Einführung in die Methoden der Künstlichen Intelligenz

Informierte Suche

PD Dr. David Sabel

SoSe 2014

Stand der Folien: 29. Mai 2023

Informierte Suche

Die Suche nennt man **informiert**, wenn (zusätzlich) eine Bewertung aller Knoten des Suchraumes angegeben werden kann.

Knotenbewertung

- Schätzfunktion
- Ähnlichkeit zum Zielknoten oder auch
- Schätzung des Abstands zum Zielknoten
- Bewertung des Zielknotens: Sollte Maximum / Minimum der Schätzfunktion sein

Heuristische Suche

Eine Heuristik (Daumenregel) ist eine Schätzfunktion, die in vielen praktischen Fällen, die richtige Richtung zum Ziel angibt.

Suchproblem ist äquivalent zu:

Minimierung (bzw. Maximierung) einer Knotenbewertung (einer Funktion) auf einem (implizit gegebenen) gerichteten Graphen

Variante:
Maximierung in einer Menge oder in einem n-dimensionaler Raum.

Beispiel: 8-Puzzle

Start:

8		1
6	5	4
7	2	3

Ziel:

1	2	3
4	5	6
7	8	

Bewertungsfunktionen (Beispiele):

1. $f_1()$ Anzahl der Plättchen an der falschen Stelle
2. $f_2()$ Anzahl der Züge (ohne Behinderungen zu beachten), die man braucht, um Endzustand zu erreichen.

Beispiel: 8-Puzzle (2)

$S_1 =$

2	3	1
8	7	6
	5	4

$f_1(S_1) = 7$
$f_2(S_1) = 2 + 1 + 1 + 3 + 1 + 0 + 2 + 2 = 12$

$S_2 =$

2	3	1
	7	6
8	5	4

$f_1(S_2) = 7$
$f_2(S_2) = 2 + 1 + 1 + 3 + 1 + 0 + 2 + 1 = 11$

$\Rightarrow f_2$ ist genauer.

Bergsteigerprozedur (Hill-climbing)

- Auch als Gradientenaufstieg bekannt
- Gradient: Richtung der Vergrößerung einer Funktion (Berechnung durch Differenzieren)

Parameter der Bergsteigerprozedur

- Menge der initialen Knoten
- Nachfolgerfunktion (Nachbarschaftsrelation)
- Bewertungsfunktion der Knoten, wobei wir annehmen, dass Zielknoten maximale Werte haben (Minimierung erfolgt analog)
- Zieltest

Bergsteigen

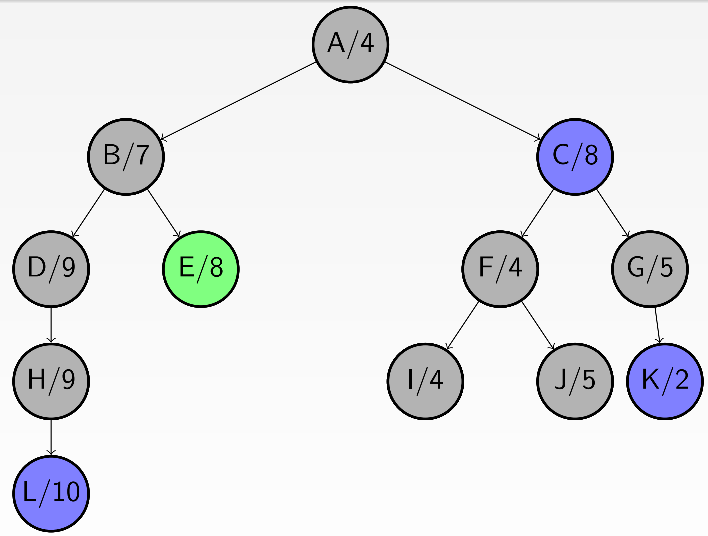
Algorithmus **Bergsteigen**

Datenstrukturen: L : Liste von Knoten, markiert mit Weg dorthin
h sei die Bewertungsfunktion der Knoten
Eingabe: L sei die Liste der initialen Knoten, absteigend sortiert entsprechend h

Algorithmus:

1. Sei K das erste Element von L und R die Restliste
2. Wenn K ein Zielknoten, dann stoppe und gebe K markiert mit dem Weg zurück
3. Sortiere die Liste $NF(K)$ absteigend entsprechend h und entferne schon besuchte Knoten aus dem Ergebnis. Sei dies die Liste L'.
4. Setze $L := L'$++R und gehe zu **??**.

