

1 Rational Agents

Ein Agent ist ein System, das seine Umgebung durch Sensoren wahrnimmt und durch Aktoren auf diese Umgebung einwirkt. Er agiert autonom, um bestimmte Ziele zu erreichen.

Sensoren: Nehmen Informationen aus der Umgebung auf (z.B. **Kamera, Mikrofon**). Die aufgenommenen Informationen werden als **"Percepts"** (Wahrnehmungen) bezeichnet.

Aktoren: Führen Aktionen aus, die die Umgebung beeinflussen (z.B. **Motor, Greifarm, Lautsprecher**).

Rationalität: Zu jedem Zeitpunkt diejenige Aktion auswählt, die seinen erwarteten Nutzen (sein **Performanzmass**) maximiert.

Rationalität vs. Perfektion: Ein rationaler Agent trifft die **bestmögliche Entscheidung, basierend auf den ihm zur Verfügung stehenden Informationen**. Sollten diese Informationen fehlerhaft oder unvollständig sein, kann ein rationaler Agent eine objektiv falsche Entscheidung treffen. Er handelt aus seiner Perspektive optimal.

Beispiel: Ein Agent, der eine Strasse überquert, schaut nach links und rechts. Wenn er von einem Meteoriten getroffen wird, war sein Handeln trotzdem rational, da er dieses Ereignis nicht vorhersehen konnte.

2 Problemformulierung für Suche

Bevor ein Problem mit einem Algorithmus gelöst werden kann, muss es formal bechrieben werden. Dies geschieht durch die Definition folgender Komponenten:

- Zustandsraum (State Space):** Die Menge aller möglichen Zustände, in denen sich die Umgebung befinden kann.
z.B. **Routenplanung: Alle Städte auf der Karte**
- Anfangszustand (Initial State):** Der Zustand, in dem der Agent seine Suche beginnt.
z.B.: **In der Startstadt -> Arad.**
- Aktionen (Actions):** Eine Beschreibung der möglichen Aktionen, die dem Agenten in einem bestimmten Zustand zur Verfügung stehen.
z.B.: **Von einer Stadt aus zu allen direkt verbundenen Nachbarstädten fahren.**
- Zieltest (Goal Test):** Eine Funktion, die überprüft, ob ein gegebener Zustand ein Zielzustand ist.
z.B. **Ist der aktuelle Zustand (Stadt) "Bukarest".**
- Pfadkosten (Path Costs):** Eine Funktion, die jeder Aktion bzw. jedem Pfad (einer Sequenz von Aktionen) numerische Kosten zuweist. Das Ziel ist oft, einen Pfad mit minimalen Gesamtkosten zu finden.
z.B.: **Die Entfernung in Kilometern zwischen zwei Städten.**

2 Informierte (heuristische) Suche

Diese Algorithmen nutzen zusätzliches, problem-spezifisches Wissen, um die Suche gezielt in Richtung des Ziels zu lenken:

- Heuristik h(n):** Eine Funktion, die die Kosten von einem Knoten n bis zum Ziel schätzt. Eine gute Heuristik ist entscheidend für die Effizienz der Suche.
- Greedy Best-First-Search:** Expandiert immer den Knoten, der dem Ziel am nächsten zu sein scheint. Nutzt als alleiniges Kriterium die Heuristik: **f(n) = h(n)**.
Ist schnell, aber nicht vollständig und nicht optimal!
- A*-Suche:** Kombiniert die Stärken von UCS und Greedy-Search. Expandiert den Knoten mit den geringsten geschätzten Gesamtkosten vom Start zum Ziel.

Nutzt die Evaluationsfunktion: f(n) = g(n) + h(n), wobei:
 - g(n):** Die tatsächlichen Kosten vom Startknoten bis zum Knoten n (wie bei UCS).
 - h(n):** Die geschätzten Kosten vom Knoten n bis zum Ziel (wie bei Greedy Search).

- Konzepte für **A*-Optimalität:**
- Zulässige Heuristik (Admissible Heuristic):** Eine Heuristik h(n) ist zulässig, wenn sie die tatsächlichen Kosten zum Ziel niemals überschätzt. Sie ist optimistisch:
Bedingung: **h(n) ≤ h*(n)**, wobei **h*(n)** die echten Kosten sind. Ist die Heuristik zulässig, findet A* immer eine optimale Lösung (**auf Bäumen**).

