Grundlagen der künstlichen Intelligenz: Hausaufgabe 2

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

Aufgabe 1

Alle Aufgaben wurden in **Clojure1 1.5** geschrieben und benutzen nichts außer *clojure.core* als namespace.

Backtracking

```
(def counter (atom 0))

(defn solve-stack [n]
  "exacly the algorithm mentioned in the exercise"
  (let [domains (vec (for [i (range n)] (vec (range n))))
        constrains [(fn [state] (= (count (distinct state)) (count state)))
                    (fn [state] (let [j (dec (count state))]
                                  (every? true? (for [i (range j)]
                                                  (not= (Math/abs (- i j))
                                                        (Math/abs (- ((domains i) (nth state (- j i)))
                                                                     ((domains j) (first state))))))))))]]
    (loop [[x & xs] [[]]]
      (if (or (nil? x) (= n (count x)))
        x
        (recur (concat (for [i (domains (count x))
                             :when (every? true? (map #(% (cons i x)) constrains))]
                         (do (swap! counter inc)
                             (cons i x))) xs))))))
```

Erklärung:

- **domains** repräsentiert die Domain einer jeden Dame und sieht für `n = 2` so aus: `[[0, 1], [0, 1]]`.
- **constrains** ist eine Liste aus Funktionen die jeweils einen Belegungszustand bekommen, wobei eine Belegung eine Liste von 0 bis `n` Elementen ist, z.B. `[0, 1]`. Sie stellen die constrains dar. Eine Funktion testet das keine Dame eine andere horizontal schlagen kann und die andere, dass sie es nicht diagonal können. Die Funktionen liefern ein `Bool` zurück.
- **loop** dies ist die Rekursion und der eigentliche Algorhitmus. Zum Start bekommt die Funktion einen Stack mit einer leeren Belegung `[[]]`, durch destructuring setzen wird `x []` zugewiesen. Als Rekursionsanker wird überprüft ob `x` einen Wert hat, also ob der Stack leer ist, oder ob wir eine gültige Belegung gefunden haben. In dieser Implementierung ist eine gültige Belegung eine Liste der Länge `n`. Falls eine der Bedingungen stimmt, wird die `x` zurückgegeben, also entweder eine gültige Belegung oder `nil` falls es keine Belegung gibt. Anonsten werden mittels list-comprehension die neuen Werte für den Stack bestimmt, wobei nur Werte auf den Stack gelegt werden die alle constrains beachten (backtracking). Der `counter` wird um für jeden expandierten Knoten um eins erhöht und zum Schluss werden die Werte vorne an den Stack gehängt.

Backtracking + Forward Checking + Minimum Remaining Values

```
(def counter (atom 0))

(defn solve-stack-fc-mrv [n]
  "exacly the algorithm mentioned in the exercise with forward checking and mrv"
  (let [domains-initial (zipmap (range n) (repeat (range n)))
        constrains [(fn [[x1 y1] [x2 y2]] (not= y1 y2))
                    (fn [[x1 y1] [x2 y2]]
                      (not= (Math/abs (- x1 x2))
                            (Math/abs (- y1 y2))))]
        forward-check (fn [x1 y1 domains]
                        (apply merge
                               (for [[x2 domain] domains]
                                 {x2 (filter (fn [y2] (every? true? (map #(% [x1 y1] [x2 y2]) constrains)))
    (loop [[[state domains] & xs] [[{} domains-initial]]]
      (if (or (nil? state) (= n (count state)))
        (map second (sort state))
        (let [[x domain] (rand-nth ((comp second first) (sort (group-by (comp count second) domains))))
              domains (dissoc domains x)
              states (filter #(every? (comp (partial < 0) count) (second %))
                             (for [y domain]
                               [(assoc state x y) (forward-check x y domains)]))]
          (recur (do (swap! counter (partial + (count states)))
                     (concat states xs))))))))
```

