Algorithmen und Datenstrukturen

Vorlesung #11 – Heuristische und Approximative Algorithmen

Benjamin Blankertz

Lehrstuhl für Neurotechnologie, TU Berlin

benjamin.blankertz@tu-berlin.de

26 · Jun · 2019

Themen der heutigen Vorlesung

- Heuristische Algorithmen
  - *Best-First* Algorithmus für kürzeste Pfade von s nach t
  - A* Algorithmus für kürzeste Pfade von s nach t
  - A* in generellem Kontext
  - Heuristische Lösung für das *Travelling Salesman Problem* (TSP)
- Approximative Algorithmen
  - Metrisches TSP: gut, aber nicht beliebig gut approximierbar
  - Allgemeines TSP: nicht approximierbar
  - 0/1-Rucksackproblem: beliebig gut approximierbar
  - Approximationsschema

Heuristische Algorithmen

- **Heuristiken** sind problem-spezifische Informationen, die es erlauben eine Lösungssuche für eine bestimmte Problemklasse zielgerichteter durchzuführen, als z.B. durch eine uninformierte Durchsuchung des Lösungsraums.
- **Heuristische Algorithmen** finden eine (möglichst) optimale Lösung in einem (meist exponentiell) großen Lösungsraum unter Verwendung einer Heuristik.

Heuristische Algorithmen

- **Heuristiken** sind problem-spezifische Informationen, die es erlauben eine Lösungssuche für eine bestimmte Problemklasse zielgerichteter durchzuführen, als z.B. durch eine uninformierte Durchsuchung des Lösungsraums.
- **Heuristische Algorithmen** finden eine (möglichst) optimale Lösung in einem (meist exponentiell) großen Lösungsraum unter Verwendung einer Heuristik.
- Oft gibt es keine Garantien für eine schnellere Laufzeit im Vergleich zu herkömmlichen Ansätzen.
- Manchmal gibt es auch keine Garantie, dass eine optimale Lösung oder eine Lösung mit vorgegebener maximaler Abweichung vom Optimum gefunden wird.
- Dennoch sind einige heuristische Algorithmen in der Praxis sehr nützlich.
- Dies wird dann anhand von systematischen experimentellen Algorithmenanalysen belegt.

Heuristische Algorithmen

- Heuristischen Verfahren folgen meist einem der folgenden Ansätze:

1. Verfahren, die eine Lösung sukzessiv von null einer Heuristik folgend aufbauen
2. Verfahren, die eine schnell hergestellte, suboptimale Lösung schrittweise mit einer Heuristik verbessern

Heuristische Algorithmen

- Heuristischen Verfahren folgen meist einem der folgenden Ansätze:

1. Verfahren, die eine Lösung sukzessiv von null einer Heuristik folgend aufbauen
2. Verfahren, die eine schnell hergestellte, suboptimale Lösung schrittweise mit einer Heuristik verbessern

- In der zweiten Variante kommen oft (nicht optimale) Greedy Algorithmen für die Erstellung einer Ausgangslösung zum Einsatz.

Noch einmal kürzeste Wege

- In vielen Anwendungen stellt sich die Frage nach kürzesten Wegen von einem Startpunkt zu einem Zielpunkt (z.B. Navigation, Robotik, Computerspiele).
- Zur Modellierung können Graphen verwendet werden.
- Dabei geht es auch um Wege auf freien Flächen. Die Knoten sind also nicht unbedingt Kreuzungspunkte, sondern können eine Diskretisierung der Fläche sein.

Noch einmal kürzeste Wege

- In vielen Anwendungen stellt sich die Frage nach kürzesten Wegen von einem Startpunkt zu einem Zielpunkt (z.B. Navigation, Robotik, Computerspiele).
- Zur Modellierung können Graphen verwendet werden.
- Dabei geht es auch um Wege auf freien Flächen. Die Knoten sind also nicht unbedingt Kreuzungspunkte, sondern können eine Diskretisierung der Fläche sein.
- In diesen Fällen werden oft Gitternetze als Graphen verwendet.
- Gleichlange Wegstrecken können unterschiedliche Kosten haben, je nach Begebenheit der Landschaft.
- Da die Graphen sehr groß sein können, ist Effizienz wichtig.
- Aufgabe: Finde möglichst effizient den 'kürzesten Weg' (Weg mit geringsten Kosten) zu gebenem Start- und Zielknoten in einem gewichteten (ggf. gerichteten) Graphen.

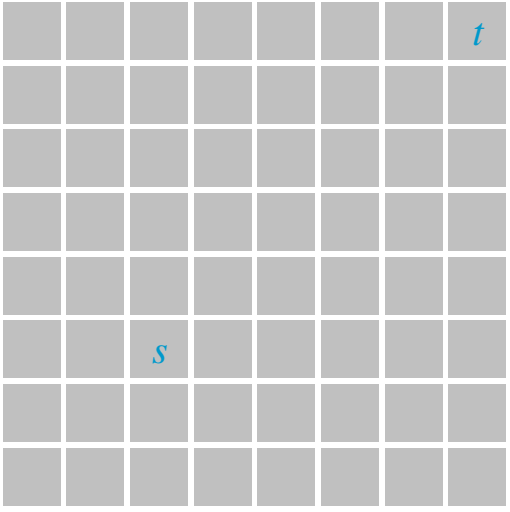
Der kürzeste Weg von s nach t

- **Ziel:** Finde den kürzesten Weg von einem Startknoten s zu einem Zielknoten t in einem gewichteten Graphen.
- Der Dijkstra Algorithmus findet die kürzesten Wege von s zu allen anderen Knoten. Wir können den Algorithmus auch bei gegebenem Ziel benutzen:
- Sobald der Zielknoten aus der Warteschlange kommt, wird die Suche gestoppt.

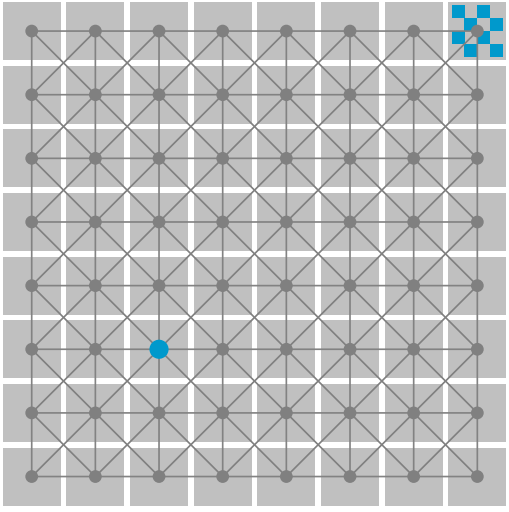
Der kürzeste Weg von s nach t

- **Ziel:** Finde den kürzesten Weg von einem Startknoten s zu einem Zielknoten t in einem gewichteten Graphen.
- Der Dijkstra Algorithmus findet die kürzesten Wege von s zu allen anderen Knoten. Wir können den Algorithmus auch bei gegebenem Ziel benutzen:
- Sobald der Zielknoten aus der Warteschlange kommt, wird die Suche gestoppt.
- Dijkstra baut den Suchbaum nach aufsteigender Länge der kürzesten Pfade auf. Die Suche geht gleichermaßen in alle Richtungen - es gibt es keine Ausrichtung auf den Zielknoten. Das ist bei gegebenem Zielknoten nicht sonderlich effizient.
- Mit einer Heuristik $h(v)$, die den Abstand jedes Knotens zum Ziel schätzt, können wir vielversprechende Knoten zuerst explorieren.
- Für kürzeste Wege auf Landkarten ist z.B. die Luftliniendistanz eine solche Schätzung.

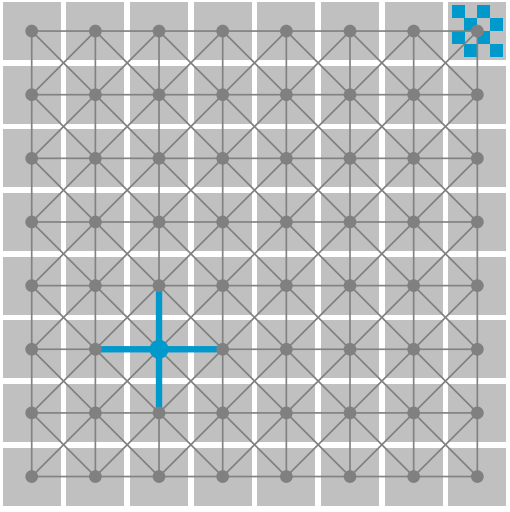
Dijkstra durch Heuristik verbessern



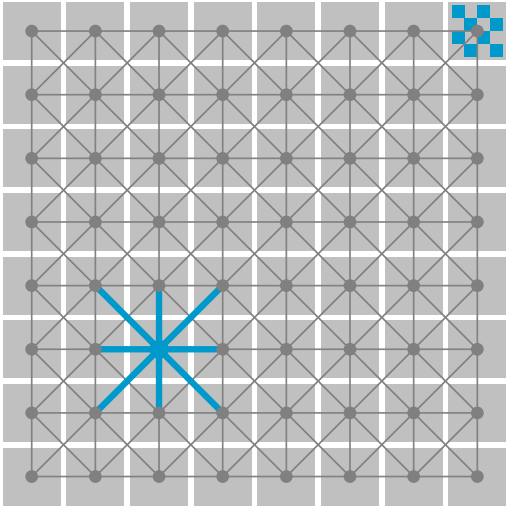
Dijkstra durch Heuristik verbessern



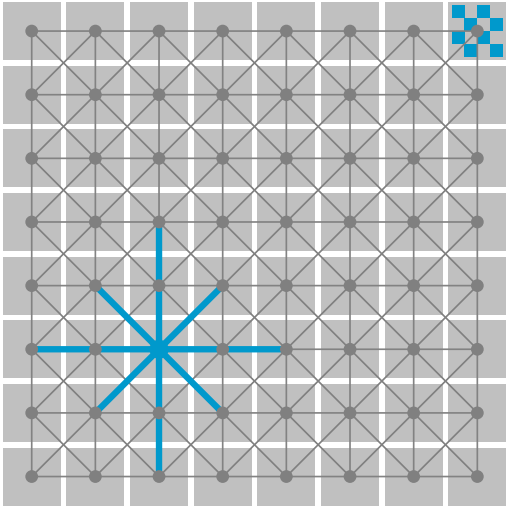
Dijkstra



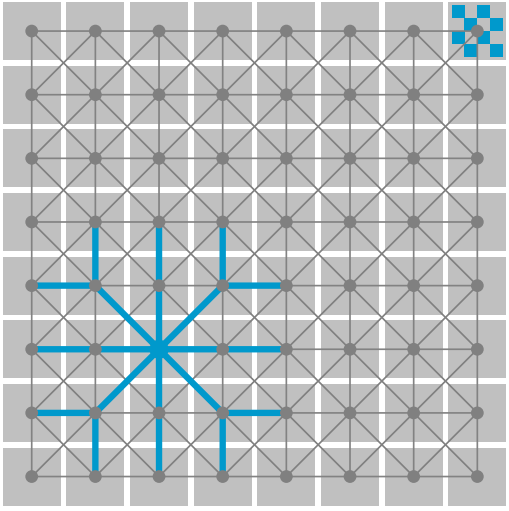
Dijkstra



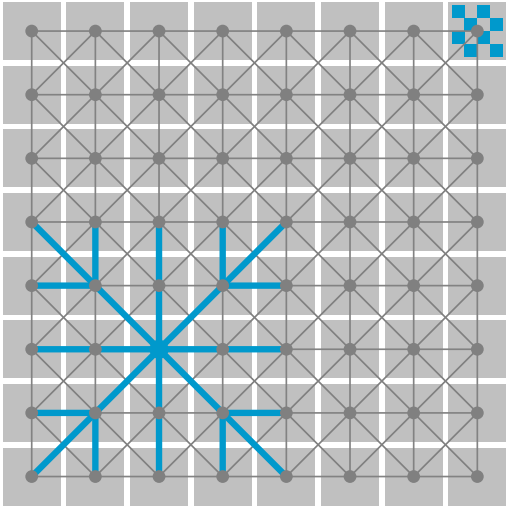
Dijkstra



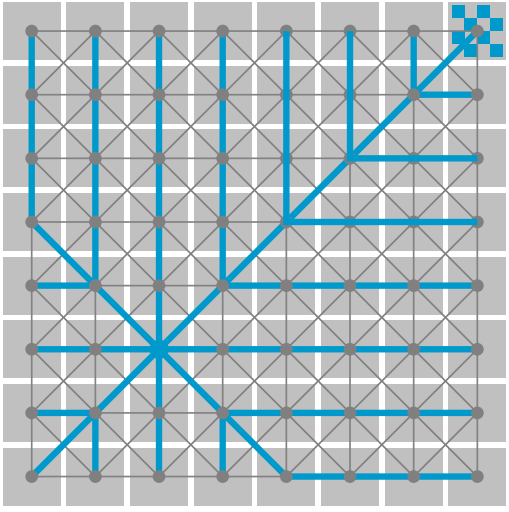
Dijkstra



Dijkstra

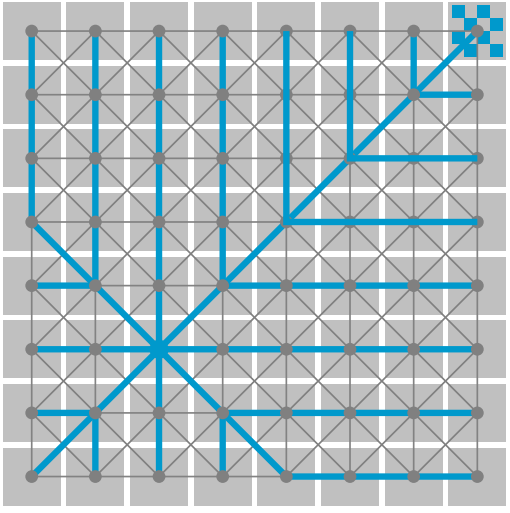


Dijkstra

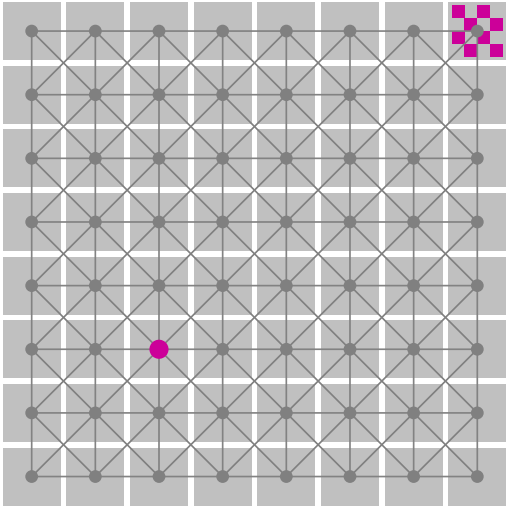


Dijkstra durch Heuristik verbessern

Dijkstra

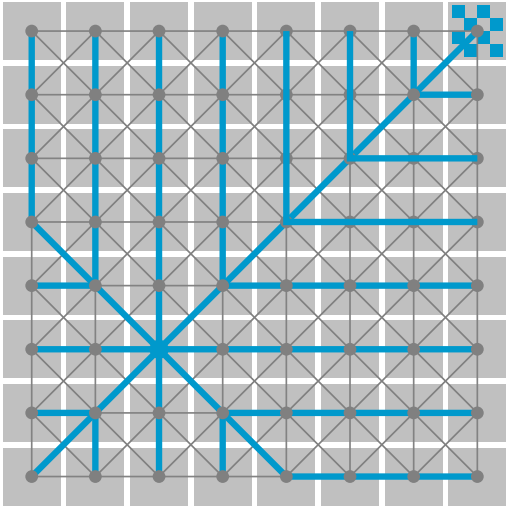


Ideal (mit Heuristik?)

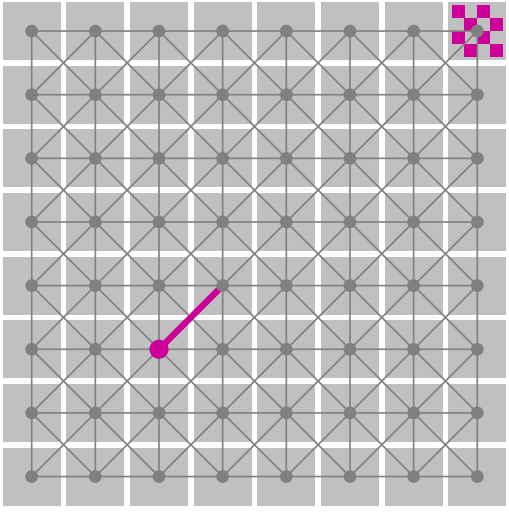


Dijkstra durch Heuristik verbessern

Dijkstra

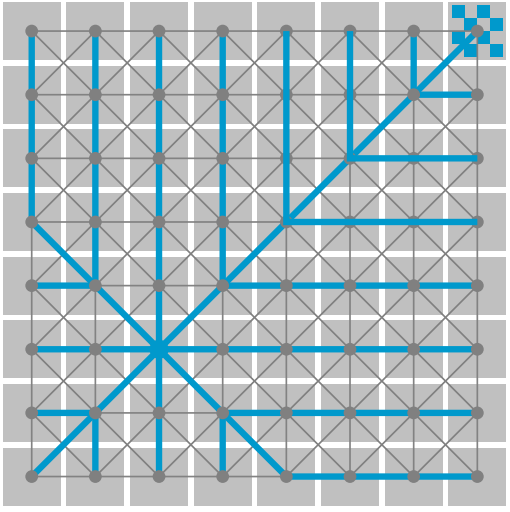


Ideal (mit Heuristik?)