- Konsistente Heuristik (Consistent/Monotone Heuristic):** Eine Heuristik ist konsistent, wenn für jeden Knoten n und jeden seiner Nachfolger n' die geschätzten Kosten von n nicht grösser sind als die Kosten des Schrittes nach n' + die geschätzten Kosten von n'.
Bedingung: **h(n) ≤ c(n, a, n') + h(n')**
Jede konsistente Heuristik ist auch zulässig. Ist die Heuristik konsistent, ist A* optimal und am effizientesten (**auf Graphen**).

```
1 def a_star_search(problem, heuristic):
2     frontier = []
3
4     heappush(frontier, (heuristic(problem.initial, problem.goal),
5                           Node(problem.initial)))
6
7     visited = {problem.initial}
8
9     while frontier:
10         _, curr = heappop(frontier)
11
12         if problem.goal_test(curr.state):
13             return curr
14
15         for next in curr.expand(problem):
16
17             if next.state not in visited:
18                 visited.add(next.state)
19
20                 cost = heuristic(next.state, problem.goal) +
21                     next.path_cost
22
23                 heappush(frontier, (cost, next))
24
25     return None
```

Titel / Begriffe		Hohe Relevanz
Keywords	Kombinationen möglich	
Beispiele		Gleiche Nummern = Gleiches Thema (1, 2, 3)

1 PEAS-Modell

PEAS-Modell ist ein strukturierter Rahmen, um die Aufgabe eines Agenten zu definieren. Es besteht aus vier Komponenten:

- P - Performance Measure (Performanzmass):** Wie wird der Erfolg des Agenten objektiv gemessen? Dies sollte ein klares Kriterium sein, das den Zustand der Umgebung bewertet:
Beispiel Staubsauger: Menge des aufgesaugten Schmutzes, Energieverbrauch und benötigte Zeit.
- E - Environment (Umgebung):** In welchem Umfeld agiert der Agent?
Beispiel Staubsauger: Ein oder mehrere Räume, Bodenbelag, Möbel, andere bewegliche Objekte (Menschen, Haustiere).
- A - Actuators (Aktoren):** Welche Werkzeuge hat der Agent, um Aktionen auszuführen?
Beispiel Staubsauger: Räder zum Bewegen, Saugvorrichtung, Bürsten.
- S - Sensors (Sensoren):** Wie nimmt der Agent seine Umgebung wahr?
Beispiel Staubsauger: Kamera zur Objekterkennung, Infrarotsensoren zur Abstandsmessung, Schmutzsensor.

2 Uninformierte (blinde) Suche - Algorithmen

Diese Algorithmen nutzen zur Suche keine zusätzliche Informationen über die Entfernung zum Ziel. Sie arbeiten sich systematisch durch.

Algo	Wie	Vollständigkeit	Optimal	Zeit-O	Raum-O
BFS - Breitensuche	Expandiert Knoten Schicht für Schicht - FIFO	Ja	Ja, wenn alle Pfadkosten gleich sind	O(b^d)	O(b^d)
DFS - Tiefensuche	Expandiert immer den tiefsten Knoten zuerst - LIFO	Nein, kann in Endlosschleifen gelangen	Nein	O(b^m)	O(bm)
UCS - Uniform-Cost -Suche	Expandiert Knoten mit geringsten Pfadkosten (Prio-Queue)	Ja	Ja	$O(b^{1+\frac{C^*}{\epsilon}})$	$O(b^{1+\frac{C^*}{\epsilon}})$
IDS - Iterative Tiefensuche	Führt wiederholt eine begrenze DFS mit steigendem Tiefenlimit durch	Ja	Ja, wenn alle Pfadkosten gleich sind	O(b^d)	O(bd)

b: Branching Factor, d: Tiefe des flachsten Ziels, m: Maximale Tiefe, C*: Kosten der optimalen Lösung, ε: Minimale Kosten einer Aktion

3 Grundlagen der Spieltheorie

Ein Spiel wird formal durch **drei grundlegende Komponenten** definiert:

- Spieler (Players):** Die Entscheidungsträger im Spiel
- Aktionen (Actions):** Die Menge der möglichen Züge, die ein Spieler ausführen kann. Ein Aktionsprofil ist eine Kombination von Aktionen, bei der jeder Spieler eine Aktion gewählt hat.
- Präferenzen/Auszahlungen (Preferences/Payoffs):** Der Nutzen (Utility), den ein Spieler für jedes mögliche Ergebnis (Aktionsprofil) des Spiels erhält. Diese Präferenzen werden typischerweise in einer Payoff-Matrix dargestellt.

Beispiel: Payoff-Matrix (Spieler 1 wählt Zeilen, Spieler 2 wählt Spalten)

	Spieler 2: Aktion C	Spieler 2: Aktion D
Spieler 1: Aktion A	Nutzen S1, Nutzen S2	Nutzen S1, Nutzen S2
Spieler 1: Aktion B	Nutzen S1, Nutzen S2	Nutzen S1, Nutzen S2

Beispiel: Eisdielen-Entscheidung

	Eisdiele B: Preis beibehalten	Eisdiele B: Preis senken
Eisdiele A: Preis beibehalten	100€, 100€	50€, 120€
Eisdiele A: Preis senken	120€, 50€	70€, 70€

Diverses

Performanzmass vs. Utilityfunction: Performanzmass → externer Faktor für Evaluation der Aktion. Utility-Funktion: Funktion intern in Agent um anhand möglicher Optionen eine Entscheidung zu treffen.