Erklärung

- **domains-initial** repräsentiert die initiellen Domains einer jeden Dame und sieht für `n = 2` so aus: `{0: [0 1], 1: [0 1]}`. Also dictionary/map wo die keys die jeweilige Dame sind und die values die Domain der Dame.
- **constrains** ist eine Liste aus Funktionen die jeweils einen 2 x/y Koordinaten bekommen, jede Koordinate ist die Position einer Dame. Sie überprüfen ob die Damen sich horizontal oder diagonal schlagen können.
- **forward-check** Dies ist der forward-check, die Funktion bekommt die Position der zu setzenden Dame `x1 y1` und die aktuellen Domains `domains`, wobei die Domain der zu setzenden Dame entfernt wurde. Sie liefert die *bereinigten* Domains zurück.
- **loop** Jeder *Node* wird mittels eines Vektors aus Belegung und Domains dargestellt, der Start-stack besteht damit aus einem *Node* der wiederum aus einer leeren Belegung hat und den initiellen Domains besteht: `[ domains-initial]`. Der Rekursionsanker ist ähnlich wie zuvor. Wenn die Funktion nicht terminiert bestimmt sie nach **mrv-Prinzip** den nächsten Wert `(rand-nth ...)`. Die Domain für diesen Wert wird entfernt (siehe forward-check), mittels list-comprehension werden alle gültigen nächsten Zustände ermittelt, bei welchen der Forward-check die Domains eingeschränkt hat. Falls es keine gültigen Folgezustände für die aktuelle Belegung gibt, ist **states** leer, und im nächsten Rekursionschritt wird der eine andere Belegung getestet: `(recur (concat states xs))`. `(swap!..)` ist dafür da den Counter hochzuzählen.

Backtracking + Arc Consistency + Minimum Remaining Values

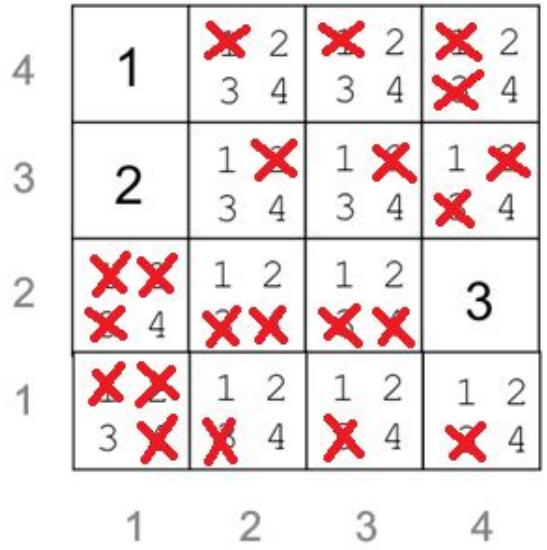
```
(defn solve-stack-arc-mrv [n]
  "exacly the algorithm mentioned in the exercise with arc-consistency and mrv"
  (let [domains-initial (zipmap (range n) (repeat (range n)))
        constrains [(fn [[x1 y1] [x2 y2]] (not= y1 y2))
                    (fn [[x1 y1] [x2 y2]]
                      (not= (Math/abs (- x1 x2))
                            (Math/abs (- y1 y2))))]
        arc-consistency (fn [domains]
                          (let [arcs (for [[x1 d1] domains
                                           [x2 d2] domains
                                           :when (not= x1 x2)]
                                       [x1 x2])]
                            (loop [[[x1 x2] & xs] arcs domains domains]
                              (if (nil? x1) domains
                                (let [d1 (domains x1)
                                      d2 (domains x2)
                                      d1-new (loop [[y1 & ys] d1 d1new []]
                                               (if (nil? y1)
                                                 d1new
                                                 (let [xy (for [y2 d2]
                                                            [[x1 y1] [x2 y2]])
                                                       xy1 (map (comp second first)
                                                                (filter (fn [[[x1 y1] [x2 y2]]]
                                                                          (every? true? (map #(% [x1 y1]
                                                                                                 [x2 y2])
                                                                                             constrains))) xy))]
                                                   (if (empty? xy1)
                                                     (recur ys d1new)
                                                     (recur ys (cons y1 d1new))))))
                                      new-arcs (for [[x3 y3] domains
                                                     :when (and (not= x3 x2) (not= x3 x1))] [x3 x1])]
                                  (if (not= (sort d1-new) (sort d1))
                                    (recur (concat xs new-arcs) (assoc domains x1 d1-new))
                                    (recur xs domains))))))))]
    (loop [[[state domains] & xs] [[{} domains-initial]]]
      (if (or (nil? state) (= n (count state)))
        (map second (sort state))
        (let [[x domain] (rand-nth ((comp second first)
                                    (sort (group-by (comp count second)
                                                    (filter (fn [[x d]] (not (state x))) domains)))))
              states (for [y domain
                           :let [domains-new (arc-consistency (assoc domains x [y]))]
                           :when (empty? (filter (fn [[x d]] (zero? (count d))) domains-new))]
                       [(assoc state x y) domains-new])]
          (recur (do (swap! counter (partial + (count states)))
                     (concat states xs))))))))
```

Erklärung

- **domains-initial und constrains** Genau wie in forward-check.
- **arc-conistency** Bekommt als Eingabe alle Domains, wobei die Domains von gesetzten Damen einstellig sind. Benutzt viel Rekursion und sieht unheimlich komplex wegen der Berechnung von `d1-new` aus, könnte vermutlich deutlich vereinfacht werden, dafür fehlt mir aber gerade die Zeit.
- **loop** Ziemlich genau wie beim forward-check, nur entfernen wir nicht die belegten Variablen aus `domains` da wir die Belegung für den arc-consistency check brauchen. Dadurch hat sich aber etwas die Berechnung für die nächste Varibale geändert (nur Variablen aus `domains` deren key noch nicht in `state` ist): `(filter (fn [[x d]] (not (state x))) domains)`. Für den aktuellen Zustand werden dann wieder die Folgezustände asugerechnet, wobei keiner eine leere domain haben darf: `:when (empty? (filter (fn [[x d]] (zero? (count d))) domains-new))`.

