Grundlagen der künstlichen Intelligenz: Hausaufgabe 1

Tom Nick - 340528
Niklas Gebauer - 340942
Leonard Witte - 341457
Johannes Herrmann - xxxxxx

Aufgabe 1

a) **Zustandsraum:** (a, b) wobei $a \in \{A, B, C, Z, i, j\}, b \in \{0, \ldots, 100\}$
wobei a die aktuelle Position und b den Ladezustand beschreibt.
Anfangszustand: $(A, 100)$
Zielzustand: (Z, c) wobei $c \geq 50$.
Aktionen:

1.
$$\mathsf{fahren}(start, ziel, energie) : (a, b) \rightarrow (x, y)$$
mit
$$\begin{aligned} a = start \land x = ziel \land \\ y = b - energie \land y \geq 0 \land \\ (start, ziel, energie) \in \{&(A, Z, 95), (Z, A, 95), \\ &(A, i, 50), (i, A, 50), \\ &(i, Z, 100), (Z, i, 100), \\ &(i, j, 50), (j, i, 50), \\ &(i, B, 45), (B, i, 45), \\ &(j, Z, 40), (Z, j, 40), \\ &(j, C, 20), (C, j, 20), \\ &(Z, C, 10), (Z, C, 10)\} \end{aligned}$$

2.
$$\mathsf{laden}(zustand) : (a, b) \rightarrow (x, y)$$
mit
$$zustand \in \{i, j\} \land y = 100 \land zustand = a = x$$

Aktionskosten:

$$\mathsf{kosten}(aktion) : Aktion(args) \rightarrow \mathbb{R}_{\geq 0}$$

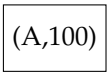
mit
$$\begin{aligned} \mathsf{fahren}(A, i, 50), \mathsf{fahren}(i, A, 50), \mathsf{fahren}(i, j, 50), \mathsf{fahren}(j, i, 50) &\mapsto 100 \\ \mathsf{fahren}(i, Z, 100), \mathsf{fahren}(Z, i, 100), \mathsf{laden}(i), \mathsf{laden}(j) &\mapsto 200 \\ \mathsf{fahren}(A, Z, 95), \mathsf{fahren}(Z, A, 95) &\mapsto 170 \\ \mathsf{fahren}(i, B, 45), \mathsf{fahren}(B, i, 45), \mathsf{fahren}(j, Z, 40), \mathsf{fahren}(Z, j, 40) &\mapsto 80 \\ \mathsf{fahren}(j, C, 20), \mathsf{fahren}(C, j, 20) &\mapsto 25 \\ \mathsf{fahren}(Z, C, 10), \mathsf{fahren}(C, Z, 10) &\mapsto 20 \end{aligned}$$

b) **Verzweigungsgrad:** maximal 5 (in Zustand i am größten)
Tiefe: Eine obere Abschätzung ist die Anzahl der Positionen*Anzahl der Ladezustände. Die Ladezustände können wir in Fünferschritten zählen, da die Energiekosten alle durch 5 teilbar sind. Wir hätten dann also eine maximale Tiefe von 6*(100/5)=6*20=120.

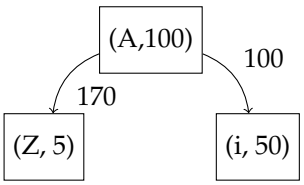
c) Da wir den schnellsten Weg finden wollen und nicht uniforme Aktionskosten haben, wäre 'Branch & Bound' am besten als Suchalgorithmus geeignet.

d)

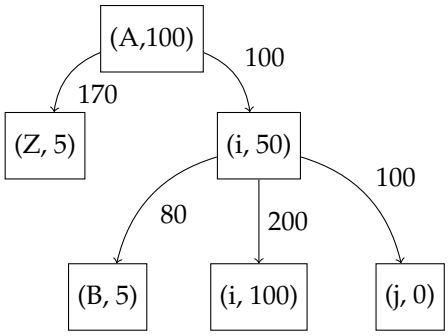
1.



2.

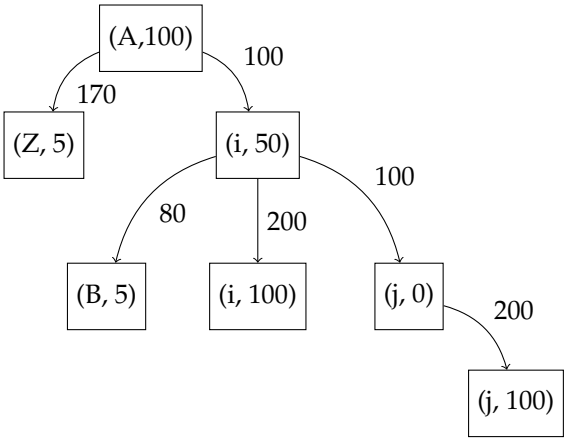


3.

