

Optimierungspotential bei MST-Heuristik

- **Allgemeine Strategie der MST-Heuristik:**
  - Ausgangspunkt ist Graph G

Anmerkung: Ein Multigraph ist ein Graph, bei dem die Kantenmenge als Multimenge modelliert ist.

1. **Konstruiere MST T**
2. **Erweitere T zu einem Multigraph T^***
   und zwar so, dass T^* **Euler-Tour** enthält
3. **„Verkürze“ die Euler-Tour, um hamiltonschen Kreis in G zu erhalten**

© 2006-2024 Jörg Striegnitz

Optimierungspotential bei MST-Heuristik

- **Allgemeine Strategie der MST-Heuristik:**
  - Ausgangspunkt ist Graph G

1. **Konstruiere MST T**

Anmerkung: Ein Multigraph ist ein Graph, bei dem die Kantenmenge als Multimenge modelliert ist.

2. **Erweitere T zu einem Multigraph T^***
   und zwar so, dass T^* **Euler-Tour** enthält

3. **„Verkürze“ die Euler-Tour, um hamiltonschen Kreis in G zu erhalten**

- **Bisher**: „Verdoppeln“ der Kantenzahl
- Eventuell zu teuer
  - vielleicht geht es mit weniger Kanten; billigeren Kanten?

© 2006-2024 Jörg Striegnitz

Eulerpfad und eulersche Graphen

D **Eulerpfad**

Sei $G = (V, E)$ ein Graph und $p = v_1, \cdots, v_k$ ein Kreis. p heißt **Eulerpfad**, falls

$$E = \bigcup_{i=1}^{k-1} (v_i, v_{i+1}) \quad \wedge \quad k - 1 = |E|$$

gilt (jede Kante in G wird genau einmal durchlaufen).

Enthält ein Graph einen Eulerpfad, so nennen wir ihn **eulerschen Graphen**.

- **Beachte**
  - Knoten können in einem Eulerpfad mehrfach besucht werden
  - In eulerschen Graphen müssen alle Knotengrade gerade sein
    - klar, wir müssen jeweils „hinein“ und „hinaus“

© 2006-2024 Jörg Striegnitz

Satz über eulersche Graphen

S **Über eulersche Graphen**

Sei $G = (V, E)$ ein zusammenhängender Graph. G ist genau dann eulersch, wenn der Grad aller Knoten gerade ist.

© 2006-2024 Jörg Striegnitz

Satz über eulersche Graphen

S Über eulersche Graphen

Sei $G = (V, E)$ ein zusammenhängender Graph. G ist genau dann eulersch, wenn der Grad aller Knoten gerade ist.

Beweis: Die Richtung "eulersch" $\Rightarrow$ "alle Knoten haben geraden Grad" ist trivial.
Da alle Knoten geraden Grad haben, muss es in G einen Kreis geben. Seien $P = v_1, \cdots, v_k$ ein Kreis mit **maximaler** Kantenzahl und $E_p = \{(v_1, v_2), \cdots, (v_{k-1}, v_k), \}$. Wir zeigen, dass dann $E_p = E$ gelten muss.

Angenommen, $E_p \neq E$, dann betrachten wir

$$G' = (V, E - E_p)$$

Da G zusammenhängend ist, existiert ein Knoten u, der sowohl in G' als auch in P vorkommt und der Knotengrad von u in G' muss mindestens zwei sein. Betrachte nun einen Kreis P' in G', der u enthält - dieser muss existieren, da alle Knoten in G' geraden Knotengrad haben müssen.

Dann können wir aber leicht einen Kantenzug in G erstellen, der mehr Kanten durchläuft als P, was im Widerspruch zu der Annahme steht, dass P ein Kreis maximaler Kantenzahl war.

© 2006-2024 Jörg Striegnitz

Satz über ungerade Knotengrade

S **Über Knoten ungeraden Grades**

Sei $G = (V, E)$ ein Graph. Die Zahl der Knoten mit ungeradem Knotengrad in G ist gerade.

© 2006-2024 Jörg Striegnitz

Satz über ungerade Knotengrade

S **Über Knoten ungeraden Grades**

Sei $G = (V, E)$ ein Graph. Die Zahl der Knoten mit ungeradem Knotengrad in G ist gerade.

Beweis: Die Summe aller Knotengrade ist $2 \cdot |E|$. Da $2 \cdot |E|$ gerade ist, muss die Anzahl der Knoten mit ungeradem Grad gerade sein.

© 2006-2024 Jörg Striegnitz

Satz über ungerade Knotengrade

S **Über Knoten ungeraden Grades**

Sei $G = (V, E)$ ein Graph. Die Zahl der Knoten mit ungeradem Knotengrad in G ist gerade.

Beweis: Die Summe aller Knotengrade ist $2 \cdot |E|$. Da $2 \cdot |E|$ gerade ist, muss die Anzahl der Knoten mit ungeradem Grad gerade sein.

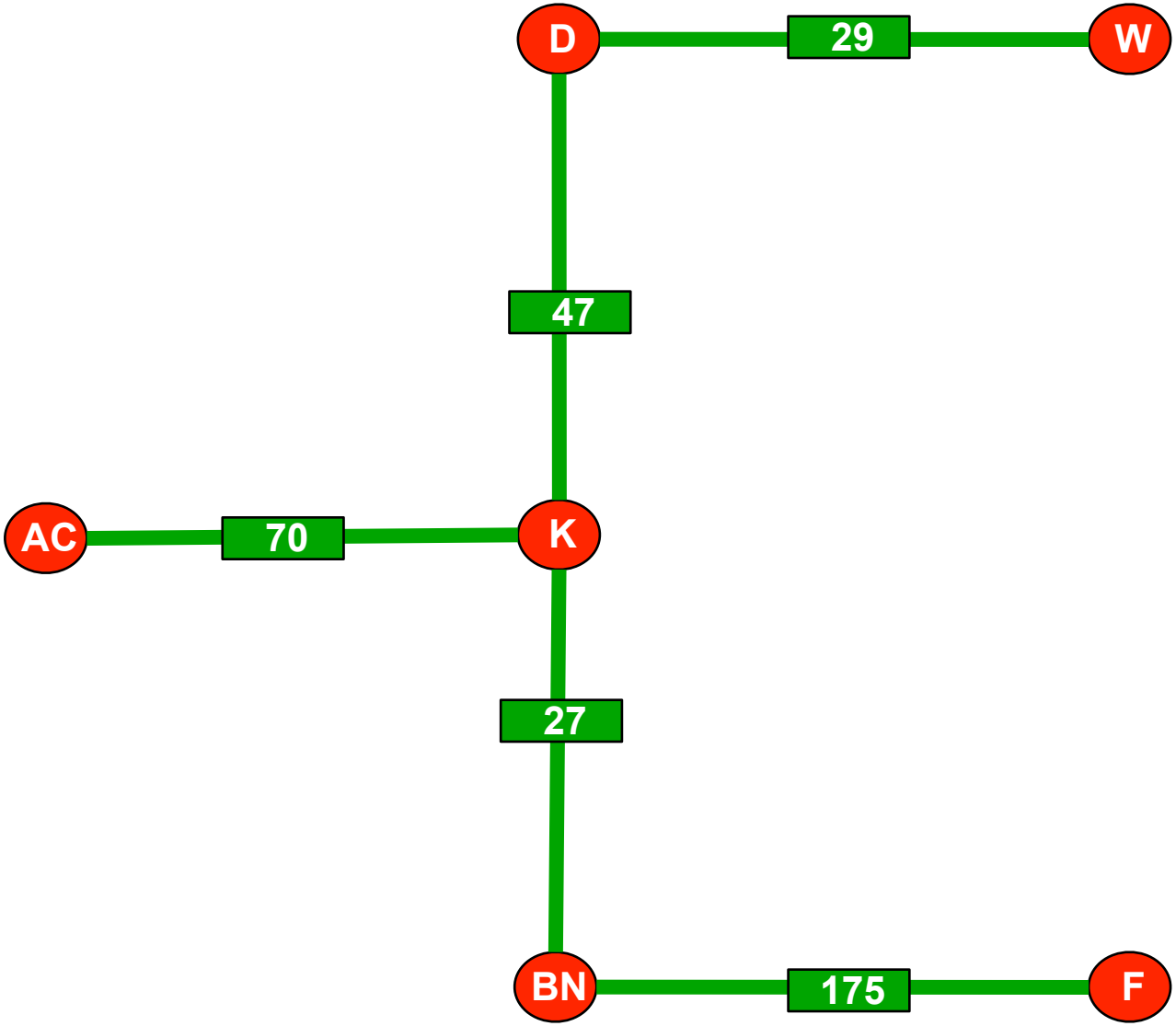
Neues Verfahren MST ➡ eulerscher Graph

- Betrachte Knoten ungeraden Grades in MST $S=\{v_1,..,v_{2m}\}$
  - deren Zahl ist gerade!
- Finde perfektes Matching minimalen Gewichts dieser Knoten in G
  - möglich, wegen gerader Knotenzahl!
  - erhalte Kantenmenge $E_m=\{(v_{i1},v_{j1}),\ldots,(v_{i2m},v_{j2m})\}$
- Konstruiere neuen Graph aus MST und E_m
  - dieser Graph muss jetzt eulersch sein

