Algorithmen und Datenstrukturen

Vorlesung #11 – Heuristische und Approximative Algorithmen

Benjamin Blankertz

Lehrstuhl für Neurotechnologie, TU Berlin

benjamin.blankertz@tu-berlin.de

20 · Jun · 2023

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)
- Lehrevaluation
- 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.
- 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 anhand von systematischen experimentellen Algorithmenanalysen belegt.

Heuristische Algorithmen

- Heuristischen Verfahren folgen meist einem der folgenden Ansätze:

1. Eine Lösung sukzessiv einer Heuristik folgend von null aufbauen

2. 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.

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.
- **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*.

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 über freien Flächen ohne vorgegebene Wege.
- 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

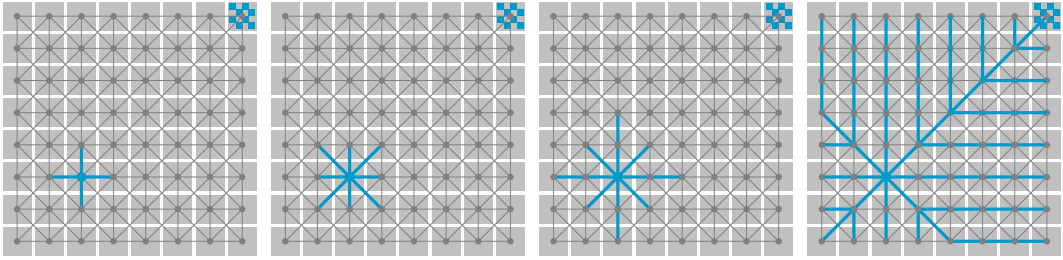
Ziel

Finde den kürzesten Weg von einem Startknoten s zu einem Zielknoten t in einem gewichteten Graphen.

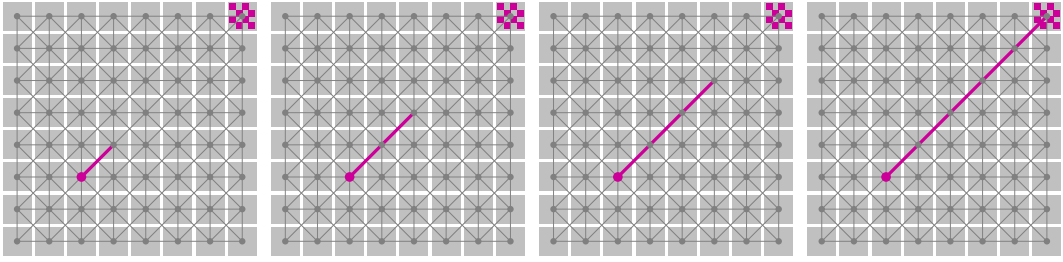
- Mit dem Dijkstra SSSP-Algorithmus:
- Stoppe Suche, sobald der Zielknoten aus der Warteschlange kommt.
- Diese Suche hat keine Ausrichtung auf den Zielknoten: nicht effizient.
- Mit einer Heuristik $h(v)$, die den Abstand jedes Knotens zum Ziel schätzt, können wir vielversprechende Knoten zuerst explorieren.
- Für Landkarten ist z.B. die Luftliniendistanz eine solche Schätzung.

Dijkstra durch Heuristik verbessern

Dijkstra



Ideal (mit welcher Heuristik?)



Der kürzeste Weg von s nach t einer Heuristik folgend

- Der (Greedy) **Best-First Algorithmus** funktioniert ähnlich wie Breitensuche.
- Grundlage: Heuristik zur Schätzung des Abstands zum Ziel für jeden Knoten
- Ausgehend von s werden kreuzende Kanten benutzt.
- Wähle Endknoten mit kleinstem Abstand zum Ziel (gemäß Heuristik)
- Greedy Auswahl!