Aufgabe 2

a) Das Zuordnungsproblem in 2-konsistentem Zustand:



b) Wir wählen für den nächsten Schritt die MRV-Heuristik, da wir Variablen mit einem kleinen Wertebereich möglichst schnell belegen wollen, um früh auf mögliche Fehler zu stoßen. Allerdings ergibt es nach dem Herstellen des 2-konsistenten Zustands keinen Sinn, Variablen auszuwählen, die einen ein-elementigen Wertebereich haben, denn aufgrund der 2-Konsistenz kann keine zukünftige Belegung anderer Variablen dazu führen, dass die Ein-elementigen nicht mehr belegt werden könnten. Daher können sie später belegt werden, wenn der Rest des Zuordnungsproblems gelöst ist.
Da wir im nächsten Schritt Forward Checking verwenden werden, ist es sinnvoll MCV als Tie-Breaker zu verwenden, um eine Variable zu wählen, die den Wertebereich von möglichst vielen anderen Variablen betrifft, falls es mehrere Variablen gibt, nachdem die MRV-Heuristik angewendet wurde.
Für unser Problem sind die Wertebereiche der Variablen $(2,2)$, $(3,2)$, $(4,3)$, und (4.4) im 2-konsistenten Zustand alle gleich klein (zwei Elemente).
Der Tie-Breaker MCV liefert uns nun $(2,2)$ und $(3,2)$ da diese jeweils Constraints zu noch 5 unbelegt Variablen haben (die anderen beiden kandidaten jeweils nur 4). Wir wählen eine dieser Variablen zufällig für unseren nächsten Schritt.

c) Sei die von uns zufällig gewählte Variable der beiden oben genannten Kandidaten $(2,2)$. Dann belegen wir diese mit dem Wert 1 und erhalten so mittels Forward-Checking folgende eingeschränkte Wertebereiche:
$(2,1) = \{2,4\}$
$(2,3) = \{3,4\}$
$(3,2) = \{2\}$
Alle anderen Wertebereiche bleiben unverändert.

Aufgabe 3

a) Anzahl der Konflikte zu Beginn (alle Knoten weiss): 10

Schritt	Knoten	Farbe	resultierende Konflikte
1.	B	Grau	6
2.	A	Schwarz	4
3.	C	Schwarz	2
4.	F	Grau	1

Ergebnis: Der Algorithmus terminiert, ohne eine Lösung (vollständige und konsistente Belegung) zu finden.

b) Wir ändern die Tie-Break-Regel nun so, dass wir den lexikographisch größten Knoten wählen, wenn die Anzahl der resultierenden Konflikte gleich ist. Somit würde der Algorithmus wie folgt ablaufen:

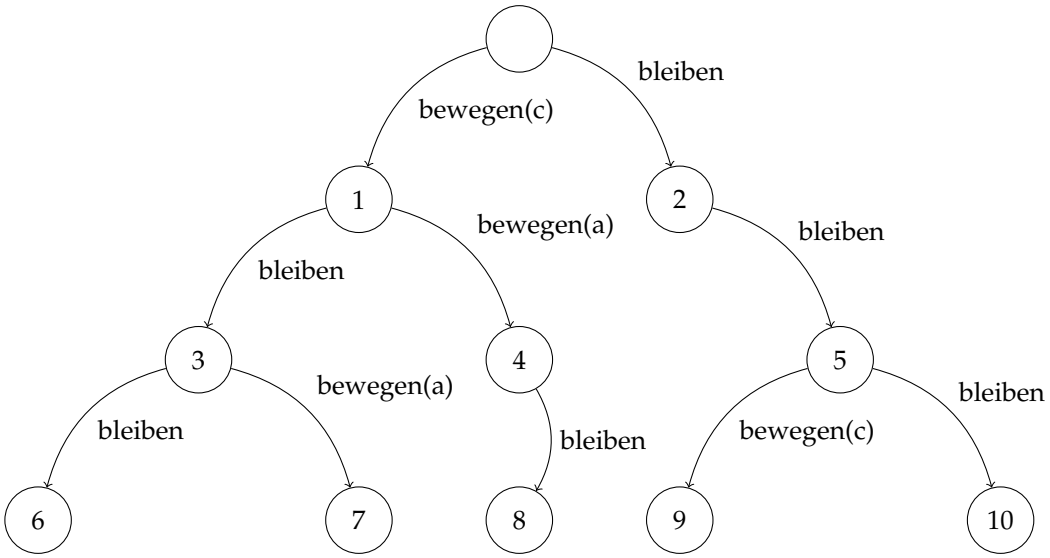
Schritt	Knoten	Farbe	resultierende Konflikte
1.	B	Grau	6
2.	G	Grau	4
3.	E	Schwarz	2
4.	A	Schwarz	0