© 2006-2024 Jörg Striegnitz

MST ➡ eulerscher Graph - Version 2

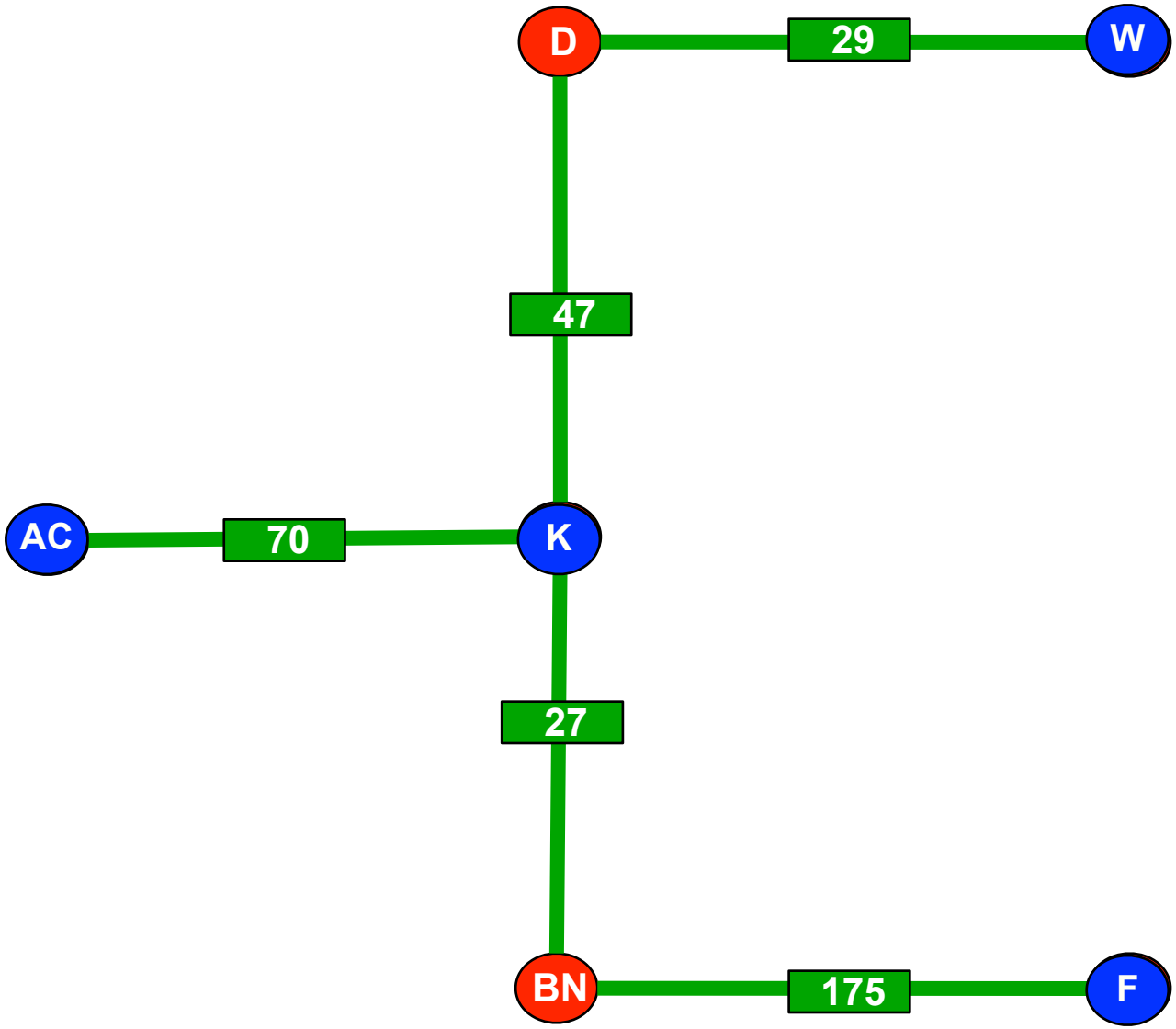
1. Knoten ungeraden Grades identifizieren



© 2006-2024 Jörg Striegnitz

MST ➡ eulerscher Graph - Version 2

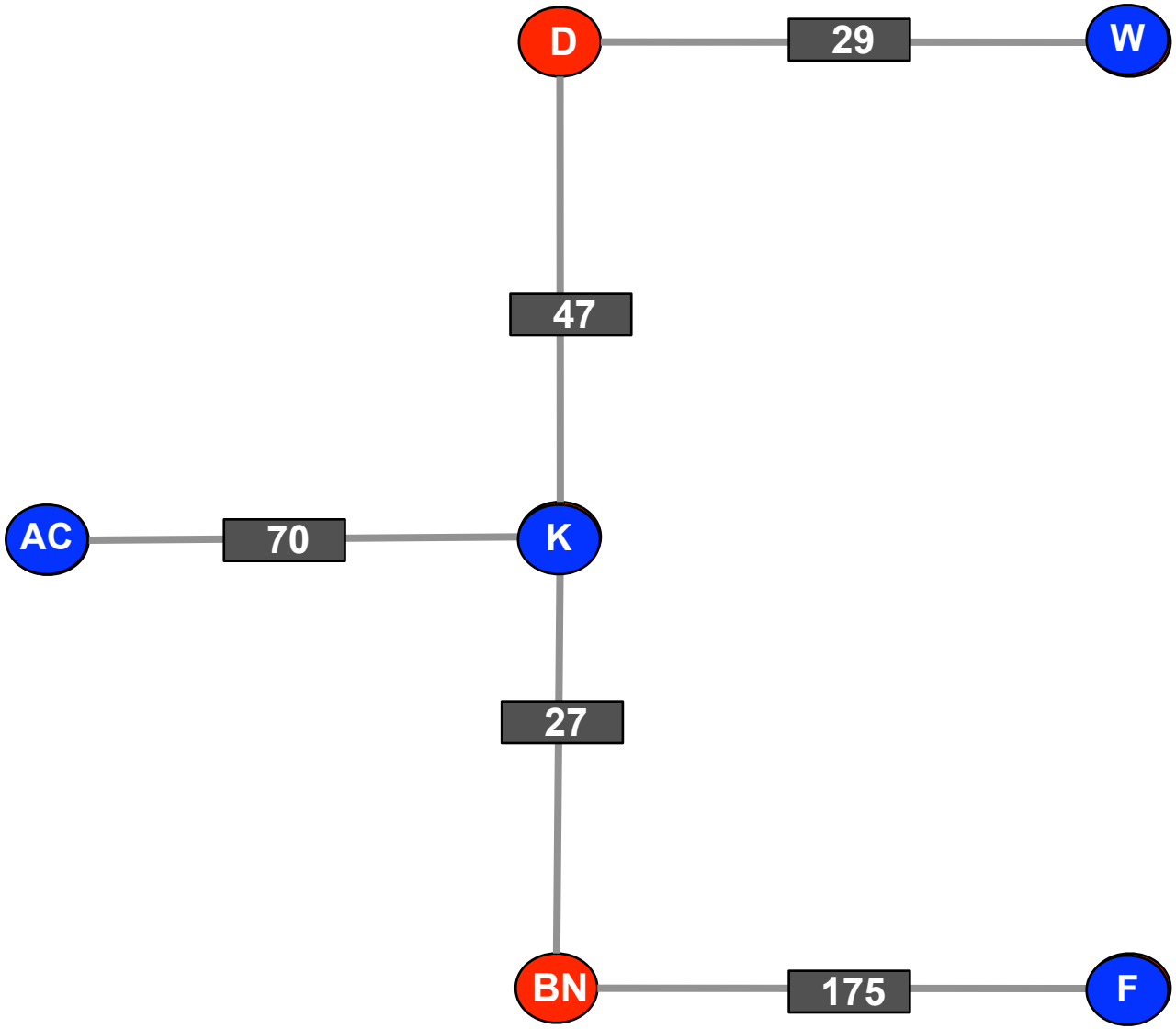
1. Knoten ungeraden Grades identifizieren