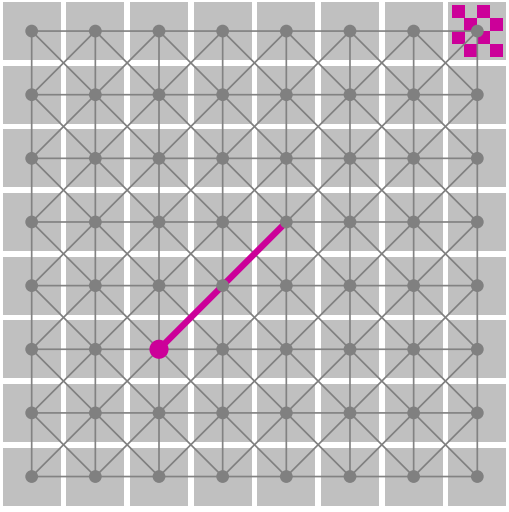


Dijkstra durch Heuristik verbessern

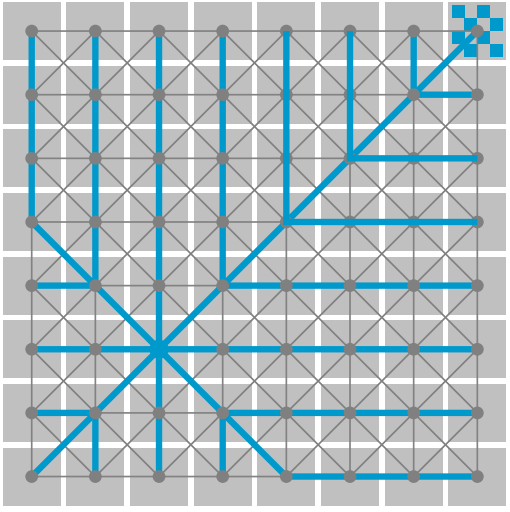
Dijkstra



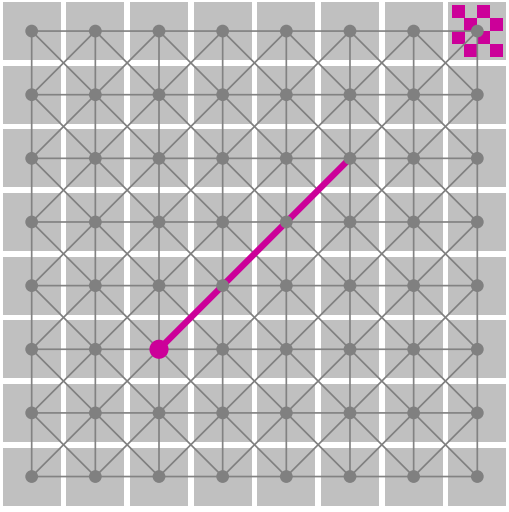
Ideal (mit Heuristik?)



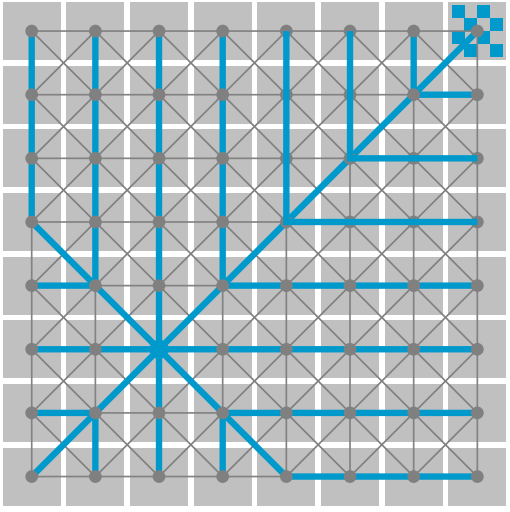
Dijkstra



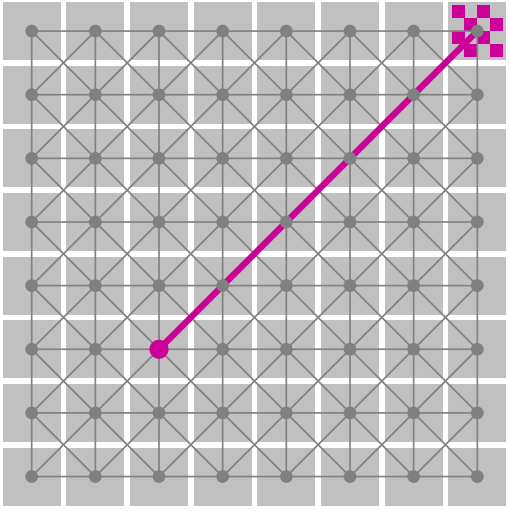
Ideal (mit Heuristik?)



Dijkstra



Ideal (mit Heuristik?)



Der kürzeste Weg von s nach t einer Heuristik folgend

- Der (Greedy) **Best-First Algorithmus** funktioniert ähnlich wie Breitensuche.
- Er basiert auf einer Heuristik, die für jeden Knoten eine Schätzung seines Abstands zum Ziel angibt.
- Ausgehend von s wird immer ein neuer Knoten besucht, der von einem der schon besuchten Knoten über eine Kante erreichbar ist.
- Unter diesen erreichbaren Knoten wird derjenige ausgewählt, dessen Abstand zum Ziel gemäß der Heuristik am kleinsten ist.
- Besuchte Knoten werden später nicht noch einmal besucht (Greedy Auswahl).

Der kürzeste Weg von s nach t einer Heuristik folgend

- Der (Greedy) **Best-First Algorithmus** funktioniert ähnlich wie Breitensuche.
- Er basiert auf einer Heuristik, die für jeden Knoten eine Schätzung seines Abstands zum Ziel angibt.
- Ausgehend von s wird immer ein neuer Knoten besucht, der von einem der schon besuchten Knoten über eine Kante erreichbar ist.
- Unter diesen erreichbaren Knoten wird derjenige ausgewählt, dessen Abstand zum Ziel gemäß der Heuristik am kleinsten ist.
- Besuchte Knoten werden später nicht noch einmal besucht (Greedy Auswahl).

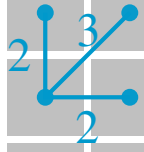
- **Definition:** Für einen Pfad p bezeichnen wir mit $c(p)$ die **Kosten des Pfades** (bzw. die Pfadlänge), also die Summe der Gewichte seiner Kanten.

Pseudocode für den BEST-FIRST Algorithmus

```
1  Q: PQ mit Priorisierung der Knoten durch Heuristik h
2  M: boolean Array der Größe V
3  parent: int Array der Größe V
4
5  add(Q, s, h(s))
6  M ← {s}
7  while Q ≠ ∅
8    v ← poll(Q)                  // v wird besucht
9    if v = t return true         // Pfad von s nach t gefunden
10   for each w with v→w ∈ E and w ∉ M
11     add w to M                 // w wurde entdeckt
12     parent[w] = v
13     add(Q, w, h(w))
14   end                          // v fertig bearbeitet
15 end
16 return false                   // kein Pfad gefunden
```

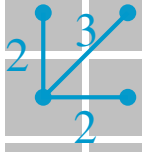
Distanzmaß:

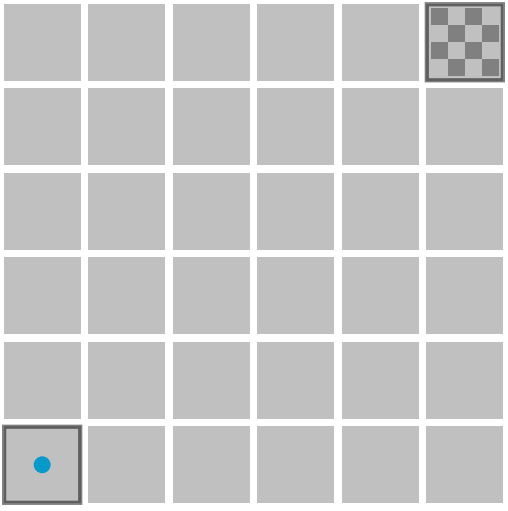
(Gewichte des Graphen)



Distanzmaß:

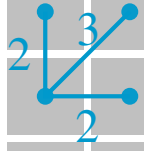
(Gewichte des Graphen)





Distanzmaß:

(Gewichte des Graphen)



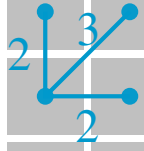
10	8	6	4	2	0
11	9	7	5	3	2
12	10	8	6	5	4
13	11	9	8	7	6
14	12	11	10	9	8
15	14	13	12	11	10

Wir nehmen hier als Heuristik die tatsächliche Distanz zum Ziel im (unblockierten) Gittergraphen.

Best-First in Aktion

Distanzmaß:

(Gewichte des Graphen)

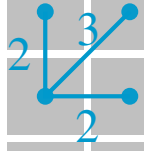


10	8	6	4	2	0
11	9	7	5	3	2
12	10	8	6	5	4
13	11	9	8	7	6
14	12	11	10	9	8
15	14	13	12	11	10

Best-First in Aktion

Distanzmaß:

(Gewichte des Graphen)

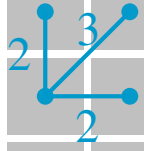


10	8	6	4	2	0
11	9	7	5	3	2
12	10	8	6	5	4
13	11	9	8	7	6
14	12	11	10	9	8
15	14	13	12	11	10

Best-First in Aktion

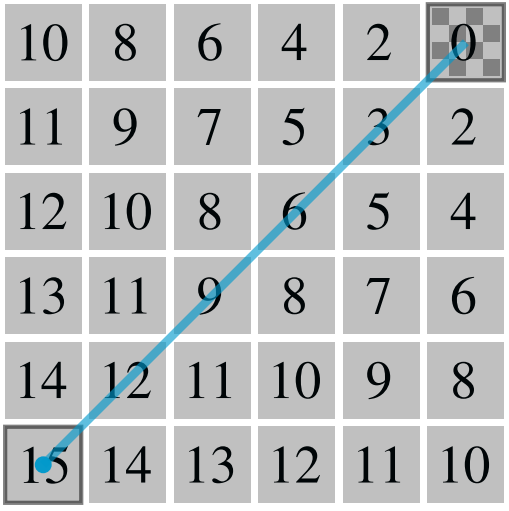
Distanzmaß:

(Gewichte des Graphen)

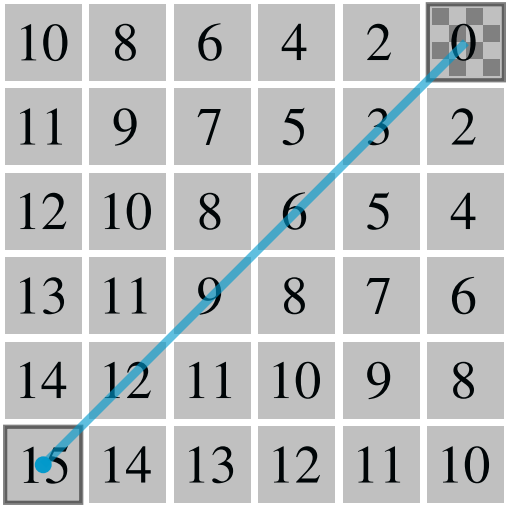


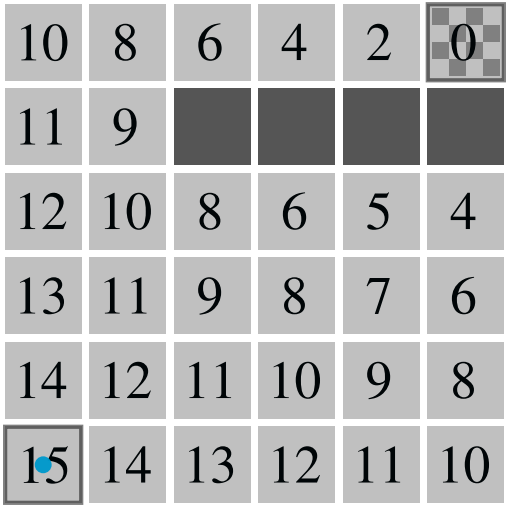
10	8	6	4	2	0
11	9	7	5	3	2
12	10	8	6	5	4
13	11	9	8	7	6
14	12	11	10	9	8
15	14	13	12	11	10

