Grundlagen der Künstlichen Intelligenz

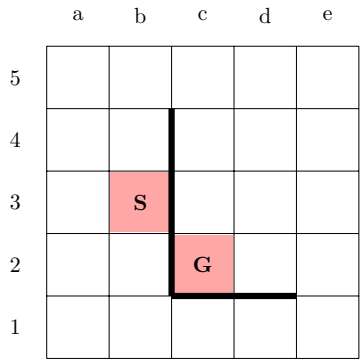
Prof. Dr. J. Boedecker, Prof. Dr. W. Burgard, Prof. Dr. F. Hutter, Prof. Dr. B. Nebel
M. Krawez, T. Schulte
Sommersemester 2018

Universität Freiburg
Institut für Informatik

Übungsblatt 2 — Lösungen

Aufgabe 2.1 (Informierte Suche)

Ein Haushalts-Roboter will den kürzesten Pfad von S (Start) nach G (Ziel) bestimmen. Der Roboter kann sich in jedem Schritt um eine horizontal oder vertikal verbundenen Zelle fortbewegen, sofern diese nicht durch eine Wand (dicke schwarze Linie) von der aktuellen Zelle getrennt ist. Jede Bewegung hat uniforme Kosten von 1. Abbildung (a) zeigt den Initialzustand, Abbildung (b) die Heuristikwerte.



(a) Initialzustand

	a	b	c	d	e
5	5	4	3	4	5
4	4	3	2	3	4
3	3	2	1	2	3
2	2	1	0	1	2
1	3	2	1	2	3

(b) Heuristikwerte

(a) Führen Sie eine A*-Suche durch, um den kürzesten Pfad von S nach G zu bestimmen. Tragen Sie von allen generierten Knoten die f-Werte in die entsprechenden Zellen in Abbildung (a) ein. Alle anderen Zellen sollen frei bleiben.

Lösung:

	a	b	c	d	e
5	8	6	6	8	
4	6	4	6	8	
3	4	**S**	6	8	
2	4	2	**G**		
1	6	4	4	6	8

($e1$ is optional)

(b) Geben Sie die Definition einer *zulässigen Heuristik* an. Ist die Heuristik aus Abbildung (b) zulässig?

Lösung:

A heuristic h is admissible if $h(n) \leq h^\star(n)$ for all n, where $h^\star$ is the optimal heuristic. I.e. h is admissible if it never over estimates the cost of the cheapest solution from n to a goal. The heuristic shown in figure (b) is admissible. (It's the Manhattan Distance heuristic.)

(c) Wieviele Knoten muss A* expandieren, wenn h^* als Heuristik verwendet wird, wobei $h^*(n)$ die tatsächlichen Kosten eines optimalen Plans von n zum Ziel G angibt?

Lösung:

Je nach Implementation entweder 6 oder 7 (je nachdem ob der Zielzustand expandiert wird oder nicht). Die Implementierung aus der Vorlesung benötigt 7 Expansionen (b3, b4, b5, c5, c4, c3, c2).

Aufgabe 2.2 (Lokale Suche)

Wir werden nun *Hill-Climbing* im selben Zusammenhang (Roboternavigation auf einem Grid mit Wänden als Hindernisse) untersuchen.

(a) Erklären Sie, wie *hill-climbing* als Methode zum Erreichen eines bestimmten Punktes in der Ebene durchgeführt werden würde.

(b) Erläutern Sie mit Hilfe eines Beispiels, warum nicht-konvexe Hindernisse (bestehend aus mehreren Wänden) zu einem lokalen Maximum für den Algorithmus führen können.

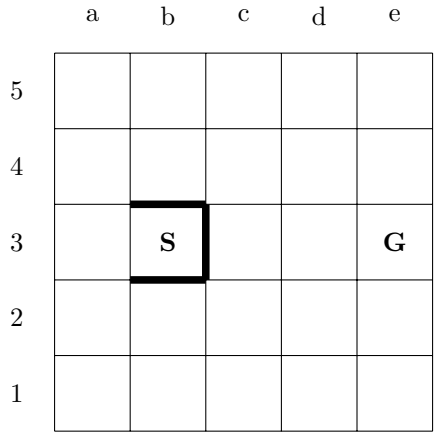
(c) Ist es möglich, bei konvexen Hindernissen zu einer nicht-optimalen Lösung zu gelangen?

(d) Würde *simulated annealing* bei dieser Problemfamilie immer aus lokalen Maxima herausfinden? Begründen Sie Ihre Antwort.

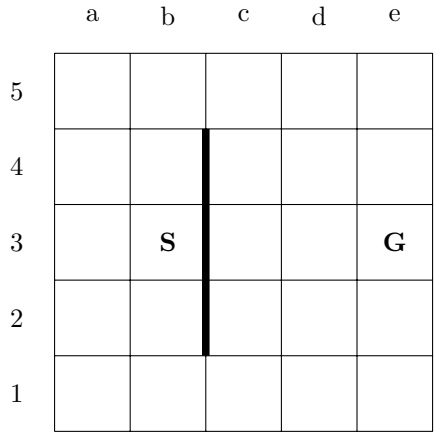
Lösung:

(a) Hill-climbing is surprisingly effective at finding reasonable if not optimal paths for very little computational cost, and seldom fails in two dimensions. Let $d(x, y)$ be the straight-line distance between point x and point y, and let the value of a point p be $-d(p, goal)$. An agent can move to any vertex accessible by a single straight-line movement, and when hill-climbing it will choose the accessible vertex nearest the goal.

(b) This can occur when an agent is at a non-convex obstacle as shown in the figure below.



