1 Planning

1.1 Einführung in Classical Planning

Planung (Planning) ist der Prozess, eine Sequenz von Aktionen zu finden, die einen Ausgangszustand in einen Zielzustand überführt, wobei logisches Schließen (Reasoning) verwendet wird, anstatt durch Versuch und Irrtum zu lernen (wie beim Reinforcement Learning).

Definition: Planning

Planning ist die Aufgabe, eine **Sequenz von Aktionen** zu erstellen, die ein **Ziel** (Goal) erreicht, ausgehend von einem **Initialzustand** (Initial State).

1.1.1 Eigenschaften der Umgebung

In der klassischen Planung (Classical Planning) gehen wir von einer spezifischen Umgebung aus:

- **Fully observable (Vollständig beobachtbar):** Der Agent kennt den kompletten Zustand der Welt.
- **Deterministic (Deterministisch):** Das Ergebnis einer Aktion ist genau vorherbestimmbar (kein Zufall).
- **Finite (Endlich):** Es gibt eine endliche Anzahl an Zuständen und Aktionen.
- **Static (Statisch):** Die Welt ändert sich nur durch die Aktionen des Agenten.
- **Discrete (Diskret):** Zustände und Zeitschritte sind diskret.

1.2 Planning vs. Problem Solving

Klassische Suchalgorithmen (Problem Solving) und Planung lösen ähnliche Probleme, unterscheiden sich jedoch fundamental in der Repräsentation.

- **Problem Solving:** Betrachtet Zustände, Ziele und Aktionen als *Black Boxes*. Der Agent versteht die innere Struktur der Zustände nicht.
- **Planning:** Nutzt **explizite Repräsentationen** (meist in First-Order Logic). Zustände, Ziele und Aktionen werden durch logische Sätze beschrieben.

Vorteile der Planung:

- **Reasoning:** Der Planer kann die Effekte von Aktionen analysieren.
- **Dekomposition:** Probleme können in Teilprobleme zerlegt werden, was die Komplexität drastisch reduziert (von exponentiell auf linear oder polynomial, je nach Abhängigkeit).

1.2.1 Problem-Dekomposition

1. **Decomposable Problems:** Teilziele sind komplett unabhängig (z. B. *have(milk)* und *have(bread)*). Sie können separat gelöst werden.
2. **Nearly Decomposable Problems:** Teilziele interagieren, aber die Interaktionen sind handhabbar (z. B. Routenplanung für Pakete: Erst Verteilung auf Flughäfen, dann lokale Auslieferung).

1.3 Logische Grundlagen und Notation

Die Planung verwendet oft eine Prolog-ähnliche Notation der Prädikatenlogik (First-Order Logic - FOL).

- **Konstanten:** Objekte (beginnen mit Kleinbuchstaben oder Zahlen), z. B. `bob`, `1`.
- **Variablen:** Platzhalter für Objekte (beginnen mit Großbuchstaben), z. B. `X`, `Person`.

- **Prädikate:** Relationen zwischen Objekten, z. B. `parent(pam, bob)`.
- **Regeln:** Implikationen, z. B. `Head :- Cond1, Cond2`.

1.4 Situation Calculus

Der Situation Calculus ist ein Ansatz, um Veränderungen in der Welt mittels Logik zu modellieren.

1.4.1 Kernkonzepte

- **Situation (s):** Ein Schnappschuss der Welt zu einem bestimmten Zeitpunkt.
- **Situations-Variable:** Jedes Prädikat, das sich ändern kann, erhält ein zusätzliches Argument für die Situation.
- **Result-Funktion:** $result(a, s)$ gibt die neue Situation zurück, die entsteht, wenn Aktion a in Situation s ausgeführt wird.

Beispiel für eine logische Regel im Situation Calculus:

$$at(A, Y, result(walk(Y), S)) \leftarrow at(A, X, S)$$

Bedeutung: Agent A ist am Ort Y in der Situation, die aus dem Gehen nach Y resultiert, falls A vorher in Situation S am Ort X war.

1.4.2 Probleme des Situation Calculus

The Frame Problem

Das **Frame Problem** beschreibt die Schwierigkeit, auszudrücken, was sich durch eine Aktion *nicht* ändert. In der klassischen Logik muss explizit gesagt werden, dass alles, was nicht durch die Aktion verändert wird, gleich bleibt (Frame Axioms).

- **Representational Frame Problem:** Man müsste für jede Kombination aus Aktion und Prädikat eine Regel schreiben, die besagt, dass das Prädikat unverändert bleibt. Dies führt zu einer riesigen Wissensbasis.
- **Inferential Frame Problem:** Der Beweiser (Theorem Prover) verschwendet die meiste Zeit damit, zu beweisen, dass sich Dinge *nicht* geändert haben.

Weitere Probleme:

- **Qualification Problem:** Schwierigkeit, *alle* Bedingungen zu nennen, die erfüllt sein müssen, damit eine Aktion erfolgreich ist (z. B. "Auto springt an" → Batterie voll, Tank voll, Zündkerzen okay, keine Banane im Auspuff...).
- **Ramification Problem:** Schwierigkeit, alle impliziten Seiteneffekte einer Aktion zu beschreiben (z. B. wenn ich einen Koffer bewege, bewegen sich auch die Socken darin).

1.5 STRIPS (Stanford Research Institute Problem Solver)

STRIPS ist eine Restriktion der allgemeinen Logik, um Planung effizienter zu machen und das Frame Problem zu umgehen.

