Konzepte zur Entwicklung eines automatischen Spielers für Malefiz

Michael Barth

Alexander Knapp

Philipp Meier

LMU

Institut für Informatik
Ludwig-Maximilians-Universität München

Allgemeines - Positionierung

- Spiele haben
  - Regelmenge
  - Zustandsraum
- Kontinuierlicher Zustandsraum
  - Beispiel: Fussball, Tennis, Kegeln ...
- Diskreter Zustandsraum
  - Beispiel: Kartenspiele, Brettspiele ...

Allgemeines - Spieltypen

	perfekte Information	unvollständige Information
deterministisch	**Dame,** **Schach,** **Go,** **...**	**Schafkopf** **(nach dem Geben)** **...**
zufällig	**Malefiz,** **Backgammon,** **...**	**Poker,** **Romme,** **Siedler von Catan,** **...**

Allgemeines - Abstraktion des Spiels

Darstellung der Spielfelder als Baum/Graph

- Berechnung der Zugmöglichkeiten: Suche in Bäumen/Graphen

Darstellung der Züge als Baum/Graph

- Bewertung von Spielzuständen
- Aufgabe: Finden von Wegen zu besonders günstigen Spielzuständen
  d. h. gute Situation für mich / schlechte Situation für den Gegner

Allgemeines - Spielbäume

- Jeder Knoten entspricht einem Spielzug
- Reihenfolge enspricht den alternierenden am Zug befindlichen Spielern
- Zufallselemente (Würfel) werden als zusätzliche Knoten eingeschoben

Überblick

- Kürzeste Wege in Graphen
- Blinde Suche in Bäumen und Graphen
- Heuristische Suche in Bäumen
- Suche in Spielbäumen
- Spiele mit Zufall

Literatur

– S. J. Russell, P. Norvig. *Artificial Intelligence: A Modern Approach*. 2. Auflage, 2003.
– J. P. Bigus, J. Bigus. *Constructing Intelligent Agents with Java*. 1998.

Kürzeste Wege in Graphen

Initialisierung: $\textsc{PathCost}(\mathit{initial}) = 0$,
$\textsc{PathCost}(\mathit{node}) = \infty$ für alle Knoten des Graphen außer *initial*.

```
proc ShortestPath(initial, fringe) ≡
  ⌈closed ← ∅
   fringe ← Insert(initial, fringe)
   do
      if Empty?(fringe) then return  fi
      node ← RemoveMin(fringe)
      if node ∉ closed
        then closed ← closed ∪ {node}
             foreach ⟨s, v⟩ ∈ Successors(node) do
               if PathCost[s] > PathCost[node] + v
                 then PathCost[s] ← PathCost[node] + v
                      ParentNode[s] ← node
                      fringe ← Insert(s, fringe) fi
             od fi
   od⌋
```

Problemrahmen

- Zustandsraum
  - Menge von Zuständen
  - Anfangszustand
- Nachfolgerfunktion
  - bestimmt die Nachfolgerzustände eines Zustands
  - Erreichen von Nachfolgezuständen über Aktionen $n \xrightarrow{a} n'$
- Zielüberprüfung
  - Funktion auf Zuständen
  - Kriterium, ob das Ziel erreicht ist
- Pfadkosten
  - Kosten, einen Zustand auf einem bestimmten Pfad vom Anfangszustand zu erreichen

Problemrahmen für Malefiz

- Zustandsraum — Spielbrett
  - Menge von Zuständen — Spielfelder
  - Anfangszustand
- Nachfolgerfunktion — Nachbarschaft
  - bestimmt die Nachfolgerzustände eines Zustands
  - Erreichen von Nachfolgezuständen über Aktionen $n \xrightarrow{a} n'$
- Zielüberprüfung
  - Funktion auf Zuständen
  - Kriterium, ob das Ziel erreicht ist
- Pfadkosten
  - Kosten, einen Zustand auf einem bestimmten Pfad vom Anfangszustand zu erreichen

Spielgraphen

- Anfangszustand
  - gegeben durch den Spielstand
- Nachfolgerfunktion
  - Paare aus Zug und Spielstand
  - liefert die Spielstände nach den möglichen Zügen
- Endkriterium
  - Prüfung, ob das Spiel zu Ende ist
  - definiert die Endzustände des Spiels
- Bewertungsfunktion
  - Bewertung eines Endzustandes nach Gewinn oder Verlust
  - Spielwert eines Endzustandes

Allgemeine Suche in Bäumen

```
proc TreeSearch(problem, fringe) ≡
  ⌈fringe ← Insert(MakeNode(InitialState[problem]), fringe)
   do
      if Empty?(fringe) then return Failure fi
      node ← RemoveFirst(fringe)
      if GoalTest[problem](State[node]) then return Solution(node) fi
      fringe ← InsertAll(Expand(node, problem), fringe)
   od⌋
```