Best-First in Aktion

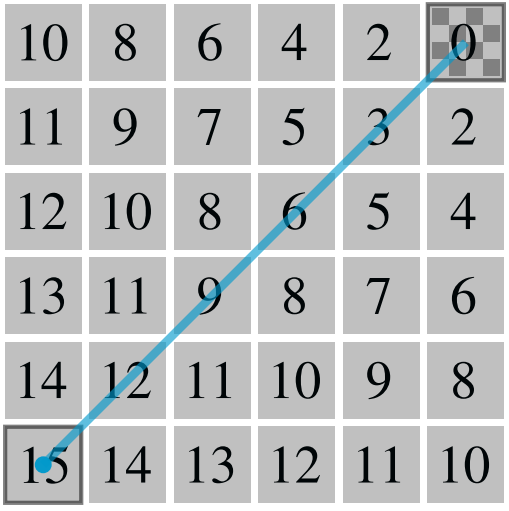


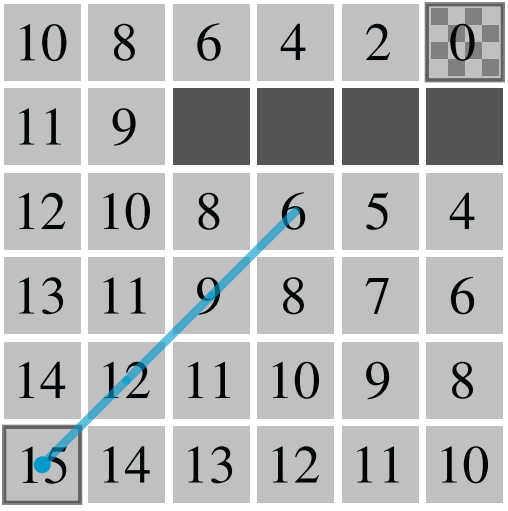
Best-First in Aktion



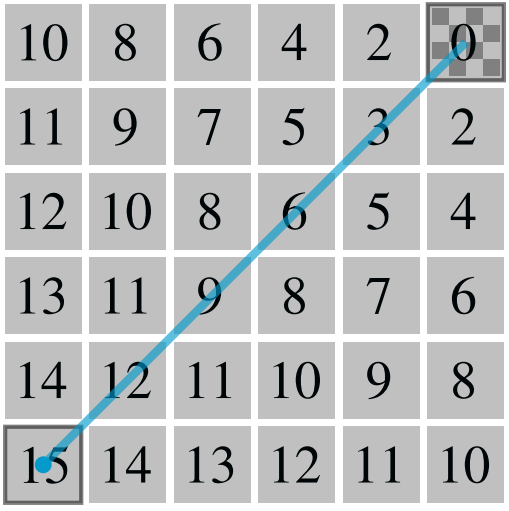


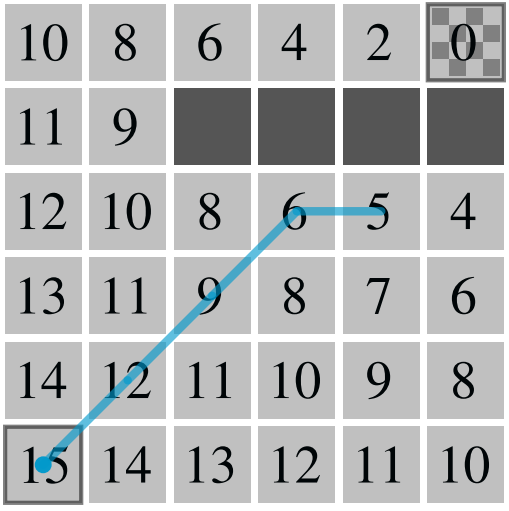
Best-First in Aktion



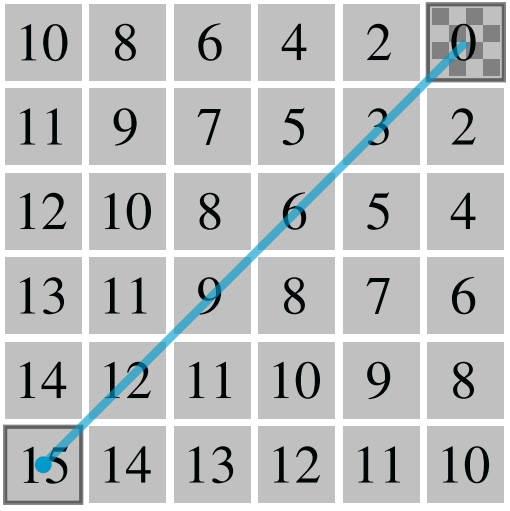


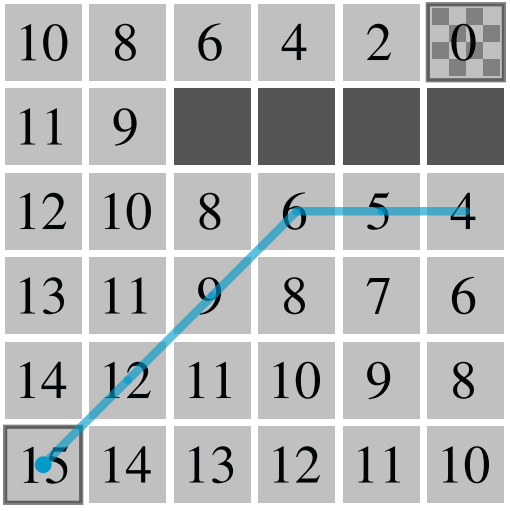
Best-First in Aktion



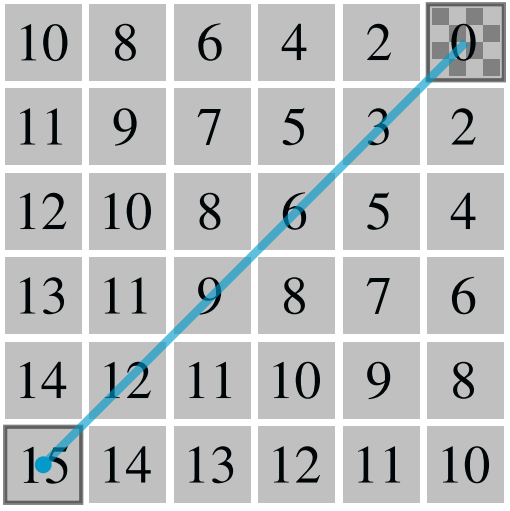


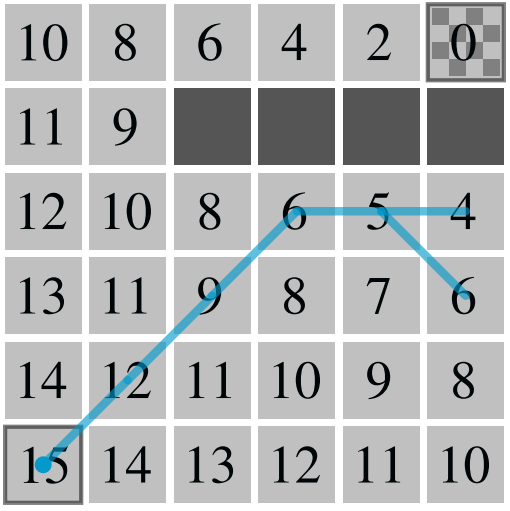
Best-First in Aktion



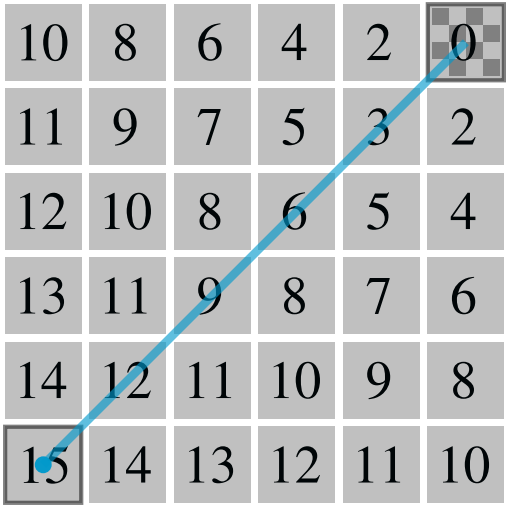


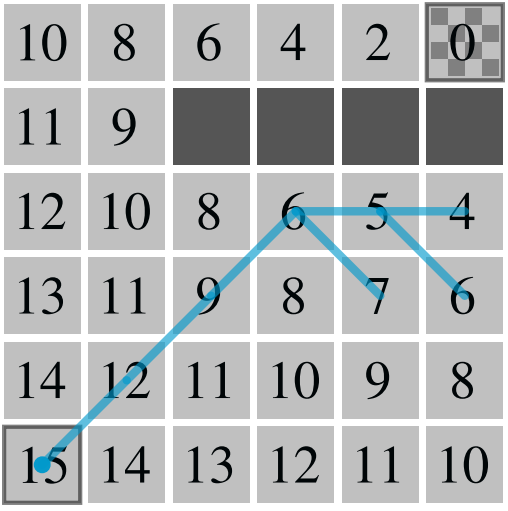
Best-First in Aktion



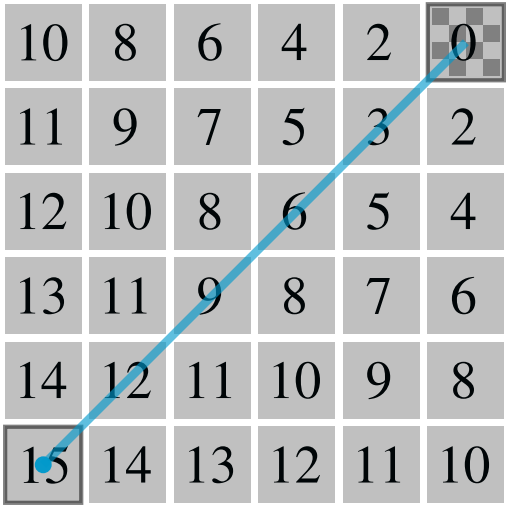


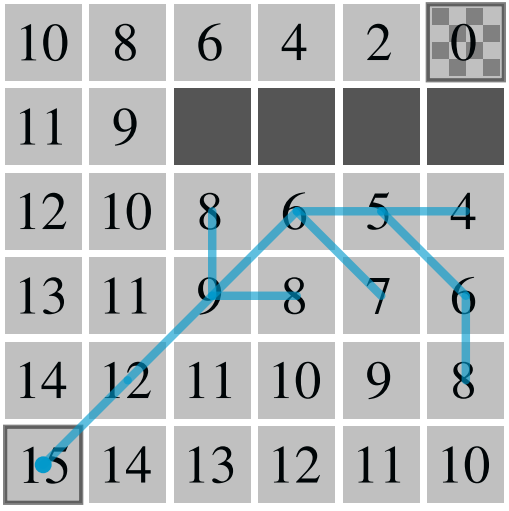
Best-First in Aktion



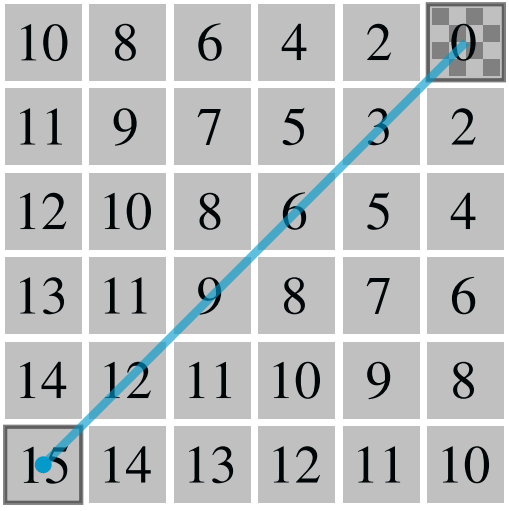


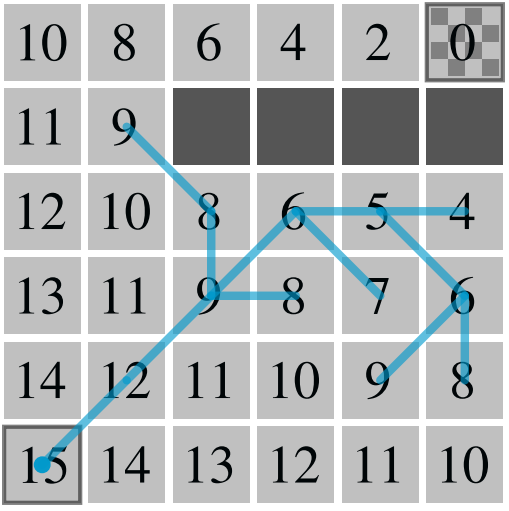
Best-First in Aktion



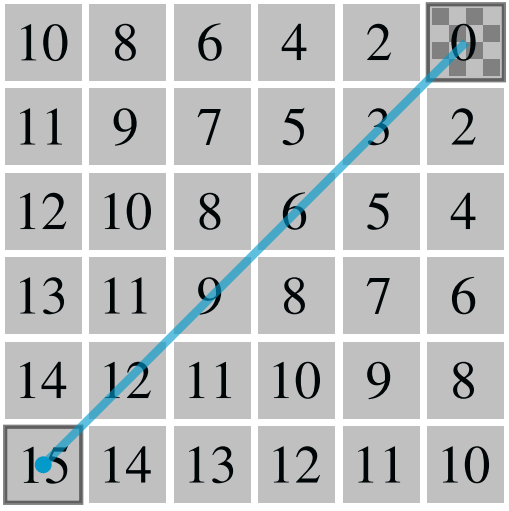


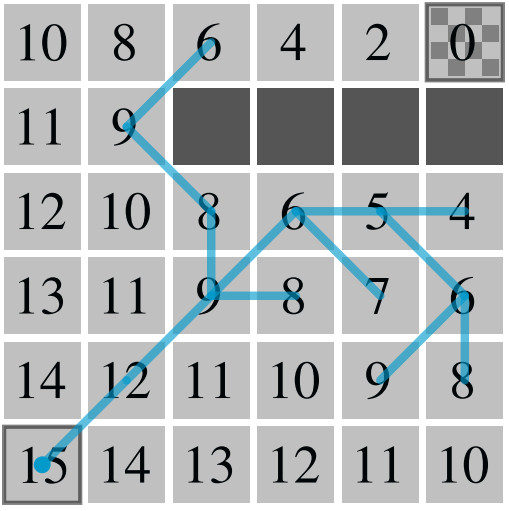
Best-First in Aktion



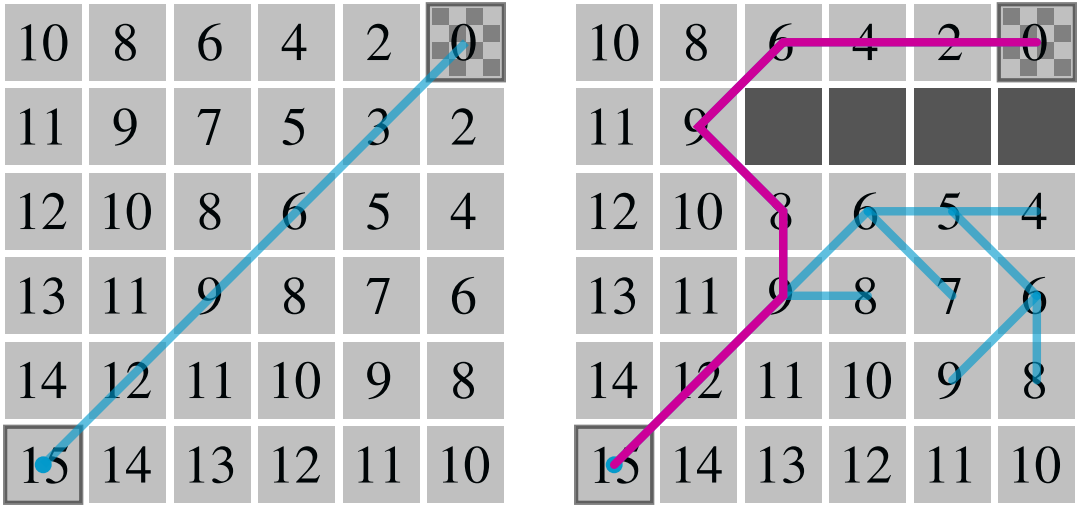


Best-First in Aktion





Best-First in Aktion



- Best-First scheitert bei schwierigeren Beispielen (d.h. der Algorithmus findet nicht die *optimale* Lösung), da er sich zu stark auf die Heuristik verlässt.

Der A* Algorithmus

- Der **A* Algorithmus** ("A-Stern", *A-star*) kombiniert
  - Vorwärtskosten (*forward cost*): von einem Knoten zum Ziel, geschätzt duch die Heuristik $h(v)$ mit den
  - Rückwärtskosten (*backward cost*): vom Start zum einem Knoten, schrittweise beim Programmablauf aktualisiert.
- Genauer: Die Rückwärtskosten entsprechen der Länge des kürzesten bislang bekannten Weges von s nach v, $dist[v]$, die während der Suche wie bei Dijkstra aktualisiert wird.

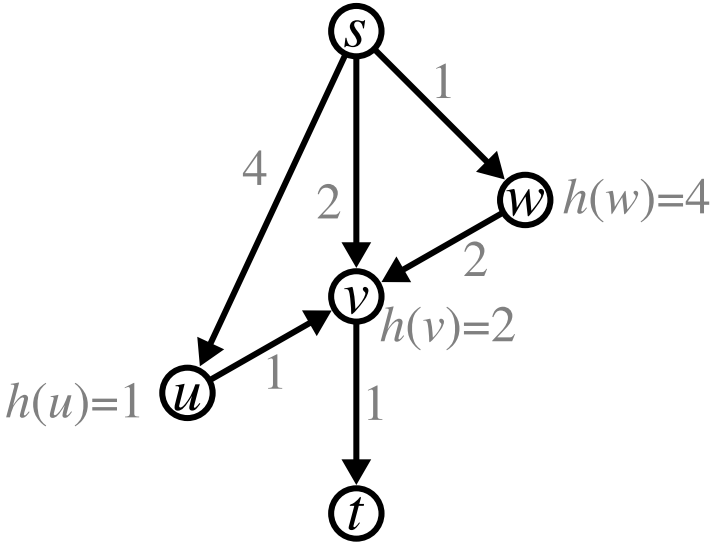
Der A* Algorithmus

- Der **A* Algorithmus** ("A-Stern", *A-star*) kombiniert
  - Vorwärtskosten (*forward cost*): von einem Knoten zum Ziel, geschätzt duch die Heuristik $h(v)$ mit den
  - Rückwärtskosten (*backward cost*): vom Start zum einem Knoten, schrittweise beim Programmablauf aktualisiert.
- Genauer: Die Rückwärtskosten entsprechen der Länge des kürzesten bislang bekannten Weges von s nach v, $dist[v]$, die während der Suche wie bei Dijkstra aktualisiert wird.
- Ausgehend von s wird immer ein neuer Knoten besucht, der von einem der schon besuchten Knoten über eine Kante erreichbar ist.
- Unter diesen erreichbaren Knoten wird derjenige Knoten v ausgewählt, für den die geschätzte Länge des Weges von s nach t über v

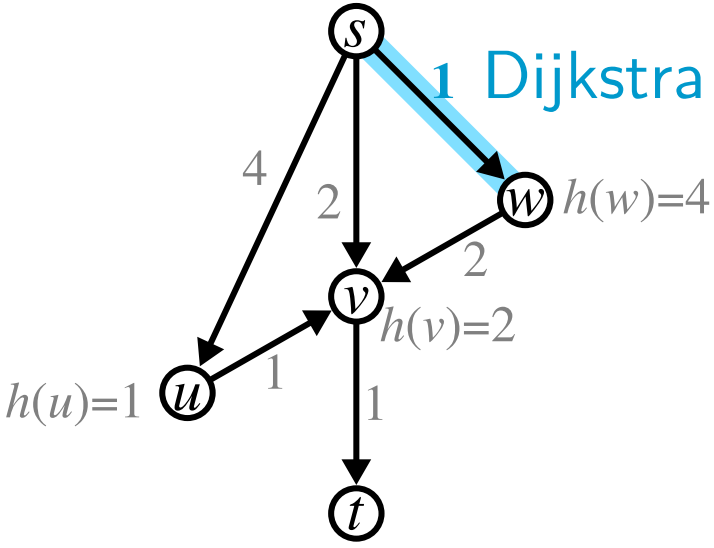
  $$f[v] = dist[v] + h(v)$$

  am kürzesten ist.
- Einige Details des Algorithmus werden auf den nächsten Seiten geklärt.

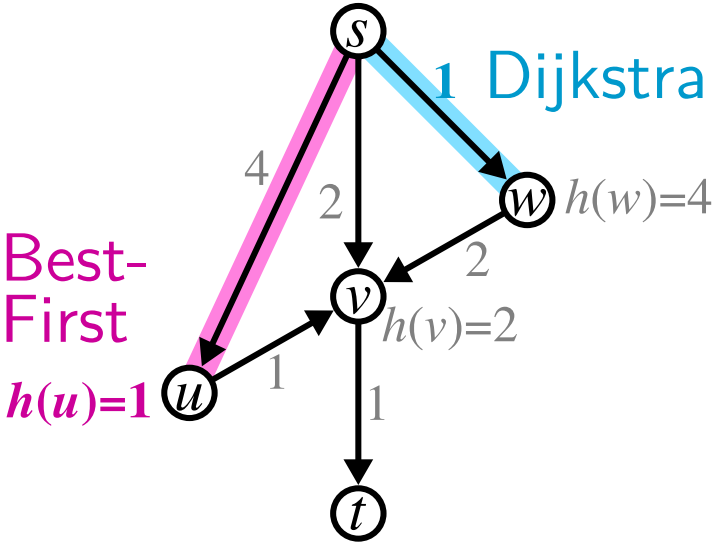
Vergleich der Kantenauswahl zwischen Dijkstra, Best-First und A*



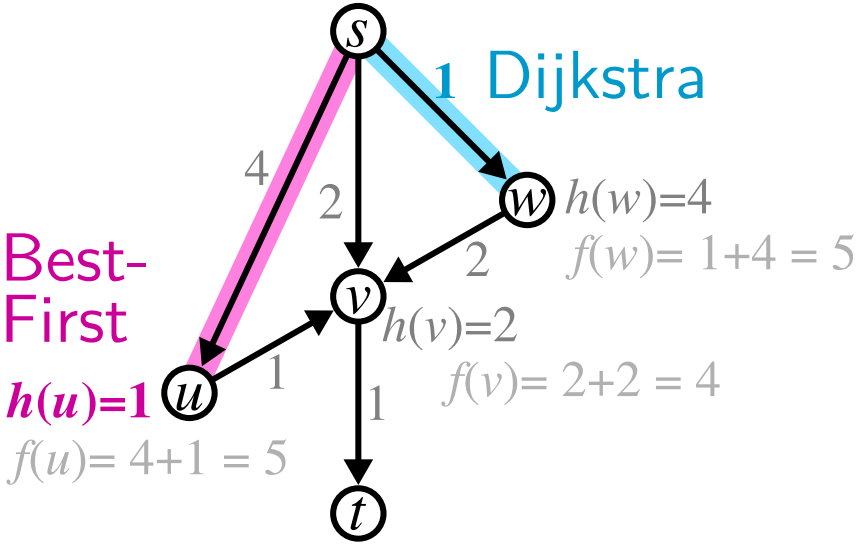
Vergleich der Kantenauswahl zwischen Dijkstra, Best-First und A*



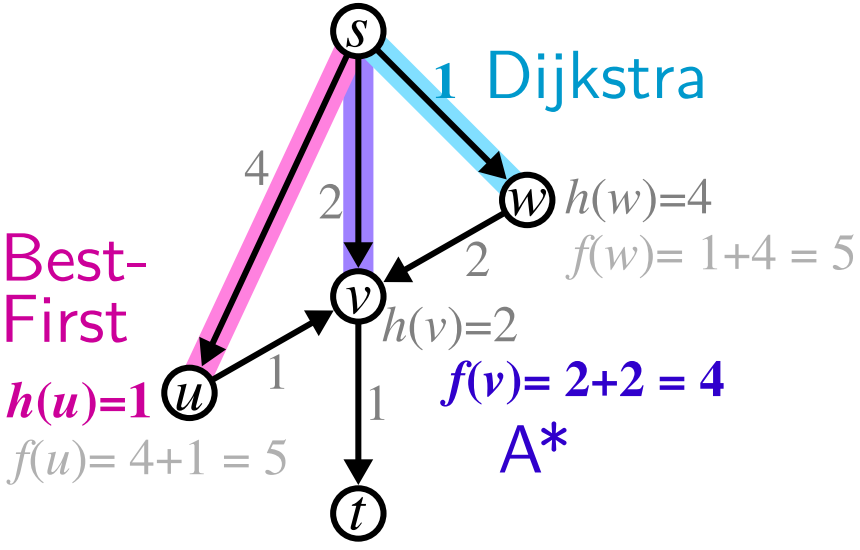
Vergleich der Kantenauswahl zwischen Dijkstra, Best-First und A*



Vergleich der Kantenauswahl zwischen Dijkstra, Best-First und A*



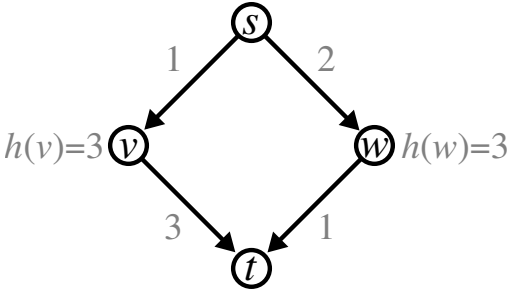
Vergleich der Kantenauswahl zwischen Dijkstra, Best-First und A*



Bemerkung: Dijkstra findet den kürzesten Weg von s nach t später.