Berechnung Lookup-Table (2x2) bei Step 1: 2x2 = 4
Bei step 2: (2x2)^2 = 16, Bei step 3: (2x2)^3 = 64

General formula for lookup table for all entries when all percepts stored: $\sum_{t=1}^n |P|^t$

Concepts to Describe Search Algo: Node expansion - generatic all successor nodes, frontier - set of all nodes available for expansion, search strat: defines which node to expand next, visited: the explored

Evaluating Search Algos: branching factor - max(nodes connected to one node), depth (d) - depth of shallowest goal node, m - max length of any path in the state space), completeness: is strategy guaranteed to find solution if one exists, optimality: does strat find best solution? time complexity: how much times does it take, space complexity: how much memory does the search require?

1 Eigenschaften von Umgebungen

Vollständig vs Partiiell Beobachtbar (Fully vs. Partially Observable):
Vollständig: Die Sensoren des Agenten erfassen zu jedem Zeitpunkt den kompletten, relevanten Zustand der Umgebung. z.B. **Schach**
Partiell: Der Agent hat nur Zugriff auf einen Teil der Umgebungsinformationen (z.B. **durch verdeckte Bereiche oder ungenaue Sensoren**) z.B. **Poker**.

Deterministisch vs. Stochastisch (Deterministic vs. Stochastic):
Deterministisch: Der nächste Zustand der Umgebung ist vollständig durch den aktuellen Zustand und die Aktion des Agenten bestimmt. z.B: **Schach**
Stochastisch: Es gibt ein Zufallselement. Die gleiche Aktion im gleichen Zustand kann zu unterschiedlichen Ergebnissen führen. z.B: **Monopoly**

Episodisch vs. Sequenziell (Episodic vs. Sequential):
Episodisch: Die Erfahrung des Agenten ist in einzelne, unabhängige Episoden unterteilt. Die Aktion jetzt beeinflusst nicht ide nächste. z.B.: **Klassifikation von Bildern (ist es ein Hund oder eine Katze?)**
Sequenziell: Die aktuelle Entscheidung beeinflusst alle zukünftigen Entscheidungen. Der Agent muss langfristig planen. z.B. **Navigationssystem (jetzt abbiegen verändert Route)**

Statisch vs. Dynamisch (Static vs. Dynamic):
Statisch: Die Umgebung ändert sich nicht, während der Agent über seine nächste Aktion nachdenkt. z.B. **Kreuzworträtsel**
Dynamisch: Die Umgebun kann sich von selbst verändern, unabhängig von den Aktionen des Agenten. z.B. **Börsen-Handel**

Diskret vs. Kontinuierlich (Discrete vs. Continious):
Diskret: Es gibt eine endliche oder abzählbar unendliche Anzahl von Zuständen, Wahrnehmungen und Aktionen (z.B: **Schach**).
Kontinuierlich: Zustände und Aktionen ändern sich fließend (z.B: **Steuerung eines Fahrzeugs.**)

Einzel- vs. Multi-Agent (Single vs. Multi-anget):
Einzalagent: Nur ein Agent agiert in der Umgebung. z.B.: **Ein Roboter in einem Labyrinth**
Multi-Agent: Mehrere Agenten agieren in der Umgebung. Diese können kooperativ oder kompetitiv sein. z.B.: **Auktions-Handel**

Bekannt vs. Unbekannt (Knows vs. Unknown):
Bekannt: Die "Regeln" der Umgebung sind dem Agenten bekannt (z.B. **Auswirkungen seiner Aktionen**). z.B: **Bekanntes Puzzle**
Unbekannt: Der Agent muss die Funktionsweise der Umgebung erst lernen. z.B. **Brandneues Videospiel**

Beispiel Autopilot: Partiiell Beobachtbar, Stochastisch, Sequenziell, Dynamisch, Kontinuerlich, Multi-Agent, Bekannt.

Beispiel E-Mail Spam Filter: Vollständig Beobachtbar, Deterministisch, Episodisch, Statisch, Diskret, Einzelagent, Unbekannt (am Anfang).

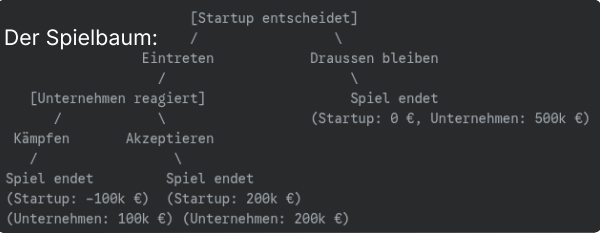
3 Sequenzielle Spiele (Sequential Games)

In sequenziellen Spielen handeln die Spieler nacheinander und können die vorherigen Züge der Gegner beobachten.