© 2006-2024 Jörg Striegnitz

MST ➡ eulerscher Graph - Version 2

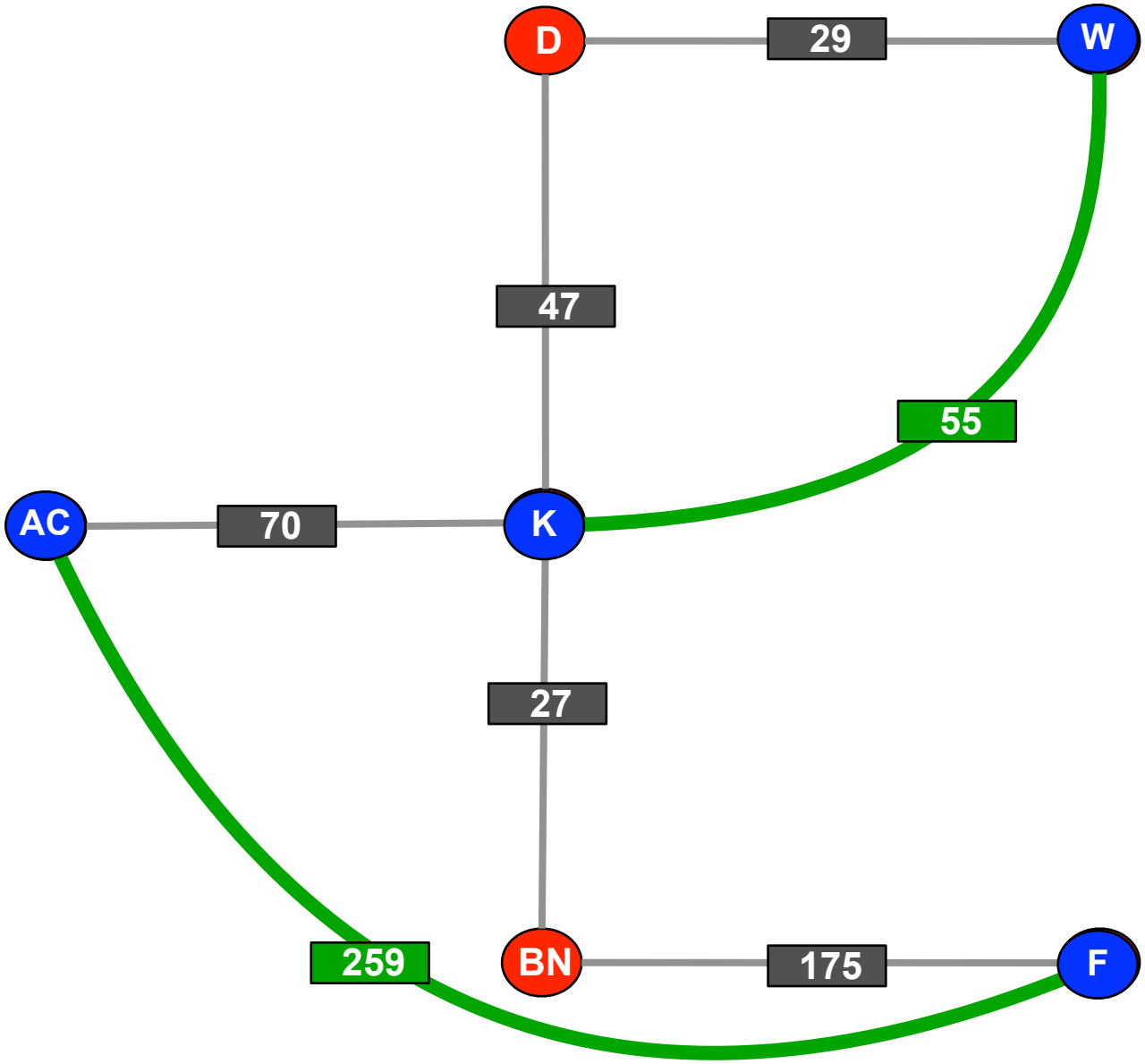
2. Minimales Matching suchen



© 2006-2024 Jörg Striegnitz

MST ➡ eulerscher Graph - Version 2

2. Minimales Matching suchen

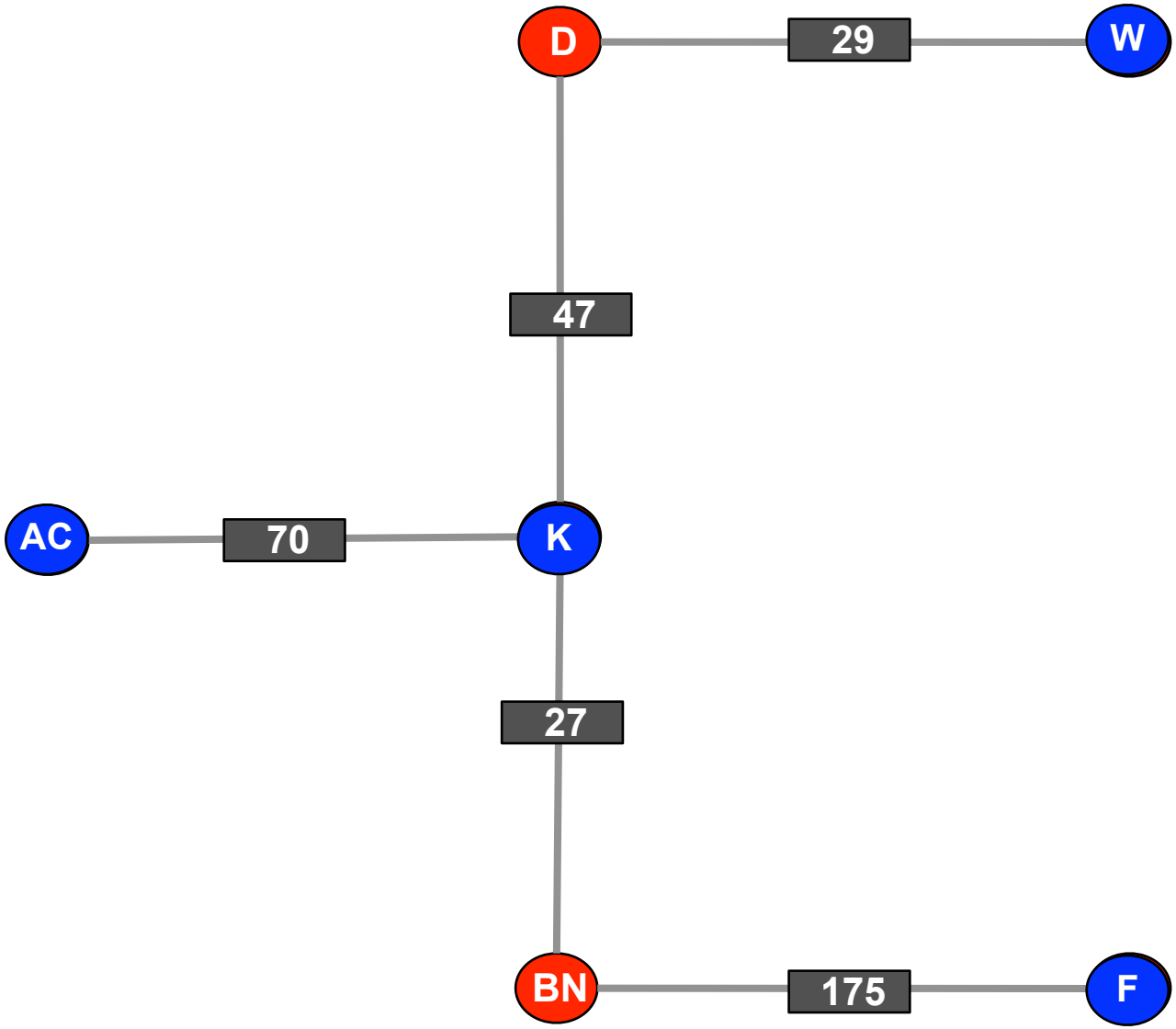


Kantensumme: 314

© 2006-2024 Jörg Striegnitz

MST ➡ eulerscher Graph - Version 2

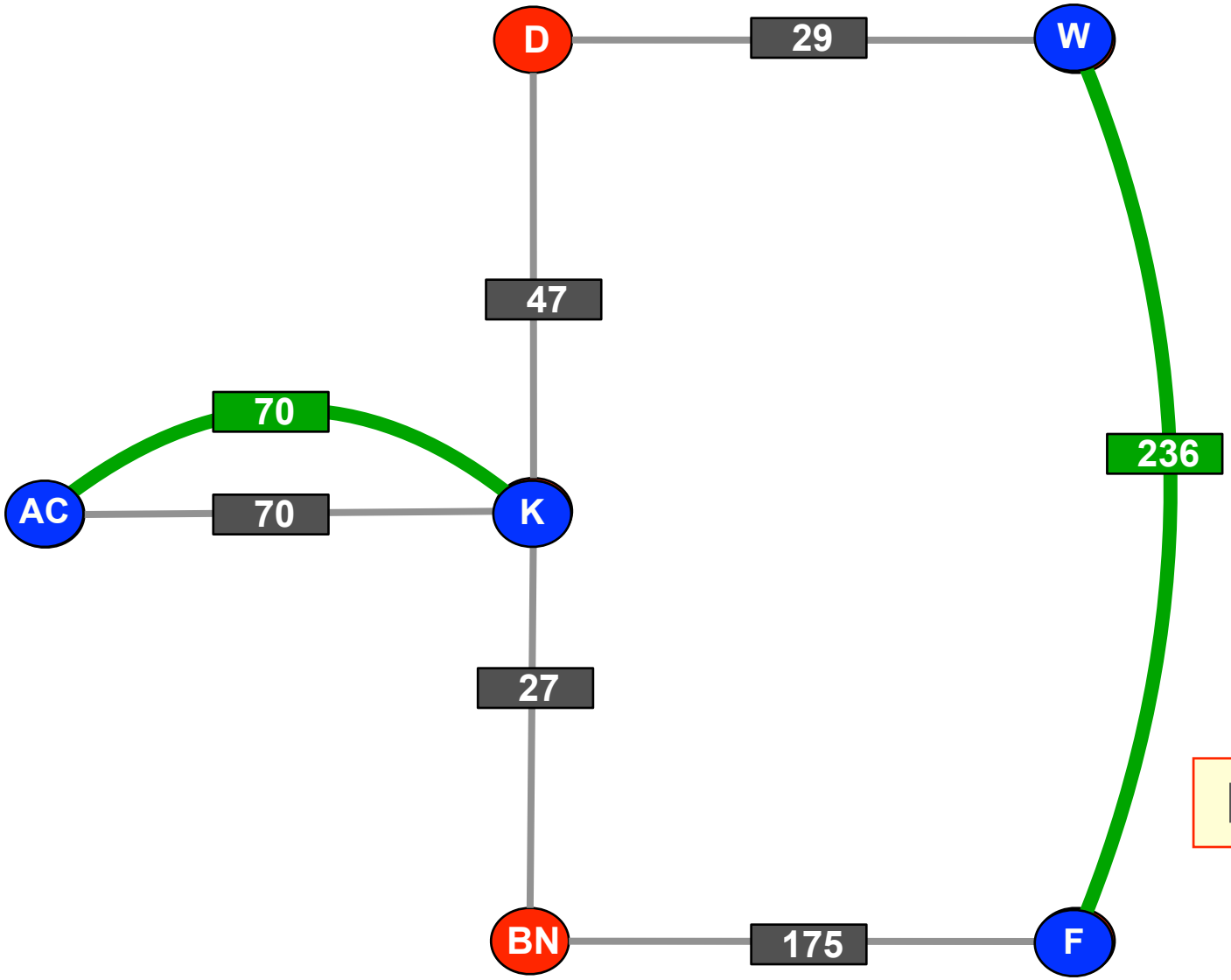