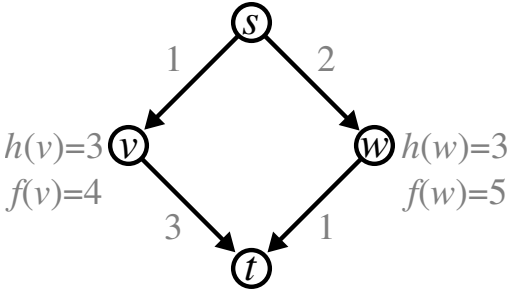
A* konkretisieren – Zulässige Heuristik

- Offensichtlich sind nur bestimme Funktionen als Heuristik hilfreich.
- Im folgenden Beispiel führt eine ungünstige Heuristik die A* Suche in die Irre:



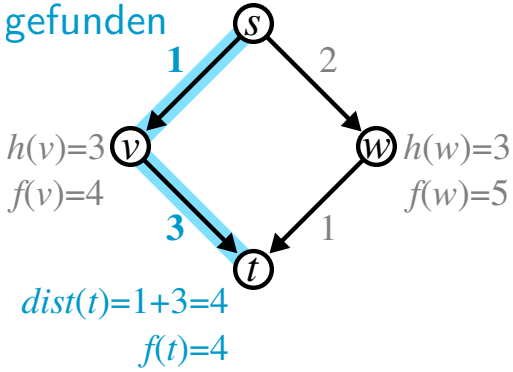
A* konkretisieren – Zulässige Heuristik

- Offensichtlich sind nur bestimme Funktionen als Heuristik hilfreich.
- Im folgenden Beispiel führt eine ungünstige Heuristik die A* Suche in die Irre:



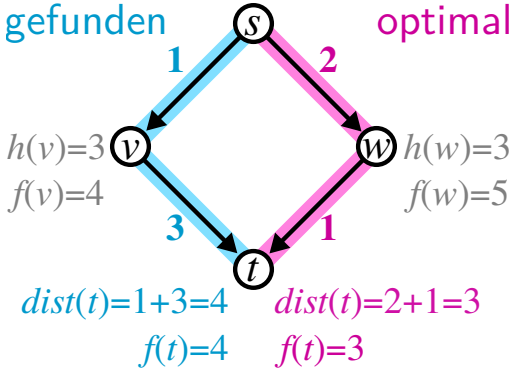
A* konkretisieren – Zulässige Heuristik

- Offensichtlich sind nur bestimme Funktionen als Heuristik hilfreich.
- Im folgenden Beispiel führt eine ungünstige Heuristik die A* Suche in die Irre:



A* konkretisieren – Zulässige Heuristik

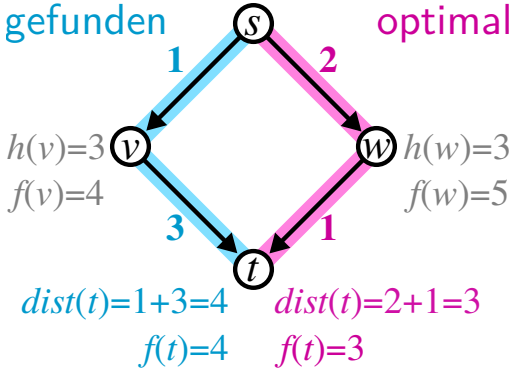
- Offensichtlich sind nur bestimme Funktionen als Heuristik hilfreich.
- Im folgenden Beispiel führt eine ungünstige Heuristik die A* Suche in die Irre:



- Wegen $f(t) < f(w)$ wird w nicht besucht und die Suche endet mit dem Pfad $s - v - t$.
- Problem: zu hoher h-Wert von w.

A* konkretisieren – Zulässige Heuristik

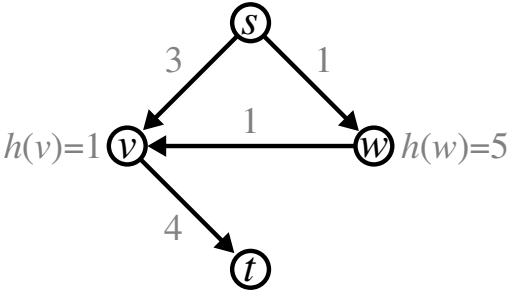
- Offensichtlich sind nur bestimme Funktionen als Heuristik hilfreich.
- Im folgenden Beispiel führt eine ungünstige Heuristik die A* Suche in die Irre:



- Wegen $f(t) < f(w)$ wird w nicht besucht und die Suche endet mit dem Pfad $s - v - t$.
- Problem: zu hoher h-Wert von w.
- Wir nennen eine Heuristik h **zulässig**, wenn $h(w)$ die Weglänge von w nach t nicht überschätzt:
- Für jeden Pfad p von w nach t muss also $h(w) \leq c(p)$ gelten.

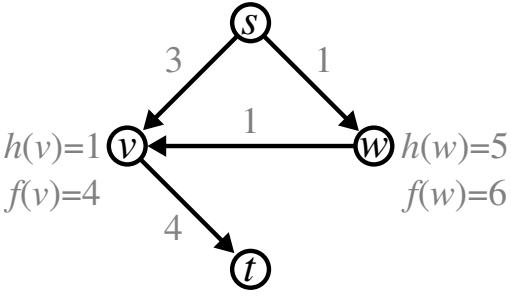
Überlegungen zur Effizienz von A*

- A* soll ein effizienter Algorithmus sein. Daher soll jede Kante wie bei Dijkstra nur einmal betrachtet werden.
- Wenn ein Knoten der PQ entnommen, also dem Suchbaum hinzugefügt wurde, darf er nicht wieder eingefügt werden.
- Genau dies kann aber selbst bei zulässigen Heuristiken erforderlich sein:



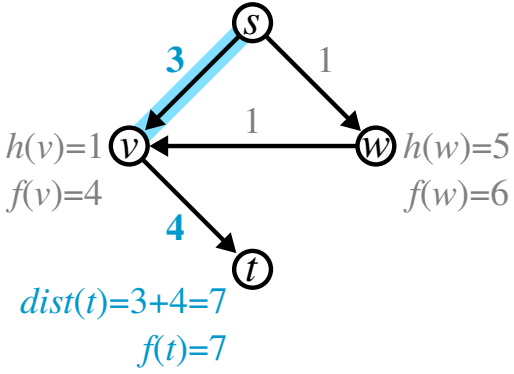
Überlegungen zur Effizienz von A*

- A* soll ein effizienter Algorithmus sein. Daher soll jede Kante wie bei Dijkstra nur einmal betrachtet werden.
- Wenn ein Knoten der PQ entnommen, also dem Suchbaum hinzugefügt wurde, darf er nicht wieder eingefügt werden.
- Genau dies kann aber selbst bei zulässigen Heuristiken erforderlich sein:



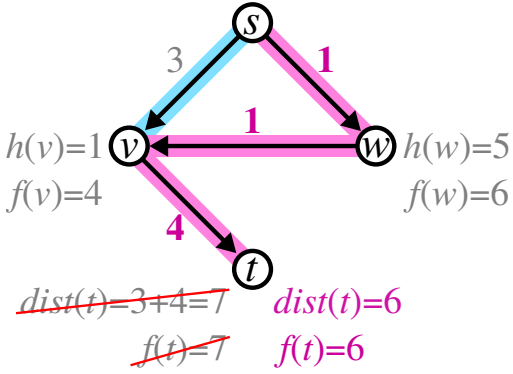
Überlegungen zur Effizienz von A*

- A* soll ein effizienter Algorithmus sein. Daher soll jede Kante wie bei Dijkstra nur einmal betrachtet werden.
- Wenn ein Knoten der PQ entnommen, also dem Suchbaum hinzugefügt wurde, darf er nicht wieder eingefügt werden.
- Genau dies kann aber selbst bei zulässigen Heuristiken erforderlich sein:



Überlegungen zur Effizienz von A*

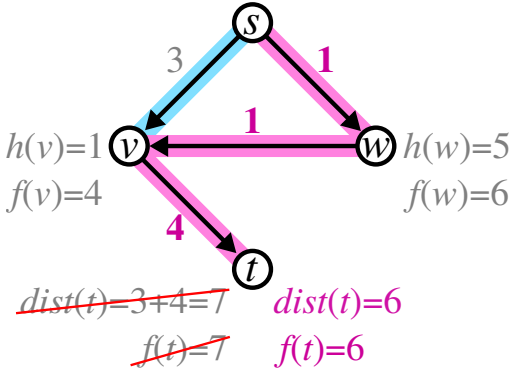
- A* soll ein effizienter Algorithmus sein. Daher soll jede Kante wie bei Dijkstra nur einmal betrachtet werden.
- Wenn ein Knoten der PQ entnommen, also dem Suchbaum hinzugefügt wurde, darf er nicht wieder eingefügt werden.
- Genau dies kann aber selbst bei zulässigen Heuristiken erforderlich sein:



- v muss zweimal besucht werden.

Überlegungen zur Effizienz von A*

- A* soll ein effizienter Algorithmus sein. Daher soll jede Kante wie bei Dijkstra nur einmal betrachtet werden.
- Wenn ein Knoten der PQ entnommen, also dem Suchbaum hinzugefügt wurde, darf er nicht wieder eingefügt werden.
- Genau dies kann aber selbst bei zulässigen Heuristiken erforderlich sein:



- v muss zweimal besucht werden.
- Problem: Der h-Wert von w ist viel höher als bei v, obwohl sie nur eine Einheit voneinander entfernt sind.
- Um eine garantiert effiziente Laufzeit zu erzielen, können konsistente Heuristiken (siehe nächste Seite) vorausgesetzt werden.

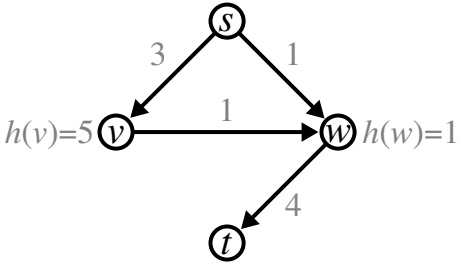
Konsistente Heuristik

- Wir nennen eine Heuristik h **konsistent** (*consistent*) oder monoton, wenn $h(t) = 0$ gilt und h die Dreiecksungleichung erfüllt, also

$$h(v) \leq weight(v \to w) + h(w)$$

für alle Knoten v und w mit $v \to w \in \boldsymbol{E}$ gilt.

Nicht konsistente Heuristik



Konsistente Heuristik

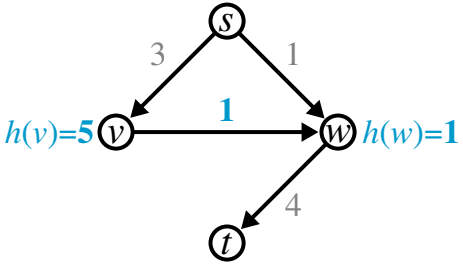
Konsistente Heuristik

- Wir nennen eine Heuristik h **konsistent** (*consistent*) oder monoton, wenn $h(t) = 0$ gilt und h die Dreiecksungleichung erfüllt, also

$$h(v) \leq weight(v \to w) + h(w)$$

für alle Knoten v und w mit $v \to w \in \boldsymbol{E}$ gilt.

Nicht konsistente Heuristik



Konsistente Heuristik

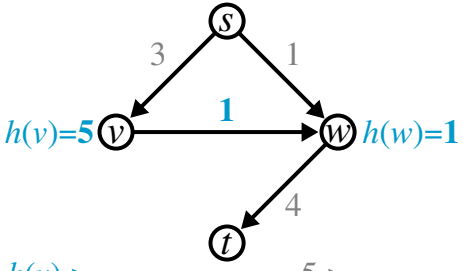
Konsistente Heuristik

- Wir nennen eine Heuristik h **konsistent** (*consistent*) oder monoton, wenn $h(t) = 0$ gilt und h die Dreiecksungleichung erfüllt, also

$$h(v) \leq weight(v \to w) + h(w)$$

für alle Knoten v und w mit $v \to w \in \boldsymbol{E}$ gilt.

Nicht konsistente Heuristik



Konsistente Heuristik

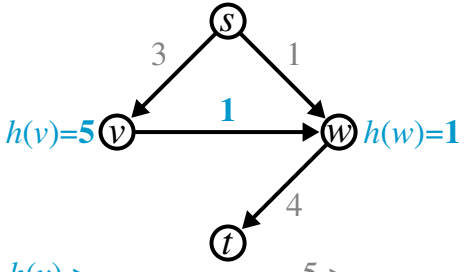
Konsistente Heuristik

- Wir nennen eine Heuristik h **konsistent** (*consistent*) oder monoton, wenn $h(t) = 0$ gilt und h die Dreiecksungleichung erfüllt, also

$$h(v) \leq weight(v \to w) + h(w)$$

für alle Knoten v und w mit $v \to w \in \boldsymbol{E}$ gilt.

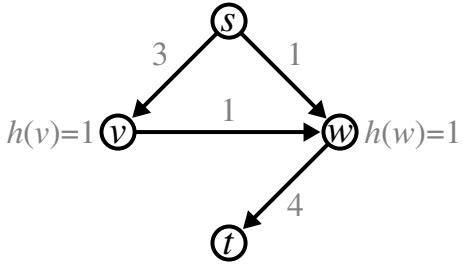
Nicht konsistente Heuristik



$h(v) >$ $\quad 5 >$

$h(w) + \text{weight}(v,w) \quad 1 + 1$

Konsistente Heuristik



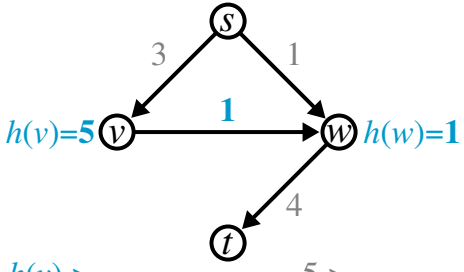
Konsistente Heuristik

- Wir nennen eine Heuristik h **konsistent** (*consistent*) oder monoton, wenn $h(t) = 0$ gilt und h die Dreiecksungleichung erfüllt, also

$$h(v) \leq weight(v \to w) + h(w)$$

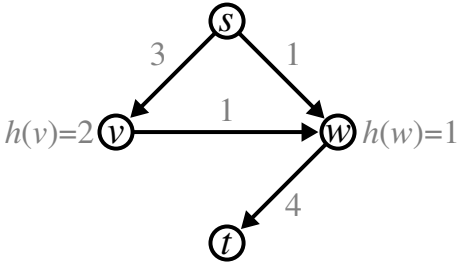
für alle Knoten v und w mit $v \to w \in \boldsymbol{E}$ gilt.

Nicht konsistente Heuristik



$h(v) >$ $5 >$
$h(w) + \text{weight}(v,w)$ $1 + 1$

Konsistente Heuristik



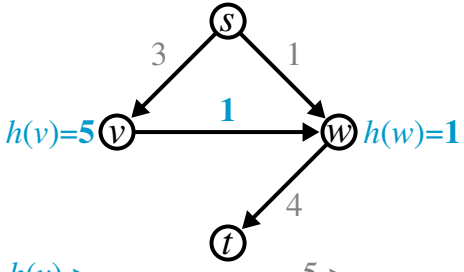
Konsistente Heuristik

- Wir nennen eine Heuristik h **konsistent** (*consistent*) oder monoton, wenn $h(t) = 0$ gilt und h die Dreiecksungleichung erfüllt, also

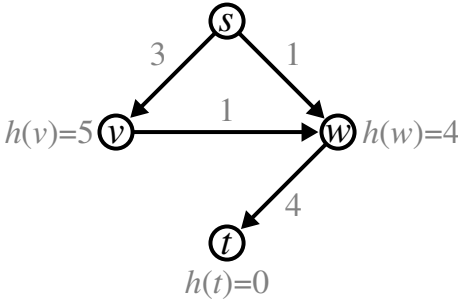
$$h(v) \leq weight(v \to w) + h(w)$$

für alle Knoten v und w mit $v \to w \in \boldsymbol{E}$ gilt.

