

# Graphentheoretische Konzepte und Algorithmen

## Thema 02

### Suchstrategien und kürzeste Wege

Julia Padberg



Hochschule für Angewandte Wissenschaften Hamburg  
*Hamburg University of Applied Sciences*

# Kantenlänge

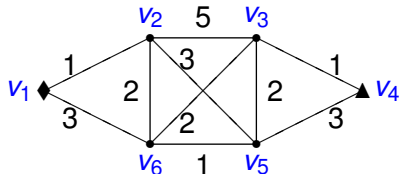
## Definition

Jeder Kante eine reelle Zahl zuordnen, die wir als **Länge** dieser Kante bezeichnen. Man spricht dann von einem Graphen mit bewerteten Kanten.

## BSP:

Sechs Städte werden durch das folgende Eisenbahnnetz miteinander verbunden. Die Länge einer Kante gibt dabei die durchschnittlich zu erwartende Fahrzeit in Stunden an.

Für einen Reisenden, der von  $\blacklozenge$ -Stadt nach  $\blacktriangle$ -City unterwegs ist, stellt sich die Frage, auf welchem Weg er am schnellsten zum Ziel kommt.



# Der Algorithmus von Dijkstra

- ▶ dient der Bestimmung kürzester Wege
- ▶ von einem fest vorgegebenen Knoten zu allen anderen Knoten
- ▶ in einem schlichten zusammenhängenden gerichteten Graphen
- ▶ mit endlicher Knotenmenge und nichtnegativ bewerteten Kanten
- ▶ und liefert einen Weg mit einer minimalen Gesamtlänge
- ▶ Bei der Anwendung dieses Verfahrens auf ungerichtete Graphen muß jede Kante durch ein Paar entgegengesetzt gerichteter Kanten ersetzt werden.

# Der Algorithmus von Dijkstra

## Vorbereitungsphase

### Algorithmus

Es bezeichne  $l_{ij}$  die Länge der Kante  $v_i v_j$ . Falls es keine Kante  $v_i v_j$  gibt, sei  $l_{ij} := \infty$ . Für jeden Knoten  $v_i \in V$  des zu untersuchenden Graphen werden drei Variable angelegt:

1.  $Entf_i$  gibt die bisher festgestellte kürzeste Entfernung von  $v_1$  nach  $v_i$  an. Der Startwert ist 0 für  $i = 1$  und  $\infty$  sonst.
2.  $Vorg_i$  gibt den Vorgänger von  $v_i$  auf dem bisher kürzesten Weg von  $v_1$  nach  $v_i$  an. Der Startwert ist  $v_1$  für  $i = 1$  und undefiniert sonst.
3.  $OK_i$  gibt an, ob die kürzeste Entfernung von  $v_1$  nach  $v_i$  schon bekannt ist. Der Startwert für alle Werte von  $i$  ist *false*.

# Der Algorithmus von Dijkstra

## Iterationsphase

### Algorithmus

#### Wiederhole

- ▶ Suche unter den Knoten  $v_i$  mit  $OK_i = false$  einen Knoten  $v_h$  mit dem kleinsten Wert von  $Entf_i$ .
- ▶ Setze  $OK_h := true$ . (Wieso?)
- ▶ Für alle Knoten  $v_j$  mit  $OK_j = false$ , für die die Kante  $v_h v_j$  existiert:
  - ▶ Falls gilt  $Entf_j > Entf_h + l_{hj}$  dann
    - ▶ Setze  $Entf_j := Entf_h + l_{hj}$
    - ▶ Setze  $Vorg_j := v_h$

solange es noch Knoten  $v_i$  mit  $OK_i = false$  gibt.

