Künstliche Intelligenz
– Aufgabenblatt 02 –

Prof. Dr. David Spieler
Hochschule München

29. März 2023

Hinweis: Implementieren Sie im Folgenden die Suchalgorithmen mit einem zusätzlichen Parameter `verbose` mit Standardwert `True`, durch welchen der jeweilige Schritt mit Hilfe von `print` ausgegeben wird. Dies erleichtert Ihnen die Fehlersuche und die Beantwortung der Fragen.

Aufgabe 1 (Suchprobleme in Graphen) *Wir möchten den kompletten Suchgraphen des Containerproblems in Python berechnen. Dazu kodieren wir einen Knoten des Containerproblems als Tupel von Tupeln.*

- *In einem solchen Tupel `x` kodiert `x[0]` die linke Position, `x[1]` die Mitte und `x[2]` die rechte Position.*
- *Eine leere Position entspricht dabei dem leeren Tupel `()`, ein einzelner Container wird durch das Tupel `('A')` oder `('B')` repräsentiert und ein Containerstapel `A` auf `B` durch `('B','A')` bzw. `'B` auf `'A'`durch `('A','B')`.*

Beispielsweise wird der Startknoten repräsentiert durch `(('B','A'),(),())` – links steht Container `A` auf Container `B` und sowohl die Mitte als auch die rechte Position sind leer.

1. *Kopieren Sie zunächst die in Aufgabenblatt 01 erstellte Klasse `Graph` in ihr Jupyter Notebook für dieses Aufgabenblatt.*
2. *Implementieren Sie eine Funktion `getSuccessorsAt(state, position)`, welche die Nachfolger von Knoten `state` als Menge zurückgibt – ausgehend von der Verschiebung eines möglichen Containers an der Position `position` (`0`: links, `1`: Mitte, `2`: rechts). Hinweis: Hier gibt es jeweils keinen oder zwei mögliche Nachfolger.*
3. *Implementieren Sie eine Funktion `getSuccessors(state)`, die alle Nachfolger von Knoten `state` als Menge zurückgibt.*

4. *Implementieren Sie eine Funktion* `buildContainerGraph()`*, die ausgehend vom Startknoten* `(('B','A'),(),())` *den kompletten Suchgraphen des Containerproblems mit allen Knoten und Kanten berechnet und zurückgibt. Hinweis: Eine Möglichkeit dies zu tun ist in einer Schleife jeweils alle Nachfolger der aktuellen Knotenmenge zu berechnen und der Knotenmenge hinzuzufügen. Die Schleife sollte enden, sobald es keine neuen Knoten mehr gibt. Vergessen Sie nicht, die Kanten der Kantenmenge hinzuzufügen.*

5. *Überprüfen Sie ihren Code, indem Sie die Knoten- und Kantenmenge mit den Abbildungen im Foliensatz vergleichen.*

Hinweis: Es ist nicht üblich, dass bei der Suche immer der komplette Graph erstellt und in einer Graph-Datenstruktur abgespeichert wird. Dies ist bei Graphen mit unendlich vielen Knoten ja auch nicht möglich. Meistens erfolgt die Exploration des Graphen direkt im Suchalgorithmus, wie in den folgenden Aufgaben.

Aufgabe 2 (Tiefensuche im Wasserproblem) *Wir möchten nun die Tiefensuche für das Wasserproblem in Python programmieren. Dazu kodieren wir einen Knoten des Wasserproblem als Tupel* `(l,r)`*, wobei* `l` *und* `r` *ganze Zahlen sind, welche den Füllstand des linken bzw. rechten Bechers mit* 4 *bzw.* 3 *Litern Fassungsvermögen repräsentieren. Wir suchen einen Pfad zu einem Zielknoten, bei dem mindestens einer der beiden Becher die gesuchten* 2 *Liter beinhaltet.*

1. *Implementieren Sie eine Funktion* `waterSuccessors(state)`*, welche die Menge aller Nachfolgeknote von Knoten* `state` *zurückgibt. Hinweis: Es gibt jeweils bis zu* 6 *Nachfolger.*

2. *Implementieren Sie eine Funktion* `dfs_nocheck(start, end, successors)`*, die einen Startknoten* `start`*, ein Endprädikat* `end` *als Funktion von einem Knoten auf* `True` *oder* `False` *und eine Nachfolgerfunktion (siehe vorherige Teilaufgabe) entgegen nimmt und einen Lösungspfad nach dem Prinzip der Tiefensuche berechnet und zurückgibt. Die optionale Überprüfung, ob ein Nachfolger im Pfad bereits besucht wurde, soll hier nicht stattfinden.*

3. *Starten Sie die Suche durch einen geeigneten Aufruf von* `dfs_nocheck`*. Was passiert?*

4. *Implementieren Sie eine Funktion* `dfs(start, end, successors)` *wie eben, jedoch diesem Mal mit der optionalen Überprüfung.*

5. *Starten Sie die Suche durch einen geeigneten Aufruf von* `dfs` *erneut. Was passiert?*

Aufgabe 3 (Breitensuche im Wasserproblem) *Wiederholen Sie die vorherige Aufgabe aber diesmal mit der Breitensuche.*

1. *Implementieren Sie eine Funktion* `bfs_nocheck(start, end, successors)` *mit der Breitensuche ohne optionalen Check.*

2. *Starten Sie die Suche durch einen geeigneten Aufruf von* `bfs_nocheck`*. Was passiert?*

3. *Implementieren Sie eine Funktion* `bfs(start, end, successors)` *wie eben, jedoch diesem Mal mit der optionalen Überprüfung.*

