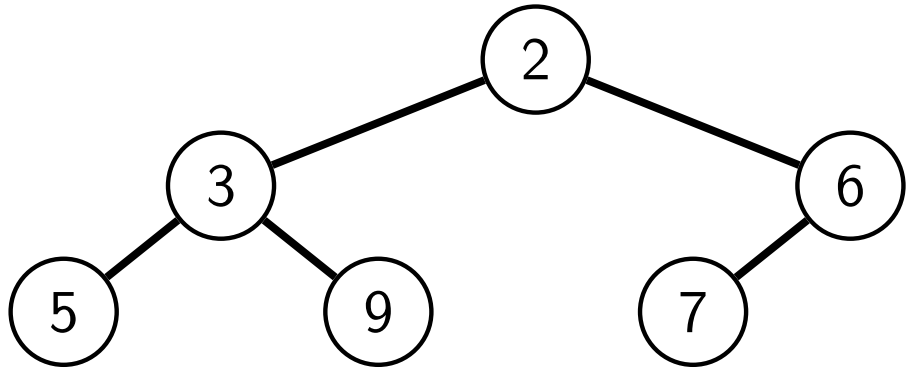
Übung zu
Algorithmen und Datenstrukturen (für ET/IT)

Wintersemester 2012/13

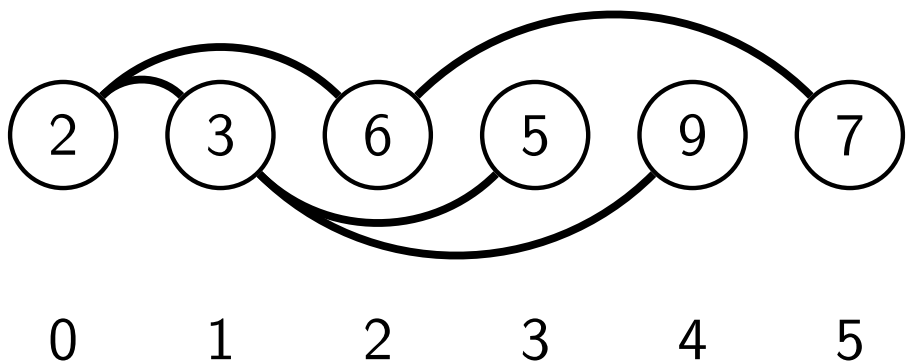
Jakob Vogel

Computer-Aided Medical Procedures
Technische Universität München

Heap im Speicher



- Wurzel bei Index 0
- Vaterknoten zu gegebenem Index i bei $\lfloor (i-1)/2 \rfloor$
- Kindknoten bei $2 \cdot i + 1$ und $2 \cdot i + 2$

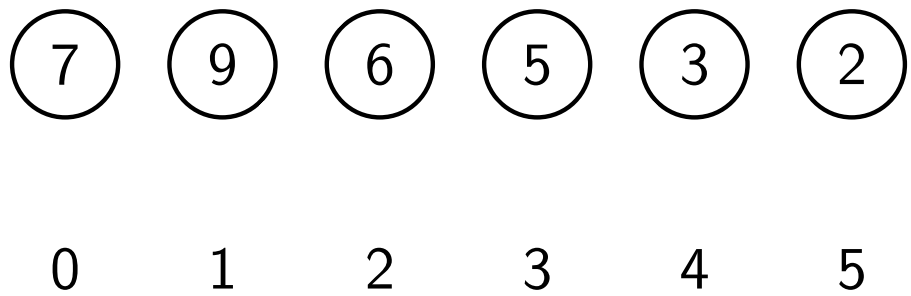


Heap Sort

- Idee: Behandle Array als Binärbaum und stelle Heap-Eigenschaft her (**buildMinHeap**)
- Möglichkeit 1: Lese wiederholt das Minimum aus (**extractMin**)
  - Aufsteigende Ordnung bei Min-Heap
  - Zusätzlicher Speicherplatz erforderlich
- Möglichkeit 2: Tausche Wurzel ans Ende, ignoriere diesen Wert in Zukunft, und lasse dann die neue große Wurzel absinken (**minHeapify**)
  - Absteigende Ordnung bei Min-Heap
  - In-place