Beispiel Bergsteigen



L = [A]L = [C,B] ++ [] = [C,B]L = [G,F] ++ [B] = [G,F,B]L = [K] ++ [F,B] = [K,F,B]L = [] ++ [F,B] = [F,B]L = [J,I] ++ [B] = [J,I,B]L = [] ++ [I,B] = [I,B]L = [] ++ [B] = [B]L = [D,E] ++ [] =

Eigenschaften der Bergsteigerprozedur

- Entspricht einer gesteuerten Tiefensuche mit Sharing
- daher nicht-vollständig
- Platzbedarf ist durch die Speicherung der besuchten Knoten exponentiell in der Tiefe.

Varianten

- Optimierung einer Funktion ohne Zieltest:
- Bergsteige ohne Stack, stets zum nächst höheren Knoten
- Wenn nur noch Abstiege möglich sind, stoppe und gebe aktuellen Knoten aus
- Findet lokales Maximum, aber nicht notwendigerweise globales

Hillclimbing in Haskell

```
hillclimbing cmp heuristic goal successor start =
 let -- sortiere die Startknoten
   list = map (\k -> (k,[k])) (sortByHeuristic start)
 in go list []
  where
   go ((k,path):r) mem
    | goal k    = Just (k,path) -- Zielknoten erreicht
    | otherwise =
       let -- Berechne die Nachfolger (nur neue Knoten)
           nf = (successor k) \\ mem
           -- Sortiere die Nachfolger entsprechend der Heuristik
           l’ = map (\k -> (k,k:path)) (sortByHeuristic nf)
       in go  (l’ ++ r) (k:mem)
   sortByHeuristic = sortBy (\a b  -> cmp (heuristic a) (heuristic b))
```


Best-First-Suche

- Ähnlich zum Hillclimbing, aber:
- Wählte stets als nächsten zu expandierenden Knoten, den mit dem besten Wert
- Änderung im Algorithmus: sortiere alle Knoten auf dem Stack

Best-First-Suche

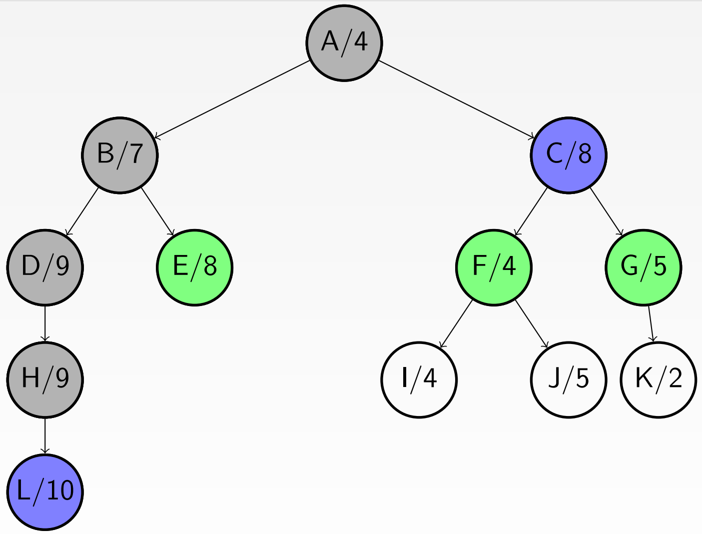
Algorithmus **Best-First Search**

Datenstrukturen:
Sei L Liste von Knoten, markiert mit dem Weg dorthin.
h sei die Bewertungsfunktion der Knoten

Eingabe: L Liste der initialen Knoten, sortiert, so dass die besseren Knoten vorne sind.
Algorithmus:

1. Wenn L leer ist, dann breche ab
2. Sei K der erste Knoten von L und R die Restliste.
3. Wenn K ein Zielknoten ist, dann gebe K und den Weg dahin aus.
4. Sei $N(K)$ die Liste der Nachfolger von K. Entferne aus $N(K)$ die bereits im Weg besuchten Knoten mit Ergebnis $\mathcal{N}$
5. Setze $L := \mathcal{N}$ ++ R
6. Sortiere L, so dass bessere Knoten vorne sind und gehe zu **??**.