- Expand liefert die Nachfolger eines Knotens
- Datenstruktur für *fringe* entscheidet über Durchlaufstrategie

Expandieren von Knoten

proc EXPAND(*node*, *problem*) $\equiv$
  $\lceil$ *successors* $\leftarrow \emptyset$
    foreach $\langle$*action*, *result*$\rangle \in$ SUCCESSORS[*problem*](STATE[*node*]) **do**
      s $\leftarrow$ NEWNODE
      STATE[*s*] $\leftarrow$ *result*
      PARENTNODE[*s*] $\leftarrow$ *node*
      ACTION[*s*] $\leftarrow$ *action*
      PATHCOST[*s*] $\leftarrow$ PATHCOST[*node*] $+$ STEPCOST(*node*, *action*, *s*)
      DEPTH[*s*] $\leftarrow$ DEPTH[*node*] $+ 1$
      successors $\leftarrow$ *successors* $\cup \{s\}$
    od
    return *successors* $\rfloor$

Allgemeine Suche in Graphen

```
proc GRAPHSEARCH(problem, fringe) ≡
  ⌈closed ← ∅
   fringe ← INSERT(MAKENODE(INITIALSTATE[problem]), fringe)
   do
      if EMPTY?(fringe) then return FAILURE fi
      node ← REMOVEFIRST(fringe)
      if GOALTEST[problem](STATE[node]) then return SOLUTION(node) fi
      if STATE[node] ∉ closed
        then closed ← closed ∪ {STATE[node]}
             fringe ← INSERTALL(EXPAND(node, problem), fringe) fi
   od⌋
```

Heuristische Suche

- Einführung einer heuristischen Funktion h
  - h bewertet einen Knoten n mit den geschätzten Kosten des günstigsten Pfades von n zum Ziel.
- Heuristik kann im EXPAND-Schritt von TREESEARCH zur Reihung nach erfolgversprechenden Nachfolgern benutzt werden.
  - Zum Beispiel *Greedy Best-First Search*: Reihung der expandierten Knoten aufsteigend nach h
- Heuristik oft aus Abstraktionen des Problems
  - Zum Beispiel Approximation einer Städtetour durch euklidische Distanz

A*-Suche

- Bewertung eines Knotens n nicht direkt nach heuristischer Funktion h, sondern durch

  $$f(n) = g(n) + h(n)$$

  mit $g(n)$ die tatsächlichen Kosten vom Startknoten nach n

- Für Vollständigkeit auf Bäumen Einschränkung auf zulässige Heuristiken: h überschätzt die Kosten zu einem Zielknoten nie

- Für Vollständigkeit auf Graphen Einschränkung auf konsistente Heuristiken:

  $$\forall n \xrightarrow{a} n' \,.\, h(n) \leq c(n, a, n') + h(n')$$

  mit $c(n, a, n')$ die tatsächlichen Kosten von n nach n' mit Aktion a

Optimale Strategie für Zwei-Spieler-Spiele

Zwei Spieler: MIN und MAX

- Nullsummenspiel
- MAX (am Zug) versucht seinen Spielwert zu maximieren
- MIN (am Gegenzug) versucht den Spielwert für MAX zu minimieren

Minimal-maximaler Spielwert eines Spielzustands n

$$\text{minimaxValue}(n) = \begin{cases} \text{utility}(n), \\ \qquad \text{falls } n \text{ ein Endzustand} \\ \max\{\text{minimaxValue}(s) \mid s \in \text{successors}(n)\}, \\ \qquad \text{falls } n \text{ MAX-Zustand} \\ \min\{\text{minimaxValue}(s) \mid s \in \text{successors}(n)\}, \\ \qquad \text{falls } n \text{ MIN-Zustand} \end{cases}$$

Minimax-Algorithmus

proc MINIMAX($state$) $\equiv$
  $\lceil v \leftarrow$ MAXVALUE($state$)
    return action in SUCCESSORS($state$) with value $v \rfloor$

proc MAXVALUE($state$) $\equiv$
  $\lceil$**if** TERMINAL?($state$) **then return** UTILITY($state$) **fi**
    $v \leftarrow -\infty$
    foreach $\langle a, s \rangle \in$ SUCCESSORS($state$) **do**
      $v \leftarrow$ MAX(v, MINVALUE(s)) **od**
  return $v \rfloor$

proc MINVALUE($state$) $\equiv$
  $\lceil$**if** TERMINAL?($state$) **then return** UTILITY($state$) **fi**
    $v \leftarrow \infty$
    foreach $\langle a, s \rangle \in$ SUCCESSORS($state$) **do**
      $v \leftarrow$ MIN(v, MAXVALUE(s)) **od**
  return $v \rfloor$

α-β-Pruning für Zwei-Spieler-Spiele

- Einführung unterer und oberer Schranken für die Spielbewertung
  - Bisheriger maximaler MAX-Wert α
  - Bisheriger minimaler MIN-Wert β
- Abschneiden von Teilbäumen, sobald der Wert eines Knotens schlechter als α bzw. β für MAX bzw. MIN wird
- Effizienz von der Reihenfolge der untersuchten Züge abhängig

