Search

Generelles

- Unser Entscheidungsprozess ist **deterministisch**, also "unfrei"
- Projiziert auf ein Spiel ist das Gegenüber/der Gegner nicht deterministisch, daher haben wir ihn durch einen deterministischen Gegner ersetzt. Dieser Gegner war Min, als wir über MiniMax geredet haben (der böseste mögliche Gegner).

Motivation

- Das Konzept von Entscheidungsbäumen ist wichtig für das Verständnis von Entscheidungsproblemen
- Im Wahrscheinlichkeitsraum sind Baumsuchen ein Spezialfall von Monte-Carlo-Methoden um Erwartungen zu schätzen. Dies nennt man Q-Function.
- Baumsuchen sind der Hintergrund von Backtracking in Constraint Satisfaction Problems

Problemformulierung

Ein deterministisches, vollständig observierbares Problme ist definiert durch:

- eine Ausgangszustand (*initial state:* `s_o \in S`): *Wo bin ich?*
- eine "Änderungsfunktion" (*successor function:* `S \times A --> S`) *Wie ändere ich das?*
- einen Zielzustand (*goal state:* `s_{goal} \in S`) *Wo will ich hin?*
- eine "Änderungskostenfunktion" (*step cost function:* `cost(s,a,s') > 0`) *Was kostet mich das?*

Die Lösung ist eine Sequenz von Aktionen von `s_o` zu `s_{goal}`. Eine optimale Lösung hat die kleinste Summe der `pathcosts`.

Beispiel: Baumsuche

Eine **Node** beschreibt einen Knoten in einem Entscheidungsbaum, nicht aber einem Zustand. Eine Node hat `parent, children, depth und pathcost`.

Eine Suchstrategie wird durch die Sortierung der Nodes in der `fringe`, der Liste der abzuarbeitenden Nodes, definiert.

Eine Suchstrategie wird bewertet nach:

- **Vollständigkeit**: Finde ich immer eine Lösung? (wenn eine Lösung existiert)
- **Zeit-Komplexität**: Wie oft muss ich eine Node expandieren?
- **Platz-Komplexität**: Wie groß wird meine fringe maximal?
- **Optimalität**: Findet es immer die Lösung mit den geringsten Kosten?

Zeit- und Platz-Komplexität werden nach folgenden Kriterien eingestuft:

- `b` = maximale Verzweigungszahl einer Node
- `d` = Tiefe (*depth*) der günstigsten Lösung
- `m` = maximale Tiefe des Entscheidungsbaums

Suchstrategien

Breadth-First Search: *FIFO*

Expandiere die umexpandierte Node mit der geringsten Tiefe

Fringe ist eine FIFO queue, neu entdeckte Nodes kommen ans Ende

Uniform Cost Search: *Sort Fringe by Cost so far*

"Cost-Aware BFS", Fringe ist sortiert nach den Pfadkosten mit den geringsten Kosten zuerst.

--> *BFS/UCS propagieren den Baum ebenenweise von oben nach unten, die Ergebnisse sind optimal, aber die Komplexität steigt exponentiell*

Depth-First Search: *LIFO*

Expandiere die umexpandierte Node mit der höchsten Tiefe

Fringe ist ein LIFO stack, neu entdeckte Nodes kommen an den Anfang

Iterative Deepening Search: *Repeat DFS for increasing depths*

Zuerst: Depth limited Search - DFS mit beschränkter Tiefe `l` Wiederhole DLS mit stetig wachsendem `l` bis das gewünschte Ziel gefunden ist

Logischerweise: Fringe ist ein LIFO stack, neu entdeckte Nodes kommen an den Anfang

--> *DFS/IDS durchlaufen den Baum radial "von links nach rechts", die Komplexität steigt linear. Nur IDS ist optimal*

A* Algorythm: *Sort Fringe by estimated total Cost*

Spezialfall von USC: Anstatt der zurückgelegten Pfadkosten `g` wird die Summe `f` aus `g` und den erwarteten Kosten bis zum Ziel `h` gebildet.

Die Fringe wird nach den geringsten Insgesamtkosten sortiert.

Die Wahl der Heuristik ist sehr wichtig für die Zahl der expandierten Nodes.

- Die Heuristik muss immer optimistisch sein, darf die tatsächlichen Kosten vom State zum Ziel also nie unterschreiten.
- Mehrere Heuristiken sind zugelassen, eine zugelassene Heuristik heißt **admissive Heuristic**
- Eine Heuristik ergibt sich aus einer Vereinfachten Ansicht eines komplizierten Problems

Eigenschaften der Suchen

Search	Completeness	Time	Space	Optimality
BFS	Yes, is `b` is finite	`b^{d+1}`	`b^{d+1}`	Yes, if cost per step is 1
UCS	Yes, if step cost > 0	similar	similar	Yes

Search	Completeness	Time	Space	Optimality
DFS	Yes, if `d` is finite	`b^m`	`b*m`	No
IDS	Yes	`b^d`	`b*d`	Yes, id cost per step is 1
A*	Yes, mostly	exp.	exp.	Yes

Prüfungsrelevant ist

- Wissen, was die Algorythmen machen und in der Lage sein, sie im Kopf durchzugehen
- Die Komplexitäten der Algorythmen kennen
- Wissen, was eine **admissive heuristic** ist
- Bei gegebener Funktion angeben, ob es sich um eine admissive heuristic handelt