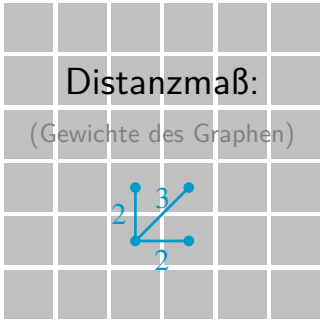
Definition Kosten eines Pfads

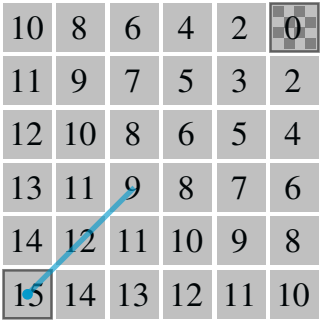
Für einen Pfad p bezeichnen wir mit $c(p)$ die **Kosten des Pfades** (bzw. die Pfadlänge), also die Summe der Gewichte seiner Kanten.

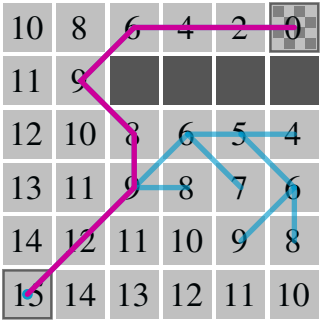
Pseudocode für den BEST-FIRST Algorithmus

```
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
```

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: von einem Knoten zum Ziel, geschätzt duch die Heuristik $h(v)$ mit den
  - Rückwärtskosten: vom Start zum einem Knoten.

- Rückwärtskosten sind $dist[v]$ analog zu Dijkstra.

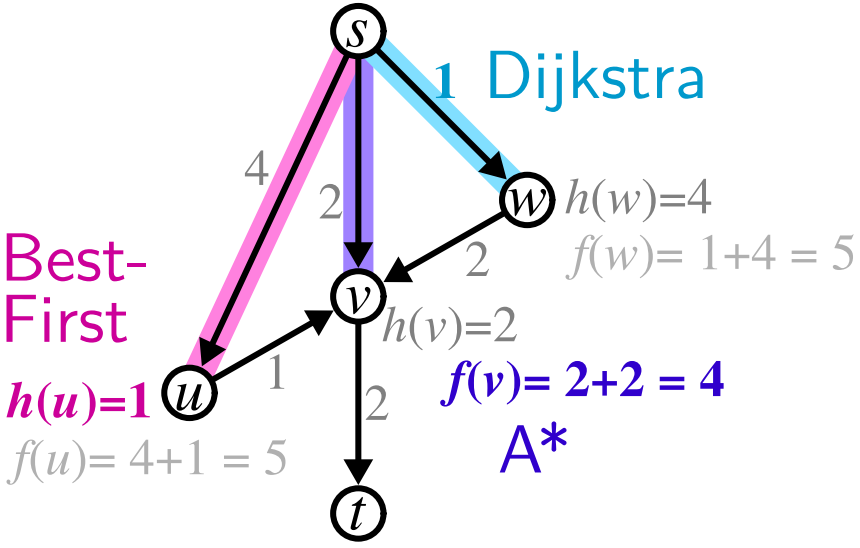
- **Ablauf:**

- Ausgehend von s werden kreuzende Kanten benutzt.

- Wähle Endknoten v mit kleinster geschätzter Länge des Weges von s nach t über v

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

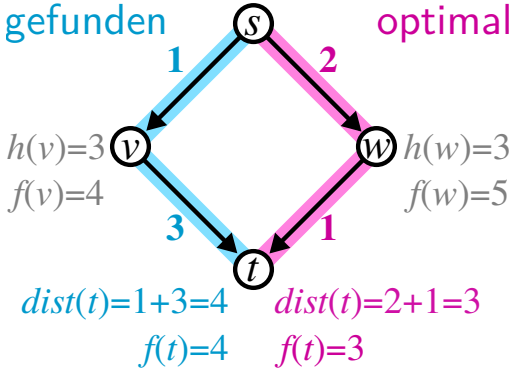
Vergleich der Kantenauswahl zwischen Dijkstra, Best-First und A*