2. Minimales Matching suchen



© 2006-2024 Jörg Striegnitz

MST ➡ eulerscher Graph - Version 2

2. Minimales Matching suchen

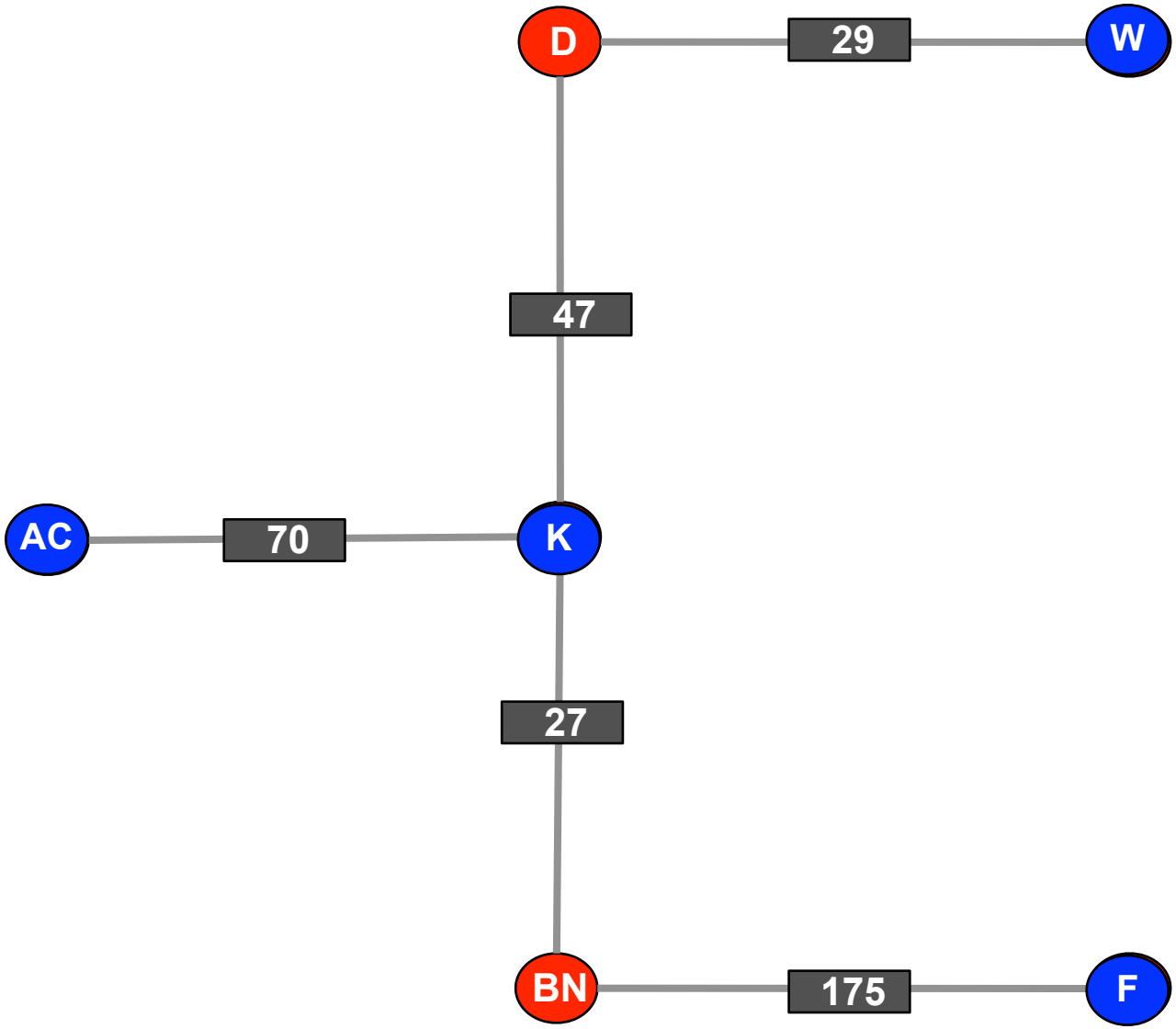


Kantensumme: 306

© 2006-2024 Jörg Striegnitz

MST ➡ eulerscher Graph - Version 2

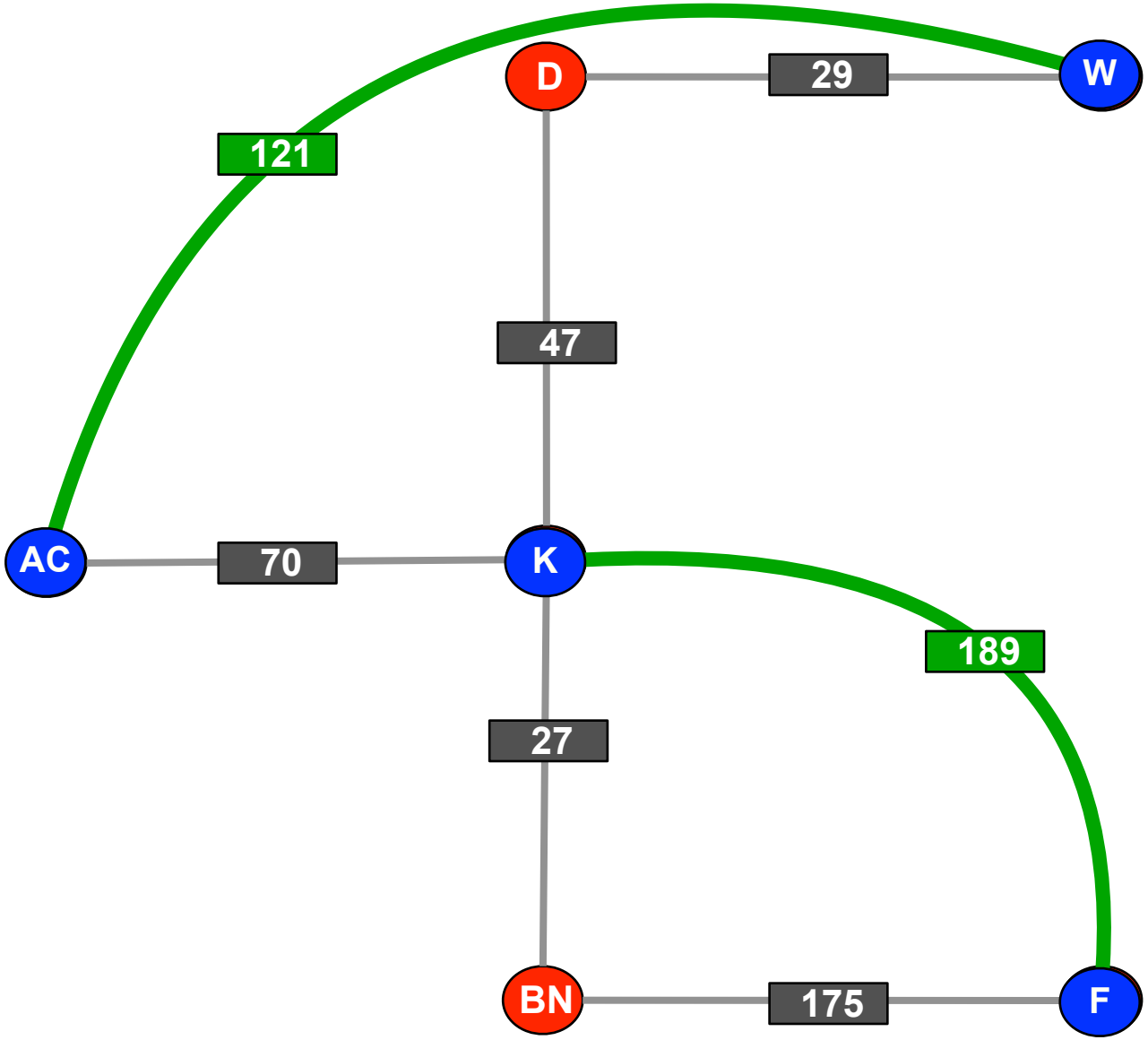
2. Minimales Matching suchen



© 2006-2024 Jörg Striegnitz

MST ➡ eulerscher Graph - Version 2