Nicht konsistente Heuristik



Konsistente Heuristik



Aus Konsistenz folgt Zulässigkeit

Wenn eine Heuristik h konsistent ist, dann ist sie auch zulässig.

Beweis.

- Wir bezeichnen die Länge bzw. die Kosten (*cost*) eines Pfades p mit $c(p)$.
- Es ist zu zeigen, dass der Wert der Heuristik an einem Knoten v_0 nicht größer ist, als die Länge eines Pfades von v_0 zum Zielknoten t.

Aus Konsistenz folgt Zulässigkeit

Wenn eine Heuristik h konsistent ist, dann ist sie auch zulässig.

Beweis.

- Wir bezeichnen die Länge bzw. die Kosten (*cost*) eines Pfades p mit $c(p)$.
- Es ist zu zeigen, dass der Wert der Heuristik an einem Knoten v_0 nicht größer ist, als die Länge eines Pfades von v_0 zum Zielknoten t.
- Sei also ein Pfad $p = v_0, \ldots, v_k$ mit $v_k = t$ gegeben und wir zeigen $h(v_0) \leq c(p)$:

$$
\begin{aligned}
h(v_0) &\leq weight(v_0 \to v_1) + h(v_1) && \text{Def. von Konsistenz} \\
&\leq weight(v_0 \to v_1) + weight(v_1 \to v_2) + h(v_2) && \text{Def. von Konsistenz} \\
&\leq \sum_{i=0}^{k-1} weight(v_i \to v_{i+1}) + h(v_k) && \text{Def. von Konsistenz} \\
&= c(p) && \text{Def. von } c \text{ und } h(v_k) = h(t) = 0
\end{aligned}
$$

Monotonie von konsistenten Heuristiken

Geschätzte Pfadlängen sind bei konsistenter Heuristik monoton steigend

Bei Verwendung einer konsistenten Heuristik wird die geschätzte Gesamtpfadlänge zum Zielknoten durch Erweiterung eines Pfades nie kürzer.

- **Beweis.** Sei p ein Pfad von s zu einem Knoten v.
- Zu zeigen: Die geschätzte Pfadlänge nach t kann nicht kleiner werden, wenn der nächste Schritt im Pfad festgelegt wird.
- Wir betrachten $p + w$, also den Pfad p verlängert um die Kante $v \to w$.

Monotonie von konsistenten Heuristiken

Geschätzte Pfadlängen sind bei konsistenter Heuristik monoton steigend

Bei Verwendung einer konsistenten Heuristik wird die geschätzte Gesamtpfadlänge zum Zielknoten durch Erweiterung eines Pfades nie kürzer.

- **Beweis.** Sei p ein Pfad von s zu einem Knoten v.
- Zu zeigen: Die geschätzte Pfadlänge nach t kann nicht kleiner werden, wenn der nächste Schritt im Pfad festgelegt wird.
- Wir betrachten $p + w$, also den Pfad p verlängert um die Kante $v \to w$.
- Die geschätzte Pfadlänge für p ist $c(p) + h(v)$ und für den verlängerten Pfad $c(p + w) + h(w)$.
- Aus der Konsistenz von h folgt:

$$\begin{aligned} c(p) + h(v) &\leq c(p) + weight(v \to w) + h(w) && \text{Konsistenz von } h \\ &= c(p + w) + h(w) && \text{Definition von } c \end{aligned}$$ □

Anforderungen an die Heuristik in A*

- Wir setzen im Folgenden voraus, dass die Heuristik konsistent ist.
- In diesem Fall muss kein Knoten mehrfach besucht werden (analog zu dem Fall nicht-negativer Gewichte bei Dijkstra).
- Viele in der Praxis benutzten Heuristikfunktionen sind konsistent, wie z.B. die Luftlinien-Distanz für Wege auf Landkarten.

Anforderungen an die Heuristik in A*

- Wir setzen im Folgenden voraus, dass die Heuristik konsistent ist.
- In diesem Fall muss kein Knoten mehrfach besucht werden (analog zu dem Fall nicht-negativer Gewichte bei Dijkstra).
- Viele in der Praxis benutzten Heuristikfunktionen sind konsistent, wie z.B. die Luftlinien-Distanz für Wege auf Landkarten.
- Bei Verwendung einer konsistenten Heuristik funktioniert A* wie Dijkstra, mit dem Unterschied, dass die Knoten in der Reihenfolge der f Schätzung exploriert werden.

Pseudocode für den A* Algorithmus

Die Implementierung von A* ist der des Dijkstra Algorithmus sehr ähnlich. Der einzige Unterschied ist, dass bei A* zur Priorisierung bei der Knotenwahl `dist[v] + h(v)` an Stelle von `dist[v]` verwendet wird.

```
1  Q : IndexMinPQ
2  for each node v
3     dist[v] ← ∞
4  end
5  dist[s] ← 0
6  add(Q, s, h(s))          // f(s) = 0 + h(s)
7  while Q ≠ ∅
8     v ← poll(Q)
9     if v = t
10       return dist[t]
11    end
12    for each w with v→w ∈ E
13       relaxAStar(v→w)
14    end
15 end
16 return inf               // no path from s to t
```

```
17 procedure relaxAStar(v→w)
18  if dist[w] > dist[v] + weight(v→w)
19    parent[w] = v
20    dist[w] = dist[v] + weight(v→w)
21    if contains(Q, w)
22      decreaseKey(Q, w, dist[w] + h(w))
23    else
24      add(Q, w, dist[w] + h(w))
25    end
26 end
```

Korrektheit und Laufzeit von A*

Korrektheit und Laufzeit von A* mit konsistenter Heuristik

Der A* Algorithmus findet bei Verwendung einer konsistenten Heuristik den kürzesten Weg zwischen einem gegeben Start- und Zielknoten in einer Laufzeit in $\boldsymbol{O}(E_0 \log V_0)$. Dabei ist V_0 die Anzahl der besuchten Knoten und E_0 die Anzahl der relaxierten Kanten.

- Vor dem Beweis diskutieren wir die Laufzeit.
- Wir haben im Prinzip dieselbe Laufzeit wie bei Dijkstra. Hier heben wir die Abhängigkeit von den tatsächlich besuchten Knoten und den tatsächlich relaxierten Kanten hervor.
- Durch die Heuristik ist in vielen Anwendungsfällen $V_0 \ll V$ und $E_0 \ll E$, also A* deutlich schneller als Dijkstra. Aber dafür gibt es keine Garantie. Es gibt auch Fälle in den $V_0 \approx V$ und $E_0 \approx E$ gilt und A* keinen (großen) Vorteil bringt.

Laufzeit von A*

Beweis der Laufzeit (impliziert Terminierung)

- Mit konsistenten Heuristiken sind wir nach der Monotonie-Eigenschaft, siehe Seite 16, in Hinblick auf Laufzeit in derselben Situation wie bei Dijkstra:
- Die Monotonie der $dist$-Werte (Dijkstra) überträgt sich wegen $f[v] = dist[v] + h(v)$ auf die f-Werte, da sich $h(v)$ beim Programmablauf nicht ändert.
- Die f-Werte, über die v in Zeile 8 ausgewählt wird, sind also monoton steigend.
- Die *while*-Schleife wird höchstens V_0-mal ausgeführt.
- Jede Kante wird daher nur einmal relaxiert.
- Damit erhalten wir die Laufzeit in $\boldsymbol{O}(E_0 \log V_0)$ wie bei Dijkstra.

Korrektheit von A* – Hilfssatz

Wir zeigen folgenden Hilfssatz:

- **Lemma:** Sei p ein Pfad $s \rightsquigarrow v$ dessen Knoten fertig bearbeitet sind, mit der Ausnahme, dass v auch in der PQ sein darf (also noch auf die Bearbeitung wartet). In diesem Fall gilt: $dist[v] \leq c(p)$.

- Beweis durch Induktion nach der Anzahl der Kanten von p. Sei u der vorletzte Knoten in p und p_0 der Subpfad $s \rightsquigarrow u$ von p.

- Da u fertig bearbeitet ist, gilt für den Nachbarknoten v die Relaxierungsbedingung

$$dist[v] \leq dist[u] + weight(u \to v)$$

- Nach IV gilt $dist[u] \leq c(p_0)$. Damit erhalten wir

$$dist[v] \leq c(p_0) + weight(u \to v) = c(p)$$

- Damit ist das Lemma bewiesen. □

Korrektheit von A*

Beweis der Korrektheit.

- Wenn A* mit einem gefundenen Pfad terminiert, dann ist $f(t)$ kleiner als die f-Werte aller Knoten in der PQ.
- In der PQ sind alle Knoten, die über kreuzende Kanten von dem Bereich der bearbeiteten Knoten erreicht werden können.
- Sei p ein anderer Pfad von s nach t. Dieser Pfad muss durch einen Knoten v laufen, der in der PQ ist. Sei p_0 der Subpfad $s \rightsquigarrow v$ und p_1 der Subpfad $v \rightsquigarrow t$.

Korrektheit von A*

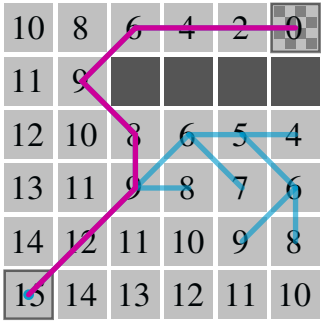
Beweis der Korrektheit.

- Wenn A* mit einem gefundenen Pfad terminiert, dann ist $f(t)$ kleiner als die f-Werte aller Knoten in der PQ.
- In der PQ sind alle Knoten, die über kreuzende Kanten von dem Bereich der bearbeiteten Knoten erreicht werden können.
- Sei p ein anderer Pfad von s nach t. Dieser Pfad muss durch einen Knoten v laufen, der in der PQ ist. Sei p_0 der Subpfad $s \rightsquigarrow v$ und p_1 der Subpfad $v \rightsquigarrow t$.

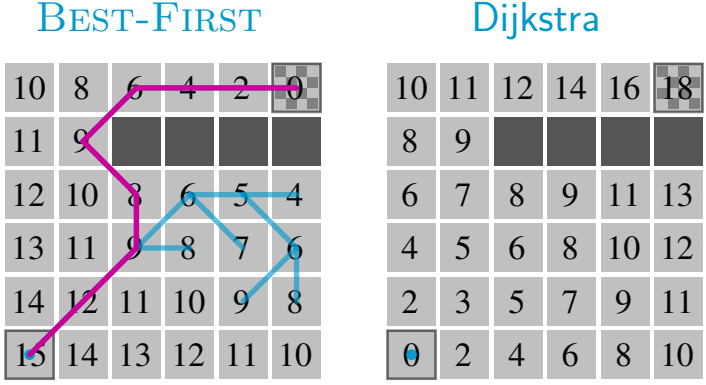
$$
\begin{aligned}
dist[t] &= f[t] && \text{da } h(t) = 0 \\
&\leq f[v] && \text{wegen Priorität der PQ} \\
&= dist[v] + h(v) && \text{Definition von } f \\
&\leq dist[v] + c(p_1) && \text{da } h \text{ zulässig ist} \\
&\leq c(p_0) + c(p_1) && \text{nach dem Lemma} \\
&= c(p)
\end{aligned}
$$

- Also kann es keinen kürzeren Pfad von s nach t geben.

Best-First



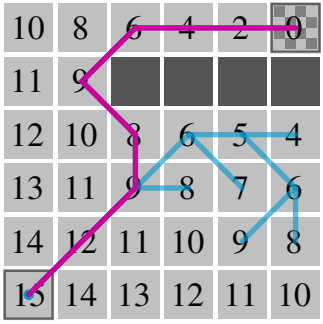
A* im Vergleich mit Best-First und Dijkstra



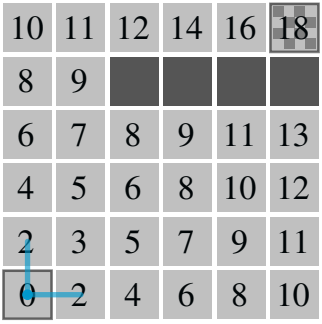
- Die Prioritäten $dist[]$ von Dijkstra und $f[]$ von A* werden erst beim Durchlauf für die besuchten Knoten bestimmt und aktualisiert. Hier sind die Werte zur Illustration von Anfang an eingetragen.

A* im Vergleich mit Best-First und Dijkstra

Best-First



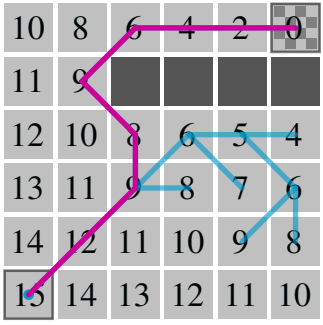
Dijkstra



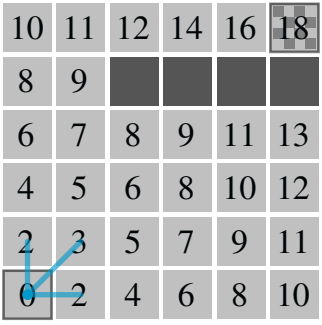
- Die Prioritäten $dist[]$ von Dijkstra und $f[]$ von A* werden erst beim Durchlauf für die besuchten Knoten bestimmt und aktualisiert. Hier sind die Werte zur Illustration von Anfang an eingetragen.

A* im Vergleich mit Best-First und Dijkstra

Best-First



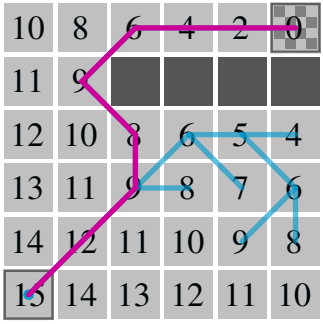
Dijkstra



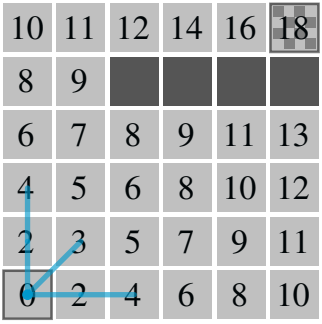
- Die Prioritäten $dist[]$ von Dijkstra und $f[]$ von A* werden erst beim Durchlauf für die besuchten Knoten bestimmt und aktualisiert. Hier sind die Werte zur Illustration von Anfang an eingetragen.

A* im Vergleich mit Best-First und Dijkstra

Best-First



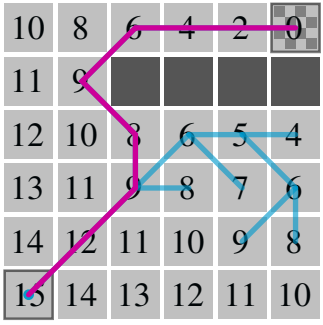
Dijkstra



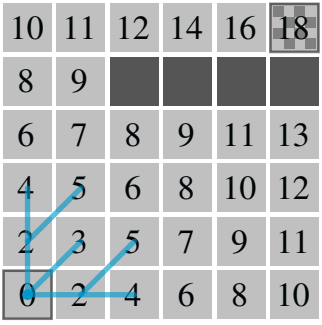
- Die Prioritäten $dist[]$ von Dijkstra und $f[]$ von A* werden erst beim Durchlauf für die besuchten Knoten bestimmt und aktualisiert. Hier sind die Werte zur Illustration von Anfang an eingetragen.

A* im Vergleich mit Best-First und Dijkstra

Best-First



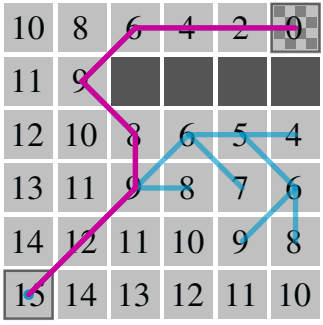
Dijkstra



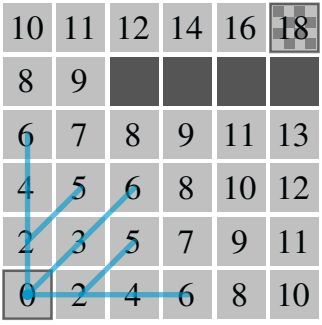
- Die Prioritäten $dist[]$ von Dijkstra und $f[]$ von A* werden erst beim Durchlauf für die besuchten Knoten bestimmt und aktualisiert. Hier sind die Werte zur Illustration von Anfang an eingetragen.

A* im Vergleich mit Best-First und Dijkstra

Best-First



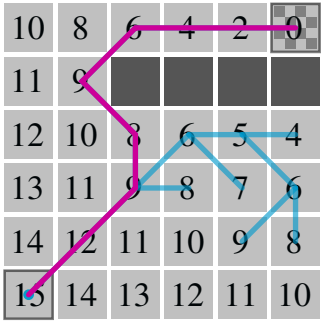
Dijkstra



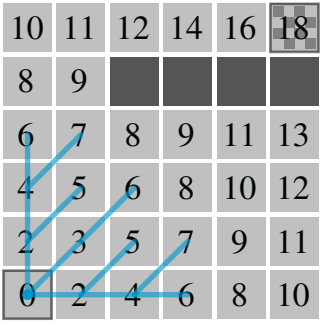
- Die Prioritäten $dist[]$ von Dijkstra und $f[]$ von A* werden erst beim Durchlauf für die besuchten Knoten bestimmt und aktualisiert. Hier sind die Werte zur Illustration von Anfang an eingetragen.