Beispiel Best-First-Suche



L = [A]L = sort ([C,B] ++ []) = [C,B]L = sort ([G,F] ++ [B]) = [B,G,F]L = sort ([D,E] ++ [G,F]) = [D,E,G,F]L = sort ([H,E] ++ [G,F] = [H,E,G,F]L = sort ([L] ++ [E,G,F] = [L,E,G,F]Zielknoten L gefunden

Best-First-Suche: Eigenschaften

- entspricht einer gesteuerten Tiefensuche
- daher unvollständig
- Platzbedarf ist durch die Speicherung der besuchten Knoten exponentiell in der Tiefe.
- Durch Betrachtung aller Knoten auf dem Stack können lokale Maxima schneller verlassen werden, als beim Hill-Climbing

Best-First-Suche in Haskell

```
bestFirstSearchMitSharing cmp heuristic goal successor start =
 let -- sortiere die Startknoten
     list = sortByHeuristic (map (\k -> (k,[k])) (start))
 in go list []
  where
   go ((k,path):r) mem
    | goal k    = Just (k,path) -- Zielknoten erreicht
    | otherwise =
       let -- Berechne die Nachfolger und nehme nur neue Knoten
           nf = (successor k) \\ mem
           -- aktualisiere Pfade
           l' = map (\k -> (k,k:path)) nf
           -- Sortiere alle Knoten nach der Heuristik
           l'' = sortByHeuristic (l' ++ r)
       in go l'' (k:mem)
   sortByHeuristic =
    sortBy (\(a,_) (b,_)-> cmp (heuristic a) (heuristic b))
```


Simulated Annealing

- Analogie zum Ausglühen: Am Anfang hohe Energie (Beweglichkeit), mit voranschreitender Zeit Abkühlung
- Suche dazu: Bei der Optimierung von n-dimensionalen Funktionen
- Ähnlich zum Bergsteigen, aber am Anfang große Sprünge (auch absteigend), später nur noch selten
- Erlaubt schnell aus lokalen Maxima rauszuspringen

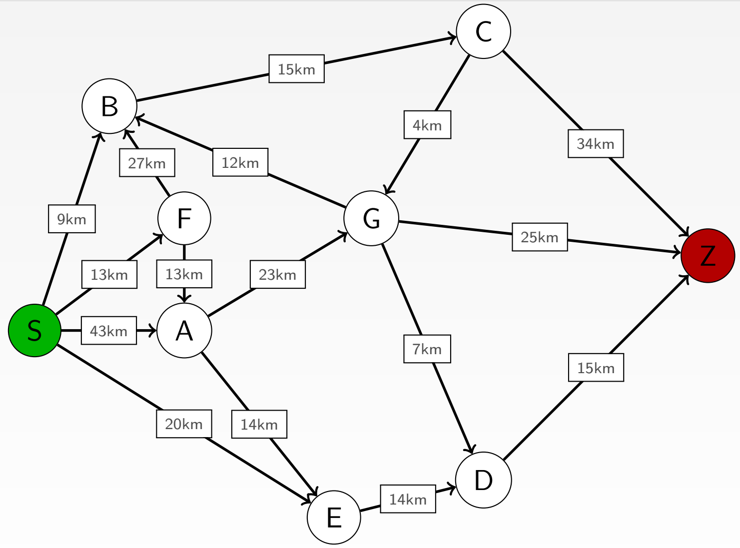
A^*-Algorithmus

Suchproblem

- Startknoten
- Zieltest Z
- Nachfolgerfunktion NF.
  Annahme: Es gibt nur eine Kante zwischen zwei Knoten.
  (Graph ist schlicht)
- Kantenkosten $g(N_1, N_2) \in \mathbb{R}$.
- Heuristik h schätzt Abstand zum Ziel

Ziel: Finde kostenminimalen Weg vom Startknoten zu einem Zielknoten

Beispiel: Routensuche



Heuristik z.B. Luftliniendistanz

Algorithmus A^***-Algorithmus**

Datenstrukturen:

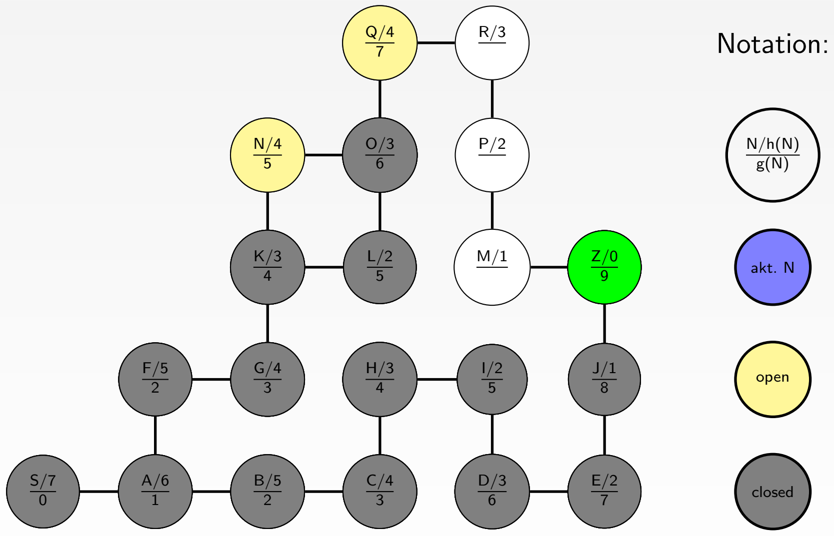
- Menge `Open` von Knoten
- Menge `Closed` von Knoten
- Wert $g(N)$ für jeden Knoten (markiert mit Pfad vom Start zu N)
- Heuristik h
- Zieltest Z
- Kantenkostenfunktion c

Eingabe:

- `Open` $:= \{S\}$, wenn S der Startknoten ist
- $g(S) := 0$, ansonsten ist g nicht initialisiert
- `Closed` $:= \emptyset$

Algorithmus:
repeat
  Wähle N aus `Open` mit minimalem $f(N) = g(N) + h(N)$
  if $Z(N)$ **then**
    break; // Schleife beenden
  else
    Berechne Liste der Nachfolger $\mathcal{N} := NF(N)$
    Schiebe Knoten N von `Open` nach `Closed`
    for $N' \in \mathcal{N}$ **do**
      if $N' \in$ `Open` $\cup$ `Closed` und $g(N) + c(N, N') > g(N')$ **then**
        skip // Knoten nicht verändern
      else
        g(N') := g(N) + c(N,N'); // neuer Minimalwert für g(N')
        Füge N' in `Open` ein und (falls vorhanden) lösche N' aus `Closed`;
      end-if
    end-for
  end-if
until `Open` $= \emptyset$
if `Open` $= \emptyset$ **then** Fehler, kein Zielknoten gefunden
else N ist der Zielknoten mit $g(N)$ als minimalen Kosten
end-if