- Darstellung als **Spielbaum (Game Tree)**:
 - Knoten: Entscheidungspunkte für die Spieler.
 - Kanten: Mögliche Aktionen
 - Blätter (Endknoten): Ergebnisse des Spiels mit den Payoffs.
- Rückwärtsinduktion (Backward Induction):**
 - Die zentrale Methode zur Lösung von sequenziellen Spielen mit perfekter Information.
 - Vorgehen (in folgender Reihenfolge):**
 - Beginne bei den letzten Entscheidungsknoten im Spielbaum.
 - Bestimme für jeden dieser Knoten die optimale Aktion für den Spieler, der an der Reihe ist. Streiche alle anderen, suboptimalen Aktionen. Der Payoff der optimalen Aktion wird zum neuen Wert dieses Knotens.
 - Gehe eine Eben im Baum zurück und wiederhole den Prozess, wobei du annimmst, dass an den nachfolgenden Knoten immer die bereits ermittelte optimale Entscheidung getroffen wird.
 - Fahre fort, bis du den Startknoten erreichst. Der verbleibende Pfad ist die Lösung.
 - Das Ergebnis der Rückwärtsinduktion ist ein **teilspielperfektes Nash-Gleichgewicht (Subgame Perfect Nash Equilibrium - SPNE)**.

3 Sequenzielle Spiele - Beispiel

Szenario: Spieler 1, ein neues Startup, das überlegt ind en Markt einzutreten. Speiler 2, ein Unternehmen, das bereits im Markt ist.



bei letztem beginnen (2.): Rational ist für Unternehmen "akzeptieren", da 200k > 100k. -> zu nächsten (1.): Firma weiss was Unternehmen macht, rational wäre "eintreten", da 100k > 0. Somit ist Pfad: Eintreten > Akzeptieren. Erebnis ist dann: 200k Startup, 200k Unternehmen. --> Das Ergebnis: teilperfekte Nash-Gleichgewicht

3 Simultane Spiele - Beispiel

Szenario: Zwei Komplizen (A und B) werden getrennt verhört. Beide können entweder schweigen oder gestehen. Wenn beide schweigen, beide milde Strafe. Wenn einer gesteht und andere nicht, geht Verräter frei und andere bekommt hohe Strafe. Wenn beide gestehen, beide mittlere Strafe.

-----	Kom. B: Schweigen	Kom. B: Gestehen
Kom. A: Schweigen	1 Jahr, 1 Jahr	10 Jahre, 0 Jahre
Kom. A: Gestehen	0 Jahre, 10 Jahre	5 Jahre, 5 Jahre

Beste Antwort für A,B (einzeln): gestehen -> 0 besser als 1 oder 5
Dominante Strategie: Gestehen
Nash-Gleichgewicht: einzige Nash-Gleichgewicht: beide gestehen, obwohl schweigen besser wäre

4 Constraint Programming (CP)

Ein Problem wird in CP durch drei grundlegende Zutaten modelliert:

- 1. **Variablen (Variables):** Die Unbekannten im Problem, für die Werte gefunden werden müssen.
- 2. **Domänen (Domains/Frames):** Die Menge der möglichen Werte, die eine Variable annehmen kann.
- 3. **Constriants (Einschränkungen):** Regeln oder Bedingungen, die festlegen, welche Kombinationen von Werten für die Variablen zulässig sind. Sie schränken den Lösungsraum ein.

Globale Constraints sind vordefinierte, mächtige Einschränkungen, die auf eine Gruppe von Variablen wirken, wie:

- **AllDifferent(vars):** Stellt sicher, dass alle Variablen in der Liste vars unterschiedliche Werte annehmen müssen. (z.B: **Sudoku, k-Färbung**).
- **AddCircuit(edges):** Stellt sicher, dass eine Menge von Kanten in einem Graphen einen einzigen geschlossenen Pfad (**Hamiltonschen Kreis**) bildet, der **jeden Knoten genau einmal besucht**. (Esentiell für Routing-Probleme wie **z.B. das Travelling Salesperson Problem**.

Symmetriebrechung (Symmetry Breaking): Dies ist eine Technik zur Steigerung der Effizienz. **Man fügt dem Modell zusätzliche, logisch redundante Constraints hinzu**, um symmetrische (d.h. im Kern identische) Lösungen auszuschliessen. Dadurch wird der Suchraum verkleinert und der Solver findet eine Lösung schneller. **Beispiel (Graphenfärbung): Lege fest, dass Knoten 1 immer Farbe 0 bekommt.**

5 Lokale Suche - Grundprinzip

Lokale Suchalgos eignen sich für grosse oder komplexe Optimierungsprobleme, bei denen der Weg zur Lösung egal ist, sondern es auf die Qualität der Endlösung ankommt.