Ergebnis: Wir finden eine Lösung für das CSP!

Aufgabe 4

a) (a, b, c, p_1, p_2, z)
a, b, c: Rohstoffe an den entsprechenden Knoten.
p_1, p_2: Positionen der entsprechenden Roboter.
z: Welcher Roboter ist am Zug.
Startzustand:
$(2, 16, 16, a, b, 1)$

b)



1: $(2, 16, 8, c, b, 2)$
2: $(1, 16, 16, a, b, 2)$
3: $(2, 8, 8, c, b, 1)$
4: $(1, 16, 8, c, a, 1)$
5: $(1, 8, 16, a, b, 1)$
6: $(2, 8, 4, c, b, 2)$
7: $(1, 8, 8, a, b, 2)$
8: $(1, 16, 4, c, a, 2)$
9: $(1, 8, 8, c, b, 2)$
10: $(0, 8, 16, a, b, 2)$

c) $6 : 4|7 : 1|8 : 11|9 : 1|10 : -6$

Nach dem Minimax-Algorithmus ist der beste Zug der Schritt zu C mit einem Ertrag von mindestens 1 nach der gegebenen Simulationstiefe.

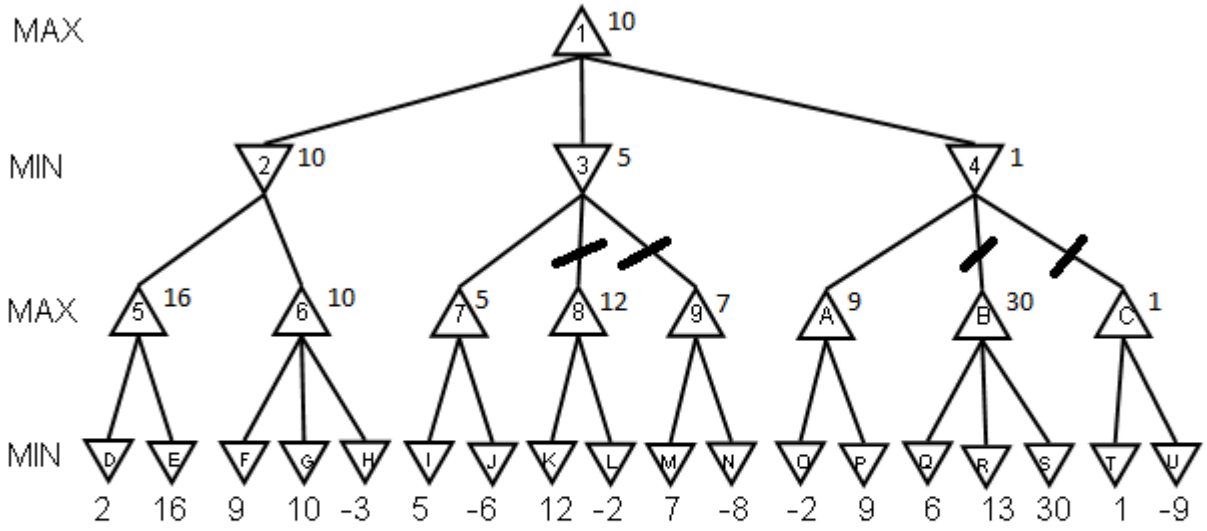
d) Die Nachfolgeknoten nach den Knoten 7 und 10 würden nach dem α -cutoff abgeschnitten werden.
Die Nachfolgeknoten von 6, 8 und 9 müssten weiter untersucht werden.

e) Bei der Aktuellen Baumtiefe sieht es so aus als sei es der beste Zug sofort zu c zu wechseln. Wenn allerdings der a-Knoten zuerst völlig ausgeschöpft wird kann Roboter 1 mit absoluter Sicherheit gewinnen.

f)

Aufgabe 5

a) Spielbaum mit allen Minimax-Werten an den Knoten:



b) Die Zweige, die durch Cut-Offs entfernt würden, sind im obigen Bild markiert.