1.5.1 Repräsentation

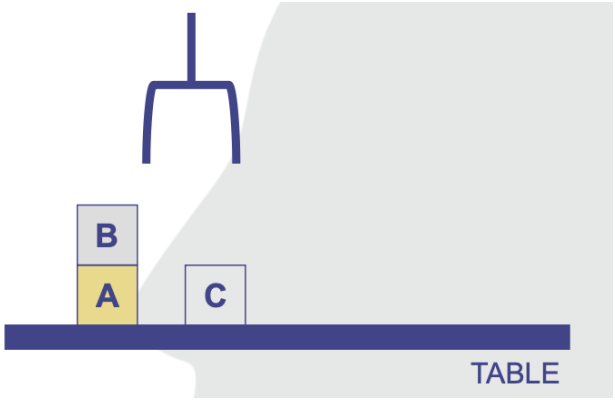
- **Zustand (State):** Konjunktion von positiven, funktionsfreien Literalen (Ground Literals).
  - **Closed World Assumption (CWA):** Alles, was nicht explizit im Zustand als wahr aufgeführt ist, ist falsch.
- **Ziel (Goal):** Konjunktion von Literalen. Ein Zustand erfüllt das Ziel, wenn er alle Ziel-Literale enthält.
- **Aktion (Action):** Definiert durch drei Komponenten:

- **Preconditions (Vorbedingungen):** Konjunktion von Literalen, die wahr sein müssen, damit die Aktion ausführbar ist.
- **ADD-List:** Literale, die nach der Aktion wahr werden (zum Zustand hinzugefügt werden).
- **DELETE-List:** Literale, die nach der Aktion falsch werden (aus dem Zustand entfernt werden).

STRIPS Assumption

Um das Frame Problem zu lösen, nimmt STRIPS an, dass jedes Literal, das **nicht** in der ADD- oder DELETE-Liste einer Aktion erwähnt wird, **unverändert** bleibt.

1.5.2 Beispiel: Blocks World



- **Prädikate:** $on(x, y)$, $on(x, table)$, $clear(x)$, $holding(x)$, $handempty$.
- **Aktionen:**
  - $stack(x, y)$: Lege x auf y.
  - $unstack(x, y)$: Nimm x von y.
  - $pickup(x)$: Nimm x vom Tisch auf.
  - $putdown(x)$: Lege x auf den Tisch.

Beispiel einer Aktion $stack(x, y)$ in STRIPS:

- **Precond:** $holding(x) \wedge clear(y)$
- **Add:** $handempty \wedge on(x, y) \wedge clear(x)$
- **Delete:** $holding(x) \wedge clear(y)$

1.6 Planungs-Algorithmen

Da STRIPS-Probleme als Zustandsraumsuche formuliert werden können, können Standard-Suchalgorithmen (wie A*) verwendet werden. Es gibt zwei Hauptrichtungen:

1.6.1 Progression (Forward Planning)

Suche vom Initialzustand zum Ziel.

1. Start: Initialzustand.
2. Finde alle anwendbaren Aktionen (deren Preconditions im aktuellen Zustand erfüllt sind).
3. Generiere Nachfolgezustände durch Anwenden von ADD- und DELETE-Listen.
4. Prüfe, ob der neue Zustand das Ziel erfüllt.

Nachteil: Hoher Verzweigungsfaktor, da viele irrelevante Aktionen möglich sind.

1.6.2 Regression (Backward Planning)

Suche vom Ziel rückwärts zum Initialzustand. Dies ist oft effizienter, da nur **relevante** Aktionen betrachtet werden. **Relevant Action:** Eine Aktion ist relevant für ein Ziel G, wenn sie mindestens ein Literal aus G in ihrer ADD-Liste hat und keines der Literale in G in ihrer DELETE-Liste steht (Konsistenz). **Inverse Action Application:**Wenn wir ein Ziel G haben und eine Aktion A rückwärts anwenden, berechnet sich das neue Teilziel (Predecessor Goal) G' wie folgt:

$$G' = (G \setminus \text{ADD}(A)) \cup \text{PRECOND}(A)$$

Das bedeutet:

1. Entferne die Effekte der Aktion, da diese durch die Aktion selbst erfüllt wurden (wir brauchen sie nicht mehr zu suchen).
2. Füge die Vorbedingungen der Aktion hinzu, da diese *vorher* erfüllt sein mussten, damit die Aktion überhaupt stattfinden kann.

1.7 Heuristiken für die Planung

Da die exakte Lösung NP-schwer ist, benötigen wir Heuristiken für die Suche.

1.7.1 Relaxed Problem

Man vereinfacht das Problem, um eine Abschätzung der Kosten (Anzahl Schritte) zu erhalten.

- **Ignore Preconditions:** Man nimmt an, jede Aktion sei immer ausführbar.
- **Ignore Delete Lists:** Man nimmt an, Aktionen machen nichts "kaputt".

Die Lösung dieses entspannten Problems dient als heuristischer Wert für das echte Problem.

1.7.2 Subgoal Independence Assumption

Man nimmt an, dass die Kosten, um eine Konjunktion von Zielen zu erreichen, gleich der Summe der Kosten der einzelnen Ziele sind.

$$Cost(G_1 \wedge G_2) \approx Cost(G_1) + Cost(G_2)$$

Achtung: Diese Heuristik ist nicht zulässig (not admissible), wenn positive Interaktionen existieren (eine Aktion erfüllt beide Ziele), und sie unterschätzt die Kosten massiv, wenn negative Interaktionen existieren (Aktionen für G_1 zerstören G_2).