Grundidee:

1. Beginne mit einer (oft zufällig gewählten) vollständigen Konfiguration (einem Zustand).
2. Bewerte die Qualität dieses ZUstands mithilfe einer Zielfunktion (Objective Function).
3. Betrachte die direkten "Nachbarn" des aktuellen Zustands.
4. Bewege dich zu einem Nachbarn mit einer besseren Bewertung.
5. Wiederhole Prozess, bis keine Verbesserung mehr möglich ist.

Anwendung: Ideal für Probleme wie das 8-Damen-Problem, Tourenplanung (TSP) oder Stundenplanerstellung, wo der Weg irrelevant ist.

Herausforderungen: Einfache lokale Suchstrategien können in suboptimalen Lösungen stecken bleiben.

- **Lokales Optimum (Maximum/Minimum):** Ein Zustand, der besser ist als alle seine direkten Nachbarn, aber nicht die global beste Lösung darstellt. Der Algorithmus ist hier "gefangen".
- **Plateau:** Ein Bereich von Zuständen mit identischer Bewertung. Der Algorithmus irrt ziellos umher, da er keine Richtung zur Verbesserung findet.
- **Grat (Ridge):** Eine schmale "Gratwanderung" von Zuständen, die zu einem globalen Optimum führen könnte, aber schwer zu folgen ist, da Bewegungen in die meisten Richtung abwärts führen.

5 Lokale Suche - Algorithmen und Techniken

- **Hill-Climbing (Bergsteigen):** "greedy local search"
Funktionsweise: Wählt immer den besten Nachbarn aus, um dorthin zu wechseln.
Nachteil: Bleibt garantiert in lokalen Optima stecken und ist daher weder vollständig noch optimal.
- **Techniken zum Entkommen aus lokalen Optima:**
Random Restarts (Zufällige Neustarts): Eine sehr effektive und einfache Strategie. Man führt denn Hill-Climbing-Algo mehrfach mit unterschiedlichen, zufälligen Startzuständen aus. Die beste gefundene Lösung über alle Durchläufe wird genommen. Je mehr Ausführung desto höher die Wahrsch'keit, das globale Optimum zu finden.
- **Genetische Algorithmen (Genetic Algorithms):**
Idee: Anstatt nur einen Zustand betrachten, wird eine ganze **Population von Zuständen** verwaltet. **Die besten Zustände überleben und kombinieren sich, um eine neue, potenziell besseren Generation von Lösungen zu erzeugen.**
Schlüsselkonzepte:
 - **1. Fitness-Funktion:** Bewertet die Qualität jedes Zustands in der Population.
 - **2. Selektion:** Individuen (Zustände) mit höhrrere Fitness haben eine höhere W'keit für die nächste Gen ausgewählt zu werden.
 - **3. Crossover (Kreuzung):** Zwei "Eltern"-Zustände werden kombiniert, um neue "Kinder"-Zustände zu erzeugen, die Merkmale beider Elternteile erben.
 - **4. Mutation:** Gelegentliche, zufällige Änderungen in einem Zustand, um die genetische Vielfalt zu erhalten und aus lokalen Optima auszubrechen.
- **Simulierte Abkühlung (Simulated Annealing):**
 - Ein stochastischer Algo, der auch Züge zu schlechteren Nachbarn erlaubt, um aus lokalen Optima zu entkommen.
 - **Grundidee:** Die W'keit, einen schlechteren Zug zu akzeptieren, hängt von einer "Temperatur" T ab.
 - Hohe Temepnatur T: Zu Beginn der Suche ist W'keit, einen schlechten Zug zu machen, hoch. Der Algo explodiert den Suchraum breit.
 - Niedrige Temperatur T: Im Laufe der Zeit "kühlt" die Temperatur ab. Die W'keit für schlechte Züge sinkt und der Algo konvergiert zunehmend zu einer guten Lösung (ähnlich wie Hill-Climbing).

Diverses

Theorem of Zermelo: Jedes endliche Zwei Personen Spiel hat immer... eine Strategie für Spieler 1 um einen Sieg zu erzwingen (first-mover-vorteil)
ODER
eine Strategie für Spieler 2, um einen Sieg zu erzwingen (second-mover-vorteil).
ODER
Eine Strategie für beide Seiten, um mindestens ein Unentschieden zu erzwingen (z.B. Schach).