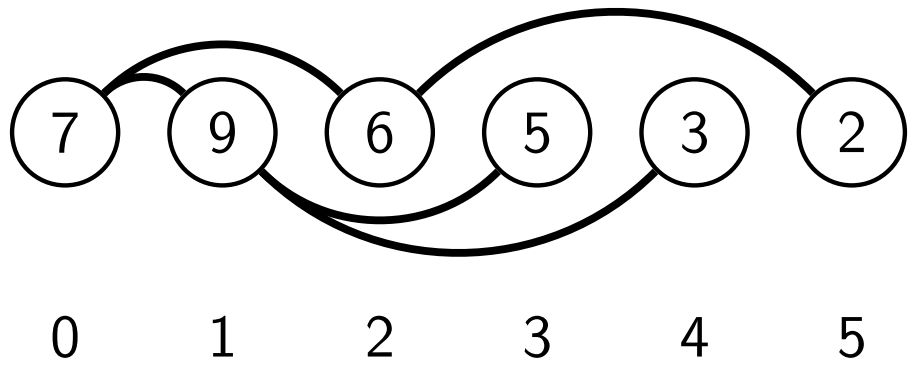
Heap Sort

- Eingabedaten



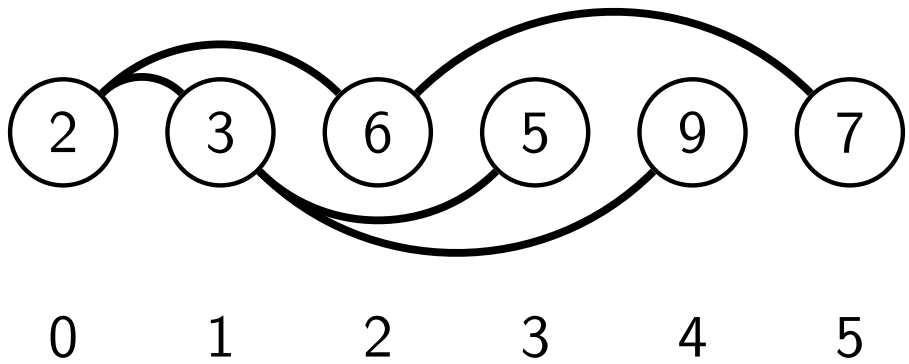
Heap Sort

- Interpretation als (fast) vollständiger Binärbaum



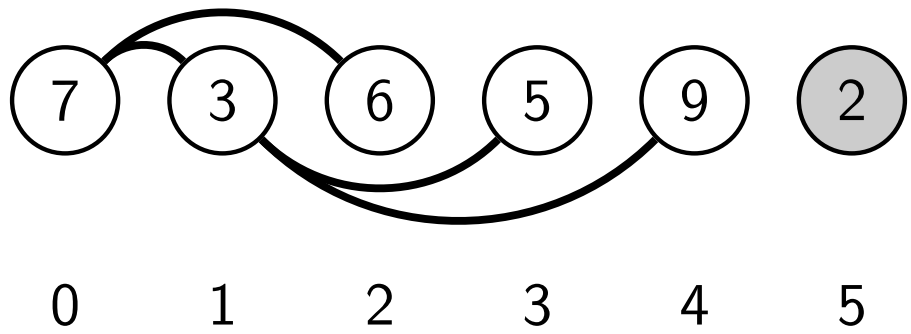
Heap Sort

- Heap-Eigenschaft hergestellt via **buildMinHeap**



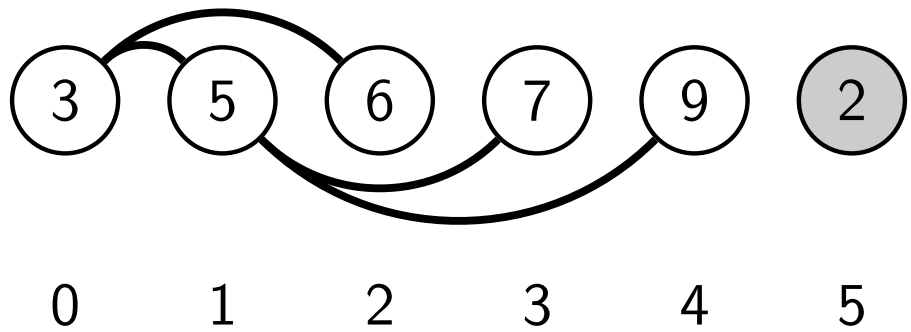
Heap Sort

- Wurzel mit Ende vertauscht, „Kürzung“



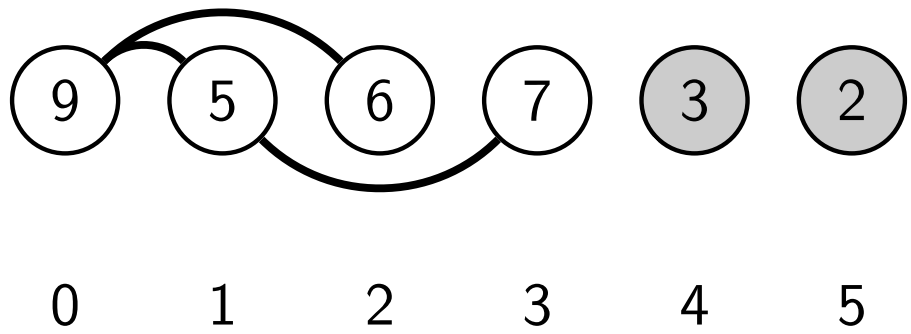
Heap Sort

- Heap-Eigenschaft hergestellt via **minHeapify** ab Wurzel



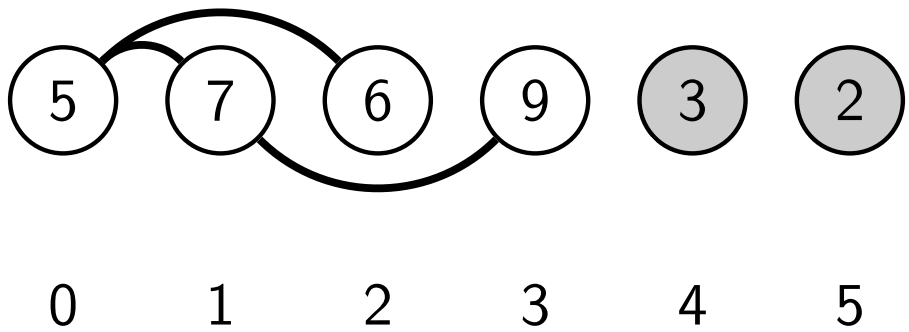
Heap Sort

- Wurzel mit Ende vertauscht, „Kürzung“



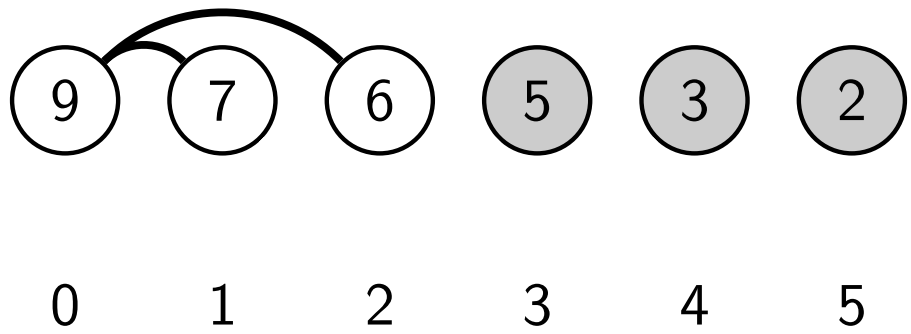
Heap Sort

- Heap-Eigenschaft hergestellt via **minHeapify** ab Wurzel



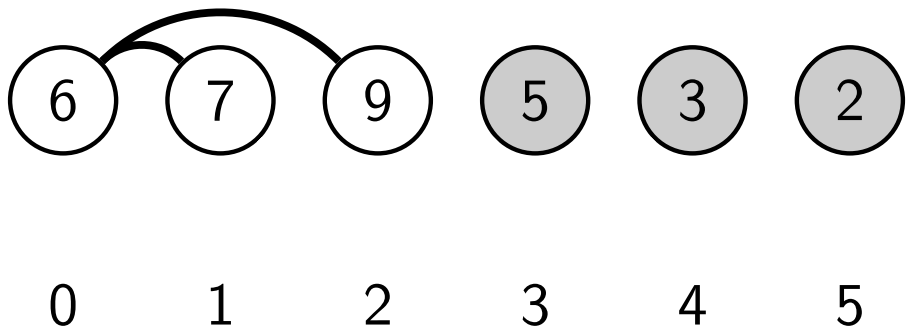
Heap Sort

- Wurzel mit Ende vertauscht, „Kürzung“



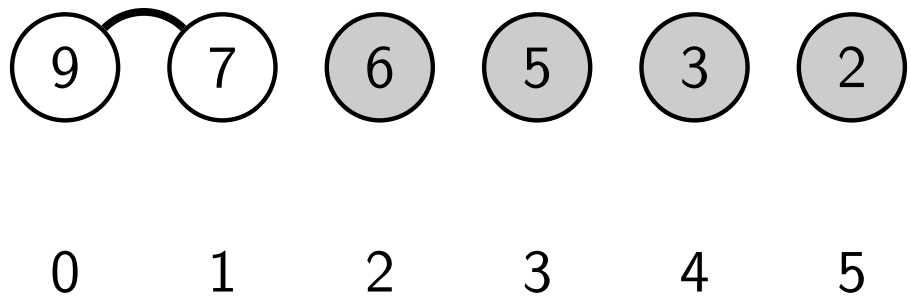
Heap Sort

- Heap-Eigenschaft hergestellt via **minHeapify** ab Wurzel



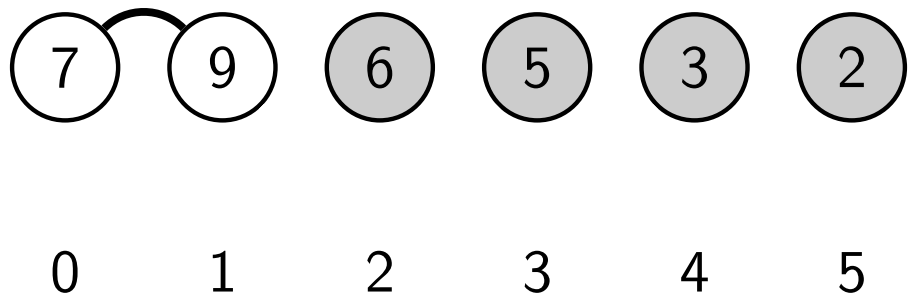
Heap Sort

- Wurzel mit Ende vertauscht, „Kürzung“



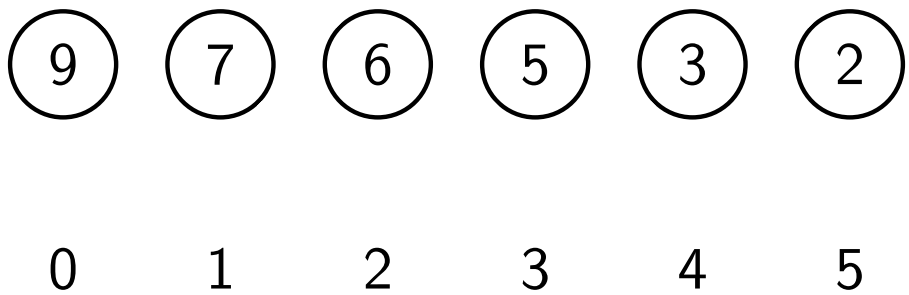
Heap Sort

- Heap-Eigenschaft hergestellt via **minHeapify** ab Wurzel



Heap Sort

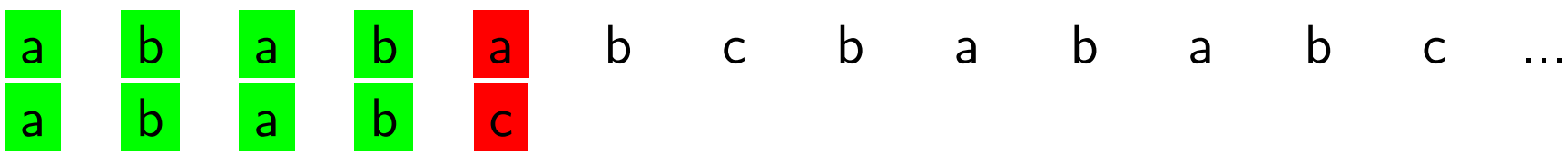
- Wurzel mit Ende vertauscht, **absteigende Ordnung!**




Code

Stringsuche

- Finde ein *Wort* w der Länge m innerhalb eines *Satzes* s der Länge $n > m$
  - Suche auf Festplatte, im Internet, ...
  - Syntax-Highlighting in der IDE
- Natürlich Teil der Standard-Bibliothek
  - `std::string::find`
- Naïver Ansatz mittels Brute-Force-Suche
  - Besuche jedes der n Zeichen des Satzes
  - Vergleiche dort dann die bis zu m Zeichen des Wortes



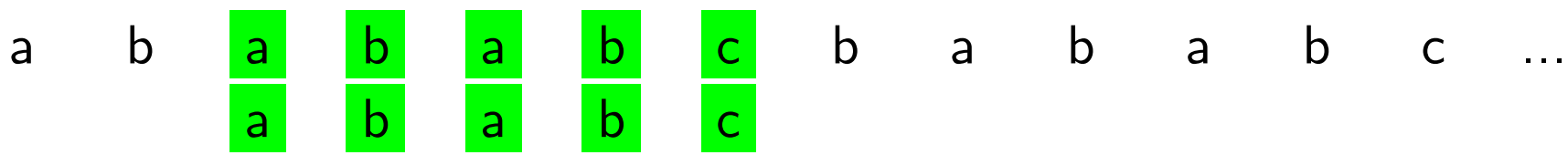
Stringsuche