Beispiel

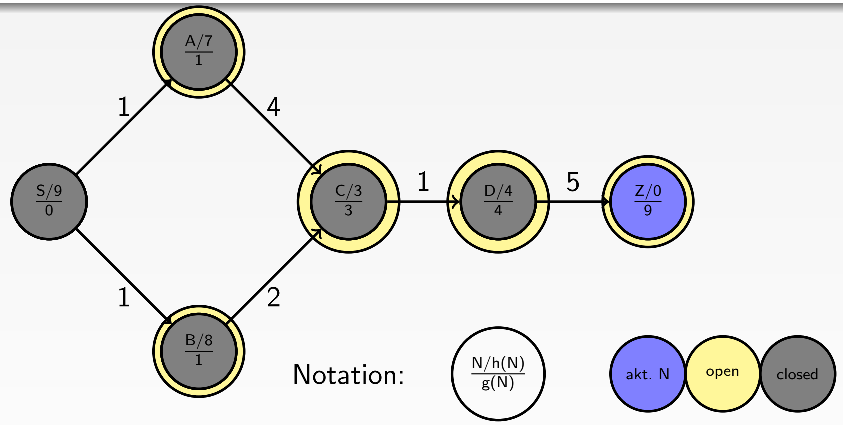


Heuristik: Rechteck-Norm $h(X) = |(y_X - y_Z)| + |x_X - x_Z|$

A^* in Haskell

```
-- Eintr"age in open / closed: (Knoten, (g(Knoten), Pfad zum Knoten))
aStern heuristic goal successor open closed
  | null open = Nothing -- Kein Ziel gefunden
  | otherwise =
     let n@(node,(g_node,path_node)) = Knoten mit min. f-Wert
                 minimumBy (\(a,(b,_)) (a',(b',_))
                    -> compare  ((heuristic a) + b) ((heuristic a') + b')) open
     in
      if goal node then Just n else  -- Zielknoten expandiert
       let nf = (successor node)   -- Nachfolger
           -- aktualisiere open und closed:
           (open',closed')  = update nf (delete n open) (n:closed)
           update [] o c = (o,c)
           update ((nfnode,c_node_nfnode):xs) o c =
            let (o',c') = update xs o c -- rekursiver Aufruf
                -- m"oglicher neuer Knoten, mit neuem g-Wert und Pfad
                newnode = (nfnode,(g_node + c_node_nfnode,path_node ++ [node]))
            in  case lookup nfnode open of -- Knoten in Open?
                 Nothing -> case lookup nfnode closed of -- Knoten in Closed?
                             Nothing -> (newnode:o',c')
                             Just (curr_g,curr_path) ->
                              if curr_g > g_node + c_node_nfnode
                               then (newnode:o',delete (nfnode,(curr_g,curr_path)) c')
                               else (o',c')
                 Just (curr_g,curr_path) ->
                  if curr_g > g_node + c_node_nfnode
                   then (newnode:(delete (nfnode,(curr_g,curr_path)) o'),c')
                   else  (o',c')
       in aStern heuristic goal successor open' closed'
```


Beispiel



$\texttt{Open} = \{S\}$ $\texttt{Closed} = \emptyset$ $N := S$

$\texttt{Open} = \{A, B\}$ $\texttt{Closed} = \{S\}$

$f(A) = 1 + 7 = 8$ $f(B) = 1 + 8 = 9$ $N := A$

$\texttt{Open} = \{B, C\}$ $\texttt{Closed} = \{A, S\}$

$f(B) = 1 + 8 = 9$ $f(C) = 5 + 3 = 8$ $N := C$

$\texttt{Open} = \{B, D\}$ $\texttt{Closed} = \{A, C, S\}$

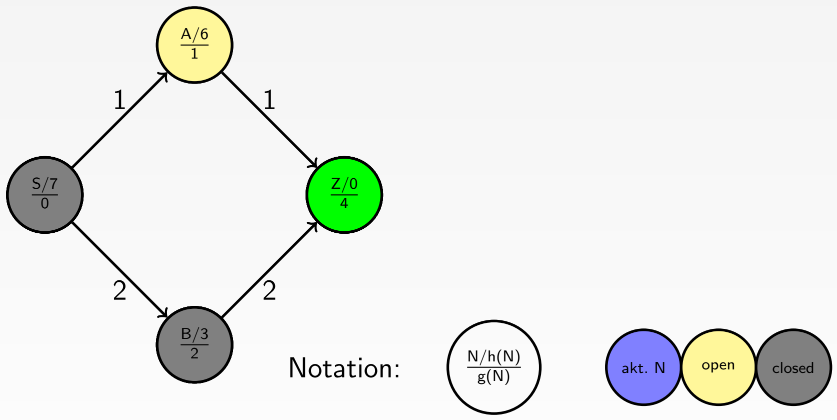
$f(B) = 1 + 8 = 9$ $f(D) = 6 + 4 = 10$ $N := B$

$\texttt{Open} = \{C, D\}$ $\texttt{Closed} = \{A, B, S\}$

Beispiel (2)

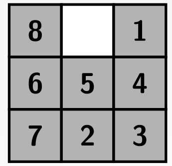
- Beispiel zeigt, dass i.A. notwendig: Knoten aus `Closed` wieder in `Open` einfügen
- Beispiel extra so gewählt!
- Beachte: Auch Kanten werden mehrfach betrachtet
- Mehr Anforderungen an die Heuristik verhindern das!

Ist A^* immer korrekt?



Nein! Die Heuristik muss **unterschätzend** sein!

Nachtrag zum 8-Puzzle



- Es gibt 9! = 362.880 verschiedene Zustände
- Davon ist die Hälfte (= 181.440) lösbar

Nachtrag zum 8-Puzzle (2)

- $h_1()$ Anzahl der Plättchen an der falschen Stelle
- $h_2()$ Anzahl der Züge (ohne Behinderungen zu beachten), die man braucht, um Endzustand zu erreichen.