## Beispiel – siehe Folie 122

Nach der Vorbereitungsphase haben *Entf*, *Vorg* und *OK* die Werte:

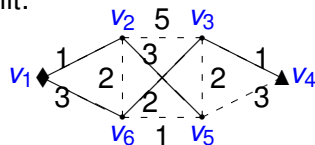
<i>i</i>	1	2	3	4	5	6
<i>Entf</i>	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
<i>Vorg</i>	1					
<i>OK</i>	<i>f</i>	<i>f</i>	<i>f</i>	<i>f</i>	<i>f</i>	<i>f</i>

Der Knoten  $v_4$  hat also von  $v_1$  den Abstand 6, und auf dem Weg von  $v_1$  nach  $v_4$  hat den vorletzten Knoten den Index  $Vorg_4 = 3$  sowie den drittletzten Knoten den Index  $Vorg_3 = 6$  und wegen  $Vorg_6 = 1$  lautet dann der komplette Weg:  $v_1, v_6, v_3, v_4$

In den Iterationsschritten ist  $v_h$  nacheinander der Knoten  $v_1, v_2, v_6, v_5, v_3, v_4$ , und am Ende haben *Entf*, *Vorg* und *OK* die Werte:

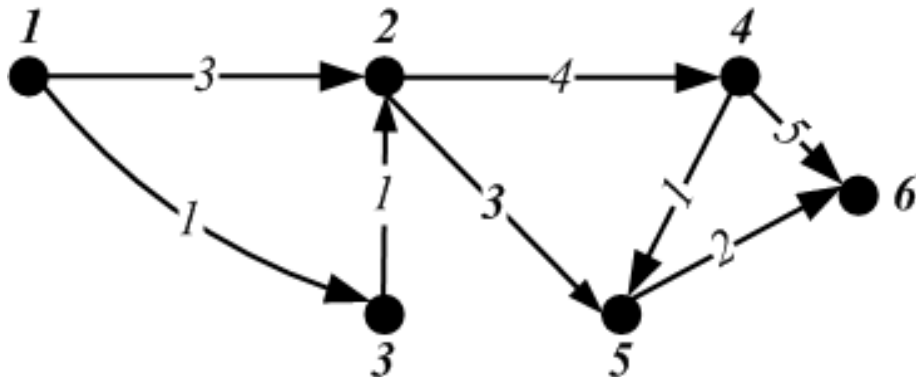
<i>i</i>	1	2	3	4	5	6
<i>Entf</i>	0	1	5	6	4	3
<i>Vorg</i>	1	1	6	3	2	1
<i>OK</i>	<i>t</i>	<i>t</i>	<i>t</i>	<i>t</i>	<i>t</i>	<i>t</i>

In der folgenden Darstellung sind die nicht zu den kürzesten Wegen gehörenden Kanten gestrichelt

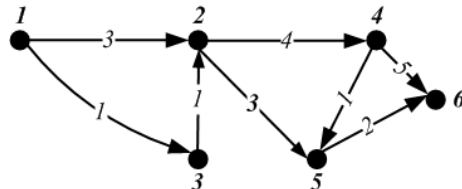


# Aufgabe 1:

Führen Sie bitte für diesen Graphen den Dijkstra-Algorithmus durch:



# Lösung von Aufgabe 1



	1	2	3	4	5	6
<i>Entf</i>	0	3	1	6	5	7
<i>Vorg</i>	1	1	1	2	2	5
<i>OK</i>	<i>t</i>	<i>t</i>	<i>t</i>	<i>t</i>	<i>t</i>	<i>t</i>



# Aufwandsabschätzung von Algorithmen

- ▶ Zeit bzw. Speicheraufwand eines Algorithmus, abhängig von einer bestimmten Eingabegröße  $n$
- ▶ bestimmte atomare Operationen innerhalb der Sprache ein konstanten, von  $n$  unabhängige Aufwand zugeordnet
- ▶ die tatsächliche Laufzeit bzw. der tatsächliche Speicheraufwand ist nicht relevant.
- ▶ Ziel einer Aufwandsabschätzung ist es daher herauszufinden, wie sich der Aufwand auf einer idealisierten (Turing-)Maschine verändert, wenn man die Eingabegröße verändert.
- ▶ Komplexitätsbetrachtungen ermöglichen den Vergleich der prinzipiellen Eigenschaften von Algorithmen
  - ▶ Z.B. das Zählen der Aufrufe einer Schlüsseloperation. Beispiel Sortieren:
    - ▶ Anzahl der Vergleiche
    - ▶ Anzahl der Vertauschungen