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

A* konkretisieren – Zulässige Heuristik

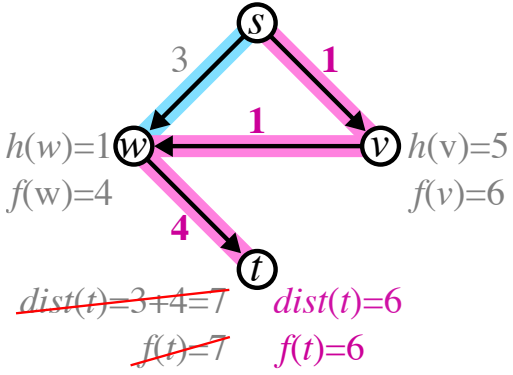
- Offensichtlich sind nur bestimme Funktionen als Heuristik hilfreich.
- Im folgenden Beispiel führt eine ungünstige Heuristik die A* Suche in die Irre:



- 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 effizient sein. Daher soll jede Kante wie bei Dijkstra nur einmal betrachtet werden:
- Wird ein Knoten der PQ entnommen, 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, trotz $g(v, w) = 1$.
- Konsistente Heuristiken (siehe nächste Seite) garantieren effiziente Laufzeit.

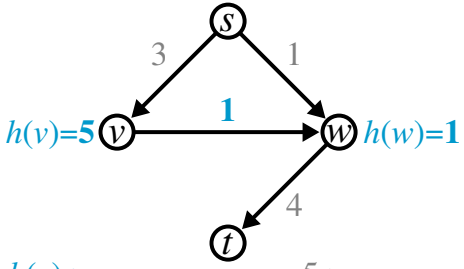
Konsistente Heuristik

- 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.

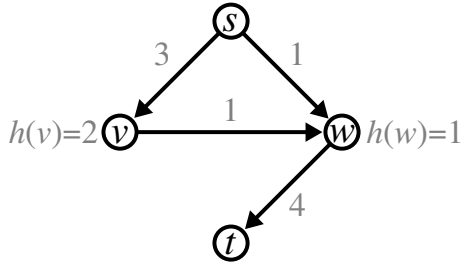
Nicht konsistente Heuristik



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

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

Konsistente Heuristik



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 $c(p)$.
- Z.z.: $h(v_0) \leq c(p)$ für beliebige Pfade p von v_0 nach t.
- Sei also ein Pfad $p = v_0, \ldots, v_k$ mit $v_k = t$ gegeben.

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

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 $v \to w$.
- 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.
- 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

Unterschied zu Dijkstra: Bei A* wird `dist[v] + h(v)` an Stelle von `dist[v]` als Priorität verwendet.

```
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 22, 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

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

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

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

- Damit ist das Lemma bewiesen. □

Korrektheit von A*

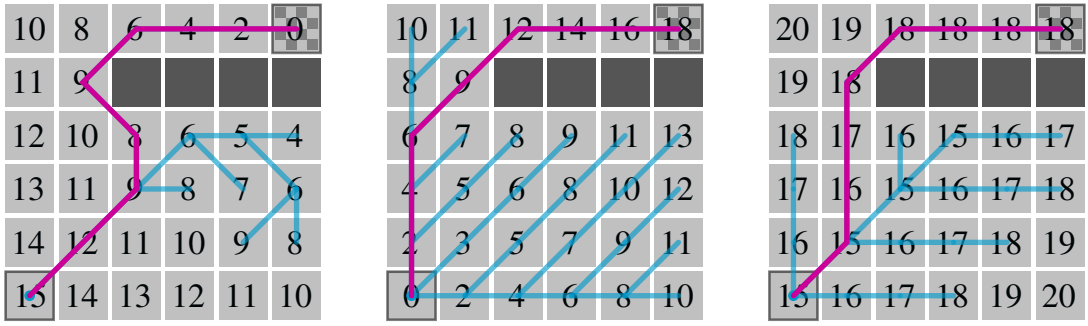
Beweis der Korrektheit.

- 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 $v \rightsquigarrow t$.

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

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

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.
- 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     psolnext ← psol with performed move
11     add(Q, psolnext, psolnext.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. In diesem Fall wird der Weg über die neue Kante gespeichert.
- 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 mindern.
- Abhilfe durch Verwendung von *Hash Sets* (nächste Vorlesung)

Lehrevaluation: Bitte beteiligen Sie sich alle!

AlgoDat Vorlesung
(ohne HA!)

AlgoDat Übung
(Rechnerübung, Online-Tutorien)