4 CP: **Forward Checking:** unvollständig, keine Lösungsgarantie. guter Tradeoff zwischen Propagation und Geschwindigkeit.
Propagation: Solver zieht logische Konsequenzen der letzten Entscheidung und verkleinert verbleibenden Suchraum.

4 Lösungsalgorithmen in CP (CSP)

- CP-Solver kombinieren zwei fundamentale Techniken:
1. **Suche (Search):**
 - a. Der systematische Teil des Algos, der den Lösungsraum durchsucht.
 - b. Die Kernmethode ist **Backtracking**. Der Solver weist einer Variable einen Wert zu und fährt mit der nächsten fort. Wenn ein Constraint verletzt wird, geht er einen Schritt zurück, macht die Zuweisung rückgängig und probiert einen anderen Wert.
 - c. Backtracking allein ist vollständig (findet eine Lösung, wenn eine existiert), aber **oft sehr langsam**.
 2. **Propagation (Constraint Propagation / Inferenz):**
 - a. Der "intelligente" Teil des Algos. Nach jeder Zuweisung einer Variable leitet der Solver logische Konsequenzen ab und reduziert die Domänen anderer Variablen.
 - b. **Forward Checking:** Eine einfache Form der Propagation. **Wenn eine Variable X den Wert v erhält, werden alle Werte, die nun ungültig sind, aus den Domänen der Nachbar-Variablen entfernt.**
 - c. **Bounds/Domain Consistency:** Eine stärkere Form. Der Solver analysiert die globalen Constraints (**z.B. AllDifferent**) tiefer, um noch mehr ungültige Werte aus den Domänen zu entfernen. Dies reduziert den Suchraum drastisch.

Die Stärke von CP liegt in der permanenten Wechselwirkung von **Suche und Propagation**. Jeder Suchschritt (Zuweisung) löst eine **Propagationswelle** aus, die den verbleibenden Suchraum verkleinert und so zukünftige Suchschritte effizienter macht.

1 Rational Agents - Types of Agents

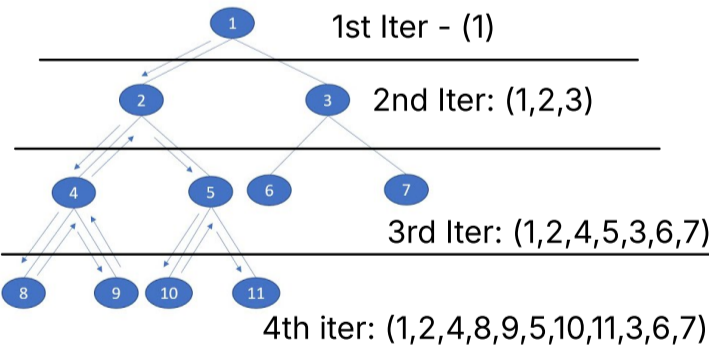
- **Simple Reflex Agent:** Handelt nur basierend auf dem aktuellen Percept (Wahrnehmung) und einfachen "Wenn-Dann"-Regeln. Hat kein Gedächtnis. **z.B: Thermostat, wenn Temp < 20°, dann heizen.**
- **Model-based Reflex Agent:** Führt ein internes Modell der Welt. Er weiss, wie die Welt funktioniert und wie seine Aktionen sie beeinflussen. Kann mit partiell beobachtbaren Umgebungen umgehen. **z.B: Adaptive Kamera im Auto: Kamera sieht Heck des vorderen Autos, nicht dessen Tempo. Wenn das Heck näher kommt und ich X schnell fahre, in der nächsten Sekunde Tempo um Y kleiner. Merkt sich vorherigen Zustand und vergleicht.**
- **Goal-based Agent:** Hat explizite Ziele. Er kann vorausschauen und Aktionen planen, die ihn zu einem Zielzustand führen (Kern von Suche und Planung) **z.B: Navigationssystem, Agent sucht im Voraus verschiedene Routen, bewertet sie und gibt die optimale Sequenz zum Ziel.**
- **Utility-based Agent:** Maximiert nicht nur das Erreichen eines Ziels, sondern den "Nutzen" (utility). Kann abwägen, welcher Zielzustand besser ist, wenn es mehrere gibt (**z.B. schneller vs. sicherer Weg**). **z.B: Reisebuchungsportal: Finde einen Flug von Zürich nach London. Ein Flug günstig aber Zwischenstopps, anderer direkt aber teuer. Utility-Agent bewertet Kunden-Happiness basierend auf einem Kompromiss zwischen Preis, Reisedauer und Komfort und schlägt Option mit dem höchsten Gesamtnutzen vor.**
- **Learning Agent:** Kann seine Leistung über die Zeit verbessern, indem er aus Erfahrungen lernt. Besteht aus Lernelement, Performanzelement, Kritiker und Problemgenerator. **z.B: Empfehlungssystem Netflix: Beobachtet vergangene Useraktionen und welche Filme User positiv bewertet. Probiert bielleich auch eine "gewagte" Empfehlung um mehr zu lernen (Problemgenerator).**

2 IDDFS - IDA* visualisiert

IDDFS geht iterativ immer eine Stufe tiefer, bis es eines der Goal-Nodes findet. Bei IDA* hängen Iterationen nicht von der Tiefe im Baum sondern von h(n) ab. Nur nodes mit **h(n) < threshold** werden expandiert.