2. Minimales Matching suchen

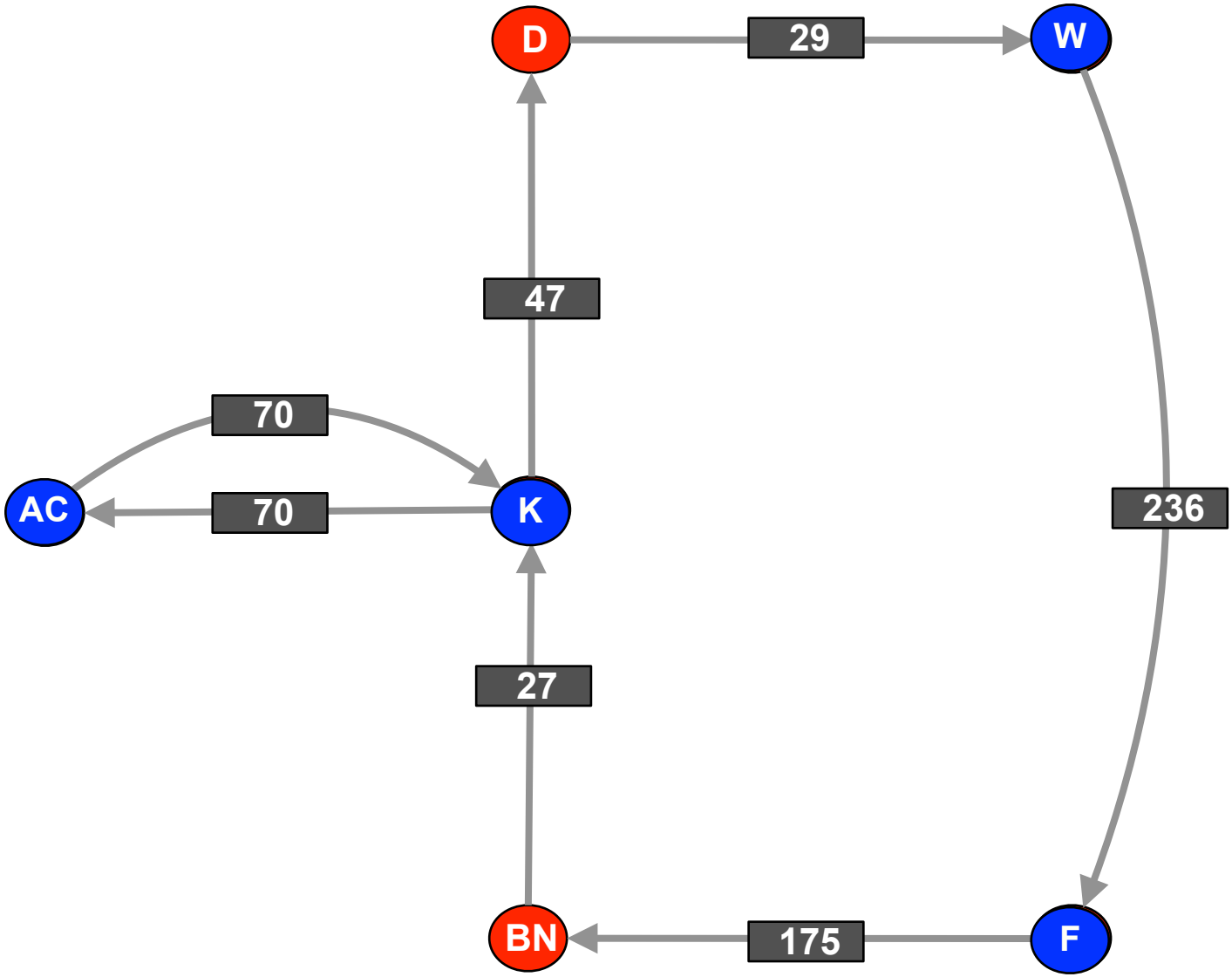


Kantensumme: 310

© 2006-2024 Jörg Striegnitz

Lösung TSP - wie vorher ...

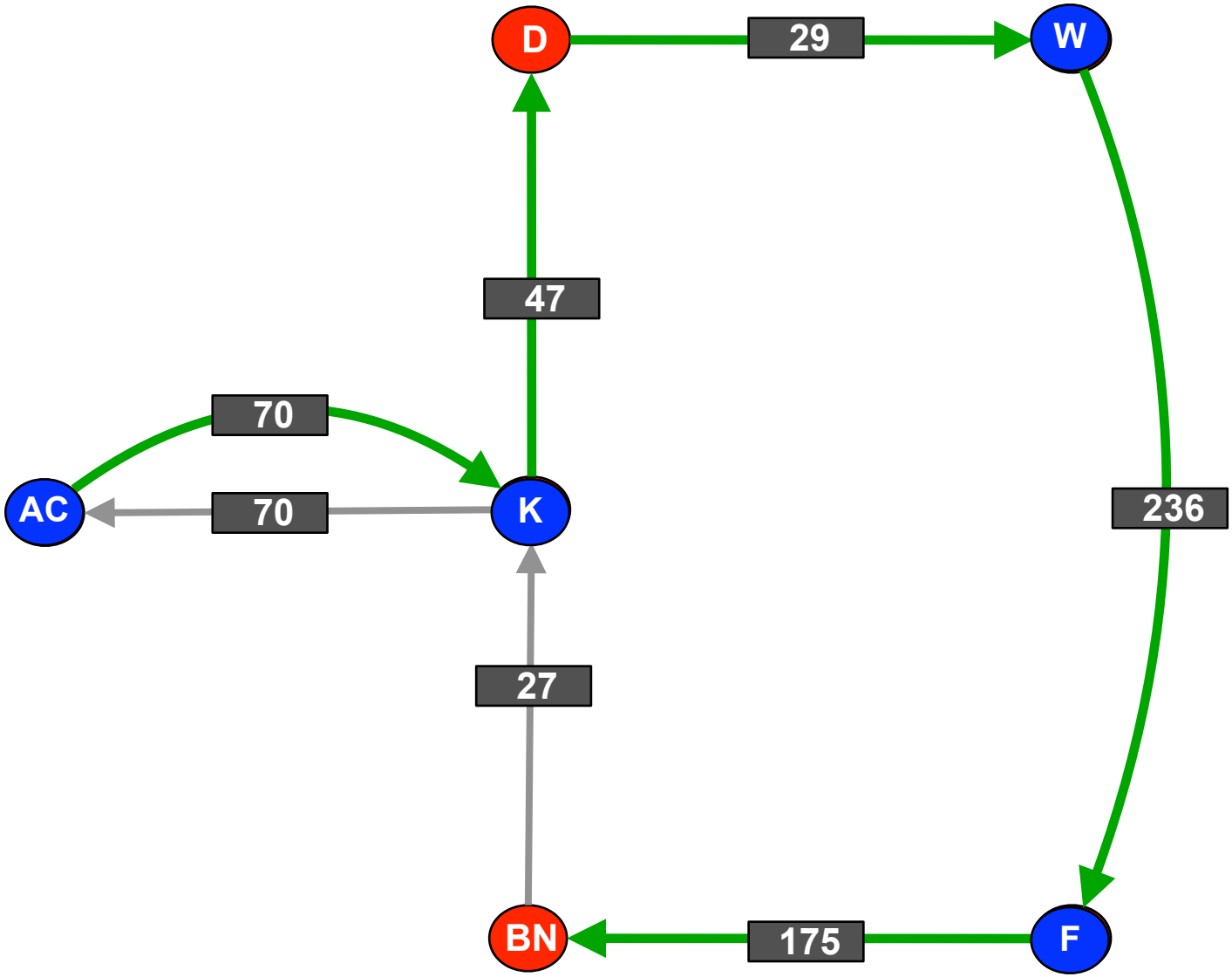
Eulerpfad bestimmen



© 2006-2024 Jörg Striegnitz

Lösung TSP - wie vorher ...

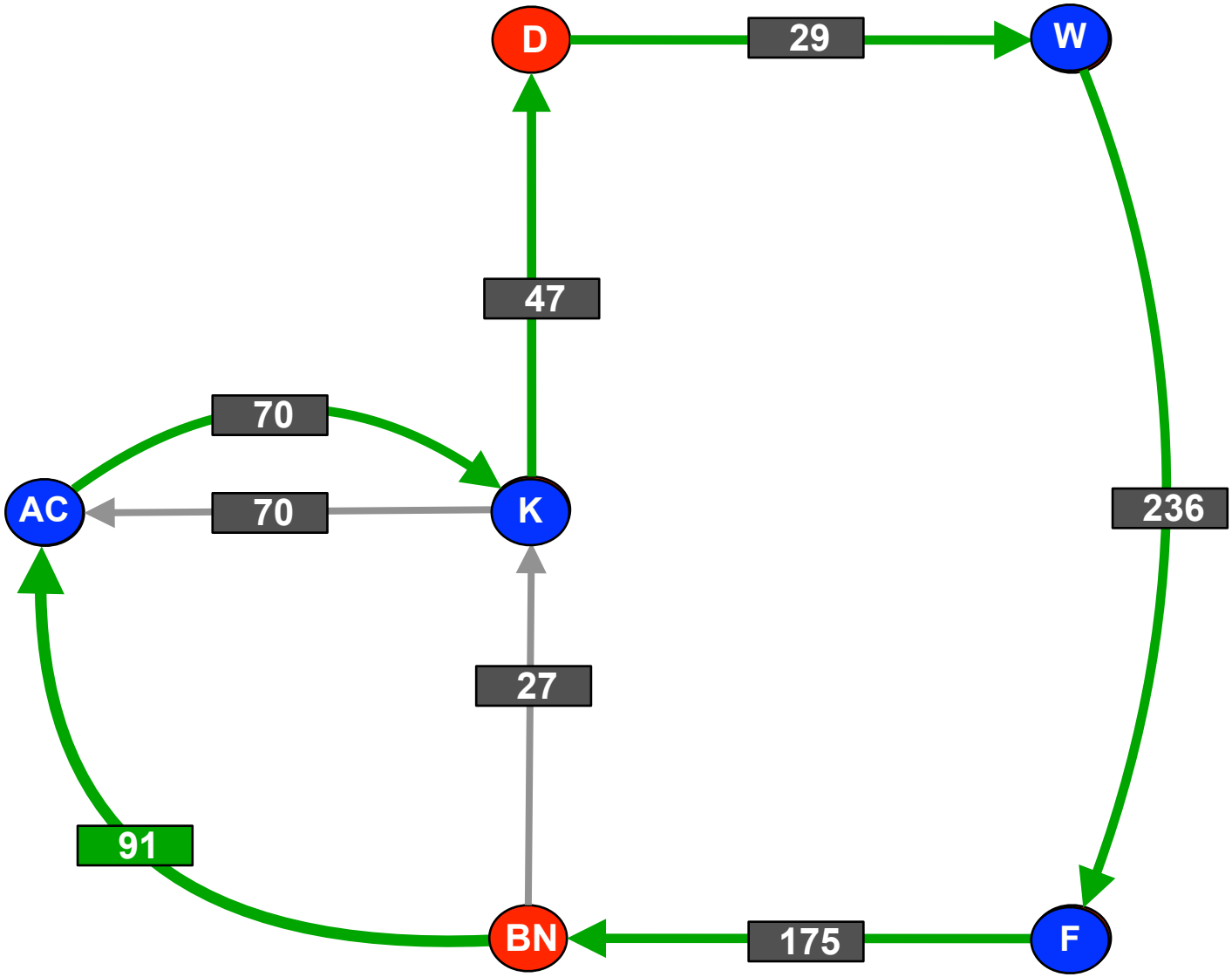
Eulerpfad zu hamiltonschem Kreis verkürzen



© 2006-2024 Jörg Striegnitz

Lösung TSP - wie vorher ...