# Größenordnungsmäßige Schreibweise

## Definition

Eine reelle Funktion  $f(x)$  heißt *von der Größenordnung*  $g(x)$  (geschrieben:  $f(x) = O(g(x))$ ), wenn  $g(x)$  das Verhalten von  $f(x)$  für hinreichend große  $x$  im wesentlichen bestimmt – präziser:

$$f(x) = O(g(x)) \implies \exists c \in \mathbb{R} : \lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = c$$

## BSP: Beispiele

$$\begin{aligned}\frac{6}{5}x^2 - 5x &= O(x^2) \\ -x^5 + \ln x &= O(x^5) \\ 2^x - 5x^3 &= O(2^x)\end{aligned}$$

# O-Kalkül

Landau-Symbole <http://alda.iwr.uni-heidelberg.de/index.php/Effizienz#Algorithmen-Komplexit.C3.A4t>

- ▶ Als Maß für das Wachstum wird häufig das  $O$ -Kalkül verwendet.
- ▶ Das wichtigste Landau-Symbol ist  $O$ , mit dem man eine obere Schranke  $f \in O(g)$  für die Komplexität angeben kann.
- ▶ Die Wachstumsrate wird dabei in der Form  $O(f(n))$  angegeben, wobei die Funktion  $f(n)$  eine Obergrenze für das Wachstum des Algorithmus ist.
- ▶ Rechenregeln:
  1.  $f(x) \in O(f(x))$
  2.  $O(O(f(x))) \in O(f(x))$
  3.  $c \cdot O(f(x)) \in O(f(x))$  für jede Konstante  $c$
  4.  $O(f(x)) + c \in O(f(x))$  für jede Konstante  $c$
  5. Sequenzregel nacheinander ausgeführter Programmteile mit  $O(f(x))$  bzw.  $O(g(x))$  :  
 $O(f(x)) + O(g(x)) \in O(\max(f(x), g(x)))$
  6. Schachtelungsregel für geschachtelte Schleifen:  
 $O(f(x)) \cdot O(g(x)) \in O(f(x) \cdot g(x))$

# Zusammenhang zwischen Komplexität und Laufzeit

- ▶ Die Dauer einer Operation sei **1 ms**,
- ▶ dann erreichen Algorithmen verschiedener Komplexität etwa folgende Leistungen:

Komplexität	Operationen		
	in 1s	in 1min	in 1h
$O(N)$	1000	60.000	3.600.000
$O(N \log_2 N)$	140	4895	204094
$O(N^2)$	32	245	1898
$O(N^3)$	10	39	153
$O(2^N)$	10	16	21

Notation	Bedeutung	Anschauliche Erklärung	Beispiele für Laufzeiten
$f \in \mathcal{O}(1)$	$f$ ist beschränkt	$f$ überschreitet einen konstanten Wert nicht (unabhängig vom Wert des Arguments).	Nachschlagen des $i$ -ten Elementes in einem <a href="#">Array</a> .
$f \in \mathcal{O}(\log x)$	$f$ <a href="#">logarithmisches</a> Wachstum	$f$ wächst ungefähr um einen konstanten Betrag, wenn sich das Argument verdoppelt. Merke: Die Basis des Logarithmus ist egal.	<a href="#">Binäre Suche</a> im sortierten Feld mit $x$ Einträgen
$f \in \mathcal{O}(\sqrt{x})$	$f$ Wachstum wie die <a href="#">Wurzelfunktion</a>	$f$ wächst ungefähr auf das Doppelte, wenn sich das Argument vervierfacht	
$f \in \mathcal{O}(x)$	$f$ <a href="#">lineares</a> Wachstum	$f$ wächst ungefähr auf das Doppelte, wenn sich das Argument verdoppelt.	Suche im unsortierten Feld mit $x$ Einträgen
$f \in \mathcal{O}(x \log x)$	$f$ super-lineares Wachstum		Fortgeschrittenere Algorithmen zum Sortieren von $x$ Zahlen <a href="#">Mergesort</a> , <a href="#">Heapsort</a>
$f \in \mathcal{O}(x^2)$	$f$ <a href="#">quadratisches</a> Wachstum	$f$ wächst ungefähr auf das Vierfache, wenn sich das Argument verdoppelt	Einfache Algorithmen zum Sortieren von $x$ Zahlen <a href="#">Bubblesort</a>
$f \in \mathcal{O}(2^x)$	$f$ <a href="#">exponentielles</a> Wachstum	$f$ wächst ungefähr auf das Doppelte, wenn sich das Argument um eins erhöht	Rekursive Variante der <a href="#">Fibonacci-Folge</a>
$f \in \mathcal{O}(x!)$	$f$ <a href="#">faktorielles</a> Wachstum	$f$ wächst ungefähr um das $(x+1)$ -fache, wenn sich das Argument um eins erhöht.	<a href="#">TSP</a> (Mit <a href="#">Enumerationsansatz</a> )