A* im Vergleich mit Best-First und Dijkstra

Best-First



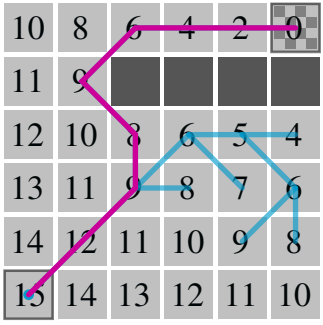
Dijkstra



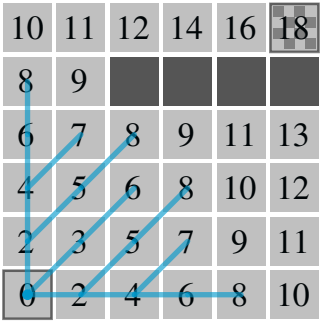
- Die Prioritäten $dist[]$ von Dijkstra und $f[]$ von A* werden erst beim Durchlauf für die besuchten Knoten bestimmt und aktualisiert. Hier sind die Werte zur Illustration von Anfang an eingetragen.

A* im Vergleich mit Best-First und Dijkstra

Best-First



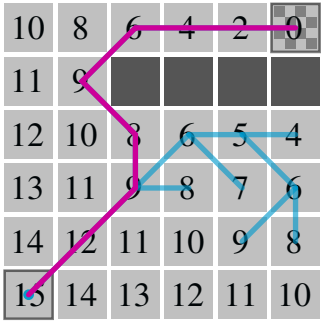
Dijkstra



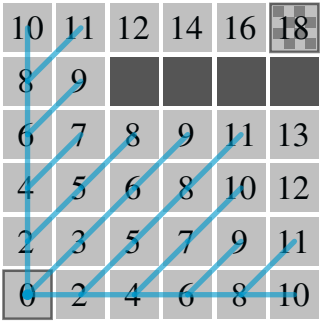
- Die Prioritäten $dist[]$ von Dijkstra und $f[]$ von A* werden erst beim Durchlauf für die besuchten Knoten bestimmt und aktualisiert. Hier sind die Werte zur Illustration von Anfang an eingetragen.

A* im Vergleich mit Best-First und Dijkstra

Best-First



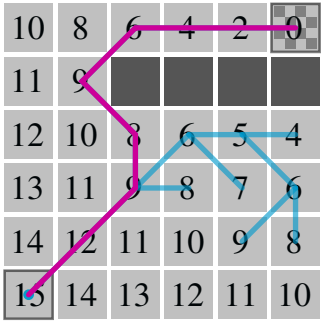
Dijkstra



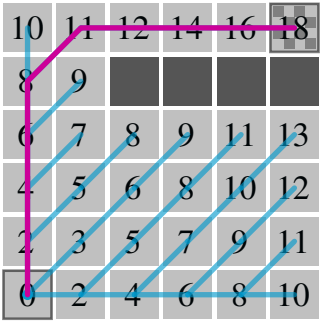
- Die Prioritäten $dist[]$ von Dijkstra und $f[]$ von A* werden erst beim Durchlauf für die besuchten Knoten bestimmt und aktualisiert. Hier sind die Werte zur Illustration von Anfang an eingetragen.

A* im Vergleich mit Best-First und Dijkstra

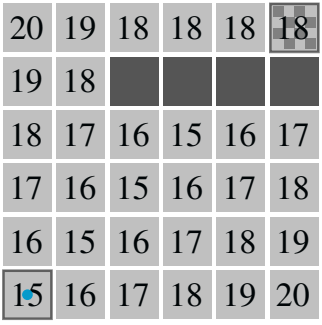
Best-First



Dijkstra



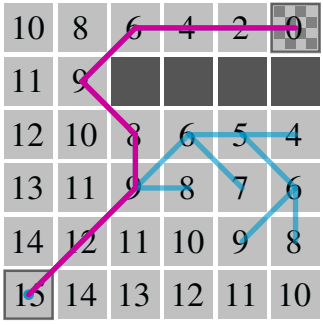
A*



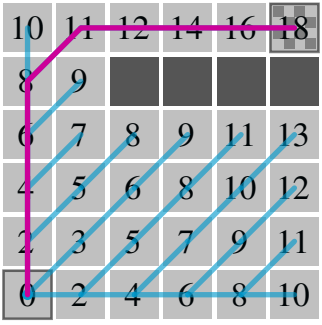
- Die Prioritäten $dist[]$ von Dijkstra und $f[]$ von A* werden erst beim Durchlauf für die besuchten Knoten bestimmt und aktualisiert. Hier sind die Werte zur Illustration von Anfang an eingetragen.

A* im Vergleich mit Best-First und Dijkstra

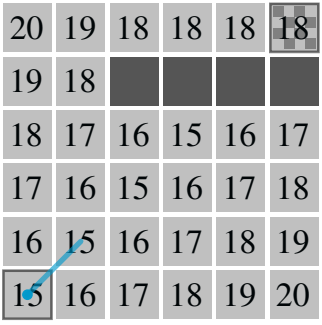
Best-First



Dijkstra



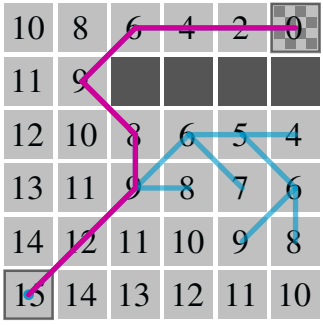
A*



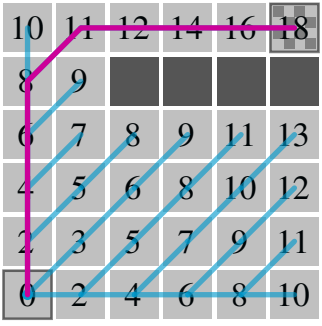
- Die Prioritäten $dist[]$ von Dijkstra und $f[]$ von A* werden erst beim Durchlauf für die besuchten Knoten bestimmt und aktualisiert. Hier sind die Werte zur Illustration von Anfang an eingetragen.

A* im Vergleich mit Best-First und Dijkstra

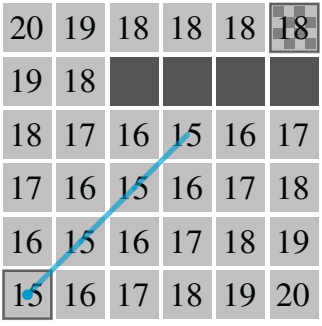
Best-First



Dijkstra

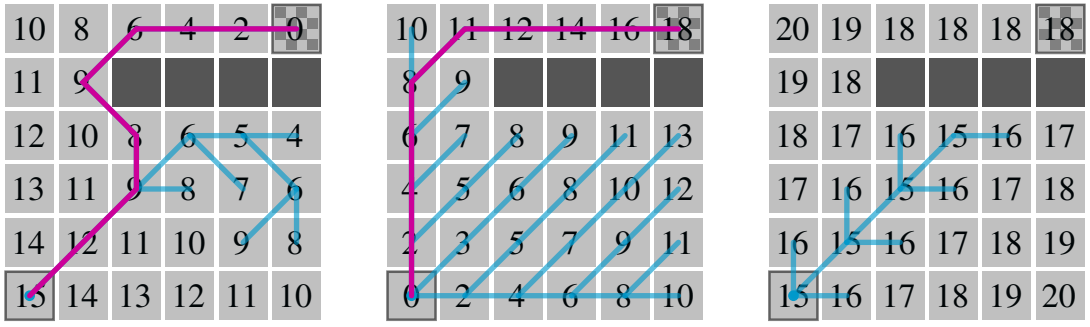


A*



- Die Prioritäten $dist[]$ von Dijkstra und $f[]$ von A* werden erst beim Durchlauf für die besuchten Knoten bestimmt und aktualisiert. Hier sind die Werte zur Illustration von Anfang an eingetragen.

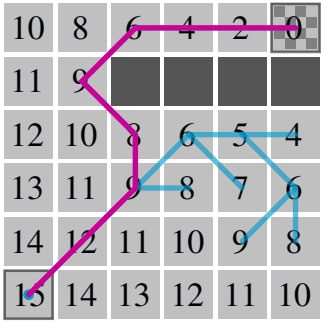
A* im Vergleich mit Best-First und Dijkstra



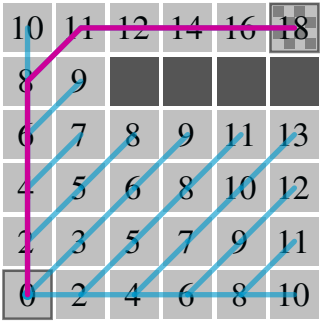
- Die Prioritäten $dist[]$ von Dijkstra und $f[]$ von A* werden erst beim Durchlauf für die besuchten Knoten bestimmt und aktualisiert. Hier sind die Werte zur Illustration von Anfang an eingetragen.

A* im Vergleich mit Best-First und Dijkstra

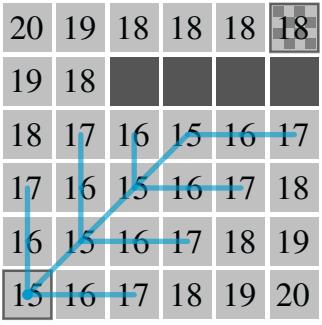
Best-First



Dijkstra

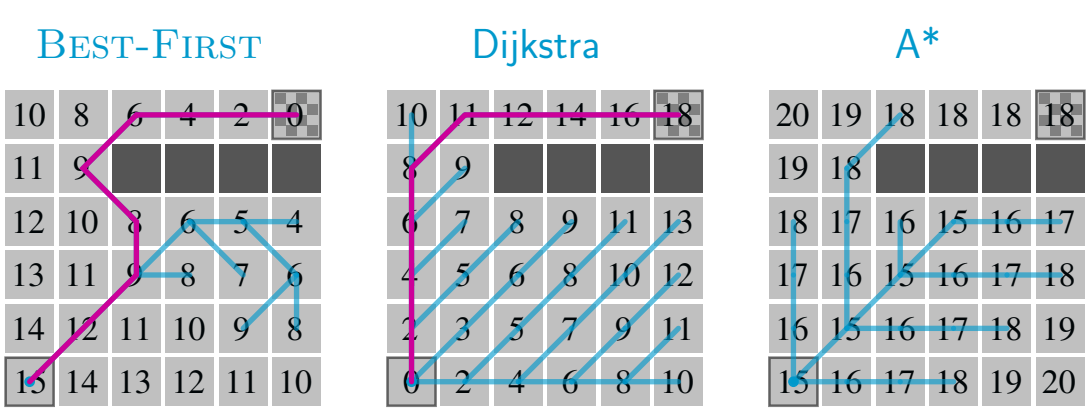


A*



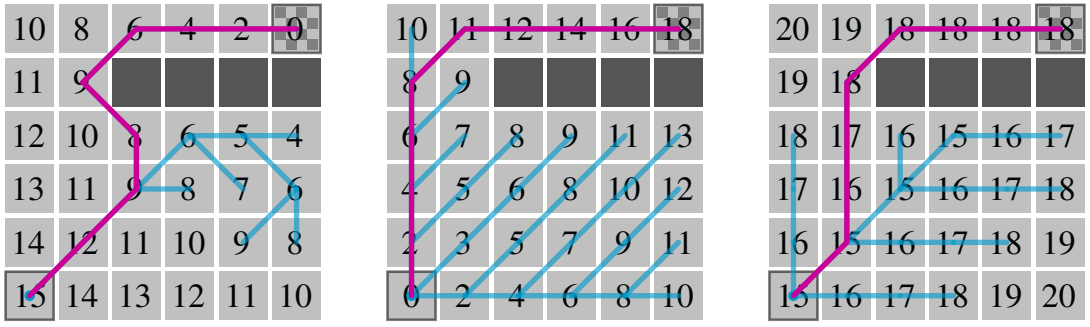
- Die Prioritäten $dist[]$ von Dijkstra und $f[]$ von A* werden erst beim Durchlauf für die besuchten Knoten bestimmt und aktualisiert. Hier sind die Werte zur Illustration von Anfang an eingetragen.

A* im Vergleich mit Best-First und Dijkstra



- Die Prioritäten $dist[]$ von Dijkstra und $f[]$ von A* werden erst beim Durchlauf für die besuchten Knoten bestimmt und aktualisiert. Hier sind die Werte zur Illustration von Anfang an eingetragen.

A* im Vergleich mit Best-First und Dijkstra



- Die Prioritäten $dist[]$ von Dijkstra und $f[]$ von A* werden erst beim Durchlauf für die besuchten Knoten bestimmt und aktualisiert. Hier sind die Werte zur Illustration von Anfang an eingetragen.
- A* findet die optimale Lösung und besucht dabei weniger Kanten als Dijkstra. Bei größeren und komplexeren Graphen ist die Einsparung oft sehr viel größer.

A* jenseits von direkten Graphenproblemen

- Die Darstellung von A* war stark an die Suche in gegebenen Graphen (mit indizierten Knoten) orientiert.
- Bei anderen Aufgaben, wie solchen, die im Konext von *Backtracking* und *Branch-and-Bound* besprochen wurden, wird der Suchbaum (oder Suchgraph) erst bei der Suche erstellt und die Knoten entsprechen Teillösungen, die nicht auf einfache Indizes reduziert werden können.

A* jenseits von direkten Graphenproblemen

- Die Darstellung von A* war stark an die Suche in gegebenen Graphen (mit indizierten Knoten) orientiert.
- Bei anderen Aufgaben, wie solchen, die im Konext von *Backtracking* und *Branch-and-Bound* besprochen wurden, wird der Suchbaum (oder Suchgraph) erst bei der Suche erstellt und die Knoten entsprechen Teillösungen, die nicht auf einfache Indizes reduziert werden können.
- Auch in diesen Fällen kann A* angewendet werden. Die Kosten sind hier die Anzahl der Lösungsschritte zur Lösung.
- Dafür wird eine Heuristik benötigt, die für jede Teillösung die Anzahl der Lösungsschritte zum Ziel schätzt.

A* jenseits von direkten Graphenproblemen

- Die Darstellung von A* war stark an die Suche in gegebenen Graphen (mit indizierten Knoten) orientiert.
- Bei anderen Aufgaben, wie solchen, die im Konext von *Backtracking* und *Branch-and-Bound* besprochen wurden, wird der Suchbaum (oder Suchgraph) erst bei der Suche erstellt und die Knoten entsprechen Teillösungen, die nicht auf einfache Indizes reduziert werden können.
- Auch in diesen Fällen kann A* angewendet werden. Die Kosten sind hier die Anzahl der Lösungsschritte zur Lösung.
- Dafür wird eine Heuristik benötigt, die für jede Teillösung die Anzahl der Lösungsschritte zum Ziel schätzt.
- Damit A* effizient ist, wird Konsistenz benötigt: Durch einen Lösungsschritt darf der Wert der Heuristik höchstens um 1 sinken.
- Mit Zulässigkeit **ohne Konsistenz** ist man im Prinzip bei *Branch-and-Bound*.

A* im Baum der Teillösungen: Pseudocode

Für eine Teillösung *psol* sei *psol.f* der f-Wert von A*, also

- Anzahl der Schritte bis Erreichen der Teillösung *psol* plus
- Wert der Heuristik für Teillösung *psol*.

```
1  Q : PriorityQueue of partial solutions
2  esol : empty partial solution
3  add(Q, esol, esol.f)
4  while Q ≠ ∅
5    psol ← poll(Q)
6    if psol is solution
7      return psol
8    end
9    for each move possible in psol
10     perform move in psol
11     add(Q, psol, psol.f)
12   end
13 end
14 return null
```

Unterschied zu A* im Graphen mit indizierten Knoten

- Warum wird hier kein *relax* verwendet?
- In *relax* wird geprüft, ob der Endknoten über die neue Kante schneller erreicht wird, als vorher (falls er schon vorher entdeckt wurde). In diesem Fall wird der Weg über die neue Kante berücksichtigt.

Unterschied zu A* im Graphen mit indizierten Knoten

- Warum wird hier kein *relax* verwendet?
- In *relax* wird geprüft, ob der Endknoten über die neue Kante schneller erreicht wird, als vorher (falls er schon vorher entdeckt wurde). In diesem Fall wird der Weg über die neue Kante berücksichtigt.
- Anders als im Graphen mit indizierten Knoten, kann hier nicht leicht festgestellt werden, ob die neue Teillösung einer schon gefundenen Teillösung entspricht.
- Dadurch können äquivalente Teillösungen in die Queue gelangen.

Unterschied zu A* im Graphen mit indizierten Knoten

- Warum wird hier kein *relax* verwendet?
- In *relax* wird geprüft, ob der Endknoten über die neue Kante schneller erreicht wird, als vorher (falls er schon vorher entdeckt wurde). In diesem Fall wird der Weg über die neue Kante berücksichtigt.
- Anders als im Graphen mit indizierten Knoten, kann hier nicht leicht festgestellt werden, ob die neue Teillösung einer schon gefundenen Teillösung entspricht.
- Dadurch können äquivalente Teillösungen in die Queue gelangen.
- Da jeweils die Teillösung mit den wenigsten Schritten der Queue zuerst entnommen wird, ist dies kein Problem für die Korrektheit des Verfahrens.
- Allerdings kann die Effizienz deutlich leiden.
- Abhilfe schafft hier die Verwendung von *Hash Sets*, die in der nächsten Vorlesung besprochen werden.

Das Problem des Handlungsreisenden

- Als zweite Herausforderung betrachten wir das **Problem des Handlungsreisenden** (*Travelling Salesman Problem*, TSP):
- Zu einem vollständigen, gewichteten Graphen soll ein Zyklus mit minimalem Gewicht bestimmt werden, der jeden Knoten genau einmal besucht (TSP-Tour).
- Ein Graph $G = (V, E)$ heißt **vollständig**, wenn er alle (nicht-reflexiven) Kanten enthält, also $E = \{v \to w \mid v, w \in V \text{ mit } v \neq w\}$ gilt.
- Das TSP ist NP-vollständig und daher eine interessante Herausforderung für die Algorithmenentwicklung.