Test mit $\sim$ 9500 Zuständen:

	h1	h2	Slowdown (Zeit h1/Zeit h2)
Mittelwert	54.35 min	1.46 min	38
Median	6.02 min	15 sec	21
Max	35.64 hrs	1.82 hrs	1307

Die Wahl der Heuristik ist wichtig!

Notationen für die Analyse

$g^*(N, N')$ = Kosten des optimalen Weges von N nach N'

$g^*(N)$ = Kosten des optimalen Weges vom Start bis zu N

$c^*(N)$ = Kosten des optimalen Weges von N bis zum nächsten Zielknoten Z.

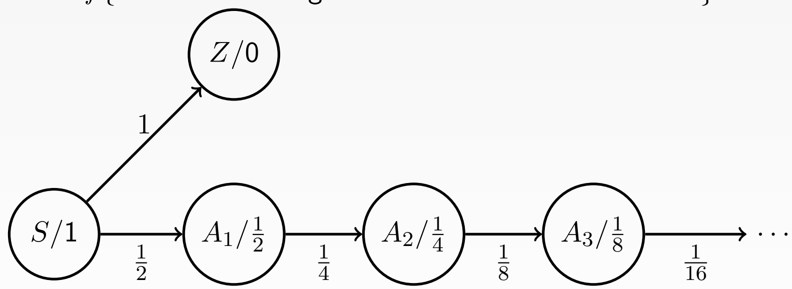
$f^*(N) = g^*(N) + c^*(N)$
(Kosten des optimalen Weges durch N bis zu einem Ziel Z)

Voraussetzungen für den A^*-Algorithmus

1. es gibt nur endlich viele Knoten N mit $g^*(N) + h(N) \leq d$, wobei $d = \mathit{inf}\{\text{Kosten aller Wege von } S \text{ zu einem Zielknoten } Z\}$.
2. Für jeden Knoten N gilt: $h(N) \leq c^*(N)$, d.h. die Schätzfunktion ist unterschätzend.
3. Für jeden Knoten N ist die Anzahl der Nachfolger endlich.
4. Alle Kanten kosten etwas: $c(N, N') > 0$ für alle N, N'.
5. Der Graph ist schlicht, d.h. zwischen zwei Knoten gibt es höchstens eine Kante

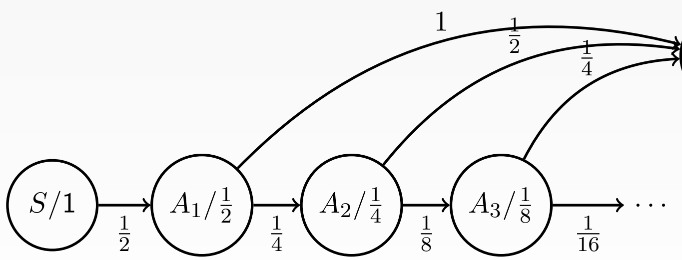
Bedingung 1 ist notwendig: Beispiele

Bedingung 1: es gibt nur endlich viele Knoten N mit $g^*(N) + h(N) \leq d$, wobei $d = \mathit{inf}\{\text{Kosten aller Wege von } S \text{ zu einem Zielknoten } Z\}$.



Bedingung 1 ist notwendig: Beispiele (2)

Bedingung 1: es gibt nur endlich viele Knoten N mit $g^*(N) + h(N) \leq d$, wobei $d = \mathit{inf}\{\text{Kosten aller Wege von } S \text{ zu einem Zielknoten } Z\}$.
zum Infimum d muss es nicht notwendigerweise auch einen endlichen Weg im Suchgraphen geben:



Bedingung 1: hinreichende Bedingungen

Sei $\varepsilon > 0$ fest.

Wenn für alle Kosten $c(N_1, N_2)$ gilt: $c(N_1, N_2) \geq \varepsilon$ und jeder Knoten hat nur endlich viele Nachfolger und h ist unterschätzend,

dann gilt auch Bedingung 1.

Korrektheit und Vollständigkeit der A^*-Suche

Wenn Voraussetzungen für den A^*-Algorithmus erfüllt, dann existiert zum Infimum d stets ein endlicher Weg mit Kosten

Notation:
$infWeg(N) := inf\{$Kosten aller Wege von N zu einem Ziel$\}$

Satz

Es existiere ein Weg vom Start S bis zu einem Zielknoten. Sei $d = infWeg(S)$. Die Voraussetzungen für den A^*-Algorithmus seien erfüllt. Dann existiert ein optimaler Weg von S zum Ziel mit Kosten d.