# Komplexität des Dijkstra-Algorithmus

Für den Dijkstra-Algorithmus gilt:

Eingabeumfang und Arbeitsaufwand haben größenordnungsmäßig die Werte

Umfang der Eingabedaten :  $O(|V|^2)$

Anzahl der Arbeitsschritte :  $O(|V|^2)$

Komplexität in Abhängigkeit vom Graphen :  $O(|V|^2)$

Aufgabe 2: Warum?

# Komplexität des Dijkstra-Algorithmus

## Lösung

### Umfang der Eingabedaten

- ▶  $|V| \cdot (|V| - 1)$  Kantenlängen, also in  $O(|V|^2)$   
(falls der Graph ungerichtet ist, nur halb so viele).

### Anzahl der Arbeitsschritte

- ▶ zur Vorbereitung müssen  $3 \cdot |V|$  Variable initialisiert werden;
- ▶ bei der Iteration muß  $(|V| - 1)$ -mal wiederholt werden:  
suche eine von höchstens  $|V| - 1$  Knoten und  
inspiziere ihre höchstens  $|V| - 1$  Nachbarn.
- ▶ zusammen höchstens  $3 \cdot |V| + 2 \cdot (|V| - 1)^2$ , also in  $O(|V|^2)$

Komplexität:  $O(|V|^2) + O(|V|^2) = O(|V|^2)$

# Komplexität des Dijkstra-Algorithmus

## Bemerkung

Unsere Abschätzung hier ist sehr grob und hat weder die Anzahl der Kanten noch geeignete Datenstrukturen zur Speicherung der noch nicht besuchten Knoten berücksichtigt.

## Aus <http://de.wikipedia.org/wiki/Dijkstra-Algorithmus>

Sei  $m$  die Anzahl der Kanten und  $n$  die Anzahl der Knoten.

- ▶ Verwaltung der Knoten mit einer Liste:  $O(n^2 + m)$
- ▶ Verwaltung der Knoten mit einer Vorrangwarteschlange:  
 $O(n \cdot (1 + T_{\text{insert}} + T_{\text{empty}} + T_{\text{extractMin}}) + m \cdot (1 + T_{\text{decreaseKey}}))$
- ▶ Verwaltung der Knoten mit einem Fibonacci-Heap:  $O(n \cdot \log n + m)$



# A\*-Algorithmus

- ▶ ein informierter Suchalgorithmus  
eine Schätzfunktion (Heuristik) hilft zielgerichtet zu suchen und damit die Laufzeit zu verringern
- ▶ zur Berechnung eines kürzesten Pfades zwischen zwei Knoten in einem Graphen mit positiven Kantengewichten
- ▶ 1968 von Peter Hart, Nils J. Nilsson und Bertram Raphael beschrieben
- ▶ ist optimal  
es wird immer die optimale Lösung gefunden, falls eine existiert.