- Finde ein *Wort* w der Länge m innerhalb eines *Satzes* s der Länge $n > m$
  - Suche auf Festplatte, im Internet, ...
  - Syntax-Highlighting in der IDE
- Natürlich Teil der Standard-Bibliothek
  - `std::string::find`
- Naïver Ansatz mittels Brute-Force-Suche
  - Besuche jedes der n Zeichen des Satzes
  - Vergleiche dort dann die bis zu m Zeichen des Wortes

a	b	a	b	a	b	c	b	a	b	a	b	c	...
	a	b	a	b	c								

Stringsuche

- Finde ein *Wort* w der Länge m innerhalb eines *Satzes* s der Länge $n > m$
  - Suche auf Festplatte, im Internet, ...
  - Syntax-Highlighting in der IDE
- Natürlich Teil der Standard-Bibliothek
  - `std::string::find`
- Naïver Ansatz mittels Brute-Force-Suche
  - Besuche jedes der n Zeichen des Satzes
  - Vergleiche dort dann die bis zu m Zeichen des Wortes



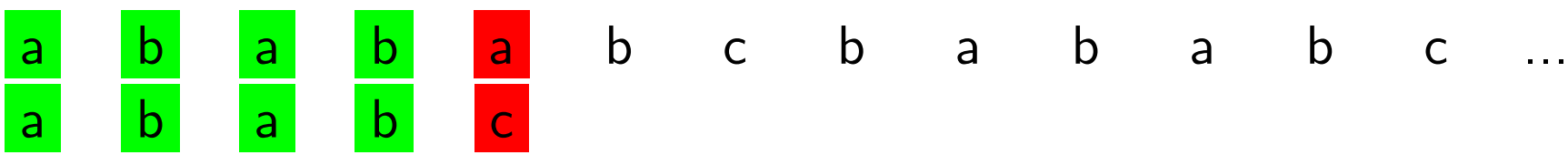

Code

Knuth-Morris-Pratt

- Wenn der Vergleich mitten im Wort scheitert, weiss man bereits über die vorhergehenden Äquivalenzen Bescheid.
- Idee: Nutze *Präfix-Tabelle*

Index:	0	1	2	3	4
Zeichen:	a	b	a	b	c
Überspringbar:	-1	0	0	1	2

  - Scheitert der Vergleich bei „c“, so stehen davor „a“ und „b“
  - „ab“ ist ein Präfix des Wortes „ababc“!
  - Spare den Vergleich der ersten beiden Zeichen des Wortes!

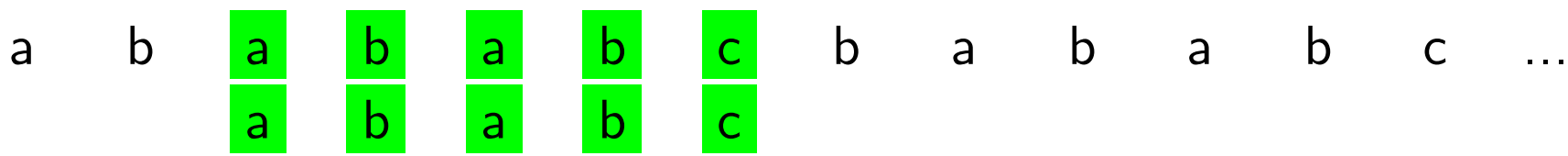


Knuth-Morris-Pratt

- Wenn der Vergleich mitten im Wort scheitert, weiss man bereits über die vorhergehenden Äquivalenzen Bescheid.
- Idee: Nutze *Präfix-Tabelle*

Index:	0	1	2	3	4
Zeichen:	a	b	a	b	c
Überspringbar:	-1	0	0	1	2

  - Scheitert der Vergleich bei „c“, so stehen davor „a“ und „b“
  - „ab“ ist ein Präfix des Wortes „ababc“!
  - Spare den Vergleich der ersten beiden Zeichen des Wortes!