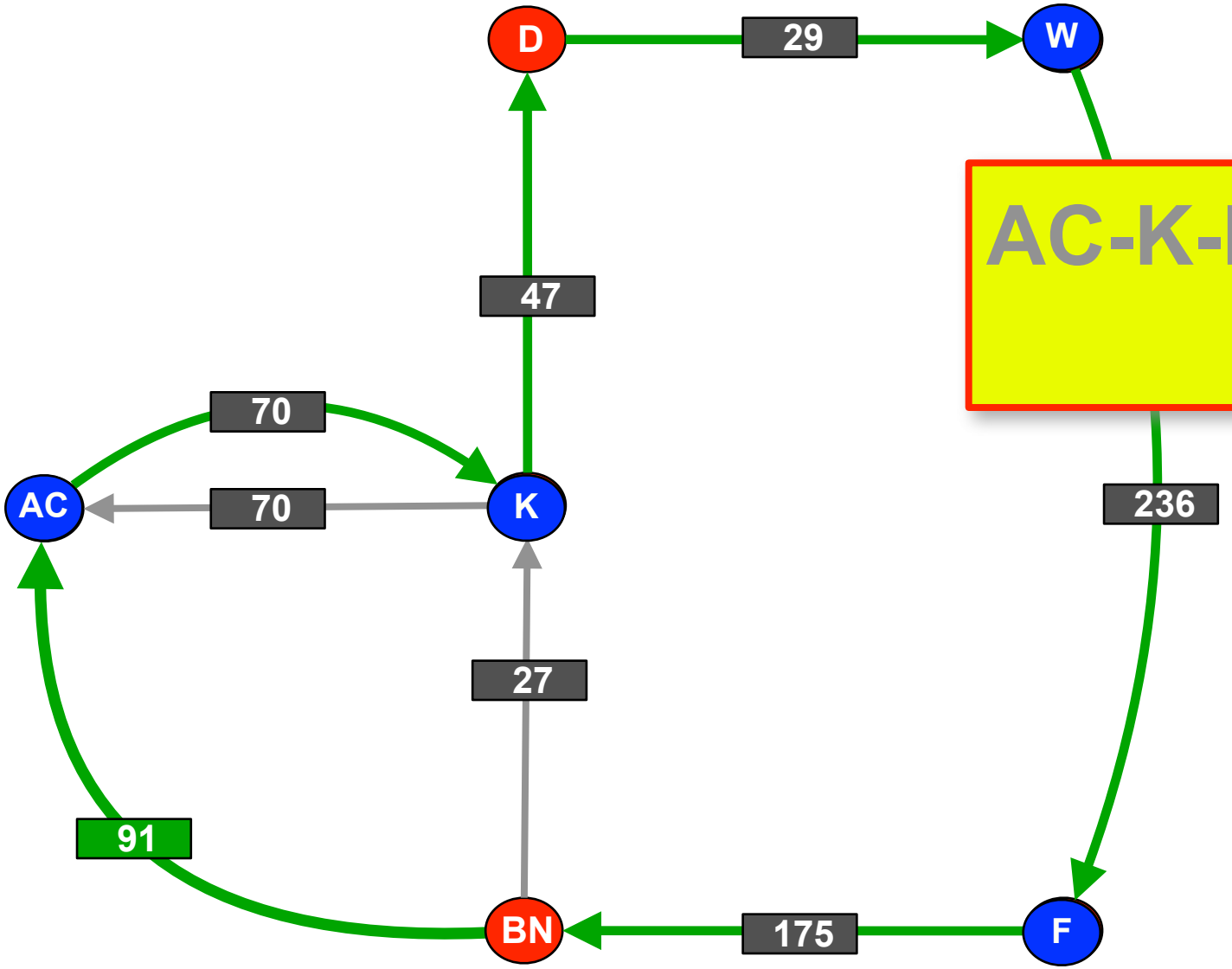
Eulerpfad zu hamiltonschem Kreis verkürzen



© 2006-2024 Jörg Striegnitz

Lösung TSP - wie vorher ...

Eulerpfad zu hamiltonschem Kreis verkürzen



AC-K-D-W-F-BN-AC

648 km

© 2006-2024 Jörg Striegnitz

Verfahren von Nico Christofides (1971)

- **Vorgehen ist als das Verfahren von Christofides bekannt**
  Bis 2014 beste bekannte Heuristik

© 2006-2024 Jörg Striegnitz

Verfahren von Nico Christofides (1971)

- **Vorgehen ist als das Verfahren von Christofides bekannt**
  Bis 2014 beste bekannte Heuristik

S Güte der Christofides-Heuristik

Sei $G = (V, E)$ ein metrischer, gewichteter Graph und p_{MST} ein Hamiltonscher Kreis, der mit der Christofides-Heuristik berechnet wurde. Dann gilt:

$$Q(p_{MST}) \leq \frac{3}{2}$$

© 2006-2024 Jörg Striegnitz

Verfahren von Nico Christofides (1971)

- **Vorgehen ist als das Verfahren von Christofides bekannt**
  Bis 2014 beste bekannte Heuristik

S Güte der Christofides-Heuristik

Sei $G = (V, E)$ ein metrischer, gewichteter Graph und p_{MST} ein Hamiltonscher Kreis, der mit der Christofides-Heuristik berechnet wurde. Dann gilt:

$$Q(p_{MST}) \leq \frac{3}{2}$$

Beweis: Seien T der MST, M das perfekte minimale Matching und H der hamiltonsche Kreis. Da der Graph metrisch war, gilt offenbar

$$\text{cost}(H) \leq \text{cost}(T) + \text{cost}(M)$$

Fener gilt

$$\text{cost}(T) \leq \text{cost}(H_{Opt})$$

da die optimale Rundreise im Vergleich zum MST mindestens eine Kante mehr enthält.

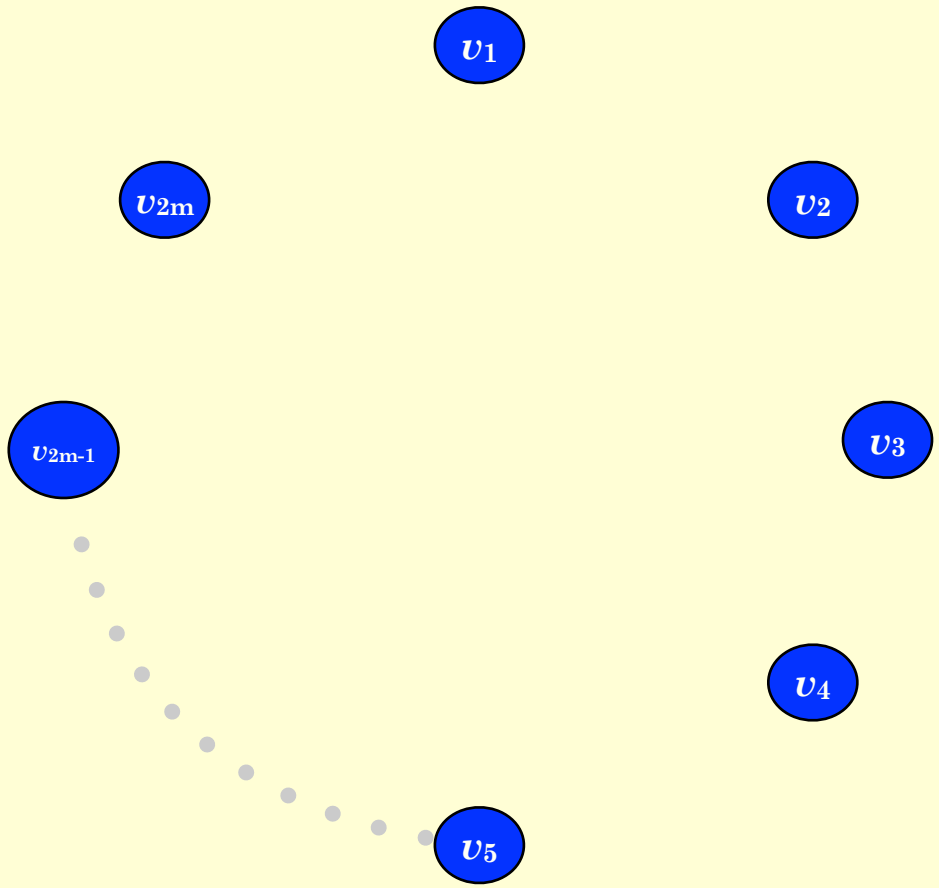
© 2006-2024 Jörg Striegnitz

Verfahren von Nico Christofides (1971)

Beweis: (Fortsetzung) Sei $S = \{v_1, \cdots, v_{2m}\}$ die Menge der Knoten mit ungeradem Grad in T. Dann können wir davon ausgehen, dass H_{Opt} diese Knoten durchläuft und zwischendurch andere Knotenmengen $\alpha_1 = \{a_{1,1}, \ldots, a_{1,k_1}\}, \cdots, \alpha_{2m} = \{a_{2m,1}, \ldots, a_{2m,k_{2m}}\}$ besucht werden ($\alpha_i = \emptyset$ ist denkbar!).