Das Problem des Handlungsreisenden

- Als zweite Herausforderung betrachten wir das **Problem des Handlungsreisenden** (*Travelling Salesman Problem*, TSP):
- Zu einem vollständigen, gewichteten Graphen soll ein Zyklus mit minimalem Gewicht bestimmt werden, der jeden Knoten genau einmal besucht (TSP-Tour).
- Ein Graph $G = (V, E)$ heißt **vollständig**, wenn er alle (nicht-reflexiven) Kanten enthält, also $E = \{v \to w \mid v, w \in V \text{ mit } v \neq w\}$ gilt.
- Das TSP ist NP-vollständig und daher eine interessante Herausforderung für die Algorithmenentwicklung.
- Hier betrachten wir zwei heuristische Algorithmen, die schnell Lösungen bestimmen, aber diese Lösungen können weit von Optimum entfernt sein.
- In den folgenden Formulierungen fassen wir das Gewicht einer Kante als seine Länge auf (kleinster Abstand $\hat{=}$ kleinstes Gewicht etc.)

Heuristische Algorithmen für das TSP

- **Nearest Insertion:** Fange mit einer Tour an, die nur die beiden am dichtesten zusammen liegenden Knoten verbindet.
- Wähle dann iterativ immer denjenigen Knoten aus, der den kleinsten Abstand zu einem der Knoten der Tour hat.
- Füge diesen Knoten so in die Tour ein, dass die Zunahme der Tourlänge möglichst klein ist. Der Knoten wird bei der Kante eingefügt, zu der er den geringsten Abstand hat.

Heuristische Algorithmen für das TSP

- **Nearest Insertion:** Fange mit einer Tour an, die nur die beiden am dichtesten zusammen liegenden Knoten verbindet.
  - Wähle dann iterativ immer denjenigen Knoten aus, der den kleinsten Abstand zu einem der Knoten der Tour hat.
  - Füge diesen Knoten so in die Tour ein, dass die Zunahme der Tourlänge möglichst klein ist. Der Knoten wird bei der Kante eingefügt, zu der er den geringsten Abstand hat.
- **Farthest Insertion:** Fange mit einer Tour an, die nur die beiden am weitesten auseinander liegenden Knoten verbindet.
  - Wähle dann iterativ immer denjenigen Knoten aus, dessen minimaler Abstand zu einem der Knoten der Tour maximal ist.
  - Füge diesen Knoten wie bei *Nearest Insertion* in die Tour ein.

Heuristische Algorithmen für das TSP

- **Nearest Insertion:** Fange mit einer Tour an, die nur die beiden am dichtesten zusammen liegenden Knoten verbindet.
  - Wähle dann iterativ immer denjenigen Knoten aus, der den kleinsten Abstand zu einem der Knoten der Tour hat.
  - Füge diesen Knoten so in die Tour ein, dass die Zunahme der Tourlänge möglichst klein ist. Der Knoten wird bei der Kante eingefügt, zu der er den geringsten Abstand hat.
- **Farthest Insertion:** Fange mit einer Tour an, die nur die beiden am weitesten auseinander liegenden Knoten verbindet.
  - Wähle dann iterativ immer denjenigen Knoten aus, dessen minimaler Abstand zu einem der Knoten der Tour maximal ist.
  - Füge diesen Knoten wie bei *Nearest Insertion* in die Tour ein.
  - Beide Varianten können mit einer Laufzeit in $\boldsymbol{O}(n^2)$ implementiert werden.
  - In der Praxis findet *Farthest Insertion* meist bessere Lösungen als *Nearest Insertion*.

Approximative Algorithmen

- Heuristische Verfahren erlauben es für bestimmte Problemklassen schnelle Lösungen zu generieren.
- Diese Lösungen sind oft nicht optimal, und man bekommt keine Garantie, wie stark die Abweichung vom Optimum maximal ist.
- Eine positive Ausnahme ist der A* Algorithmus, der zumindest bei Verwendung einer konsistenten Heuristik effizient eine *optimale* Lösung findet.

Approximative Algorithmen

- Heuristische Verfahren erlauben es für bestimmte Problemklassen schnelle Lösungen zu generieren.
- Diese Lösungen sind oft nicht optimal, und man bekommt keine Garantie, wie stark die Abweichung vom Optimum maximal ist.
- Eine positive Ausnahme ist der A* Algorithmus, der zumindest bei Verwendung einer konsistenten Heuristik effizient eine *optimale* Lösung findet.
- **Approximative Algorithmen** zielen auf suboptimale Lösungen ab, um schnellere Laufzeiten zu erzielen, und geben eine obere Schranke für die maximale Abweichung vom Optimum an.
- Wenn ein Algorithmus Lösungen liefert, deren Wert C sich höchsten um den Faktor $\rho > 1$ von dem optimalen Wert C^* unterscheidet, also

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho$$

wird er als **ρ-Approximationsalgorithmus** bezeichnet. In der Definition wird das Maximum genommen, um Minimierung- und Maximierungsprobleme abzudecken.

Approximationen für das metrische TSP

- Wir betrachten wieder das Travelling Salesman Problem (TSP). Bei dem vorhin besprochenen heuristischen Ansatz bleibt unklar, wie weit die Lösung vom Optimum entfernt sein kann.
- Unter der Voraussetzung, dass die Entfernungen zwischen den Knoten die Dreiecksungleichung erfüllen (metrisches TSP), gibt es effiziente approximative Algorithmen für das TSP.

Approximationen für das metrische TSP

- Wir betrachten wieder das Travelling Salesman Problem (TSP). Bei dem vorhin besprochenen heuristischen Ansatz bleibt unklar, wie weit die Lösung vom Optimum entfernt sein kann.
- Unter der Voraussetzung, dass die Entfernungen zwischen den Knoten die Dreiecksungleichung erfüllen (metrisches TSP), gibt es effiziente approximative Algorithmen für das TSP.
- Der Christofides Algorithmus bestimmt eine Lösung in $\boldsymbol{O}(V^4)$ mit höchstens 1.5 mal den Kosten des Optimums. Dies ist die beste bekannte Approximation.
- Allerdings kann es unter P≠NP keine beliebig guten Approximationen geben. Bislang ist bekannt, dass $\frac{122}{123}$ eine obere Schranke ist [Karpinski et al, 2015].
- Wir besprechen hier eine einfachere Variante, die eine Lösung bestimmt, deren Tourlänge höchstens das doppelte des Optimums ist (2-Approximationsalgorithmus).

Pseudocode zur 2-Approximation des metrischen TSP

- Dieser Ansatz nimmt einen minimalem Spannbaum des gegebenen Graphen als Ausgangsbasis für die approximative Lösung.
- Der Spannbaum wird in einen Zyklus verwandelt, der jeden Knoten genau einmal besucht.

Listing 1: **Pseudocode zur Bestimmung einer Tour durch den Graphen G, dessen Kosten die Dreiecksungleichungen erfüllen, die maximale die doppelten Kosten der optimalen TSP-Tour hat.**

```
1 APPROX-TSP-TOUR(G)
2 bestimme einen minimalen Spannbaum T von G mit bel. Startknoten s
3 sei H die Liste der Knoten von T in Nebenreihenfolge
4 H stellt als Zyklus eine TSP−Tour dar
```

- Zur Erinnerung (Vorlesung #7): Die Nebenreihenfolge ist die Sortierung der Knoten eines Graphen in der Reihenfolge des Entdecktwerdens bei der Tiefensuche (d.h. aufsteigend nach Eingangsstempel)

Korrektheit und Approximation von Approx-TSP-Tour

Der Pseudocode Approx-TSP-Tour liefert eine korrekte TSP-Tour (jeder Knoten wird genau einmal besucht) in einer Laufzeit in $\boldsymbol{O}(V^2 \log V)$. Die Tour hat maximal die zweifache Länge der optimalen Tour.

Beweis.

- Die Laufzeit von Prim für den MST ist in $\boldsymbol{O}(E \log V)$. Die anschließende Tiefensuche zur Sortierung der Knoten ist in $\boldsymbol{O}(V^2)$. Da in der vorliegenden Situation $E \in \boldsymbol{O}(V^2)$ gilt, ist die Laufzeit wie behauptet.
- Eine TSP-Tour wird durch Entfernung einer beliebigen Kante zu einem Spannbaum, also ist die Länge des MST eine untere Schranke für die Länge der optimalen TSP-Tour TSP^*:

$$c(MST) \leq c(TSP^*)$$

2-Approximation für das metrische TSP

- Außerdem kann der MST in einen Zyklus umgewandelt werden, der fast eine (suboptimale) TSP-Tour darstellt:
- Dazu betrachten wir die vollständige Traversierung des MST. In diese Liste werden die Knoten bei einer Tiefensuche sowohl bei dem Besuch eines Knotens (Eingangsstempel), als auch beim Verlassen eines Knotens (Ausgangsstempel) eingetragen.
- Diese Traversierung W passiert jede Kante des MST zweimal, einmal in jeder Richtung.

2-Approximation für das metrische TSP

- Außerdem kann der MST in einen Zyklus umgewandelt werden, der fast eine (suboptimale) TSP-Tour darstellt:
- Dazu betrachten wir die vollständige Traversierung des MST. In diese Liste werden die Knoten bei einer Tiefensuche sowohl bei dem Besuch eines Knotens (Eingangsstempel), als auch beim Verlassen eines Knotens (Ausgangsstempel) eingetragen.
- Diese Traversierung W passiert jede Kante des MST zweimal, einmal in jeder Richtung.
- W bildet einen Zyklus, da die Tiefensuche am Ende wieder zu der Wurzel zurückkehrt und es gilt

$$c(W) \;=\; 2c(MST) \;\leq\; 2c(TSP^*)$$

- Allerdings ist der Zyklus W noch keine zulässige TSP Tour, da die Knoten mehrfach (genauer gesagt doppelt) besucht werden.
- Dies kann durch einen kleinen Umbau des Zyklus behoben werden.

2-Approximation für das metrische TSP

- Bei dem Durchlaufen von W wird jeder Knoten, der vorher schon besucht wurde, ausgelassen. Die resultierende Tour nennen wir T. Sie besucht jeden Knoten nur einmal.
- Durch das Überspringen eines Knotens wird die Länge des Zyklus kürzer (jedenfalls nicht länger):
- Sei $u - v - w$ eine Knotenfolge des ursprünglichen Zyklus, aus der der Knoten v ausgelassen wird. Wegen der Dreiecks-Ungleichung gilt:

$$c(u, w) \leq c(u, v) + c(v, w)$$

2-Approximation für das metrische TSP

- Bei dem Durchlaufen von W wird jeder Knoten, der vorher schon besucht wurde, ausgelassen. Die resultierende Tour nennen wir T. Sie besucht jeden Knoten nur einmal.
- Durch das Überspringen eines Knotens wird die Länge des Zyklus kürzer (jedenfalls nicht länger):
- Sei $u - v - w$ eine Knotenfolge des ursprünglichen Zyklus, aus der der Knoten v ausgelassen wird. Wegen der Dreiecks-Ungleichung gilt:

$$c(u, w) \leq c(u, v) + c(v, w)$$

- Daher erhalten wir insgesamt die Abschätzung

$$c(T) \leq c(W) = 2c(MST) \leq 2c(TSP^*)$$

- Da die Tour T genau der Rückgabe von Approx-TSP-Tour entspricht, ist damit die Behauptung bewiesen.

Nicht-Approximierbarkeit des allgemeinen TSP

- Nun folgt das ernüchternde Resultat, dass es für das allgemeine Handlungsreisendenproblem (als ohne die Voraussetzung der Dreiecksungleichung) gar keine Approximation geben kann.

Das allgemeine Handlungsreisendenproblem ist nicht approximierbar

Unter der Voraussetzung P≠NP gibt es für kein $\rho \geq 1$ einen ρ-Approximationsalgorithmus mit polynomieller Laufzeit für das allgemeine Handlungsreisendenproblem.

Nicht-Approximierbarkeit des allgemeinen TSP

- Nun folgt das ernüchternde Resultat, dass es für das allgemeine Handlungsreisendenproblem (als ohne die Voraussetzung der Dreiecksungleichung) gar keine Approximation geben kann.

Das allgemeine Handlungsreisendenproblem ist nicht approximierbar

Unter der Voraussetzung P≠NP gibt es für kein $\rho \geq 1$ einen ρ-Approximationsalgorithmus mit polynomieller Laufzeit für das allgemeine Handlungsreisendenproblem.

Beweis.

- Wir nehmen an, dass es für ein $\rho \geq 1$ einen ρ-Approximationsalgorithmus $\mathcal{X}$ mit polynomieller Laufzeit für das TSP gibt.
- Da die Approximation dann auch für alle größeren Zahlen gilt, können wir ρ als ganzzahlig voraussetzen (z.B. durch Aufrunden).
- Wir zeigen, dass sich mit $\mathcal{X}$ auch das Hamilton-Zyklus Problem lösen lässt.

Nicht-Approximierbarkeit des allgemeinen TSP

- Sei ein Graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ gegeben, für den die Existenz eines Hamilton-Kreis festgestellt werden soll.
- Wir definieren den vollständigen und gewichteten Graphen $\boldsymbol{G}' = (\boldsymbol{V}, \boldsymbol{E}', c)$ durch

$$\boldsymbol{E}' = \{(v, w) \in \boldsymbol{V} \times \boldsymbol{V} \mid v \neq w\}$$

$$c(v, w) = \begin{cases} 1 & \text{falls } (v, w) \in \boldsymbol{E} \\ \rho V + 1 & \text{sonst} \end{cases}$$

- Offensichtlich kann $\boldsymbol{G}'$ in polynomieller Zeit in V und E aus $\boldsymbol{G}$ erstellt werden.

Nicht-Approximierbarkeit des allgemeinen TSP

- Sei ein Graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ gegeben, für den die Existenz eines Hamilton-Kreis festgestellt werden soll.
- Wir definieren den vollständigen und gewichteten Graphen $\boldsymbol{G'} = (\boldsymbol{V}, \boldsymbol{E'}, c)$ durch

$$\boldsymbol{E'} = \{(v, w) \in \boldsymbol{V} \times \boldsymbol{V} \mid v \neq w\}$$

$$c(v, w) = \begin{cases} 1 & \text{falls } (v, w) \in \boldsymbol{E} \\ \rho V + 1 & \text{sonst} \end{cases}$$

- Offensichtlich kann $\boldsymbol{G'}$ in polynomieller Zeit in V und E aus $\boldsymbol{G}$ erstellt werden.
- $\boldsymbol{G'}$ ist so definiert, dass der ρ-Approximatiosnalgorithmus $\mathcal{X}$ für das TSP in $\boldsymbol{G'}$ die Frage nach dem Hamilton-Zyklus in $\boldsymbol{G}$ beantwortet!

Nicht-Approximierbarkeit des allgemeinen TSP

- Nach Konstruktion unterscheiden sich die Kosten für die optimale TSP-Tour um mehr als den Faktor ρ, abhängig davon, ob ein Hamilton-Zyklus in $\boldsymbol{G}$ exisitert oder nicht.
- Daher könnte der ρ-Approximationsalgorithmus $\mathcal{X}$ die Hamilton-Frage entscheiden:
- Wir wenden $\mathcal{X}$ auf das TSP $\boldsymbol{G}'$ an und unterscheiden nach den Kosten der gefundenen Tour T:

1. $c(T) \leq \rho V$: Die Tour enthält nur Kanten, die zu $\boldsymbol{G}$ gehören, da alle anderen Kanten Kosten $\rho V + 1$ haben. Daher stellt die TSP-Tour T auch einen Hamilton-Zyklus in $\boldsymbol{G}$ dar.
2. $c(T) > \rho V$: Die optimale Tour kann sich maximal um den Faktor ρ unterscheiden, hat also Kosten $> V$. Da ein Hamilton-Zyklus in $\boldsymbol{G}$ eine TSP Tour mit Kosten V wäre, kann es diesen nicht geben, wenn die optimale TSP Tour Kosten $> V$ hat.

Approximativer Ansatz für das 0/1-Rucksack Problem

- Mittels dynamischer Programmierung konnte ein Algorithmus für das 0/1-Rucksackproblem mit pseudopolynomieller Laufzeit formuliert werden.
- Die Laufzeit in $\boldsymbol{O}(KW)$ kann für große Kapazitätsgrenzen W sehr schlecht sein.
- Mit einem approximativen Algorithmus kann man effizient Lösungen finden, die beliebig nah am Optimum sind.

Approximativer Ansatz für das 0/1-Rucksack Problem

- Mittels dynamischer Programmierung konnte ein Algorithmus für das 0/1-Rucksackproblem mit pseudopolynomieller Laufzeit formuliert werden.
- Die Laufzeit in $\boldsymbol{O}(KW)$ kann für große Kapazitätsgrenzen W sehr schlecht sein.
- Mit einem approximativen Algorithmus kann man effizient Lösungen finden, die beliebig nah am Optimum sind.
- Wenn im Rahmen von dynamischer Programmierung die Laufzeit verbessert werden soll, muss die Tabelle verkleinert werden.
- Im Sinne einer approximativen Lösung ist dies im Prinzip durch eine Skalierung einer Variable der $\textsc{Opt}$ Funktion möglich.

Approximativer Ansatz für das 0/1-Rucksack Problem

- Mittels dynamischer Programmierung konnte ein Algorithmus für das 0/1-Rucksackproblem mit pseudopolynomieller Laufzeit formuliert werden.
- Die Laufzeit in $\boldsymbol{O}(KW)$ kann für große Kapazitätsgrenzen W sehr schlecht sein.
- Mit einem approximativen Algorithmus kann man effizient Lösungen finden, die beliebig nah am Optimum sind.
- Wenn im Rahmen von dynamischer Programmierung die Laufzeit verbessert werden soll, muss die Tabelle verkleinert werden.
- Im Sinne einer approximativen Lösung ist dies im Prinzip durch eine Skalierung einer Variable der OPT Funktion möglich.
- Die Anzahl der Objekte kann natürlich nicht skaliert werden. Aber auch eine Skalierung der Kapazitätsgrenze ist nicht zielführend, da die Kapazitätsgrenze exakt eingehalten werden muss.
- Im Gegensatz dazu können die Werte der Objekte skaliert werden. Daher erstellen wir nun einen neuen dynamischen Programmier-Ansatz, bei dem die OPT Funktion von dem zu erreichenden Wert abhängt.