Berechnung der Präfix-Tabelle

- Speichere pro Stelle die Zahl der vorangehenden Zeichen, die ein Präfix des gesamten Suchwortes bilden

a	b	a	b	c
-1				

- Keine vorangehenden Zeichen, daher Standardwert

Berechnung der Präfix-Tabelle

- Speichere pro Stelle die Zahl der vorangehenden Zeichen, die ein Präfix des gesamten Suchwortes bilden

a	b	a	b	c
-1	0			

- Vorangehende Zeichen Präfix, aber Wortbeginn

Berechnung der Präfix-Tabelle

- Speichere pro Stelle die Zahl der vorangehenden Zeichen, die ein Präfix des gesamten Suchwortes bilden

a	b	a	b	c
-1	0	0		

- Vorangehende Zeichen Präfix, aber Wortbeginn

Berechnung der Präfix-Tabelle

- Speichere pro Stelle die Zahl der vorangehenden Zeichen, die ein Präfix des gesamten Suchwortes bilden

a	b	a	b	c		
		a	b	a	b	c
-1	0	0	1			

- Ein vorangehendes Zeichen Präfix

Berechnung der Präfix-Tabelle

- Speichere pro Stelle die Zahl der vorangehenden Zeichen, die ein Präfix des gesamten Suchwortes bilden

a	b	a	b	c		
		a	b	a	b	c
-1	0	0	1	2		

- Zwei vorangehende Zeichen Präfix

Beispiele

```
 f l i e g e n   f l i e g e n   n a c h t s ?
-1 0 0 0 0 0 0 0 0 1 2 3 4 5 6 7 8 0 0 0 0 0 0
```

```
 s i n n   u n d   u n s i n n   s i n d . . .
-1 0 0 0 0 0 0 0 0 0 0 0 1 2 3 4 5 1 2 3 0 0 0
```

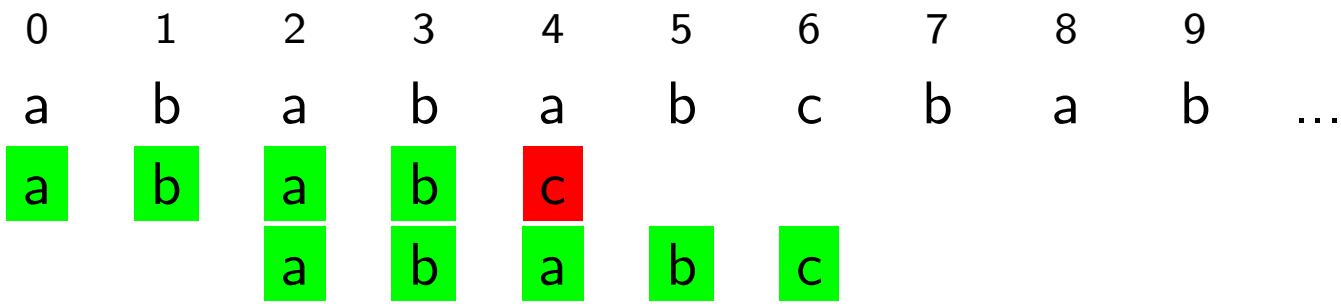
```
 c a c h e . p u s h _ b a c k ( v a l u e ) ;
-1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
```


Code

Suche

Annahme: Wir scheitern beim Vergleichen des Wortes mit dem Satz...

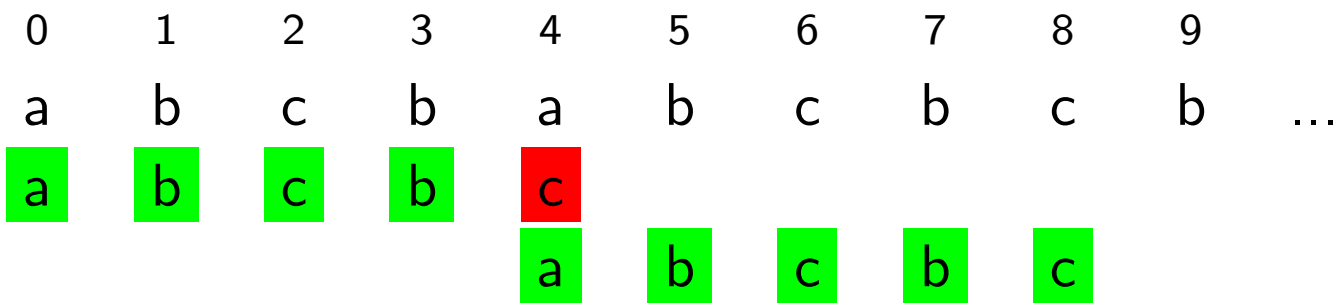
- ...an einer Stelle, wo die Präfixtabelle **nicht** 0 ist:
  - Wir wissen von dem Präfix, und fahren mitten im Wort fort, hier $i = 2$, $j = 2$ und dann Test ob $w[j] = s[i + j]$...



Suche

Annahme: Wir scheitern beim Vergleichen des Wortes mit dem Satz...

- ...an einer Stelle, wo die Präfixtabelle 0 ist:
  - Es gab kein Präfix, also müssen wir nicht zurück (wie bei Brute-Force) und können direkt weitermachen, also hier $i = 4$, $j = 0$ und dann Test ob $w[j] = s[i + j]$...




Code

Weitere Beispiele für Präfix-Tabellen

```
 s t d : : s t r i n g
-1 0 0 0 0 0 1 2 0 0 0

 i n t   s o r t ( i n t *  d a t a ,  i n t  n )
-1 0 0 0 0 0 0 0 0 0 1 2 3 0 0 0 0 0 0 0 0 1 2 3 4 0

 T U  M u e n c h e n
-1 0 0 0 0 0 0 0 0 0 0
```

Pfadsuche

- Beispiel für einen Graphenalgorithmus: Suche kürzeste bzw. günstigste Pfade in einem gewichteten Graphen!
- *Single Pair Shortest Path (SPSP)*: Finde günstigsten Pfad zwischen zwei Knoten!
- *Single Source Shortest Path (SSSP)*: Finde günstigste Pfade zu allen Knoten von einem einzigen Startknoten aus!

- Dijkstra-Algorithmus!

