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:

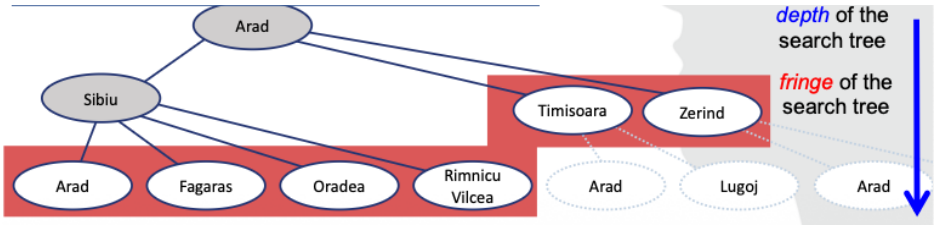
1. **Initial State (Startzustand)** s_0: Der Zustand, in dem der Agent beginnt.
2. **Actions (Aktionen)** $A(s)$: Die Menge der möglichen Aktionen in einem Zustand s.
3. **Transition Model (Überführungsmodell)** $Result(s, a)$: Beschreibt, was eine Aktion tut. Rückgabe ist der Folgezustand.
4. **Goal 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 (entdeckten), aber noch nicht expandierten (bearbeiteten) Knoten. Sie wartet darauf, vom Algorithmus besucht zu werden.
- **Expansion:** Anwenden der Aktionen auf einen Zustand, um alle direkten Nachfolger (Kindknoten) zu generieren und zur Fringe hinzuzufügen.



Unterschied Tree vs. Graph Search

- **Tree Search:** Verfolgt nicht, welche Zustände bereits besucht wurden. Es werden lediglich Knoten generiert. Dies kann dazu führen, dass redundante Pfade mehrfach besucht werden oder der Algorithmus in Zykeln (Schleifen) hängen bleibt.
- **Graph Search:** Speichert besuchte Zustände in einer *Explored Set* (Closed List). Bevor ein Knoten zur Fringe hinzugefügt wird, wird geprüft, ob der Zustand bereits bekannt ist. Dies vermeidet Redundanz und Endlosschleifen.

1.2.1 Bewertungskriterien und Parameter

Um Suchstrategien zu vergleichen, werden folgende Metriken genutzt, die auf den Eigenschaften des Suchraums basieren:

Parameter der Komplexität:

- b **(Branching factor):** Der Verzweigungsfaktor. Die maximale Anzahl an Nachfolgern, die ein Knoten haben kann.
- d **(Depth):** Die Tiefe des *flachsten* (am wenigsten Schritte entfernten) Zielknotens.
- m **(Max Depth):** Die maximale Tiefe des Suchraums (Länge des längsten möglichen Pfades). Kann unendlich (∞) sein.

Kriterien:

- **Completeness (Vollständigkeit):** Findet der Algorithmus garantiert eine Lösung, wenn eine existiert?
- **Optimality (Optimalität):** Findet er garantiert die kostengünstigste Lösung (minimale Pfadkosten)?
- **Time Complexity:** Anzahl der generierten Knoten (Rechenzeit).
- **Space Complexity:** Maximale Anzahl der Knoten, die gleichzeitig im Speicher gehalten werden müssen.

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, in der sie Knoten aus der Fringe zur Expansion auswählen.

1.3.1 Breadth-First Search (BFS) - Breitensuche

Erkundet den Suchbaum schichtweise (Ebene für Ebene).

- **Algorithmus:** Verwendet eine **FIFO-Queue** (First-In-First-Out).
  1. Wähle den *ältesten* Knoten in der Fringe (geringste Tiefe).
  2. Überprüfe auf Zielzustand.
  3. Generiere alle Nachfolger und füge sie *hinten* an die Fringe an.
- **Vollständig**: Ja (solange b endlich ist), da er irgendwann jede endliche Tiefe d erreicht.
- **Optimal**: Ja, aber **nur** wenn alle Schrittkosten gleich sind (z.B. 1). Dann ist der flachste Pfad auch der kostengünstigste.
- **Zeit:** $O(b^d)$ (Exponentiell). Alle Knoten bis Tiefe d werden generiert.
- **Speicher:** $O(b^d)$. Alle generierten Knoten der aktuellen Ebene müssen gespeichert werden.

Problem: Speicherbedarf ist das größte Problem der BFS, da die Fringe exponentiell wächst.

1.3.2 Uniform-Cost Search (UCS)

Erkundet den Suchbaum basierend auf Pfadkosten statt Tiefe. Verallgemeinerung von BFS für ungleiche Schrittkosten.

