Hilfsblatt zu allen Praktiksaufgaben

Dieses Hilfsblatt fasst das Vorgetragene nochmal kurz zusammen und bietet Hinweise zu den verschiedenen Praktikumsaufgaben.

Hintergrund: Pac-Man klassisch

Pac-Man ist ein legendäres Video-Automatenspiel, erfunden um 1980 herum - zu seinen Regeln siehe etwa https://de.wikipedia.org/wiki/Pac-Man.

Der KI1 Pac-Man

Unser Pac-Man im KI-Praktikum spielt nicht das originale Pac-Man-Spiel. Stattdessen wird in einer Labyrinth-ähnlichen Umgebung mit verschiedenen Suchstrategien experimentiert. Die Größe des Labyrinths kann je nach Beispielwelt variieren. Neben den Wänden hat unser Pac-Man nur mit den Dots zu tun, Geister kommen im ersten Teil des Praktikums nicht vor.

Unserer Pac-Man kann sich nach Norden, Osten, usw. bewegen und kann natürlich auch gar nichts tun.

Pac-Man, Suche, Knoten & Bäume

Zu den wichtigsten Fähigkeiten eines guten Pac-Man-Spielers zählt das effiziente Fressen von Dots, mit so wenig Aktionen wie möglich. Dazu sollte das unnötig mehrfache Ablaufen von Wegen vermieden werden. In einfachen Welten ist dies noch leicht im Kopf lösbar (siehe Abbildung 1), aber wie sieht die optimale Lösung für die Welt in Abbildung 2 aus?

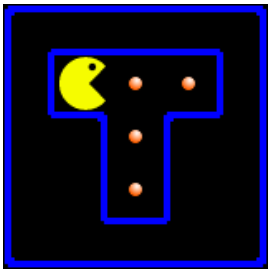

Abbildung 2: Eine einfache Pac-Man Welt.

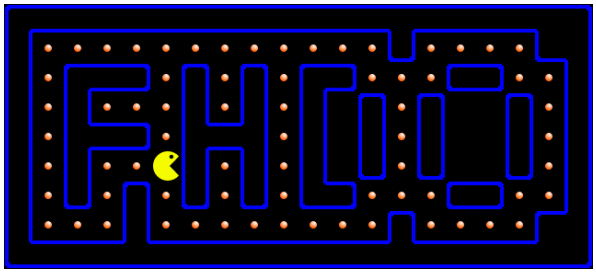

Abbildung 1: Die FH-DO Pac-Man Welt.

Um eine Lösung zu finden, können Suchstrategien verwendet werden. Diese geben eine „Ablauf“-Strategie vor (z.B. immer erst nach rechts gehen). Die Suchstrategien betrachten die Auswirkungen der verschiedenen Aktionen (GO_NORTH, GO_EAST etc.) auf einen Weltzustand in einer bestimmten Reihenfolge. Jede zulässige Aktion führt wieder zu einem neuen Weltzustand (siehe Abbildung 3), für den wieder Aktionen möglich sein können.

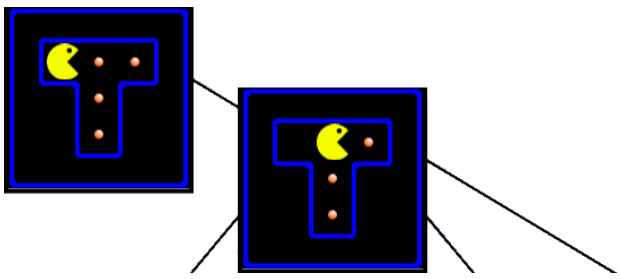

Abbildung 3: Der neue Weltzustand basierend auf der Ausgangs-Welt.

Irgendwann verfügt mindestens (i.d.R. führen vielen Lösungen zum Ziel) einer der Weltzustände über keine Dots mehr (siehe Abbildung 4). Jetzt hat der Pac-Man eine mögliche Lösung gefunden und könnte über diesen Weltzustand auf die Aktionen rückschließen, mit denen er von seiner aktuellen Welt diesen Weltzustand erreichen kann (sofern er nicht noch nach weiteren, vielleicht besseren Lösungen suchen möchte).

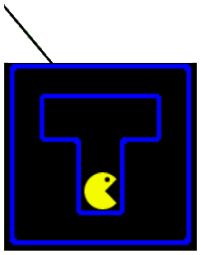

Abbildung 4: Ein möglicher Endzustand der Suche.

Die Praktika

In den folgenden Praktika werden die Suchen aus den Vorlesungen implementiert und diese bauen basierend auf der aktuellen Welt des Pac-Mans einen Baum auf, in dem jeder Knoten einen Weltzustand darstellt und die ausgeführten Aktionen die Verbindungen zwischen den Knoten. Manche der Knoten sind mögliche (bessere oder schlechtere) Lösungen der Suchen.

Praktikum 1: Einführung in den Server

MyAgent & Pac-Man-Server

Im Praktikum 1 lernt man den Pac-Man-Server und den MyAgent-Client kennen. Unserer Pac-Man wird durch die Klasse *MyAgent* gesteuert. Die Klasse *MyAgent* stellt zusammen mit der *Pac-Man-Server.jar* eine Client-Server-Architektur da.