H_{Opt} hat demnach folgende Gestalt:

$$H_{Opt} = v_1, \alpha_1, v_2, \alpha_2, \cdots, \alpha_{2m-1}, v_{2m}, \alpha_{2m}, v_1$$



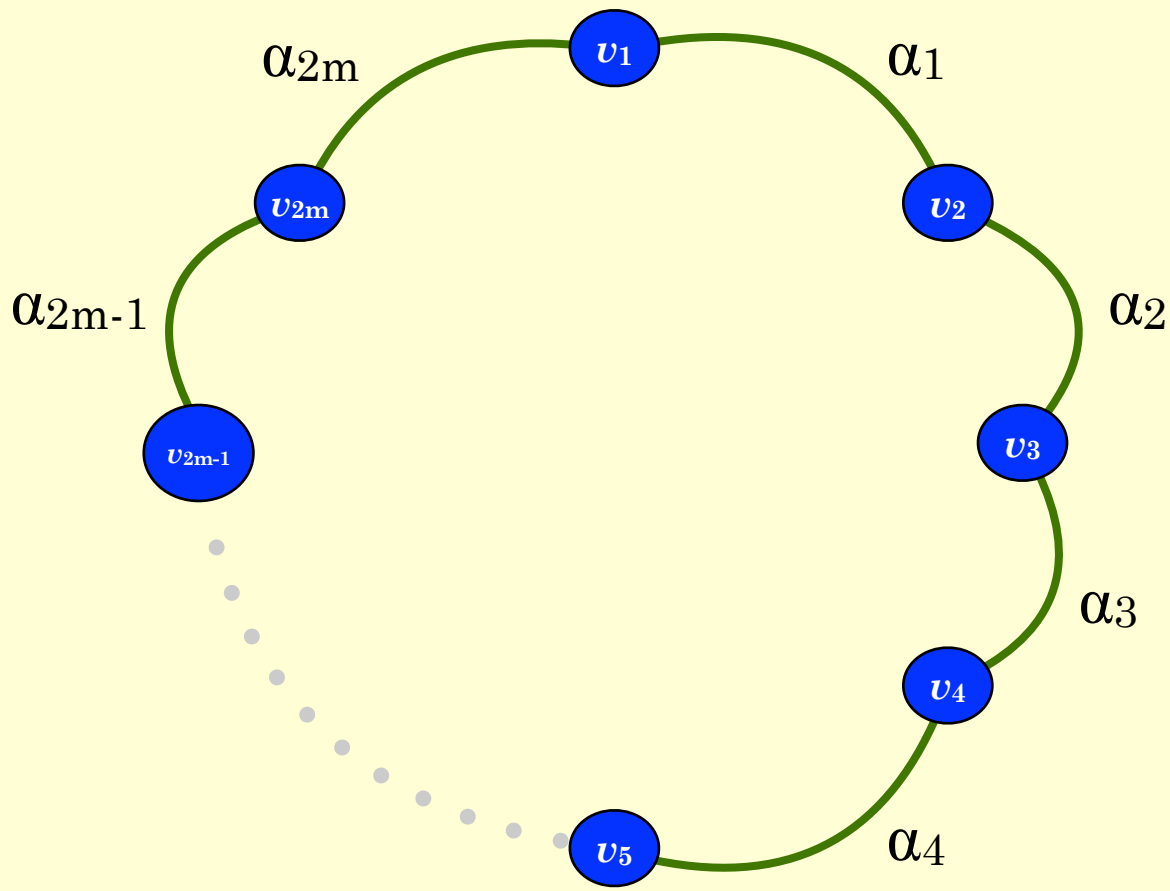
© 2006-2024 Jörg Striegnitz

Verfahren von Nico Christofides (1971)

Beweis: (Fortsetzung) Sei $S = \{v_1, \cdots, v_{2m}\}$ die Menge der Knoten mit ungeradem Grad in T. Dann können wir davon ausgehen, dass H_{Opt} diese Knoten durchläuft und zwischendurch andere Knotenmengen $\alpha_1 = \{a_{1,1}, \ldots, a_{1,k_1}\}, \cdots, \alpha_{2m} = \{a_{2m,1}, \ldots, a_{2m,k_{2m}}\}$ besucht werden ($\alpha_i = \emptyset$ ist denkbar!).

H_{Opt} hat demnach folgende Gestalt:

$$H_{Opt} = v_1, \alpha_1, v_2, \alpha_2, \cdots, \alpha_{2m-1}, v_{2m}, \alpha_{2m}, v_1$$



Optimale Rundreise H_{Opt}

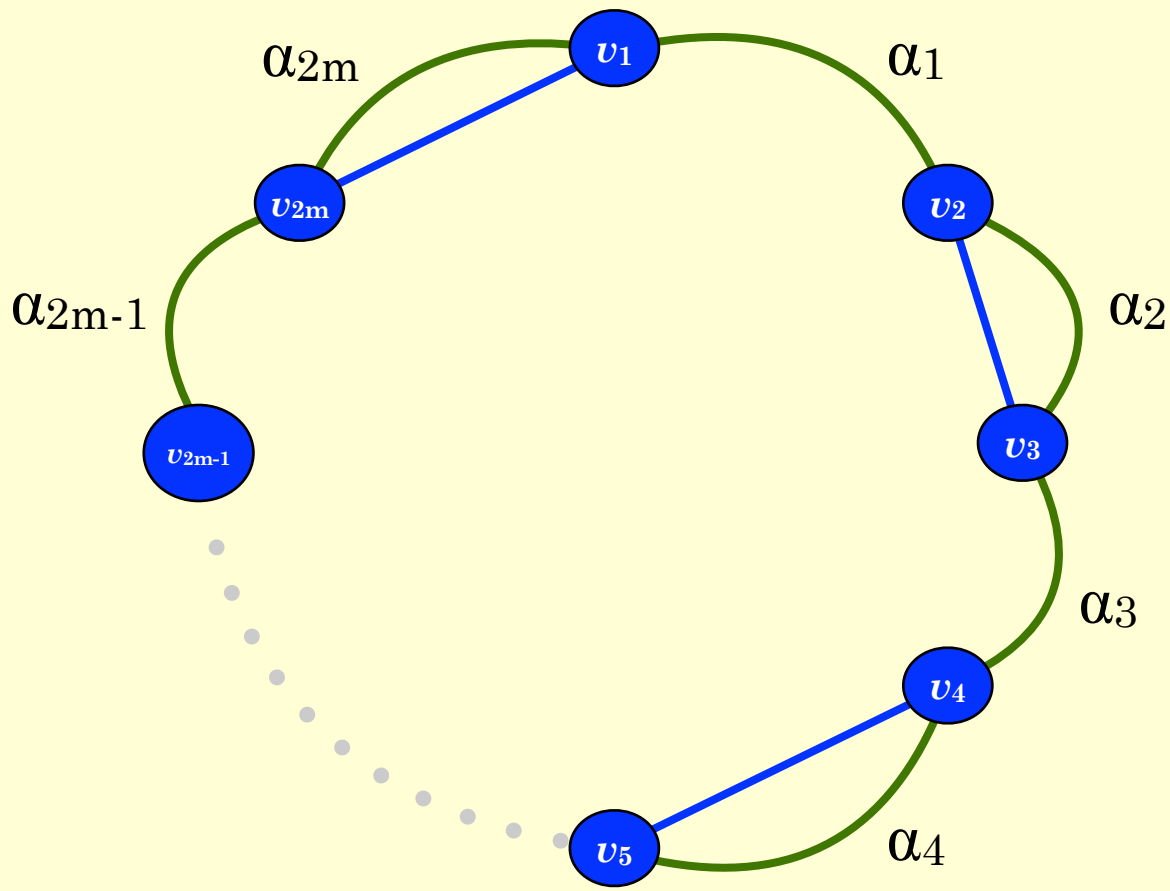
© 2006-2024 Jörg Striegnitz

Verfahren von Nico Christofides (1971)

Beweis: (Fortsetzung) Sei $S = \{v_1, \cdots, v_{2m}\}$ die Menge der Knoten mit ungeradem Grad in T. Dann können wir davon ausgehen, dass H_{Opt} diese Knoten durchläuft und zwischendurch andere Knotenmengen $\alpha_1 = \{a_{1,1}, \ldots, a_{1,k_1}\}, \cdots, \alpha_{2m} = \{a_{2m,1}, \ldots, a_{2m,k_{2m}}\}$ besucht werden ($\alpha_i = \emptyset$ ist denkbar!).

H_{Opt} hat demnach folgende Gestalt:

$$H_{Opt} = v_1, \alpha_1, v_2, \alpha_2, \cdots, \alpha_{2m-1}, v_{2m}, \alpha_{2m}, v_1$$