- **Algorithmus:** Verwendet eine **Priority Queue**, sortiert nach $g(n)$.
  1. $g(n)$: Kumulierte Kosten vom Start bis zum Knoten n.
  2. Wähle den Knoten mit dem *geringsten* $g(n)$ aus der Fringe.
  3. WICHTIG: Der Zieltest erfolgt erst bei der **Expansion** (Entnahme aus Fringe), nicht bei der Generierung, um sicherzustellen, dass kein billigerer Weg existiert.
- **Vollständig**: Ja, wenn jede Schrittkosten $\geq \epsilon > 0$ ist (keine unendlich kleinen Schritte oder Nullkosten-Zyklen).
- **Optimal**: Ja, da billigere Pfade immer vor teureren expandiert werden.

- **Komplexität:** $O(b^{1+\lfloor C^*/\epsilon \rfloor})$.
  - C^*: Kosten der optimalen Lösung.
  - ϵ: Minimale Kosten eines Einzelschritts.
  - Kann schlechter als b^d sein, wenn viele kleine Schritte möglich sind.

1.3.3 Dijkstra-Algorithmus

Der Dijkstra-Algorithmus ist die **Graph-Search-Variante** der Uniform-Cost Search. Er erweitert UCS um eine Speicherkomponente für bereits besuchte Zustände.

- **Unterschied zu UCS:** Während UCS (als Tree Search) redundante Pfade mehrfach berechnen und in Zyklen geraten kann, verhindert Dijkstra dies durch das Speichern besuchter Knoten.
  1. Initialisiere Fringe mit Startzustand und setze *Explored Set* auf leer.
  2. Solange Fringe nicht leer ist:
  3. Wähle Knoten mit kleinstem $g(n)$ und entferne ihn aus der Fringe.
  4. Wenn Zielzustand: Rückgabe Lösung.
  5. Füge Knoten zum *Explored Set* hinzu.
  6. Expandiere Knoten: Füge Nachfolger nur zur Fringe hinzu, wenn sie weder in der Fringe noch im *Explored Set* sind.
- **Eigenschaften:** Erbt Vollständigkeit und Optimalität von UCS (bei nicht-negativen Kantenkosten), ist aber effizienter in Graphen mit vielen Pfaden zum gleichen Zustand.

[Image of Dijkstra algorithm flowchart]

1.3.4 Depth-First Search (DFS) - Tiefensuche

Erkundet Pfade bis in die Tiefe, bevor zurückgegangen wird (Backtracking).

- **Algorithmus:** Verwendet eine **LIFO-Queue / Stack** (Last-In-First-Out).
  1. Wähle den *neuesten* Knoten in der Fringe (tiefste Ebene).
  2. Generiere Nachfolger und lege sie *vorne* auf den Stack.
  3. Wenn ein Blattknoten (oder Sackgasse) erreicht wird, wird dieser verworfen und der nächste Knoten vom Stack (Geschwisterknoten) bearbeitet.
- **Vollständig**: Nein. Kann in unendlichen Pfaden ($m = \infty$) oder Schleifen (ohne Graph Search) hängen bleiben, selbst wenn das Ziel flach liegt.
- **Optimal**: Nein. Findet den ersten Pfad (oft links im Baum), nicht zwingend den kürzesten.
- **Zeit:** $O(b^m)$. Schlecht, wenn die maximale Tiefe m viel größer ist als die Zieltife d ($m \gg d$).
- **Speicher:** $O(b \cdot m)$ (Linear!). Nur der aktuelle Pfad (Tiefe m) und die Geschwister auf jeder Ebene (b) müssen gespeichert werden. Sehr effizient.

1.3.5 Depth-Limited Search (DLS)

Eine DFS, die künstlich beschränkt wird, um Endlos-Pfade zu vermeiden.

- **Algorithmus:** Wie DFS, aber Knoten in Tiefe l (Limit) werden behandelt, als hätten sie keine Nachfolger.
- **Parameter:** l (Tiefenlimit).
- Löst das Endlos-Pfad-Problem der DFS.
- **Unvollständig**: Wenn die Lösung tiefer liegt als das Limit ($d > l$).
- **Nicht optimal**: Wie DFS.

1.3.6 Iterative Deepening Search (IDS)

Kombiniert die Vorteile von BFS (Vollständigkeit/Optimalität) und DFS (Speichereffizienz).