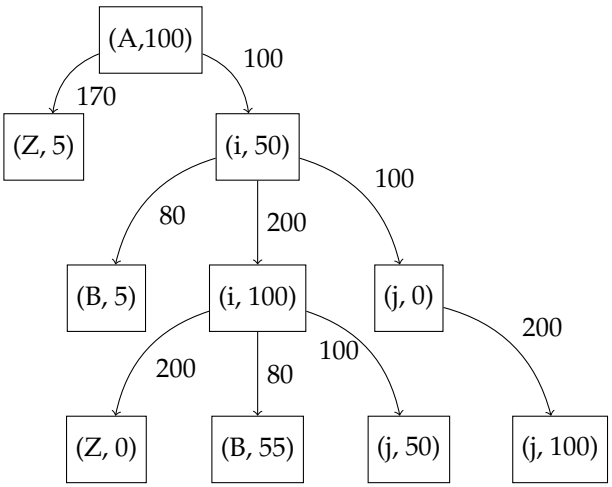


4. Die dritte und vierte Expansion von dem Knoten (Z,5) und (B,5) bewirken keine Veränderung des Baumes, da Sie keine Nachfolger haben. (Energie reicht nicht aus um zu einem anderen Knoten zu fahren)

5.



6.



e) Die klassische dynamische Programmierung würde, wenn man auf einen Knoten stößt, der schon einmal im Baum vorhanden ist (bei dem sich also das Auto an der gleichen Stelle befindet und die gleiche Ladung aufweist) immer den Knoten bevorzugen, dessen Pfad die geringeren akkumulierten Aktionskosten hat. Dies wird durch eine Tabelle garantiert, die beim Suchen im Speicher gehalten wird und für jeden Zustand (also jede Kombination aus Ladestand und Position) die geringsten Pfadkosten enthält, mit denen dieser Zustand bisher erreicht wurde. Normalerweise verhindert dies Zyklen und das unnötige mehrmalige Untersuchen von Pfaden, wenn diese nicht die optimale Lösung versprechen, da man davon ausgeht, dass die jeweils günstigste Teillösung auch zur günstigsten Gesamtlösung führt.
In unserem Fall werden trotzdem 'unnötige' Pfade untersucht, wenn man nicht beachtet, dass eine Position bei höheren Pfadkosten (also der Zustand unter Ausklammerung der Ladung) nur sinnvollerweise ein zweites Mal untersucht werden sollte, wenn die Ladung höher ist, als die beim letzten Besuch mit geringeren Pfadkosten. Dies könnte jedoch ohne Probleme und großen Aufwand als Constraint beim Benutzen der dynamischen Programmierung übernommen werden.

Aufgabe 2

a) **Zustandsraum:** $((w_1, x_1, y_1), (w_2, x_2, y_2), (w_3, x_3, y_3), (w_4, x_4, y_4), (w_5, x_5, y_5))$ wobei $x_i, y_i \in \{(a, b) \mid a \in \{1, \ldots, 5\}, b \in \{1, \ldots, 5\}\}$ mit $i \in \{1, \ldots, 5\}$
Wir können maximal 5 Stapel bilden, da wir 5 Kisten haben. Jede Kiste wird beschrieben durch ein Tupel von x und y welche den Stapel sowie die Höhe in diesem Stapel beschreiben.w_1 beschreibt den Wert der Kiste, man könnte auch die Reihenfolge dafür benutzen, aber dadurch werden die Formalitäten kniffliger.
Anfangszustand: $((1, 1, 2), (2, 1, 1), (3, 1, 4), (4, 1, 3), (5, 1, 5))$
Zielzustand: $((1, x_1, 5), (2, x_2, 4), (3, x_3, 3), (4, x_4, 2), (5, x_5, 1))$ mit $x_1 = x_2 \wedge x_2 = x_3 \wedge x_3 = x_4 \wedge x_4 = x_5$
Aktionen:

1. $$\text{bewegen}(von, nach, ((w_1, x_1, y_1), (w_2, x_2, y_2), (w_3, x_3, y_3), (w_4, x_4, y_4), (w_5, x_5, y_5)) \rightarrow ((c_1, a_1, b_1), (c_2, a_2, b_2), (c_3, a_3, b_3), (c_4, a_4, b_4), (c_5, a_5, b_5))$$

   die Bedingungen in Python/Javascript-Pseudocode:

```
# w bezeichnet auch den Index in der liste
current = [[w_1,x_1, y_1], [w_2,x_2, y_2], [w_3,x_3, y_3],
           [w_4,x_4, y_4], [w_5,x_5, y_5]]
oberste_kiste_von = current.filter(lambda x: x[1] == von)
                           .max(key=lambda x, y: x[2] > y[2])
oberste_kiste_nach = current.filter(lambad x: x[1] == nach)
                            .max(key=lambda x, y: x[2] > y[2])
position = oberste_kiste_von[0 - 1] # TODO
return current[0:positon] +
       [[oberste_kiste_von[0], nach, oberste_kiste_nach[2] + 1]] +
       current[position:]
```

b) Eine zulässige Heuristik ist:

Aufgabe 3

a) Die iterative Tiefensuche vereint die Vorteile von Tiefen- und Breitensuche. Es handelt sich ebenfalls um eine uninformierte Suche, die für Probleme mit uniformen Aktionskosten die Optimalität der Breitensuche und den geringen Speicherverbrauch der Tiefensuche vereint. Im Gegensatz zu normalen Tiefensuche ist sie auch Vollständig für Suchbäume mit unendlich langen Pfaden.
Dies wird dadurch erreicht, dass in jedem Iterationsschrit eine Tiefensuche bis zu einer begrenzten Tiefe ausgeführt wird. Nach jedem Iterationsschritt wird diese Tiefe um 1 erhöht. Damit werden beispielsweise, wenn man bei der Tiefe 0 anfängt, erst alle Knoten, die über Pfade der Länge 0 erreichtw erden können untersucht. Dann alle Knoten, die über Pfade der Länge 1 erreicht werden können und so weiter.
Dadurch, dass jedes Mal eine Tiefensuche stattfindet, ist der Speicherbedarf gering, da immer nur der aktuell betrachtete Ast im Speicher gehalten werden muss. Durch die Begrenzung der Tiefe wird, sofern eine Lösung vorhanden ist, wie bei der Breitensuche bei uniformen Aktionskosten auch eine Lösung auf der niedrigsten Tiefe gefunden, auf der Lösungen gefunden werden können. Wenn mehrere Lösungen auf dieser Ebene existieren, ist es egal, welche von diesen wir finden, da alle die gleichen Kosten haben.

b) Beim vollständigen Durchsuchen des Baumes müssen der Wurzelknoten 6 Mal (also die Tiefe+1), alle Knoten der Tiefe 1 5 Mal, alle Knoten der Tiefe 2 4 Mal usw. generiert werden. Es ergibt sich also folgende Rechnung:

$$6 * 1 + 5 * 35 + 4 * 35^2 + 3 * 35^3 + 2 * 35^4 + 35^5 = 55656831$$

Bei der normalen Tiefensuche wird jeder Knoten genau ein mal generiert, wir haben also:

$$1 + 35 + 35^2 + 35^3 + 35^4 + 35^5 = 54066636$$

Insgesamt haben wir in diesem Fall bei der iterativen Tiefensuche also nur einen Overhead von ungefähr 3 Prozent:

$$55656831/54066636 = 1.029411761$$

c) Je höher der Verzweigungsfaktor ist (solange er nicht unendlich ist), desto geringer ist der prozentuale Overhead der iterativen Tiefensuche, denn die Anzahl der Blätter wächst mit einer Vergrößerung des Verzweigungsfaktors sehr stark. Die Blätter machen den größten Teil des Suchbaumes aus und werden sowohl bei der Tiefensuche als auch bei der iterativen Tiefensuche nur einmal generiert. Die mehrfache Generierung der Knoten geringerer Tiefen fällt bei einem hohen Verzweigungsfaktor also nicht so sehr ins Gewicht, wie bei einem niedrigen Verzweigungsfaktor, da der Unterschied der Anzahl an Knoten zweier Tiefen bei großem Verzweigungsfaktor wesentlich höher ist.
So sind bei einem Verzweigungsfaktor von 2 beispielsweise immer doppelt so viele Knoten in der nächst tieferen Ebene, wie in der Ebene darüber. Beim Verzweigungsfaktor 5 sind es hingegen schon fünf Mal so viele Knoten.

Aufgabe 4

a)

Schritt	Expandiert	Queue	Anmerkung
1.	A(35)	AC(30), AB(40)	-
2.	AC(30)	ACZ(35), ACD(40), AB(40)	ACA (Zyklus), ACB (Dyn. Progr.)