4. *Starten Sie die Suche durch einen geeigneten Aufruf von* `bfs` *erneut. Was passiert?*

5. *Starten Sie die Suche mit* `bfs` *erneut, jedoch wählen Sie dieses Mal ein unerreichbares Ziel. Was passiert? Können Sie die Funktion* `bfs` *verbessern?*

Aufgabe 4 (Das Steineproblem) *Für das Steineproblem kodieren wir einen Zustand mit Hilfe eines 7-stelligen Tupeln und den Zeichen* `'W'`*,* `'B'` *und* `' '` *für einen weißen Stein, einen schwarzen Stein und das leere Feld. Der Startzustand wird also repräsentiert durch* `('W', 'W', 'W', ' ', 'B', 'B', 'B')`*.*

1. *Implementieren Sie eine Funktion* `computeG(f, t)`*, welche die tatsächlichen Kosten von Zustand* `f` *zu Zustand* `t` *berechnet.*

2. *Implementieren Sie eine Funktion* `computePathG(p)`*, welche die tatsächlichen Kosten eines Pfades* `p` *(einer Liste von Zuständen) berechnet.*

3. *Implementieren Sie eine Funktion* `stoneSuccessors(state)` *welche die Nachfolgezuständen* `successor` *von Zustand* `state` *mit den entsprechenden Kosten* `g` *als Menge von Tupeln* `(successor, g)` *berechnet.*

4. *Implementieren Sie eine angepasste Version der DFS, welche die Kosten* `g` *ignoriert und generieren Sie damit eine Lösung. Wieviele Schritte hat Ihre Lösung und berechnen Sie deren Gesamtkosten mit* `computePathG(p)`*. Wie lange dauert die Ausführung? Ist die Lösung gut?*

5. *Implementieren Sie eine angepasste Version der BFS, welche die Kosten* `g` *ignoriert und generieren Sie damit eine Lösung. Wieviele Schritte hat Ihre Lösung und berechnen Sie deren Gesamtkosten mit* `computePathG(p)`*. Wie lange dauert die Ausführung? Ist die Lösung gut?*

6. *Implementieren Sie eine Funktion* `stoneH(state)`*, welche die im Foliensatz vorgestellte Heuristik h für Zustand* `state` *berechnet.*

7. *Implementieren Sie die Bergsteigersuche in* `hill_climb(start, end, successors, h, verbose=True)`*. Generieren Sie damit eine Lösung. Wieviele Schritte hat Ihre Lösung und berechnen Sie deren Gesamtkosten mit* `computePathG(p)`*. Wie lange dauert die Ausführung? Ist die Lösung gut?*

8. *Implementieren Sie mit* `bestfs(start, end, successors, h, verbose=True)` *die Bestensuche. Generieren Sie damit eine Lösung. Wieviele Schritte hat Ihre Lösung und berechnen Sie deren Gesamtkosten mit* `computePathG(p)`*. Wie lange*

dauert die Ausführung? Ist die Lösung gut? Hinweis: Für die effiziente Verwaltung der Kandidaten und die Extraktion des besten Kandidaten bietet sich eine `PriorityQueue` *aus der Bibliothek* `queue` *an.*

Aufgabe 5 (Der A*-Algorithmus) *Wir wollen nun den A*-Algorithmus auf dem Steineproblem testen.*

- *Implementieren Sie eine Klasse* `AStarNode`*, die einen* `predecessor`*-Knoten speichert, als auch die entsprechenden Werte* `g`*,* `h` *und* `f`*. Implementieren Sie auch* `AStarNode.__lt__(self,other)`*.*
- *Implementieren Sie den A*-Algorithmus* `astar(start, end, successors, h)`*.*
- *Generieren Sie damit eine Lösung für das Steineproblem. Wie lange dauert die Ausführung? Ist die gefundene Lösung gut?*

Aufgabe 6 (Labyrinth) *Wir testen nun den A*-Algorithmus an einem Labyrinth.*

```
maze = [
    "################",
    "#          #    #",
    "#          ###  #",
    "#            #  #",
    "#            #  #",
    "#       #### #  #",
    "#       #    #  #",
    "#       #    #  #",
    "#       #    #  #",
    "#       # #  #  #",
    "#       # #  #  #",
    "#       # #     #",
    "#       # #######",
    "#       #        #",
    "#       #        #",
    "################"
]
start = (1,1)
end = (14,14)
```

- *Implementieren Sie eine Funktion* `mazeSuccessors(position)`*, die ausgehend von einer Position* `position = (x,y)` *im Labyrinth, die Menge der Nachbarn berechnet. Als Nachbarn gelten die vier Nachbarfelder links, recht, oben und unten aber nur, wenn es sich dabei um einen Weg* `' '` *und keine Mauer* `'#'` *handelt. Hinweis: In* `maze` *erhalten Sie das Feld mit den Koordinaten* (x, y) *durch* `maze[y][x]`*!*
- *Implementieren Sie die Manhattan-Distanz als Heuristik* `mazeH(position)`*.*

- *Starten Sie den A*-Algorithmus im Labyrinth, um einen Weg von* `start` *bis* `end` *zu finden. Findet der Algorithmus eine Lösung und ist sie optimal? Wie lange dauert es? Hinweis: Schreiben Sie sich dazu eine Funktion* `markSolution(maze, path)`*, die das Labyrinth auf der Konsole ausgibt zusammen mit dem Pfad gekennzeichnet durch* `'.'`*-Zeichen.*