- **Algorithmus:** Führt DLS wiederholt mit steigendem Limit aus.
  1. Starte DLS mit Limit $l = 0$.
  2. Wenn kein Ziel gefunden: Verwerfe kompletten Suchbaum.
  3. Erhöhe Limit $l = l + 1$ und starte DLS von vorn.
- **Vollständig**: Ja, wie BFS.
- **Optimal**: Ja, wie BFS (bei gleichen Schrittkosten), da das Ziel auf der flachstmöglichen Ebene zuerst gefunden wird.
- **Zeit:** $O(b^d)$. Knoten werden mehrfach generiert (Knoten auf Level 1 werden d-mal generiert, auf Level d nur 1-mal). Da die Blätterzahl bei exponentiellem Wachstum dominiert, ist der Overhead gering (ca. 11% bei $b = 10$).
- **Speicher:** $O(b \cdot d)$ (Linear). Verhält sich bzgl. Speicher wie DFS.

Fazit: IDS ist oft die bevorzugte uninformierte Suchmethode für große Suchräume, wenn die Zieltife unbekannt ist.

1.4 Informierte Suche (Heuristische Suche)

Nutzt problemspezifisches Wissen in Form einer **Heuristikfunktion** $h(n)$, um die Suche effizienter Richtung Ziel 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

Versucht, den Knoten zu expandieren, der dem Ziel am nächsten zu sein *scheint*.

- **Algorithmus:** Priority Queue sortiert nach $h(n)$.
  1. Bewertungsfunktion $f(n) = h(n)$.
  2. Wähle Knoten mit *kleinstem* $h(n)$.
- Ignoriert die bereits zurückgelegten Kosten ($g(n)$).
- **Vollständig**: Nein (wie DFS, kann in Schleifen geraten oder Sackgassen folgen).
- **Optimal**: Nein.
- **Zeit/Speicher:** $O(b^m)$ im schlechtesten Fall. Mit guten Heuristiken oft drastisch schneller.

1.4.2 A* Search (A-Star)

Kombiniert UCS (Vermeidung langer Pfade) und Greedy (Fokus auf Ziel).

- **Algorithmus:** Priority Queue sortiert nach $f(n)$.
  1. **Bewertungsfunktion:** $f(n) = g(n) + h(n)$
  2. $g(n)$: Tatsächliche Kosten vom Start bis n (Vergangenheit).
  3. $h(n)$: Geschätzte Kosten von n bis zum Ziel (Zukunft).
  4. $f(n)$: Geschätzte **Gesamtkosten** des Pfades durch n.

5. Expandiere Knoten mit minimalem $f(n)$.

- Verhält sich wie UCS, wenn $h(n) = 0$.

1.5 Heuristiken für A*

Damit A* optimal ist, muss die Heuristik je nach Suchart (Tree oder Graph Search) bestimmte mathematische Eigenschaften erfüllen.

1.5.1 Admissibility (Zulässigkeit)

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

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

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

- **Bedeutung:** Wenn A* eine Lösung mit Kosten C findet, garantiert die Zulässigkeit, dass es keinen übersehenen Pfad mit Kosten $< C$ geben kann, da dieser eine optimistische Schätzung $< C$ gehabt hätte und zuerst expandiert worden wäre.
- 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 die geschätzten Kosten entlang eines Pfades nicht sinken. Für jeden Knoten n und jeden Nachfolger n' gilt:

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

(Dreiecksungleichung).

- Die Kostenreduktion der Heuristik ($h(n) - h(n')$) darf nicht größer sein als die tatsächlichen Schrittkosten $c(n, a, n')$.
- **Folge:** Die $f(n)$-Werte ($g + h$) entlang eines Pfades steigen monoton an.
- Notwendig für Optimalität bei **Graph Search**.
- Konsistenz impliziert Admissibility.
- Bei konsistenten Heuristiken ist der erste gefundene Pfad zu einem Knoten garantiert der optimale.

1.5.3 Dominanz von Heuristiken

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

- h_2 ist "näher" an den wahren Kosten (h^*) und damit genauer.
- **Effekt:** A* mit h_2 muss weniger Knoten expandieren als mit h_1, da Knoten mit $f(n) > C^*$ früher erkannt und abgeschnitten werden.

	Search Cost (nodes generated)			Effective Branching Factor		
d	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: $h_{MAN}(n) \geq h_{MIS}(n) \geq 0$. Daher dominiert die Manhattan-Distanz die “Misplaced Tiles”-Heuristik. Da sie näher an den echten Kosten ist, macht sie A* effizienter.