Optimale Rundreise H_{Opt}

Perfektes Matching M_1

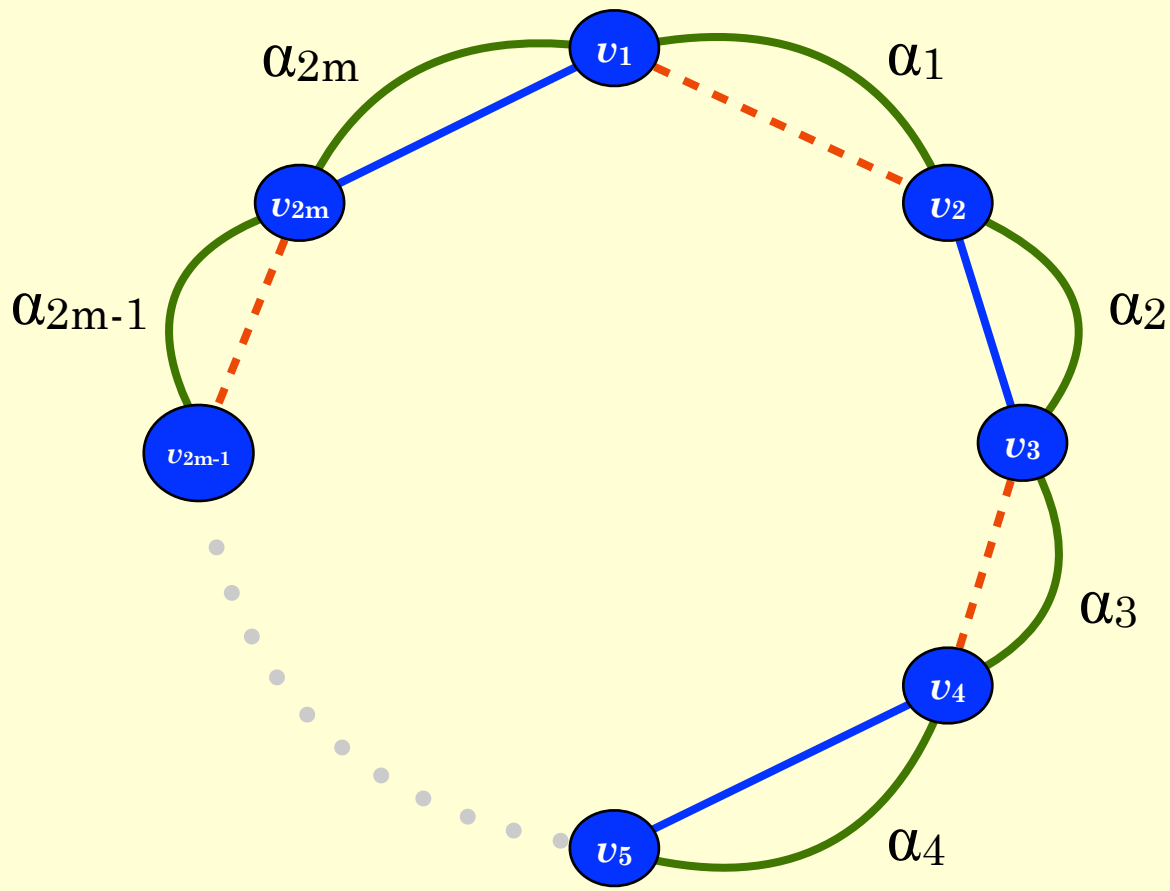
© 2006-2024 Jörg Striegnitz

Verfahren von Nico Christofides (1971)

Beweis: (Fortsetzung) Sei $S = \{v_1, \cdots, v_{2m}\}$ die Menge der Knoten mit ungeradem Grad in T. Dann können wir davon ausgehen, dass H_{Opt} diese Knoten durchläuft und zwischendurch andere Knotenmengen $\alpha_1 = \{a_{1,1}, \ldots, a_{1,k_1}\}, \cdots, \alpha_{2m} = \{a_{2m,1}, \ldots, a_{2m,k_{2m}}\}$ besucht werden ($\alpha_i = \emptyset$ ist denkbar!).

H_{Opt} hat demnach folgende Gestalt:

$$H_{Opt} = v_1, \alpha_1, v_2, \alpha_2, \cdots, \alpha_{2m-1}, v_{2m}, \alpha_{2m}, v_1$$



Optimale Rundreise H_{Opt}

Perfektes Matching M_1

Perfektes Matching M_2

© 2006-2024 Jörg Striegnitz

Verfahren von Nico Christofides (1971)

Beweis: (Fortsetzung) Wegen der Dreiecksungleichung gilt

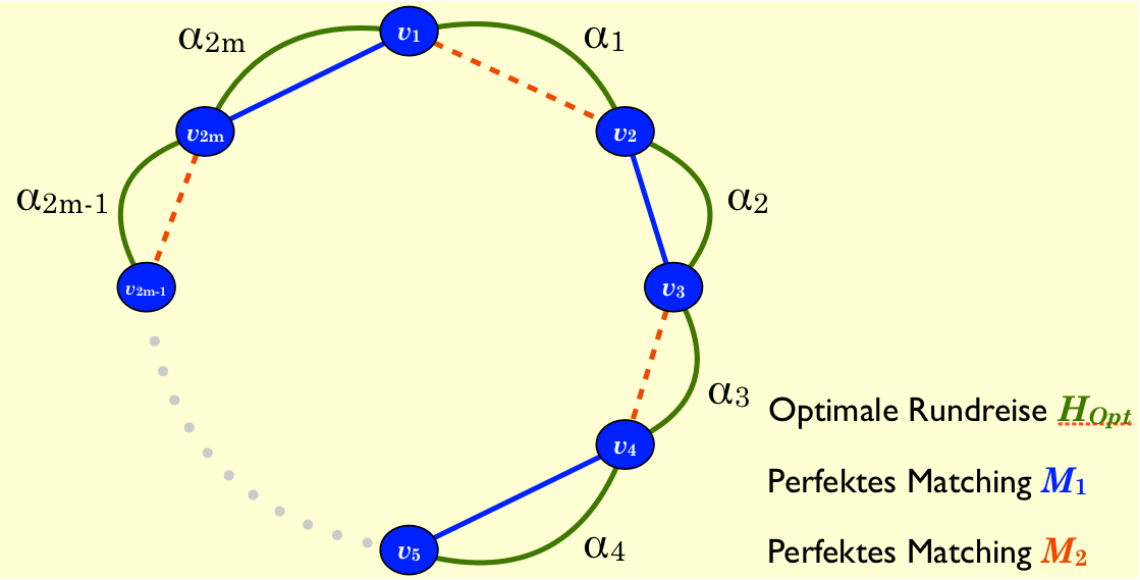
$$\text{cost}(H_{Opt}) \geq \text{cost}(M_1) + \text{cost}(M_2)$$

Und da M_1 und M_2 perfekte Matchings sind, gilt

$$\text{cost}(M_1) \geq \text{cost}(M) \quad \wedge \quad \text{cost}(M_2) \geq \text{cost}(M)$$

und somit

$$\frac{1}{2} \cdot \text{cost}(H_{Opt}) \geq \text{cost}(M)$$



© 2006-2024 Jörg Striegnitz

Verfahren von Nico Christofides (1971)

Beweis: (Fortsetzung) Wegen der Dreiecksungleichung gilt

$$\text{cost}(H_{Opt}) \geq \text{cost}(M_1) + \text{cost}(M_2)$$

Und da M_1 und M_2 perfekte Matchings sind, gilt

$$\text{cost}(M_1) \geq \text{cost}(M) \quad \wedge \quad \text{cost}(M_2) \geq \text{cost}(M)$$

und somit

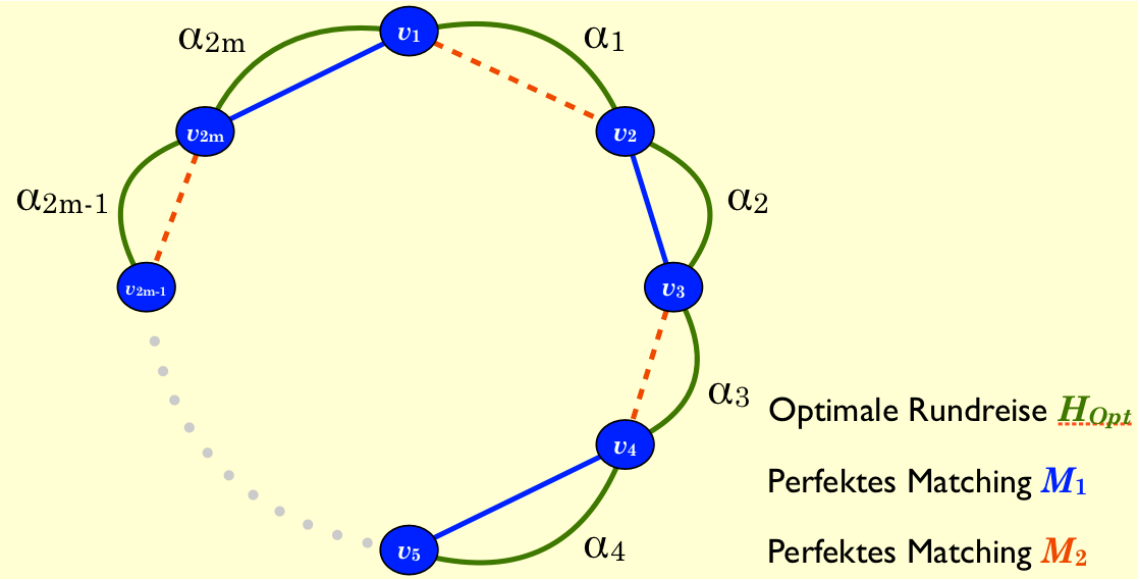
$$\frac{1}{2} \cdot \text{cost}(H_{Opt}) \geq \text{cost}(M)$$

Damit haben wir

$$\left| \begin{array}{rcl} \text{cost}(H) & \leq & \text{cost}(M) + \text{cost}(T) \\ \text{cost}(M) & \leq & \frac{1}{2} \cdot \text{cost}(H_{Opt}) \\ \text{cost}(T) & \leq & \text{cost}(H_{Opt}) \end{array} \right|$$

und es folgt sofort

$$\text{cost}(H) \leq \frac{3}{2} \cdot \text{cost}(H_{Opt})$$



© 2006-2024 Jörg Striegnitz

Ausblick

- **Das beste, bekannte Approximationsverfahren hat eine garantierte Güte von 1,4**
  Schöne-Ohren-Zerlegung von Sebö und Vygen (2014)

- **Neben den dargestellten Eröffnungsverfahren (dem Finden einer Lösung), gibt es noch Verbesserungsverfahren**
  z.B. simulated annealing, genetische Algorithmen

- **Das TSP gehört zu den schwersten, bekannten Problemen**
  Komplexitätsklasse NP – man nimmt an, dass es keinen Polynomialzeit-Algorithmus gibt

- **Die Praxis zeigt, das oft z.B. die *Nearest Neighbor*-Heuristik brauchbare Ergebnisse liefern kann**

© 2006-2024 Jörg Striegnitz