0/1-Rucksack Problem – Duales Dynamisches Programm

- Wir definieren eine rekursive Funktion $\textsc{Opt}(k, v)$, die angibt wieviel Gewicht mindestens notwendig ist, um mit einer Auswahl aus Objekten $1, \ldots, k$ einen vorgegebenen Wert v (oder mehr) zu erreichen und ∞, falls der Wert v gar nicht mit den Objekten erreicht werden kann.
- V sei die Summe aller Werte: $V = \sum_{k=1}^{K} v_k$.
- Der Definitionsbereich von $\textsc{Opt}$ ist $0 \leq k \leq K$ und $0 \leq v \leq V$.

0/1-Rucksack Problem – Duales Dynamisches Programm

- Wir definieren eine rekursive Funktion $\textsc{Opt}(k, v)$, die angibt wieviel Gewicht mindestens notwendig ist, um mit einer Auswahl aus Objekten $1, \ldots, k$ einen vorgegebenen Wert v (oder mehr) zu erreichen und ∞, falls der Wert v gar nicht mit den Objekten erreicht werden kann.
- V sei die Summe aller Werte: $V = \sum_{k=1}^{K} v_k$.
- Der Definitionsbereich von $\textsc{Opt}$ ist $0 \leq k \leq K$ und $0 \leq v \leq V$.
- Den Wert $v = 0$ erreicht man ohne Objekte, also ist die Gewichtsgrenze $\textsc{Opt}(k, 0) = 0$ für alle k.
- Mit $k = 0$ Objekten, kann man keinen Zielwert $v > 0$ erreichen.

0/1-Rucksack Problem – Duales Dynamisches Programm

- Wir definieren eine rekursive Funktion $\textsc{Opt}(k, v)$, die angibt wieviel Gewicht mindestens notwendig ist, um mit einer Auswahl aus Objekten $1, \ldots, k$ einen vorgegebenen Wert v (oder mehr) zu erreichen und ∞, falls der Wert v gar nicht mit den Objekten erreicht werden kann.
- V sei die Summe aller Werte: $V = \sum_{k=1}^{K} v_k$.
- Der Definitionsbereich von $\textsc{Opt}$ ist $0 \leq k \leq K$ und $0 \leq v \leq V$.
- Den Wert $v = 0$ erreicht man ohne Objekte, also ist die Gewichtsgrenze $\textsc{Opt}(k, 0) = 0$ für alle k.
- Mit $k = 0$ Objekten, kann man keinen Zielwert $v > 0$ erreichen.

$$\textsc{Opt}(k, v) = \begin{cases} 0 & \text{falls } v = 0 \\ \infty & \text{falls } k = 0 \,\&\, v > 0 \\ \max(\underbrace{w_k + \textsc{Opt}(k-1, \max(0, v - v_k))}_{k \text{ ausgewählt}}, \underbrace{\textsc{Opt}(k-1, v)}_{k \text{ nicht ausgewählt}}) & \text{sonst} \end{cases}$$

Analyse der dualen 0/1-Rucksack Lösung

Korrektheit und Laufzeit der dualen 0/1-Rucksack Lösung

Der Algorithmus basierend auf dynamischer Programmierung mit der $\textsc{Opt}$ Funktion von der vorigen Seite findet die optimale Lösung des 0/1-Rucksack Problems mit ganzzahligen Werten und Gewichten in einer Laufzeit in $\boldsymbol{O}(KV)$, wobei V die Summe der Werte aller Objekte ist: $V = \sum_{k=1}^{K} v_k$.

- Die Tabelle, die für die dynamische Programmierung benötigt wird, hat die Größe $(K+1)(V+1)$.
- Das Berechnen jedes Tabelleneintrags kann gemäß der $\textsc{Opt}$ Funktion in konstanter Zeit ausgeführt werden.
- Insgesamt kann also die Tabelle in einer Laufzeit in $\boldsymbol{O}(KV)$ bestimmt werden. □

Analyse der dualen 0/1-Rucksack Lösung

- Nachtrag: Der Lösungswert steht nicht unbedingt am Ende der Tabelle in dem Eintrag (K, V), sondern er muss aus der Tabelle herausgesucht werden.
- Man sucht den größten Wert v, für den $OPT(K, v) \leq W$ gilt.

Analyse der dualen 0/1-Rucksack Lösung

- Nachtrag: Der Lösungswert steht nicht unbedingt am Ende der Tabelle in dem Eintrag (K, V), sondern er muss aus der Tabelle herausgesucht werden.
- Man sucht den größten Wert v, für den $OPT(K, v) \leq W$ gilt.

Vergleich der Lösungsvarianten für das Rucksackproblem:

- Laufzeit von Variante 1 mit $\textsc{Opt}(k, W)$ ist in $\boldsymbol{O}(KW)$.
- Laufzeit von Variante 2 mit $\textsc{Opt}(k, v)$ ist in $\boldsymbol{O}(KV)$. Für den maximalen Wert der Objekte $\bar{v} = \max_k(v_k)$ erhalten wir die grobere Abschätzung $\boldsymbol{O}(K^2\bar{v})$.

Analyse der dualen 0/1-Rucksack Lösung

- Nachtrag: Der Lösungswert steht nicht unbedingt am Ende der Tabelle in dem Eintrag (K, V), sondern er muss aus der Tabelle herausgesucht werden.
- Man sucht den größten Wert v, für den $OPT(K, v) \leq W$ gilt.

Vergleich der Lösungsvarianten für das Rucksackproblem:

- Laufzeit von Variante 1 mit $\textsc{Opt}(k, W)$ ist in $\boldsymbol{O}(KW)$.
- Laufzeit von Variante 2 mit $\textsc{Opt}(k, v)$ ist in $\boldsymbol{O}(KV)$. Für den maximalen Wert der Objekte $\bar{v} = \max_k(v_k)$ erhalten wir die grobere Abschätzung $\boldsymbol{O}(K^2\bar{v})$.
- Welche Variante effizienter ist, hängt von der Kapazitätsgrenze und der Größe der Werte ab.
- Die zweite Variante hat den Vorteil, dass sie die Grundlage für ein Approximationsschema liefert.

Approximationsschema

Approximationsschema

Ein **Approximationsschema** für ein Optimierungsproblem ist ein Algorithmus, der zu jeder Eingabe und jedem $\varepsilon > 0$ eine $(1+\varepsilon)$-Approximation liefert.

- Nun ist nicht nur interessant, wie sich die Laufzeit in Abhängigkeit von der Eingabe verhält, sondern auch, wie die Laufzeit von ε abhängt.
- Man nennt ein Approximationsschema ein **Approximationsschema mit vollständig polynomieller Laufzeit**, falls seine Laufzeit polynomiell in der Größe der Eingabe und in $\frac{1}{\varepsilon}$ ist.

Approximationsschema für das 0/1-Rucksack Problem

- Seien K Objekte mit Gewichten w_k und Werten v_k sowie eine Kapazitätsgrenze W gegeben. Sei $\bar{v} = \max_k(v_k)$ der maximale Wert eines Objektes.
- Zu der gegebenen Approximationsgüte $\varepsilon > 0$ definieren wir $E = \varepsilon \frac{\bar{v}}{K}$.
- Wir gehen davon aus, dass es keine Objekte mit einem Gewicht $w_k > W$ gibt. Dies ist keine Einschränkung, weil solche Objekt sowieso nicht ausgewählt werden könnten.
- Wir benutzen E als Skalierungsfaktor und berechnen neue Werte $v'_k = \lfloor \frac{v_k}{E} \rfloor = \lfloor \frac{K}{\varepsilon} \frac{v_k}{\bar{v}} \rfloor$. Die skalierten Werte liegen also zwischen 0 und $\frac{K}{\varepsilon}$.

Approximationsschema für das 0/1-Rucksack Problem

- Seien K Objekte mit Gewichten w_k und Werten v_k sowie eine Kapazitätsgrenze W gegeben. Sei $\bar{v} = \max_k(v_k)$ der maximale Wert eines Objektes.
- Zu der gegebenen Approximationsgüte $\varepsilon > 0$ definieren wir $E = \varepsilon \frac{\bar{v}}{K}$.
- Wir gehen davon aus, dass es keine Objekte mit einem Gewicht $w_k > W$ gibt. Dies ist keine Einschränkung, weil solche Objekt sowieso nicht ausgewählt werden könnten.
- Wir benutzen E als Skalierungsfaktor und berechnen neue Werte $v'_k = \lfloor \frac{v_k}{E} \rfloor = \lfloor \frac{K}{\varepsilon} \frac{v_k}{\bar{v}} \rfloor$. Die skalierten Werte liegen also zwischen 0 und $\frac{K}{\varepsilon}$.
- Nun wird der zuvor skizzierte Algorithmus auf das Problem mit den skalierten Werten v'_k angewendet.

Approximationsschema für den 0/1-Rucksack

Der oben beschriebene Algorithmus ist ein Approximationsschema mit vollständig polynomieller Laufzeit.

Approximationsgüte

- Sei S eine optimale Lösung für das Originalproblem die Wert $V^* = \sum_{k \in S} v_k$ erzielt und S' eine optimale Lösung für die skalierten Werte v'_k (vom Approx.-Alg.).
- Da S' optimal für die skalierten Werte v'_k ist, erhalten wir:

$$\sum_{k \in S'} v'_k \geq \sum_{k \in S} v'_k = \sum_{k \in S} \lfloor \frac{v_k}{E} \rfloor \geq \sum_{k \in S} (\frac{v_k}{E} - 1) \geq \frac{V^*}{E} - K$$

Approximationsgüte

- Sei S eine optimale Lösung für das Originalproblem die Wert $V^* = \sum_{k \in S} v_k$ erzielt und S' eine optimale Lösung für die skalierten Werte v'_k (vom Approx.-Alg.).
- Da S' optimal für die skalierten Werte v'_k ist, erhalten wir:

$$\sum_{k \in S'} v'_k \;\geq\; \sum_{k \in S} v'_k \;=\; \sum_{k \in S} \lfloor \frac{v_k}{E} \rfloor \;\geq\; \sum_{k \in S} (\frac{v_k}{E} - 1) \;\geq\; \frac{V^*}{E} - K$$

- Nun können wir die Werte der Lösung des Approximations-Algorithmus bezogen auf die Originalwerte abschätzen:

$$\sum_{k \in S'} v_k \;\geq\; \sum_{k \in S'} v'_k E \;\geq\; (\frac{V^*}{E} - K)E \;=\; V^* - KE \;\geq\; V^*(1 - \varepsilon)$$

wobei die letzte Ungleichung mit $\bar{v} \leq V^*$ aus $E \;=\; \varepsilon \frac{\bar{v}}{K} \;\leq\; \varepsilon \frac{V^*}{K}$ folgt.

Approximationsgüte

- Sei S eine optimale Lösung für das Originalproblem die Wert $V^* = \sum_{k \in S} v_k$ erzielt und S' eine optimale Lösung für die skalierten Werte v'_k (vom Approx.-Alg.).
- Da S' optimal für die skalierten Werte v'_k ist, erhalten wir:

$$\sum_{k \in S'} v'_k \;\geq\; \sum_{k \in S} v'_k \;=\; \sum_{k \in S} \lfloor \frac{v_k}{E} \rfloor \;\geq\; \sum_{k \in S} (\frac{v_k}{E} - 1) \;\geq\; \frac{V^*}{E} - K$$

- Nun können wir die Werte der Lösung des Approximations-Algorithmus bezogen auf die Originalwerte abschätzen:

$$\sum_{k \in S'} v_k \;\geq\; \sum_{k \in S'} v'_k E \;\geq\; (\frac{V^*}{E} - K)E \;=\; V^* - KE \;\geq\; V^*(1 - \varepsilon)$$

  wobei die letzte Ungleichung mit $\bar{v} \leq V^*$ aus $E \;=\; \varepsilon \frac{\bar{v}}{K} \;\leq\; \varepsilon \frac{V^*}{K}$ folgt.

- Laufzeit: Die Originaltabelle hat die Größe $K \times V \;\leq\; K \times K\bar{v}$, also ist skalierte Tabelle kleiner oder gleich $K \times \frac{K\bar{v}}{E} \;=\; K \times \frac{K\bar{v}K}{\varepsilon\bar{v}} \;=\; K \times \frac{K^2}{\varepsilon}$.
- Somit ist die Laufzeit in $\boldsymbol{O}(\frac{K^3}{\varepsilon})$.

Resümee zu Approximativen Algorithmen

- Bei der Approximation von NP-vollständigen Optimierungsproblemen kann folgendes passieren:

1. Das Problem lässt sich überhaupt nicht approximieren, egal wie lax die Approximationsgüte angesetzt wird. Beispiel: TSP ohne Dreiecksungleichung
2. Das Problem lässt sich approximieren, aber nur bis zu einer gewissen Grenze. Beispiel: TSP mit Dreiecksungleichung
3. Das Problem kann beliebig gut approximiert werden. Beispiel: 0/1-Rucksack

Resümee zu Approximativen Algorithmen

- Bei der Approximation von NP-vollständigen Optimierungsproblemen kann folgendes passieren:

1. Das Problem lässt sich überhaupt nicht approximieren, egal wie lax die Approximationsgüte angesetzt wird. Beispiel: TSP ohne Dreiecksungleichung
2. Das Problem lässt sich approximieren, aber nur bis zu einer gewissen Grenze. Beispiel: TSP mit Dreiecksungleichung
3. Das Problem kann beliebig gut approximiert werden. Beispiel: 0/1-Rucksack

- Bei dem letzten Punkt unterscheidet man noch danach, wie stark die Approximationsgüte die Laufzeit beeinträchtigt.
- Im guten Fall ist die Laufzeit polynomiell in Eingabegröße und $\frac{1}{\varepsilon}$.

Executive Summary

- **Heuristische Algorithmen** explorieren den Suchraum zunächst in eine Richtung, in der die Lösung erwartet wird: Schnelle Laufzeit bei passender Heuristik, keine Verbesserung im *worst-case*.
- Teilweise suboptimale Lösungen, ohne Garantie, wie nah die gefundene Lösung am Optimum ist.

Executive Summary

- **Heuristische Algorithmen** explorieren den Suchraum zunächst in eine Richtung, in der die Lösung erwartet wird: Schnelle Laufzeit bei passender Heuristik, keine Verbesserung im *worst-case*.
- Teilweise suboptimale Lösungen, ohne Garantie, wie nah die gefundene Lösung am Optimum ist.
- **Approximative Algorithmen** geben eine Garantie, wie nah sie an die optimale Lösung kommen.
- Effizienter Ansatz für NP-vollständige Probleme. Allerdings können manche NP-vollständigen Probleme nicht effizient approximiert werden.

Anhang

Inhalt des Anhangs:

- Approximative Variante von A*: S. 48

Approximative Variante von A*

- Man kann den A* Algorithmus beschleunigen, wenn man suboptimale Lösungen in Kauf nimmt.
- Eine einfache Variante ist der **Gewichtete A*** Algorithmus (*weighted A**).
- Dabei wird eine gegebene konsistente Heuristik h mit einem konstanten Faktor $\varepsilon > 0$ multipliziert.
- Dadurch werden wenig Knoten besucht, also terminiert der Algorithmus schneller.
- Der gefundene Pfad ist maximal um den Faktor ε länger als der optimale Pfad.
- Durch die Skalierung verliert die Heuristik meist ihre Konsistenz. Daher kann der oben genannte Pseudocode nicht verwendet werden.

Literatur I

Generell:

- Schöning U. *Algorithmik (Spektrum Lehrbuch).* Spektrum Akademischer Verlag; 2001. ISBN: 978-3827410924
- Ottmann T & Widmayer P. *Algorithmen und Datenstrukturen.* Springer Verlag, 5. Auflage; 2011. ISBN: 978-3827428042
- Cormen TH, Leiserson CE, Rivest R, Stein C. *Algorithmen - Eine Einführung.* De Gruyter Oldenbourg, 4. Auflage; 2013. ISBN: 978-3486748611
- Skiena S. *The Algorithm Design Manual.* Springer; Auflage: 2nd ed. 2008. ISBN: 978-1848000698
- Dasgupta S, Papadimitriou CH, Vazirani UV. *Algorithms.* McGraw-Hill Higher Education; 2008. ISBN: 978-0073523408

Anderes Vorlesungsmaterial:

Literatur II

- Röglin H. *Skript zur Vorlesung Randomisierte und Approximative Algorithmen*, Universität Bonn, `http://www.roeglin.org/teaching/WS2011/RandomisierteAlgorithmen/RandomisierteAlgorithmen.pdf`

Originalveröffentlichungen:

- Karpinski M, Lampis M, Schmied R. *New inapproximability bounds for TSP*. Journal of Computer and System Sciences. 2015 Dec 1;81(8):1665-77.

Index

A* Algorithmus, 10
A-star, 10
A-Stern, 10
Approximationsschema, 42
  mit vollständig polynomieller Laufzeit, 42
Approximativer Algorithmus, 29

backward cost, 10
Best-First Algorithmus, 7

Farthest Insertion:, 28
forward cost, 10

Graph
  vollständiger, 27

Heuristik, 2
  konsistent, 13, 14
  zulässig, 12
Heuristische Algorithmen, 2

Konsistente Heuristik, 13, 14

Nearest Insertion:, 28

Problem des Handlungsreisenden, 27

Rückwärtskosten, 10
ρ-Approximationsalgorithmus, 29

Travelling Salesman Problem
  approximativer Ansatz, 30
  heuristischer Ansatz, 27
Travelling Salesman Problem, 27

Vorwärtskosten, 10

*weighted A**, 48

Zulässige Heuristik, 12