Korrektheit und Vollständigkeit der A^*-Suche (2)

Ein optimaler Knoten ist stets in `Open`:

Lemma

Die Voraussetzungen zum A^*-Algorithmus seien erfüllt. Es existiere ein optimaler Weg $S = K_0 \to K_1 \to K_2 \to \ldots \to K_n = Z$ vom Startknoten S bis zu einem Zielknoten Z . Dann ist während der Ausführung des A^*-Algorithmus stets ein Knoten K_i in `Open`, markiert mit $g(K_i) = g^*(K_i)$, d.h. mit einem optimalen Weg von S zu K_i.

Korrektheit und Vollständigkeit der A^*-Suche (3)

Expandierte Zielknoten sind optimal:

Lemma

Wenn die Voraussetzung für den A^*-Algorithmus erfüllt sind, gilt: Wenn A^* einen Zielknoten expandiert, dann ist dieser optimal.

Ein Zielknoten wird nach endlicher Zeit expandiert:

Lemma

Die Voraussetzungen zum A^*-Algorithmus seien erfüllt. Wenn ein Weg vom Start zum Zielknoten existiert gilt: Der A^*-Algorithmus expandiert einen Zielknoten nach endlich vielen Schritten.

Korrektheit und Vollständigkeit der A^*-Suche (4)

Zusammenfassend ergibt sich:

Theorem

Es existiere ein Weg vom Start bis zu einem Zielknoten. Die Voraussetzungen zum A^*-Algorithmus seien erfüllt. Dann findet der A^*-Algorithmus einen optimalen Weg zu einem Zielknoten.

Spezialfälle

- Wenn $h(N) = 0$ für alle Knoten N, dann ist A^*-Algorithmus dasselbe wie die sogenannte Gleiche-Kosten-Suche
- Wenn $c(N_1, N_2) = k$ für alle Knoten N_1, N_2 und $h(N) = 0$ für alle Knoten N, dann ist A^*-Algorithmus gerade die Breitensuche.

Variante: A^{*o}-Algorithmus

A^{*o}-Algorithmus

- findet alle optimalen Wege
- Abänderung am A^*-Algorithmus:
  sobald erster Zielknoten mit Wert d expandiert wurde:
  - Füge in `Open` nur noch Knoten mit $g(N) + h(N) \leq d$ ein
  - Andere Knoten kommen in `Closed`
- Stoppe erst, wenn `Open` leer ist

Theorem

Wenn die Voraussetzungen für den A^*-Algorithmus gelten, dann findet der Algorithmus A^{*o} alle optimalen Wege von S zum Ziel.

Schätzfunktionen

Definition

Wenn man zwei Schätzfunktionen h_1 und h_2 hat mit:

1. h_1 und h_2 unterschätzen den Aufwand zum Ziel
2. für alle Knoten N gilt: $h_1(N) \leq h_2(N) \leq c^*(N)$

Dann nennt man h_2 besser informiert als h_1.

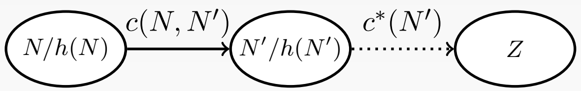
Hieraus alleine kann man noch nicht folgern, dass der A^*-Algorithmus zu h_2 sich besser verhält als zu h_1.
Notwendig ist:
Die Abweichung bei Sortierung der Knoten mittels f muss klein sein. D.h. optimal wäre $f(k) \leq f(k') \Leftrightarrow f^*(k) \leq f^*(k')$.

Monotone Schätzfunktionen

Definition

Eine Schätzfunktion $h(.)$ ist **monoton**, gdw.

- $h(N) \leq c(N, N') + h(N')$ für alle Knoten N und Nachfolger N'
- $h(Z) = 0$ für alle Zielknoten Z.

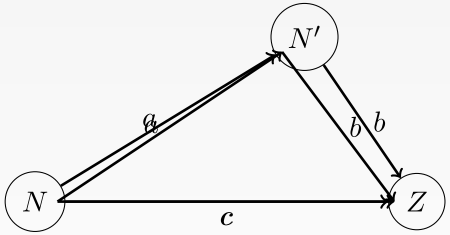


Satz: Eine monotone Schätzfunktion ist auch unterschätzend.