Weiterführende Informationen zum Server, zur Bedienung, Architektur und Einrichtung findet Ihr in der Server_Readme.

Die Methode *action* der Klasse *MyAgent* stellen den aus der Vorlesung bekannten „updateState-chooseAction cycle“ dar. Über die *action*-Methode kann der Agent auf seine aktuelle Wahrnehmung der Welt (*PacmanPercept*) und auf die Rückmeldung des Servers auf seine letzte Aktion (*PacmanActionEffect*) zugreifen.

Das PacmanPercept verfügt über folgende Methoden:

- getView():
  Gibt den aktuellen Zustand der Welt in Form eines 2D-Arrays von *PacmanTileTypes* zurück.
  PacmanTileTypes können sein:
  WALL, DOT, EMPTY, PACMAN, GHOST und GHOST_AND_DOT, POWERPILL und GHOST_AND_POWERPILL
- getPosition():
  Gibt die X- und Y-Koordinate der aktuellen Position des Pac-Man zurück.
- getGhostInfos():
  Gibt eine Liste von GhostInfo - Objekten zurück, über welche auf die Informationen der Geister zugegriffen werden kann.

Die möglichen PacmanActionEffects sind:

GAME_INITIALIZED, GAME_OVER, BUMPED_INTO_WALL, DOT_EATEN, MOVEMENT_SUCCESSFUL, BUMPED_INTO_PACMAN, ATE_GHOST, ATE_POWERPILL, ATE_GHOST_AND_DOT und ATE_GHOST_AND_POWERPILL.

In der *action*-Methode wird die nächste auszuführende Aktion des Agenten zurückgegeben.

Die möglichen PacmanActions sind:

GO_NORTH, GO_EAST, GO_SOUTH, GO_WEST, WAIT und QUIT_GAME

Praktikum 2: Knoten & Bäume

Hier noch ein paar Tipps zur Entwicklung der Knoten, die euch für die späteren Aufgaben helfen:

- Die expand-Methode soll die Aktionen des Pac-Man simulieren und z.B. bei der Bewegung auf einen Dot, diesen fressen. Da sich somit der Weltzustand ändert, müssen die neuen Kindknoten einen eigenen Weltzustand besitzen und keine Referenz auf den Weltzustand des Elternknotens (Stichwort: Call by Reference).
- Der Pac-Man kann auch nichts tun (warten). Sollte die Aktion Warten bei der Expansion von Knoten berücksichtigt werden? (in einem Spiel ohne Geister)

Praktikum 3: Die Suche

Hier noch ein paar Punkte, die zu berücksichtigen sind:

- Mit welchen Informationen startet die Suche? (Start- & Zielzustand bzw. Zieltest)
- Mit welchen Datentypen werden die noch zu expandierenden und die bereits expandierten Knoten gespeichert (openList/closedList)?
- Mit der Methode contains(object), welche von allen Java Collections unterstützt wird, kann geprüft werden, ob ein Element sich z.B. schon in einer Liste befindet. Dazu muss die equals-Methode der Knoten überschrieben werden.
- Für das Vergleichen von zwei Arrays ist die Methode Arrays.deepEquals(array1, array2) sehr hilfreich, da array1 == array2 nicht bei mehrdimensionalen Arrays funktioniert.
- Wie kann auf die Folge von notwendigen *PacmanActions* geschlossen werden, um einen bestimmten Knoten vom Anfangszustand aus zu erreichen?

Praktikum 4: Verschiedene Suchverfahren

Die verschiedenen Suchstrategien unterscheiden sich im Wesentlichen nur darin, ob und wie Knoten bewertet und ggf. sortiert werden. Beispielsweise fügt Tiefensuche die Knoten immer am Anfang einer Liste an, während Greedy-Search die Knoten nach Bewertungen sortiert hält. Abbildung 5 zeigt einen Pseudocode, welcher für alle Suchverfahren genutzt werden kann.

```
function ClosedList-Search(problem, openList) return a solution or failure
        closedList = an empty list
        openList = INSERT(MAKE-NODE(INITIAL-STATE[problem]), openList)
        loop do
                if openList is empty then return failure
                node = REMOVE-FRONT(openList)
                if GOAL-TEST(problem, STATE[node]) then return node
                if STATE[node] is not in closedList then
                        add STATE[node] to closedList
                        for child-node in EXPAND(STATE[node], problem) do
                                openList = INSERT(child-node, openList)
                        end
        end
```

Die Realisierung der Funktion "INSERT" bestimmt, welches Suchverfahren hier verfolgt wird (Tiefensuche, Breitensuche, Greedy Suche, UCS, A*).

Abbildung 5: Pseudocode für alle Suchverfahren.

Hier noch ein paar Punkte, die zu berücksichtigen sind:

- Wie unterscheidet man innerhalb der *Suchen*-Klasse welche Suchstrategie ausgeführt wird?
- Wie werden die Knoten in den Listen organisiert?
- Wie könnte eine brauchbare heuristische Zustandsbewertung, in Bezug auf das Ziel alle Dots zu fressen, aussehen? (für die heuristischen Suchstrategien)