Dijkstra-Algorithmus

$pred = \{\emptyset, ..., \emptyset\}$; $d = \{\infty, ..., \infty\}$; $d[v_{\text{start}}] = 0$;
MinPriorityQueue Q mit Elementen v, Schlüsseln d;

while ($\neg$isEmpty(Q)) {
    v = extractMin(Q);

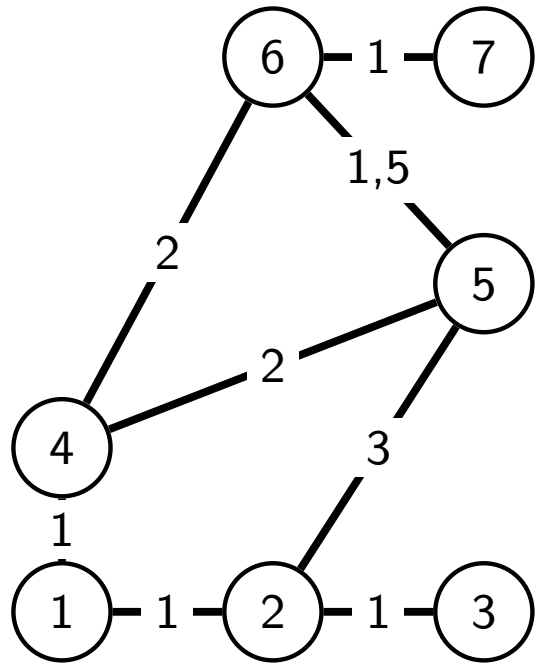
    // Ankunft am Zielknoten bei SPSP
    if ($v = v_{\text{stop}}$) {
        return;
    }

    // Für jeden unbesuchten Nachbarn n
    foreach (Knoten $n : \{v, n\} \in E \wedge n \in Q$) {
        // Günstigeren Pfad nach n über v gefunden
        if ($d[v] + w[\{v, n\}] < d[n]$) {
            $pred[n] = v$; $d[n] = d[v] + w[\{v, n\}]$;
            decreaseKey($Q, n, d[n]$);
        }
    }
}

Beispiel: Kürzester Pfad von 1 nach 7

- Initialisiere Startknoten 1.

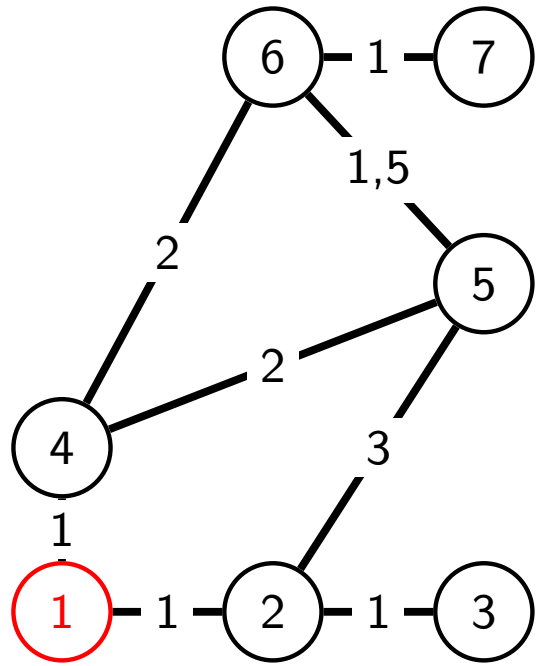
	1	2	3	4	5	6	7
pred							
dist	0	∞	∞	∞	∞	∞	∞
Q	1	2	3	4	5	6	7



Beispiel: Kürzester Pfad von 1 nach 7

- Besuche Knoten 1, da minimale Distanz.

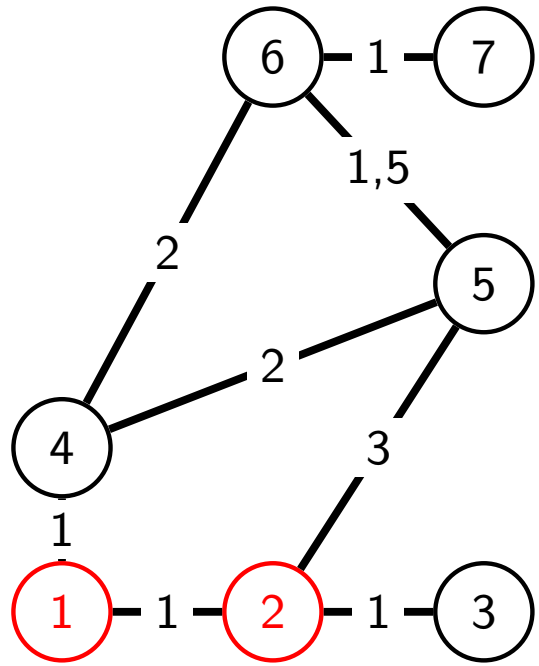
	1	2	3	4	5	6	7
pred		1		1			
dist	0	1	∞	1	∞	∞	∞
Q	2	4	3	5	6	7	



Beispiel: Kürzester Pfad von 1 nach 7

- Besuche Knoten 2.

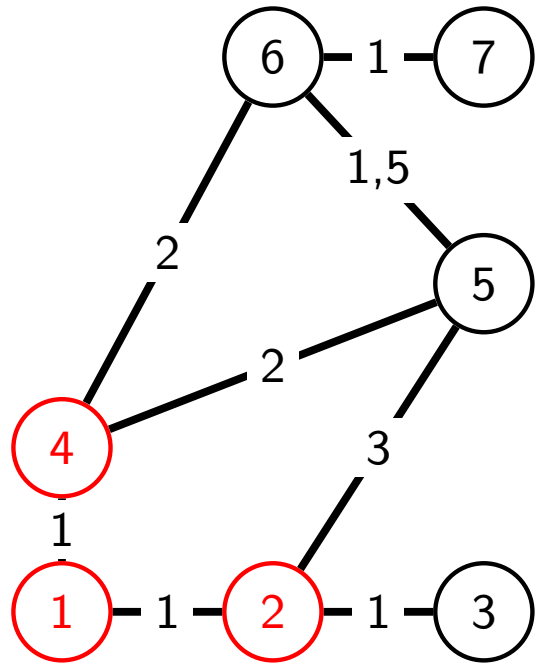
	1	2	3	4	5	6	7
pred		1	2	1	2		
dist	0	1	2	1	4	∞	∞
Q	4	3	5	6	7		



Beispiel: Kürzester Pfad von 1 nach 7

- Besuche Knoten 4, Update für Knoten 5.

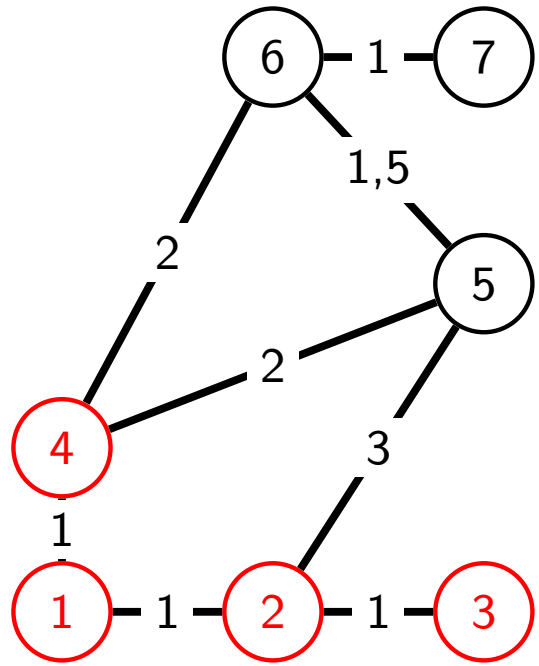
	1	2	3	4	5	6	7
pred		1	2	1	4	4	
dist	0	1	2	1	3	3	∞
Q	3	5	6	7			



Beispiel: Kürzester Pfad von 1 nach 7

- Besuche Knoten 3, keine unbesuchten Nachbarn.

