

1 AI101-03 Uninformed and Informed Search

1.1 Einführung und Problemdefinition

Problemlösende Agenten sind zielbasierte Agenten, die atomare Repräsentationen verwenden (Zustände als Black Boxes). Der Prozess besteht aus vier Phasen:

1. **Zielformulierung:** Definition des Ziels basierend auf der aktuellen Situation.
2. **Problemformulierung:** Entscheidung über zu betrachtende Aktionen und Zustände.
3. **Suche:** Prozess des Findens einer Aktionssequenz, die zum Ziel führt.
4. **Ausführung:** Durchführung der gefundenen Aktionen.

Wohlformuliertes Suchproblem

Ein Suchproblem wird durch fünf Komponenten definiert:

1. **Initial State (Startzustand)** s_0 : Der Zustand, in dem der Agent beginnt.
2. **Actions (Aktionen)** $A(s)$: Die Menge der möglichen Aktionen in einem Zustand s .
3. **Transition Model (Überführungsmodell)** $Result(s, a)$: Beschreibt, was eine Aktion tut. Rückgabe ist der Folgezustand.
4. **Goal Test (Zieltest)**: Bestimmt, ob ein Zustand ein Zielzustand ist.
5. **Path Cost (Pfadkosten)** $c(s, a, s')$: Additive Kostenfunktion. Meistens sind Schrittkosten nicht-negativ.

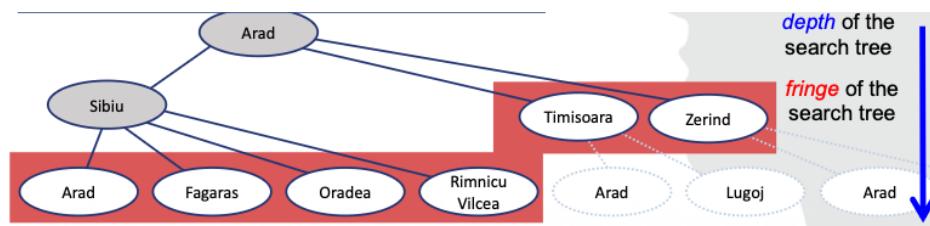
Lösung: Eine Sequenz von Aktionen, die vom Startzustand zum Ziel führt.

Optimale Lösung: Eine Lösung mit den geringsten Pfadkosten.

1.2 Suchalgorithmen: Tree Search vs. Graph Search

Der Kern aller Suchalgorithmen ist die Expansion von Zuständen.

- **Fringe (Open List):** Menge der generierten, aber noch nicht expandierten Knoten.
- **Expansion:** Anwenden der Aktionen auf einen Zustand, um Kindknoten zu generieren.



Unterschied Tree vs. Graph Search

- **Tree Search:** Verfolgt nicht, welche Zustände bereits besucht wurden. Kann in Schleifen geraten und redundante Pfade mehrfach besuchen.
- **Graph Search:** Speichert besuchte Zustände in einer *Explored Set* (Closed List), um Redundanz und Schleifen zu vermeiden.

1.2.1 Bewertungskriterien für Suchstrategien

- **Completeness (Vollständigkeit):** Findet der Algorithmus garantiert eine Lösung, wenn eine existiert?
- **Optimality (Optimalität):** Findet er die kostengünstigste Lösung?

- **Time Complexity:** Wie lange dauert die Suche? (Anzahl generierter Knoten).
- **Space Complexity:** Wie viel Speicher wird benötigt? (Maximale Anzahl Knoten im Speicher).

Parameter der Komplexität:

- b : Verzweigungsfaktor (Branching factor) - max. Anzahl Nachfolger eines Knotens.
- d : Tiefe des flachsten Zielknotens.
- m : Maximale Tiefe des Suchraums (kann ∞ sein).

1.3 Uninformierte Suche (Blind Search)

Uninformierte Strategien haben keine Information darüber, wie nah ein Zustand am Ziel ist. Sie unterscheiden sich nur in der Reihenfolge der Knotencxpansion.

1.3.1 Breadth-First Search (BFS) - Breitensuche

Expandiert den flachsten Knoten in der Fringe zuerst (FIFO-Queue).

- **Vollständig:** Ja (wenn b endlich).
- **Optimal:** Ja, aber nur wenn alle Schritt kosten gleich sind (z.B. 1). Sonst nicht.
- **Zeit:** $O(b^d)$ (Exponentiell).
- **Speicher:** $O(b^d)$ (Jeder generierte Knoten muss gespeichert werden).

Problem: Speicherbedarf ist das größte Problem der BFS.

1.3.2 Uniform-Cost Search (UCS)

Expandiert den Knoten mit den geringsten Pfadkosten $g(n)$ zuerst (Priority Queue).

- Äquivalent zu BFS, wenn alle Schritt kosten gleich sind.
- **Vollständig:** Ja (wenn Kosten $\epsilon > 0$).
- **Optimal:** Ja.
- **Komplexität:** Hängt von den Kosten ab, kann schlechter als b^d sein, wenn viele Schritte mit kleinen Kosten existieren.

1.3.3 Depth-First Search (DFS) - Tiefensuche

Expandiert den tiefsten Knoten in der Fringe zuerst (LIFO-Queue / Stack).