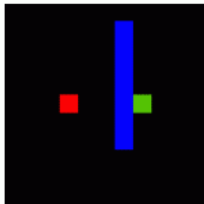
# Grundidee

Die Knoten werden während der Suche in drei verschiedene Klassen eingeteilt:

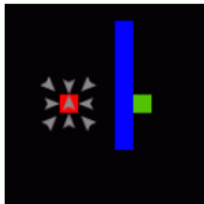
- ▶ unbekannte Knoten:  
noch nicht gefunden, noch kein Weg bekannt.  
Jeder Knoten (außer dem Startknoten) ist zu Beginn des Algorithmus unbekannt.
- ▶ bekannte Knoten:  
bekannte (suboptimale) Wege mit ihrem f-Wert in der Open List  
Wahl des vielversprechendsten Knoten  
anfangs nur der Startknoten
- ▶ abschließend untersuchte Knoten:  
kürzeste Wege in der Closed List  
anfangs leer.

# Beispiel

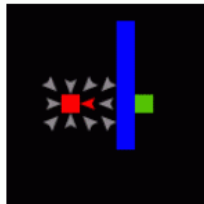
[http://www.geosimulation.de/methoden/a\\_stern\\_algorithmus.htm](http://www.geosimulation.de/methoden/a_stern_algorithmus.htm)



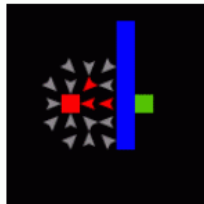
a



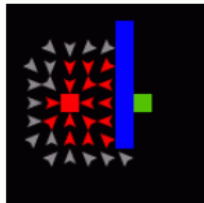
b



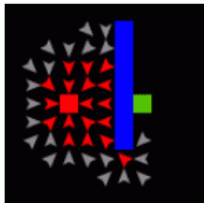
c



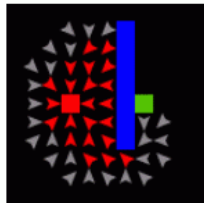
d



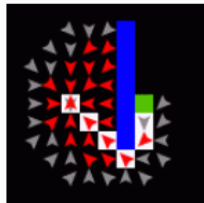
e



f



g



h

# A\*-Algorithmus: Initialisierung

## Algorithmus

Knoten  $V = \{v_1, \dots, v_n\}$

offenen Liste:  $OL$  nur der Startknoten  $v_1$  mit  $f_1 = h_1$ ,  $g_1 = 0$  und  $p_1 = v_1$

geschlossene Liste:  $CL$  leer

heuristische Knotenwerte :  $h_i \in \mathbb{R}^+$

Kantenlängen zwischen  $v_i$  und  $v_j$ :  $l_{ij} \in \mathbb{R}^+$

Vorgänger:  $p_i \in V$

aktueller Schätzwert  $f_i = \infty$  für  $i > 1$

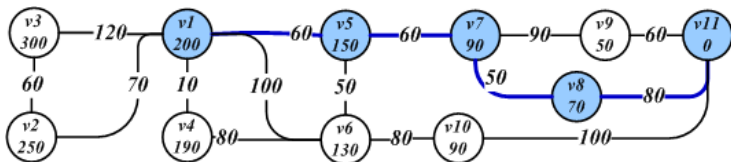
zurückgelegter Weg  $g_i = \infty$  für  $i > 1$

# A\*-Algorithmus

## Algorithmus

1. Knoten  $v_k$  mit niedrigsten  $f_k$  in  $OL$  suchen
2. Knoten  $v_k$  in die  $CL$  schieben.
3. Für die adjazente Knoten  $v_j$ , die nicht in der  $CL$  sind: wird geprüft, ob  $g_j > g_k + l_{k,j}$ . Falls  
JA, wird der aktuelle Knoten  $v_k$  Vorgängerknoten:  $p_j := v_k$   
und neuer g- und der f-Wert:  $g_j := g_k + l_{k,j}$  und  $f_j := g_j + h_j$
4. Falls der Zielknoten in der geschlossenen Liste; gehe zu 5.  
Falls kein Zielknoten gefunden und offene Liste leer; gehe zu 6.  
Sonst; gehe zu 1.
5. Der Pfad läßt sich vom Zielknoten aus mittels der Vorgänger bis zum Startknoten zurück verfolgen.
6. Es gibt keinen Pfad.