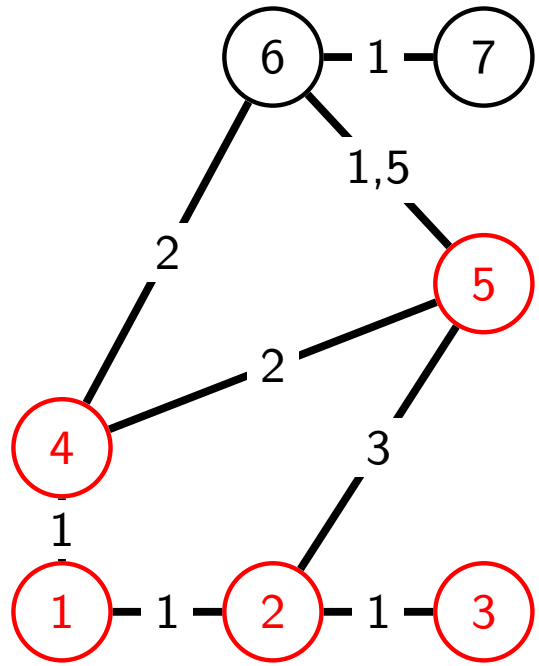
	1	2	3	4	5	6	7
pred		1	2	1	4	4	
dist	0	1	2	1	3	3	∞
Q	5	6	7				



Beispiel: Kürzester Pfad von 1 nach 7

- Besuche Knoten 5, Pfad zu Knoten 6 über 5 schlechter.

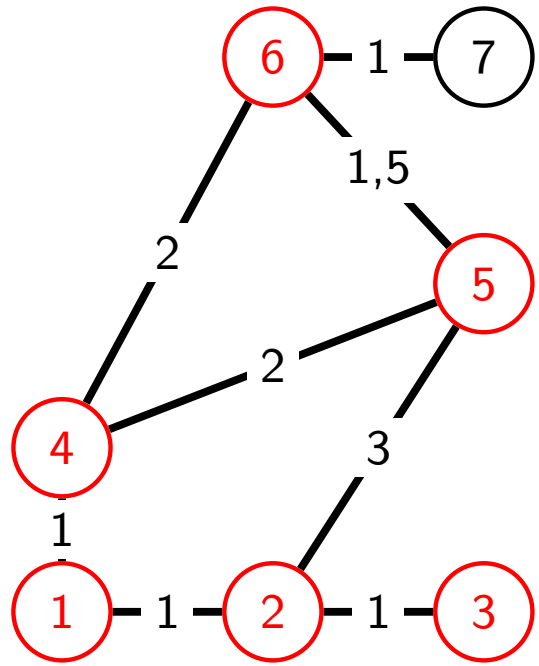
	1	2	3	4	5	6	7
pred		1	2	1	4	4	
dist	0	1	2	1	3	3	∞
Q	6	7					



Beispiel: Kürzester Pfad von 1 nach 7

- Besuche Knoten 6.

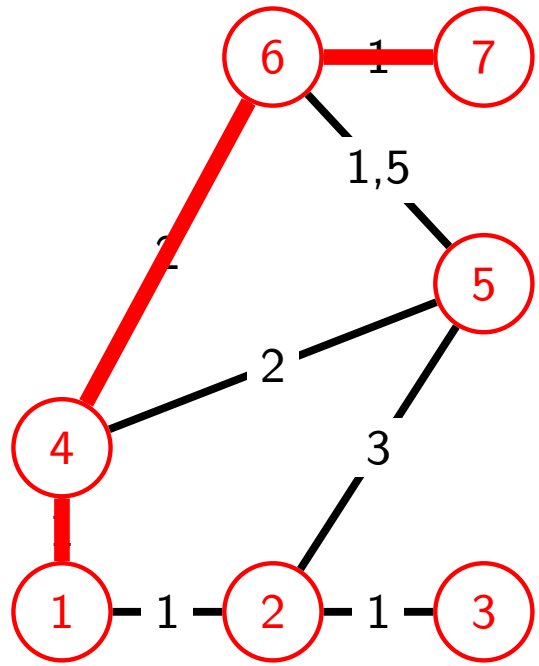
	1	2	3	4	5	6	7
pred		1	2	1	4	4	6
dist	0	1	2	1	3	3	4
Q	7						



Beispiel: Kürzester Pfad von 1 nach 7

- Erreiche Ziel 7, Ergebnis $1 \to 4 \to 6 \to 7$ mit Kosten 4.

	1	2	3	4	5	6	7
pred		1	2	1	4	4	6
dist	0	1	2	1	3	3	4
Q							



Pfad auslesen

- Start beim Zielknoten
- Iteratives Einfügen des Vorgängers des aktuellen Pfadanfangs
- Abbruch beim Startknoten

$path = \{v_{\text{stop}}\};$
while $(path[0] \neq v_{\text{start}})$ {
    $path = \{pred[path[0]]\} \cup path;$
}