- **Vollständig:** Nein (kann in unendlichen Pfaden oder Schleifen hängen bleiben, außer bei Graph Search in endlichen Räumen).
- **Optimal:** Nein (findet irgendeinen Pfad, nicht zwingend den kürzesten).
- **Zeit:** $O(b^m)$. Schlecht, wenn $m \gg d$.
- **Speicher:** $O(b \cdot m)$ (Linear!). Nur der aktuelle Pfad und Geschwisterknoten werden gespeichert.

1.3.4 Depth-Limited Search (DLS)

DFS mit einem vordefinierten Tiefenlimit l .

- Löst das Endlos-Pfad-Problem der DFS.
- Unvollständig, wenn Lösung tiefer als l ($d > l$).
- Nicht optimal.

1.3.5 Iterative Deepening Search (IDS)

Kombiniert die Vorteile von BFS (Vollständigkeit) und DFS (Speichereffizienz). Führt DLS mit Limit $l = 0, 1, 2, \dots$ nacheinander aus.

- **Vollständig:** Ja.
- **Optimal:** Ja (bei gleichen Schrittkosten).
- **Zeit:** $O(b^d)$. Knoten werden mehrfach generiert, aber da die unterste Ebene die Mehrheit ausmacht, ist der Overhead gering (ca. 11% mehr Aufwand bei $b = 10$).
- **Speicher:** $O(b \cdot d)$ (Linear).

Fazit: IDS ist oft die bevorzugte uninformierte Suchmethode für große Suchräume mit unbekannter Tiefe.

1.4 Informierte Suche (Heuristische Suche)

Nutzt problem spezifisches Wissen in Form einer **Heuristikfunktion** $h(n)$, um die Suche zu lenken.

Heuristik $h(n)$

$h(n)$ = geschätzte Kosten vom Knoten n zum Ziel.

- $h(n) \geq 0$
- Für Zielknoten gilt $h(Goal) = 0$.

1.4.1 Greedy Best-First Search

Expandiert den Knoten, der dem Ziel am nächsten scheint.

- **Bewertungsfunktion:** $f(n) = h(n)$.
- **Vollständig:** Nein (wie DFS, kann in Schleifen geraten).
- **Optimal:** Nein.
- **Zeit/Speicher:** $O(b^m)$ im schlechtesten Fall. Gute Heuristiken können dies drastisch verbessern.

1.4.2 A* Search (A-Star)

Kombiniert UCS und Greedy. Minimiert die geschätzten Gesamtkosten des Pfades durch n .

- **Bewertungsfunktion:** $f(n) = g(n) + h(n)$
- $g(n)$: Tatsächliche Kosten vom Start bis n - \downarrow Vergangenheit.
- $h(n)$: Geschätzte Kosten von n bis zum Ziel - \downarrow Zukunft.
- $f(n)$: Geschätzte Gesamtkosten des Pfades durch n .

1.5 Heuristiken für A*

Damit A* optimal ist, muss die Heuristik bestimmte Eigenschaften erfüllen.

1.5.1 Admissibility (Zulässigkeit)

Eine Heuristik $h(n)$ ist **admissible**, wenn sie die Kosten zum Ziel *niemals überschätzt*.

$$0 \leq h(n) \leq h^*(n)$$

(wobei $h^*(n)$ die wahren Kosten zum Ziel sind).

- Notwendig für Optimalität bei **Tree Search**.
- Beispiel Luftlinie: Die direkte Distanz ist immer kürzer oder gleich der Straßenentfernung.

1.5.2 Consistency (Konsistenz / Monotonie)

Eine Heuristik $h(n)$ ist **consistent**, wenn für jeden Knoten n und jeden Nachfolger n' gilt:

$$h(n) \leq c(n, a, n') + h(n')$$

(Dreiecksungleichung). a ist die angewandte Aktion ($c(n, a, n')$: die Kosten genau dieses Schritts vom Zustand n nach n' unter Verwendung der Aktion a)

- Notwendig für Optimalität bei **Graph Search**.
- Konsistenz impliziert Admissibility.
- Bei konsistenten Heuristiken steigen die $f(n)$ -Werte entlang eines Pfades monoton an (oder bleiben gleich).

1.5.3 Dominanz von Heuristiken

Wenn $h_2(n) \geq h_1(n)$ für alle n (und beide zulässig sind), dann **dominiert** h_2 die Heuristik h_1 .

- h_2 ist näher an den wahren Kosten (h^*).
- A* mit h_2 expandiert weniger Knoten als mit h_1 und ist effizienter.

d	Search Cost (nodes generated)			Effective Branching Factor		
	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	—	539	113	—	1.44	1.23
16	—	1301	211	—	1.45	1.25
18	—	3056	363	—	1.46	1.26
20	—	7276	676	—	1.47	1.27
22	—	18094	1219	—	1.48	1.28
24	—	39135	1641	—	1.48	1.26

1.5.4 Beispiel: 8-Puzzle Heuristiken

- $h_{MIS}(n)$: Anzahl der falsch platzierten Kacheln (Misplaced Tiles).
- $h_{MAN}(n)$: Summe der Manhattan-Distanzen aller Kacheln zu ihrer Zielposition.

Es gilt: $h_{MAN}(n) \geq h_{MIS}(n)$. Daher dominiert die Manhattan-Distanz die "Misplaced Tiles"-Heuristik und ist für A* besser geeignet.