Sollte es in der Iteration kein Goal Node finden, setzt man für die nächste Iteration den **threshold = tiefsten F-score** der Nodes wo man aufgehört hat vorher, da **h(n) > threshold** war.

(Visualisierung ist IDDFS / IDS):



3 Strategies and Equilibria

Best response function: Maximaler Payoff, Wenn man die Wahlmöglichkeit aller Spieler kennt, ist es rational die beste Antwortaktion zu nehmen.
Nash Equilibrium: Wenn die Aktion eines jeden Spielers die best response auf die Aktion eines anderen Spielers ist.
Dominante Strategie: Wenn die Strategie immer die Beste ist, egal was die anderen Spieler tun.
Dominiert Strategie: Wenn die Strategie niemals die beste Antwort auf eine Aktion eines Gegenspielers ist.
Pure strategy: Bestimmte Handlungen oder Entscheidungen, die ein Spieler im Spiel trifft. Es handelt sich um eine Strategie, bei der eine bestimmte Option aus allen möglichen Optionen ausgewählt wird, ohne dass ein Zufall oder eine Wahrscheinlichkeit im Spiel ist. **z.B. bei Schere-Stein-Papier immer Stein spielen.**
Mixed Strategy: Eine Option wird nach dem Zufallsprinzip mit einer bestimmten Wahrscheinlichkeit ausgewählt. W'keit-Verteilung über die möglichen Aktionen. **z.B. Schere-Stein-Papier: p(Rock)=0.25, p(Scissors)=0.25,p(Paper)=0.5**
Mixed Strategy Equilibrium: Wenn der Gegner etwas anderes spielt als reiner Zufall (1/3, 1/3, 1/3), dann ist die best response immer einer pure strategy. Wenn beide zufällig spielen, dann spielen sie ein Best response gegeneinander (MSE).
Strict Nash Equilibrium (stabil): wenn jmd. abweicht, wird der Payoff sofort bedeutend schlechter.
Weak Nash Equilibrium (instabil): wenn jmd. abweicht, wird der Payoff kontinuierlich (wenig) schlechter, Lawineneffekt möglich.

4 CP für Optimierungsprobleme (COP)

Ein reines Constraint-Problem fragt: "Gibt es eine gültige Lösung?". Ein Optimierungsproblem fragt: "**Was ist die beste gültige Lösung?**".

- **Zielfunktion (Objective Function):** Man fügt dem Modell eine Zielfunktion hinzu, **die für jede Lösung einen Wert berechnet**. Mit **Minimize(funktion)** oder **Maximize(funktion)** weist man den Solver an, die Lösung mit dem minimalen bzw. maximalen Wert zu finden.
- **Channeling Constraints:** Dies sind spezielle Constraints, die verschiedene Teile eines Modells miteinander verknüpfen. Sie sind oft notwendig, um die Zielfunktion zu formulieren:
 - **OnlyEnforceIf(bedingung)** ist ein typischer Channeling Constraint. Er aktiviert einen anderen Constraint nur dann, wenn eine bestimmte Bedingung (oft eine boolesche Variable) wahr ist.
 - Anwendung: **Man kann damit z.B. zählen, wie viele Ressourcen (Farben, Fahrzeuge) in einer Lösung verwendet werden, um diese Anzahl dann zu minimieren.**

Hier random Code-Beispiel:

```
1 model = cp_model.CpModel()
2 x = model.NewIntVar(0, 9, "x")
3 y = model.NewIntVar(0, 9, "y")
4 z = model.NewBoolVar("z")
5 model.Add(x + y == 10).OnlyEnforceIf(z)
6 model.Add(x < y)
7 model.AddAllDifferent([x, y])
8 # Beispiel für Circuit (Hamiltonkreis über Knoten)
9 n = 3
10 arcs = []
11 for i in range(n):
12     for j in range(n):
13         if i != j:
14             arcs.append([i, j, model.NewBoolVar(f"e{i}{j}")])
15 model.AddCircuit(arcs)
```