## Beispiel A\*



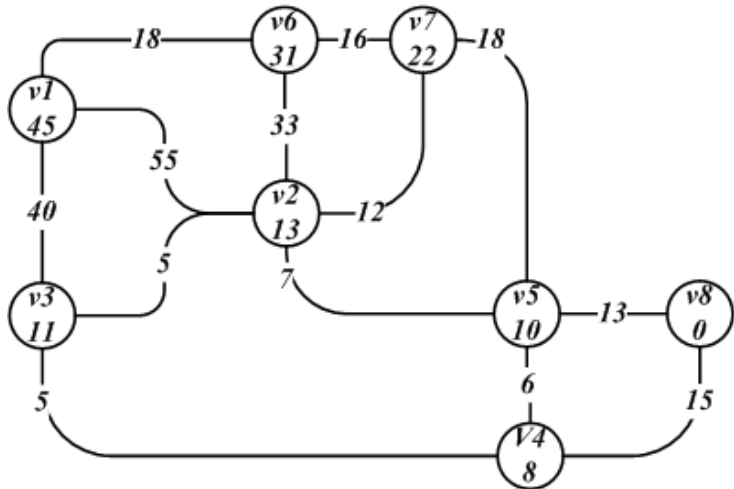
	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$	$v_6$	$v_7$	$v_8$	$v_9$	$v_{10}$	$v_{11}$
<i>Vorg</i>	$v_1$	$v_1$	$v_1$	$v_1$	$v_1$	$v_1$ $v_4$	$v_5$	$v_7$	$v_7$	$v_6$	$v_8$
<i>h</i>	200	250	300	190	150	130	90	70	50	90	0
<i>g</i>	0	70	120	10	60	100 90	120	170	210	170	250
<i>f</i>	200	320	420	200	210	230 220	210	240	260	260	250
<i>CL</i>	$t$			$t$	$t$	$t$	$t$	$t$			$t$

Länge von  $v_1$  nach  $v_{11}$ : 250

Weg  $v_1 - v_5 - v_7 - v_8 - v_{11}$

## Aufgabe 3:

Bitte berechnen Sie mit Hilfe des A\*-Algorithmus den kürzesten Weg von  $v_1$  nach  $v_8$ .



# Lösung von Aufgabe 3

	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$	$v_6$	$v_7$	$v_8$
<i>Vorg</i>	$v_1$	<del><math>v_1</math></del> <del><math>v_6</math></del> $v_3$	$v_1$	$v_3$	$v_4$	$v_1$	$v_6$	$v_4$
<i>h</i>	45	13	11	8	10	31	22	0
<i>g</i>	0	<del>55</del> <del>51</del> 45	40	45	51	18	34	60
<i>f</i>	45	<del>68</del> <del>64</del> 58	51	53	61	49	56	60
<i>CL</i>	<i>t</i>	<i>t</i>	<i>t</i>	<i>t</i>		<i>t</i>	<i>t</i>	<i>t</i>

Länge von  $v_1$  nach  $v_8$ : 60

Weg  $v_1 - v_3 - v_4 - v_8$



# Diskussion

1. Was muss für die Heuristik gelten, damit A\* klappt?  
die Heuristik muss monoton sein

(sie muss diese Bedingungen erfüllen):

- ▶ Die Kosten dürfen nie überschätzt werden.
- ▶ Für jeden Knoten  $k$  und jeden Nachfolgeknoten  $j$  von  $k$  muss gelten  
 $h(k) \leq l_{k,j} + h(j)$ .

Ist eine der beiden Bedingungen verletzt ergibt sich eine nicht optimale Lösung.

2. Ist der A\* immer besser als Dijkstra?

- ▶ nicht, wenn die Heuristik sehr komplex ist.
- ▶ nicht, wenn der Zielknoten auch noch gefunden werden muss