A* terminiert nun mit dem Ergebnis, dass ACZ der kürzeste Pfad von A nach Z ist, da ACZ in der Queue ganz vorne steht.

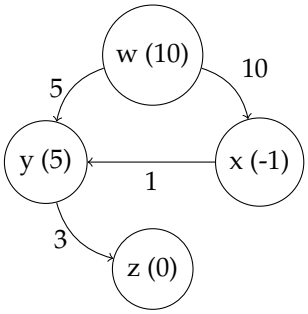
b) Die Heuristik überschätzt die Kosten von B nach Z, denn die echten Kosten sind 25, die Heuristik prognostiziert aber 35. Dadurch handelt es sich um eine unzulässige Heuristik und die Optimalität von A* ist nicht mehr gewährleistet, was sich auch in unserer Simulation zeigt. Statt AB zu expandieren und den kürzesten Weg zu finden, terminiert A* wegen der Überschätzung frühzeitig.

c) Wir wählen für B 20 statt 35 als heuristische Kostenschätzung, damit A* eine optimale Lösung liefert:

Schritt	Expandiert	Queue	Anmerkung
1.	A(35)	AB(25), AC(30)	-
2.	AB(25)	ABC(25), ABD(30), AB(40)	ABA (Zyklus), AC (Dyn. Progr.)
3.	ABC(25)	ABCZ(30), ABD(30)	ABCA(Zykl.), ABCB (Zykl.), ABCD (Dyn. Progr.)

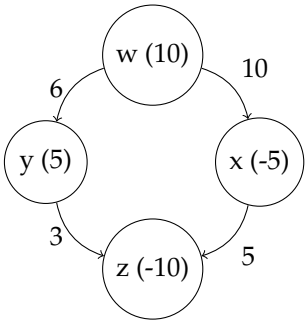
A* terminiert nun mit dem optimalen Ergebnis ABCZ als kürzesten Pfad zwischen A und Z.

d) Betrachten wir folgenden Suchgraphen, in dem der kürzeste Weg von w nach z gefunden werden soll. Der Wert der Heuristik ist für jeden Knoten in Klammern hinter dem Bezeichner notiert.



Der Wert der Heuristik für den Knoten 'x' ist negativ. Dadurch würde A* nach dem Expandieren des Startknotens 'w' vorerst 'x' expandieren, da der Wert der Funktion, die die Kosten berechnet, nach denen die Zustände in die Queue eingeornet werden, für 'x' geringer ist, als für 'y' (10+(-1) = 9 für 'x' und 5+5 = 10 für 'y').
Dadurch wird das zielgerichtete Verhalten von A* kurzzeitig negativ beeinflusst. Denn der kürzeste Weg führt offensichtlich direkt über 'y' und selbst eine uninformierte Suche, die nur die bisherigen Pfadkosten kennt (zB. Breitensuche), würde in diesem Fall vorerst 'y' expandieren. Mit negativem Wert kann die Heuristik also sogar mehr Suchaufwand als nötig verursachen, indem sie A* sozusagen 'in die Irre führt'.

e) Eine negative Heuristik kann sowohl Vollständigkeit als auch Optimalität von A* verhindern. Für die Optimalität besteht ein Problem besonders dann, wenn der Zielknoten und den Knoten auf einem suboptimalen Pfad negative Werte zugeteilt werden. Betrachten wir für die Optimalität folgendes Beispiel:



A* würde mit dem Pfad 'wxz' terminieren, obwohl 'wyz' kürzer ist. Aber sowohl f(wx) = 5 als auch f(wxz) = 5 sind kleiner als f(wy) = 11 (für f(x) = g(x) + h(x) mit g(x) sind die tatsächlichen Pfadkosten und h(x) ist der heuristische Wert). Also findet A* nicht den optimalen Pfad. Wenn dem Zielzustand kein negativer Wert zugewiesen wird, bleibt die Optimalität jedoch erhalten.

Dei Vollständigkeit bleibt bei endlichen Zustandsräumen bestehen, denn dann werden im worst case auch bei negativer Heuristik alle Pfade expandiert, eine vorhandene Lösung also gefunden. Für unendliche Zustandsräume kann allerdings keine Vollständigkeit garantiert werden, denn negative Werte einer Heuristik können anfallende Pfadkosten von unendlichen Pfaden immer kleiner halten als den Weg zur Lösung, wenn die heuristichen Werte mit dem unendlichen Pfad immer weiter ins negative wachsen.