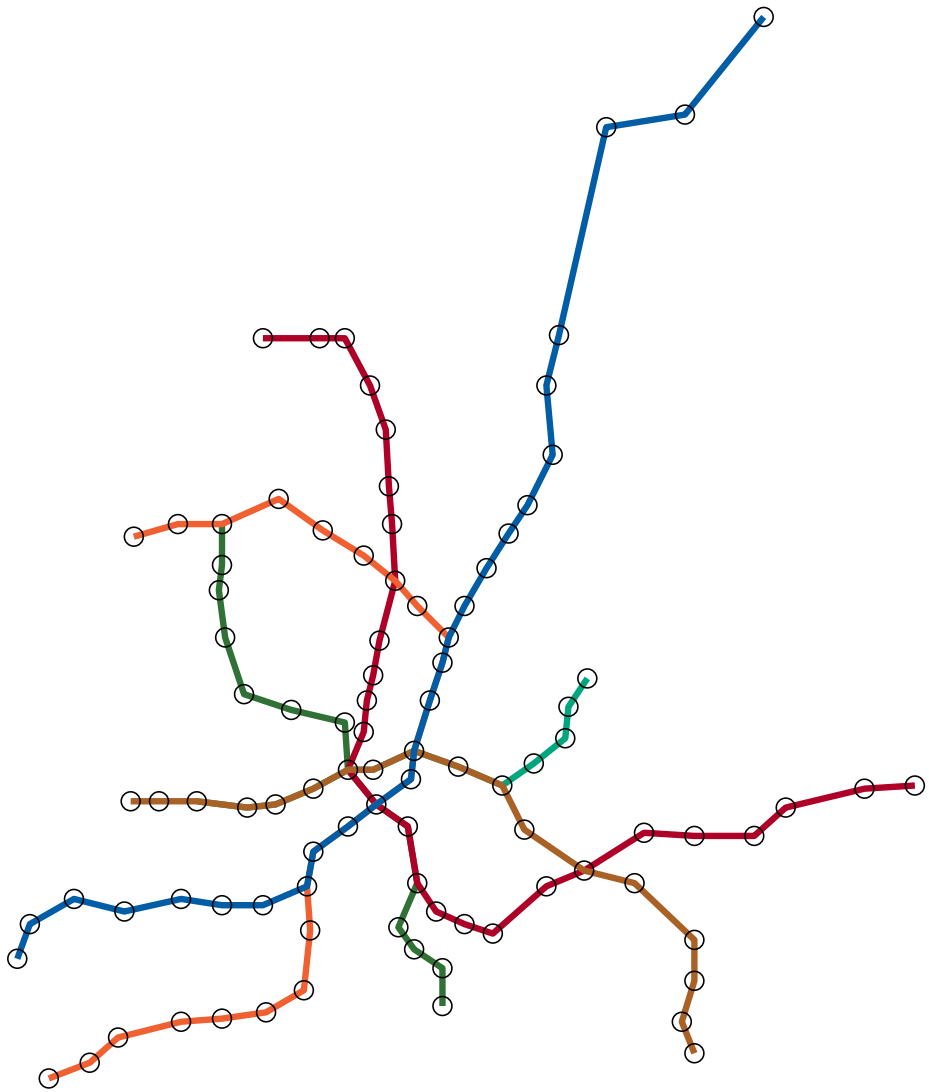

Code

Beobachtungen

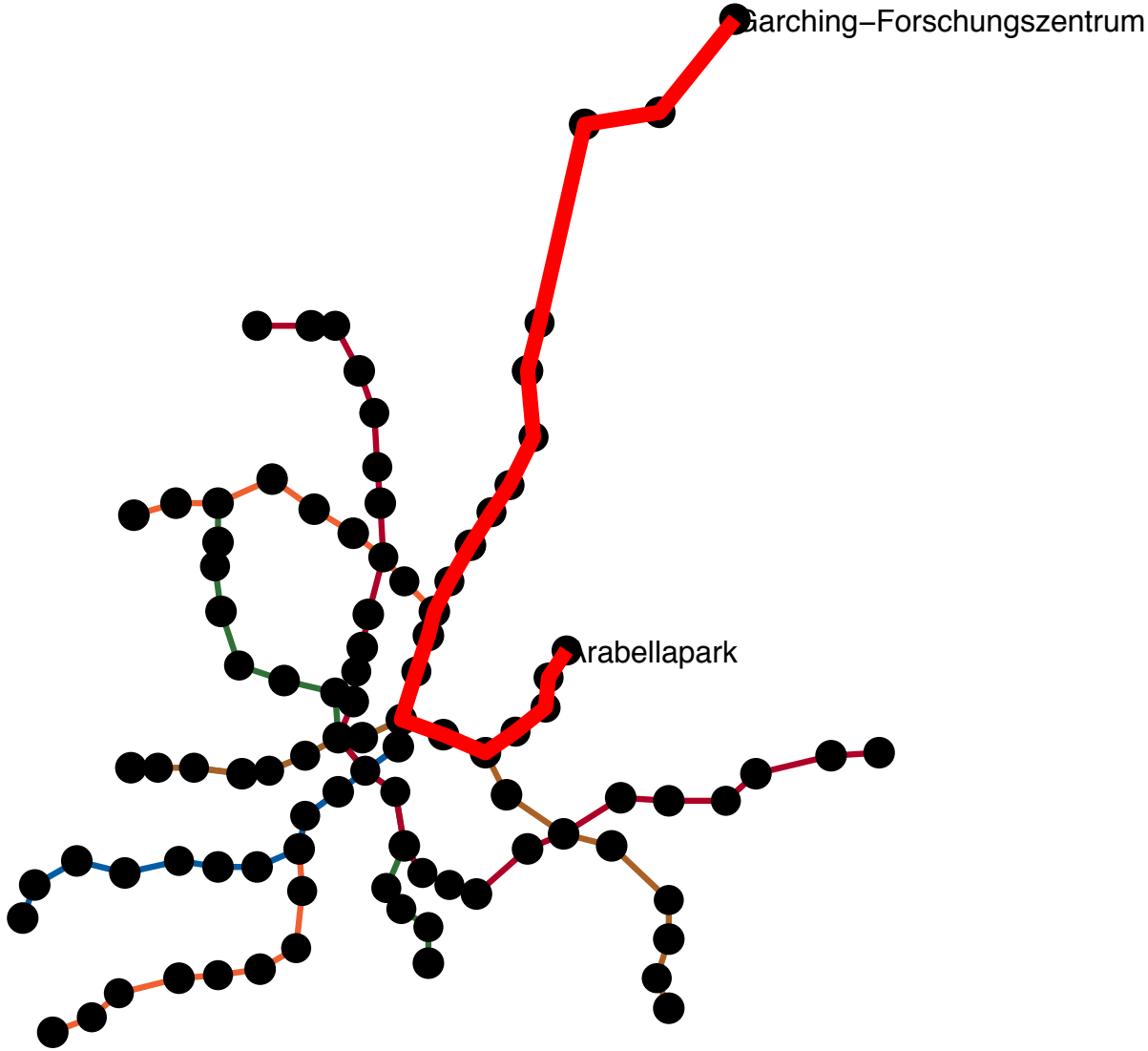
- Bei gleichem Start und anderem Ziel grundsätzlich gleicher Ablauf
- Allenfalls Abbruch in früherem Programmzustand

- *SPSP*-Lösungen beinhalten *SSSP*-Lösungen!

Beispiel: U-Bahn München mittels Dijkstra



Beispiel: U-Bahn München mittels Dijkstra




Demo

Heuristische Suche

- Gleichmäßiges Absuchen des Graphen dauert zu lange!
- Ändere Reihenfolge: Besuche zuerst Knoten, die wahrscheinlich (!) schneller ans Ziel fühen!
- Monotone Heuristik
  - Überschätzt niemals wahren Aufwand, dann beweisbar optimal.
  - Beispiel: Luftlinie, vgl. Dreiecksungleichung!