(c) It is possible but unlikely. This requires that the agent be at a vertex that is closer to the goal than any of the other vertices of the obstacle, as illustrated in the figure below.



(d) If a solution exists (the agent can't be entirely walled-in) and one chooses an appropriate annealing schedule, simulated annealing can escape local maxima for this family of problems.

Aufgabe 2.3 (Suchalgorithmen)

Beweisen Sie die folgenden Aussagen:

(a) Breitensuche ist ein Spezialfall der uniformen Kostensuche.

Lösung:

Breitensuche expandiert immer einen unexpandierten Knoten minimaler Tiefe, während uniforme Kostensuche immer einen unexpandierten Knoten mit minimalen Pfadkosten expandiert. Sind die Aktionskosten bei uniformer Kostensuche konstant 1, so hat ein Knoten genau dann Tiefe k, wenn seine Pfadkosten k betragen. Insbesondere hat er also minimale Tiefe genau dann, wenn er minimale Pfadkosten hat. Die Expansionskriterien sind somit identisch.

(b) Breitensuche, Tiefensuche und uniforme Kostensuche sind Spezialfälle der gierigen Bestensuche (greedy best-first search).

Lösung:

Dass Breitensuche ein Spezialfall der uniformen Kostensuche ist, wurde schon in der letzten Teilaufgabe gezeigt. Es bleibt also noch zu zeigen, dass Tiefensuche und uniforme Kostensuche Spezialfälle der gierigen Bestensuche sind.
Tiefensuche ist ein Spezialfall der gierigen Bestensuche für $h(n) := -depth(n)$.
Uniforme Kostensuche ist ein Spezialfall der gierigen Bestensuche für $h(n) := g(n)$

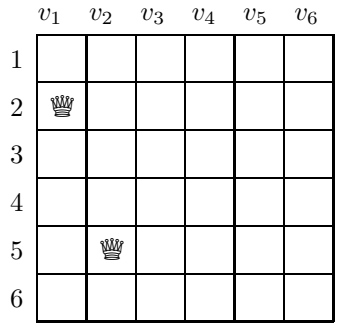
(c) Uniforme Kostensuche ist ein Spezialfall der A*-Suche.

Lösung:

Die Bewertungsfunktion bei uniformer Kostensuche ist $f(n) = g(n)$, bei A*-Suche $f(n) = g(n) + h(n)$. Setze also $h \equiv 0$.

Aufgabe 2.4 (Forward Checking / Kantenkonsistenz)

Betrachten Sie das 6-Damen Problem, bei dem 6 Spielfiguren auf einem 6×6 Felder großen Brett so platziert werden sollen, dass sich keine zwei Damen auf der selben horizontalen, vertikalen oder diagonalen Line befinden. Der Wertebereich sei $dom(v_i) = 1, \ldots, 6$ für alle Variablen $v_i \in V$. Betrachten Sie nun den Zustand $\alpha = \{v_1 \mapsto 2, v_2 \mapsto 5\}$.



(a) Erzeugen Sie Kantenkonsistenz in α. Geben Sie hierzu insbesondere die Wertebereiche der Variablen vor und nach dem Erzeugen der Kantenkonsistenz an. Sie dürfen annehmen, dass der Wertebereich von Variablen mit bereits zugewiesenen Werten nur aus dem zugewiesenen Wert besteht, während unbelegte Variablen den vollen Wertebereich haben. Wählen Sie immer diejenige Variable mit niedrigstem Index, für welche noch keine Kantenkonsistenz erzeugt wurde.

(b) Führen Sie Forward-Checking in α aus. Vergleichen Sie das Ergebnis mit (a).

Lösung:

(a)

In der Tabelle sind die Domänen nach dem Erzeugen der Kantenkonsistenz für alle Constraints an denen die jeweilige Variable beteiligt ist.

	v_1	v_2	v_3	v_4	v_5	v_6
AC-v_1 = ACv_2	{2}	{5}	{123456}	{123456}	{123456}	{123456}
AC-v_3	{2}	{5}	{13}	{123456}	{123456}	{123456}
AC-v_4	{2}	{5}	{13}	{146}	{123456}	{123456}
AC-v_5	{2}	{5}	{13}	{146}	{4}	{123456}
AC-v_4	{2}	{5}	{13}	{16}	{4}	{123456}
AC-v_6	{2}	{5}	{13}	{16}	{4}	{6}
AC-v_3	{2}	{5}	{1}	{16}	{4}	{6}
AC-v_4	{2}	{5}	{1}	{}	{4}	{6}

Immer wenn die Domäne einer Variablen reduziert wird werden alle Constraints an denen die Variable beteiligt ist wieder in die *queue* gesteckt. Dies kann dann zu weiteren Änderungen führen.
(Im zweiten Durchlauf steht AC-v_i nicht mehr für alle Constraints von v_i, sondern nur für die, die wieder in die *queue* eingefügt wurden)

(b)

	v_1	v_2	v_3	v_4	v_5	v_6
FC $dom(v)$	{2}	{5}	{13}	{146}	{134}	{346}

Wenn man die Arc Consistency von v_5 checkt, stellt man fest, dass v_3 keine erlaubten Werte mehr hat, wenn man $v_5 \mapsto 1$ oder $v_5 \mapsto 3$ zuweist. Dies wird durch forward checking nicht erkannt. Arc-Consistency hat zudem eine *queue* und propagiert Änderungen an einer Domäne weiter so dass die Constraints einer Variable eventuell mehrfach getestet werden. In diesem Fall findet AC-3 sofort heraus, dass die Constraints unerfüllbar sind, während der Such-Algorithmus mit Forward-Checking noch deutlich mehr zu tun hat.