Beweis: Induktion über die Entfernung jedes Knotens N vom Ziel

- Wenn von N aus kein Weg zum Ziel, dann $h(N)$ immer untersch.
- Wenn N ein Zielknoten ist, dann gilt $h(N) = 0$
- Induktionsschritt: Sei $N \to N'$ der optimale Präfix des Weges von N zum Ziel,
  Dann gilt: $c^*(N) = c(N, N') + c^*(N')$.
  Induktionsannahme: $h(N') \leq c^*(N')$ (da $h(N')$ unterschätzend),
  $h(N) \underbrace{\leq}_{\text{Monotonie}} c(N, N') + h(N') \underbrace{\leq}_{\text{Induktionsannahme}} c(N, N') + c^*(N') = c^*(N)$

Montonie (2)



Dreiecksungleichung: $c \leq a + b$

Monotonie (3)

Satz

Ist die Schätzfunktion h monoton, so expandiert der A^*-Algorithmus jeden untersuchten Knoten beim ersten mal bereits mit dem optimalen Wert. D.h. $g(N) = g^*(N)$ für alle expandierten Knoten.

Variante: A^* als Baumsuche

- Aktualisiere nie die $g(N)$ Werte
- Verwende keine Closed-Liste
- Gleiche Knoten kommen evtl. mehrfach in Open vor, aber mit anderen Wegen
- Platzersparnis
- Optimalität: Nur wenn h monoton ist

Weitere Folgerungen aus der Monotonie

Satz

Wenn $h(.)$ monton ist gilt:

1. Wenn N später als M expandiert wurde, dann gilt $f(N) \geq f(M)$.
2. Wenn N expandiert wurde, dann gilt $g^*(N) + h(N) \leq d$ wobei d der optimale Wert ist.
3. Jeder Knoten mit $g^*(N) + h(N) \leq d$ wird von A^{*o} expandiert.

Weitere Folgerungen aus der Monotonie (2)

Theorem

Wenn eine monotone Schätzfunktion gegeben ist, die Schätzfunktion in konstanter Zeit berechnet werden kann, dann läuft der A^*-Algorithmus in Zeit $O(|D|)$, wenn $D = \{N \mid g^*(N) + h(N) \leq d\}$.
Bei konstanter Verzweigungsrate c, und d als Wert des optimalen Weges und δ als der kleinste Abstand zweier Knoten, dann ist die Komplexität $O(c^{\frac{d}{\delta}})$.

Weitere Folgerungen aus der Monotonie (3)

Satz

- Voraussetzungen für A^*-Algorithmus erfüllt
- d die Kosten des optimalen Weges
- h_2 besser informiert als h_1
- h_1, h_2 monoton
- für i=1,2: A_i der A^{*o}-Algorithmus zu h_i

Dann: Alle Knoten N mit $g^*(N) + h_2(N) \leq d$ die von A_2 expandiert werden, werden auch von A_1 expandiert.

Fazit

- Monotone Schätzfunktion wünschenswert
- Besser informierte Schätzfunktion wünschenswert
- Für montone Heuristik ist A^* optimal

Problem: A^* verbraucht zuviel Platz (alle besuchten Knoten)

Varianten des A^*-Algorithmus

IDA* (Iterative Deepening A^*) mit Grenze d

- Ist analog zu A^*.
- es gibt keine `Open`/`Closed`-Listen, nur einen Stack mit Knoten und Wegekosten.
- der Wert $g(N)$ wird bei gerichteten Graphen nicht per Update verbessert.
- Der Knoten N wird nicht expandiert, wenn $f(N) > d$.
- das Minimum der Werte $f(N)$ mit $f(N) > d$ wird das d in der nächsten Iteration.

Platz: Linear in der Länge des optimalen Weges
Problem: Durch Nicht-Speichern der entdeckten Knoten: Eventuell exponentiell viele Pfade ablaufen (Zeit)

Varianten des A^*-Algorithmus (2)

SMA* (Simplified Memory Bounded A^*)

- Wie A^*, aber die Größe der `Open` und `Closed`-Mengen ist beschänkt
- Wenn der Platz verbraucht ist, wird der schlechteste Knoten gelöscht
- Schlechtester Knoten: Größter $f(N)$-Wert.