A*-Suche

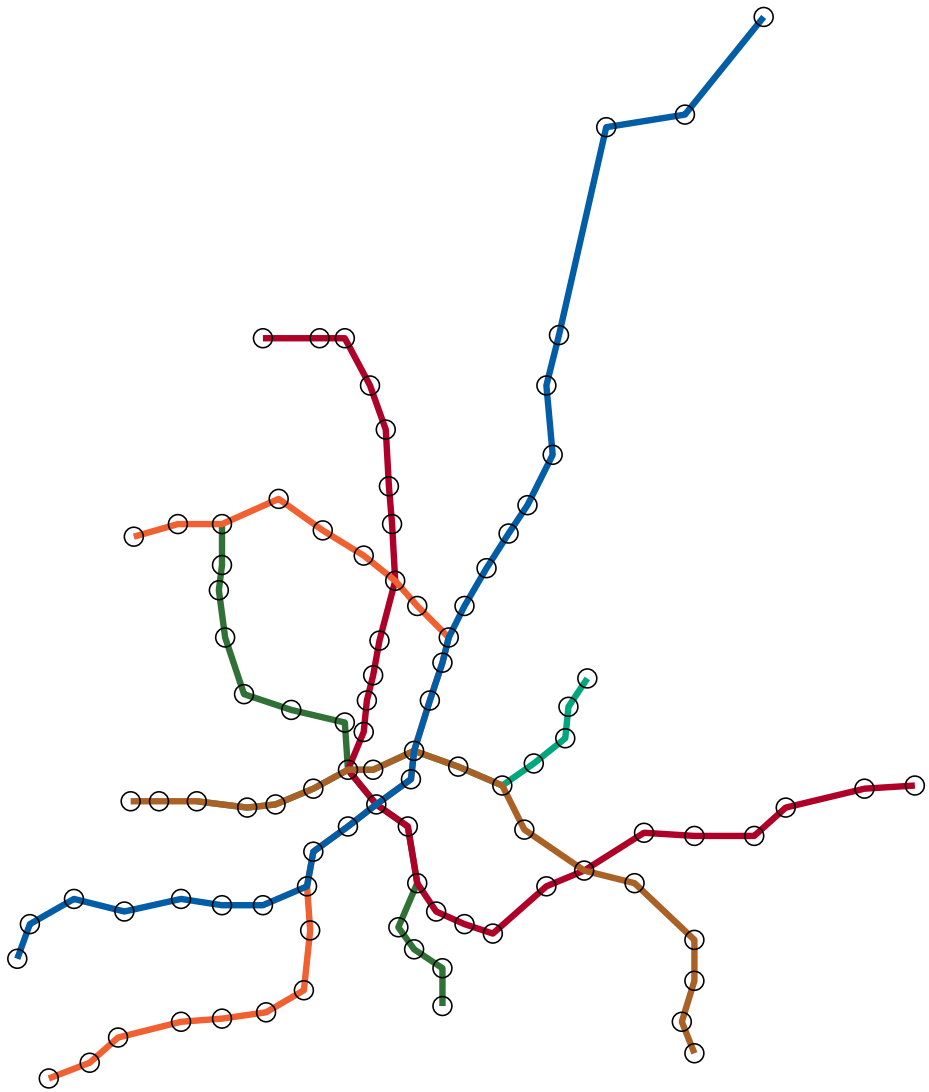
$h = \{||v_1 - v_{\text{stop}}||, ..., ||v_m - v_{\text{stop}}||\}$;
$pred = \{\emptyset, ..., \emptyset\}$; $d = \{\infty, ..., \infty\}$; $d[v_{\text{start}}] = 0$;
MinPriorityQueue Q mit Elementen v, Schlüsseln $d + h$;

while ($\neg$isEmpty(Q)) {
    v = extractMin(Q);

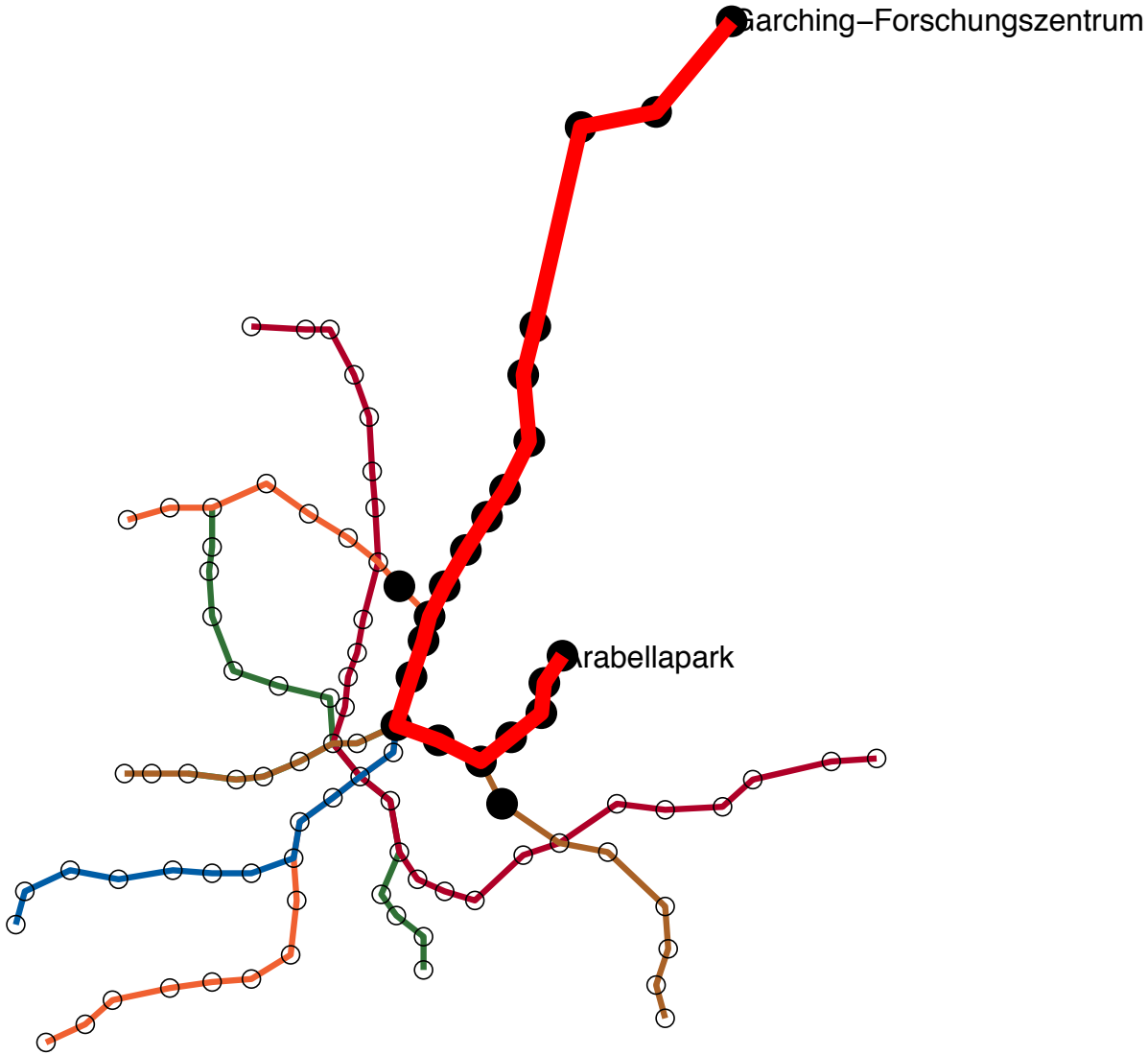
    // Ankunft am Zielknoten bei SPSP
    if ($v = v_{\text{stop}}$) {
        return;
    }

    // Für jeden unbesuchten Nachbarn n
    foreach (Knoten $n : \{v, n\} \in E \wedge n \in Q$) {
        // Günstigeren Pfad nach n über v gefunden
        if ($d[v] + w[\{v, n\}] < d[n]$) {
            $pred[n] = v$; $d[n] = d[v] + w[\{v, n\}]$;
            decreaseKey($Q, n, d[n] + h[n]$);
        }
    }
}

Beispiel: U-Bahn München mittels A*



Beispiel: U-Bahn München mittels A*




Demo