4 Checklist CP (OR-Tools)

1. Variablen & Domänen:
Korrekte Domäne? **NewIntVar(0, 0, ...)** für Ziffern (0-9) oder für Ziffern 1-9: **NewIntVar(1, 9, ...)**?
Richtiger **Variablentyp**? **NewIntVar** für Ganzzahlen vs. **NewBoolVar** für Wahr/Falsch?
Alle Variablen definiert? Fehlt eine die in einem Constraint verwendet wird?
2. Constraints:
AllDifferent vergessen? Häufiger Fehler bei **z.B. Sudoku, Zeitpläne**, etc.) **model.AddAllDifferent(...)**
Logikfehler: > statt <? == statt !=?
Arithematische Formel korekt? **Stimmt Berechnung (S*1000+E*100...)**
Indexierungsfehler? Greift CCode auf richtigen Index zu? **board[i][j] vs board[j][i]**
Off-by-One-Fehler: **range(size)** vs. **range(size-1)**
3. Optimierungsprobleme (COP):
Falsches Ziel? **model.Minimize(...)** statt **model.Maximize(...)**
Falsche Zielfunktion? Wird richtige Variable optimiert?
4. Channeling Constraints:
OnlyEnforceIf(b) **Logik** korrekt? Ist es falschrum?
Richtig: **model.Add(x > 5).OnlyEnforceIf(b)**
Falsch: **model.Add(b == 1).OnlyEnforceIf(x > 5)**

2 Iterative Deepening A* Search

Der A*-Algorithmus ist optimal und vollständig, aber im hat den Nachteil, dass sein **Speicherbedarf im schlimmsten Fall exponentiell wächst**, da er alle besuchten Knoten im Speicher halten muss.

IDA* ist die Antwort auf dieses Problem. Die Kernidee ist, die **Speichereffizienz der Tiefensuche (DFS)** mit der **Optimalitätsgarantie von A*** zu kombinieren.

Anstatt einer Tiefenbegrenzung, wie bei der normalen iterativen Tiefensuche (**IDDFS**), verwendet IDA* eine Kostenbegrenzung (einen "Threshold". Diese Begrenzung bezieht sich auf die A*-Bewertungsfunktion: **f(n) = g(n) + h(n)**)

1. **Initialisierung:** Der erste Kosten-Grenzwert (threshold) wird auf die heuristischen Kosten des Startzustands gesetzt: threshold = h(start)
2. **Iterationen:** Der Algorithmus startet eine Schleife. In jeder Iteration wird eine modifizierte Tiefensuche durchgeführt.
3. **Begrenzte Tiefensuche:** Diese Tiefensuche expandiert einen Pfad nur so lange, wie die Gesamtkosten f(n) den aktuellen threshold nicht überschreiten.
 - a. Wenn f(n) threshold, wird der Pfad abgeschnitten (pruning), und die Suche kehrt um (backtracking).
4. **Nächster Threshold:** Wenn die Suche beendet ist, ohne das Ziel zu finden, wird ein neuer threshold für die nächste Iteration festgelegt. Dieser neue Grenzwert ist das Minimum aller f(n)-Kosten, die im vorherigen Durchlauf den alten Grenzwert überschritten haben.
5. **Ende:** Dieser Prozess wird wiederholt, bis eine Lösung gefunden wird.

IDA*, ist optimal und verwendete Heuristik ist gültig, wenn wahren Kosten nie überschätzt werden. Speicher: O(bd), Zeit: O(b^d)
b: Verzweigungsfaktor, d: Tiefe der optimalen Lösung

3 Finding Mixed Strategy Equilibrium

	Player2: Left q	Player2: Right 1-q
Player1: Left p	50, 50	80, 20
Player1: Right 1-p	90, 10	20, 80

Payoffs Player 1:
Wenn er links spielt: E(links) = **q * 50 + (1 - q) * 80** = 80 - 30q
Wenn er rechts spielt E(rechts) = **q * 90 + (1 - q) * 20** = 20 + 70q
Müssen gleich sein → 80 - 30q = 20 + 70q, q = 0.6, 1-q = 0.4
Player 2 Spielt den Mix: **(0.6, 0.4)**
Payoffs Player 2:
Wenn er links spielt: E(links) = **p * 50 + (1 - p) * 10** = 10 + 40p
Wenn er rechts spielt: E(rechts) = **p * 20 + (1 - p) * 80** = 80 - 60p
Müssen gleich sein → 10 + 40p = 80 - 60p, p = 0.7, 1-p = 0.3
Player 1 spielt den Mix: **(0.7, 0.3)**
Kernaussage: Ein stabiles Gleichgewicht bei gemischten Strategien (MSNE) entsteht dadurch, dass jeder Spieler seine Wahrscheinlichkeiten so wählt, dass der Gegner zwischen seinen eigenen Aktionen indifferent wird. Spieler 1 wählt also die Strategie so, um Spieler 2 indifferent zu machen, also Strategie so wählen, dass es aus der Sicht des Gegners egal ist welche Aktion er wählt.