α-β-Pruning-Algorithmus

proc AlphaBetaSearch(*state*) $\equiv$
  ⌈$v \leftarrow$ MaxValue(*state*, $-\infty$, ∞)
  return action in Successors(*state*) with value v⌋

proc MaxValue(*state*, α, β) $\equiv$
  ⌈**if** Terminal?(*state*) **then return** Utility(*state*) **fi**
  $v \leftarrow -\infty$
  foreach $\langle a, s \rangle \in$ Successors(*state*) **do**
    $v \leftarrow$ Max(v, MinValue(s, α, β))
    if $v \geq \beta$ **then return** v **fi**
    $\alpha \leftarrow$ Max(α, v)
  od
  return v⌋

proc MinValue(*state*, α, β) analog

Strategie für Zwei-Spieler-Spiele mit Zufall

Zusätzliche Einführung von Zufallszuständen

- abwechselnd mit Spielzuständen
- Aktionen entsprechen dem Würfelwurf $n \xrightarrow{x} \langle n, x \rangle$
- Bewertung nach der Wahrscheinlichkeit $P(\langle n, x \rangle) = P(x)$

Erwarteter minimal-maximaler Spielwert eines Spielzustands n

$$\mu\text{minimaxValue}(n) = \begin{cases} \text{utility}(n), \\ \qquad \text{falls } n \text{ ein Endzustand} \\ \max\{\mu\text{minimaxValue}(s) \mid s \in \text{successors}(n)\}, \\ \qquad \text{falls } n \text{ MAX-Zustand} \\ \min\{\mu\text{minimaxValue}(s) \mid s \in \text{successors}(n)\}, \\ \qquad \text{falls } n \text{ MIN-Zustand} \\ \sum\{P(x) \cdot \mu\text{minimaxValue}(s) \mid \langle s, x \rangle \in \text{successors}(n)\}, \\ \qquad \text{falls } n \text{ Zufallszustand} \end{cases}$$

Strategie für Mehr-Spieler-Spiele

- Erweiterung der Bewertung auf Vektoren
  - Spielwert aus Sicht jedes Spielers k
- Jeder Spieler versucht seinen eigenen Spielwert zu maximieren

Minimal-maximaler Spielwert eines Spielzustands n

$$\mathsf{minimaxValue}(n) = \begin{cases} \mathsf{utility}(n), \\ \qquad \text{falls } n \text{ ein Endzustand} \\ \max_k \bigcup \{\min_k \mathsf{minimaxValue}(s) \mid s \in \mathsf{successors}(n)\}, \\ \qquad \text{falls } n \text{ ein } k\text{-Zustand} \end{cases}$$

mit $\max_k V$ ($\min_k V$) Vektoren mit maximalem (minimalem) Wert v_k für $v \in V$

Heuristische Suchbeschränkungen

- Ersetzen der Berechnung des tatsächlichen Werts eines Spielstands durch Auswertung einer Evaluierungsfunktion

- Bewertung eines Spielstandes durch gewichtete Kombination von Einzelkriterien

- Beschränkung der Suchtiefe durch Ersetzen des Terminierungstests

  if CUTOFF?(*state*, *depth*) **then return** EVAL(*state*) **fi**

  (CUTOFF? liefert $true$ für terminale Knoten)

- Bevorzugung „vielversprechender“ Züge beim α-β-Pruning

Bewertungsmöglichkeiten in Malefiz

Unmittelbare Bewertung einzelner Figuren

- Entfernung zum Ziel
  - Abhängig von Blockadesteinen
  - Eher irrelevant bei großer Entfernung
- Entfernung zu eigenen Spielfiguren
  - Relevant zur Beseitigung von Blockadesteinen
- Entfernung zu gegnerischen Spielfiguren
  - Unterscheidung zwischen Vorwärts- und Rückwärtsentfernung (in der Nähe des Ziels)
  - Trennung durch Blockadesteine

Bewertungsmöglichkeiten in Malefiz

Bewertung der Spielsituation

- Chancen zum Sieg
  - Wahrscheinlichkeit des Sieges mit wenigen Würfen
  - Anzahl der Figuren in der Nähe des Zielfeldes
  - Risiken blockiert zu werden
  - Risiken geworfen zu werden
- Mögliche Bedrohung des Gegners
  - Siegverhinderung durch Werfen
  - Möglichkeiten zu vorteilhaft zu blockieren
- Mögliche Bedrohung durch den Gegner

Bewertungsmöglichkeiten in Malefiz

Bewertung der Blockaden

- Wert der Blockadenbeseitigung
  - Verbesserung eigener Zugmöglichkeiten/Siegchancen
  - Öffnen von Wegen geringerer gegnerischer Bedrohung

- Wert des Blockierens
  - Siegchancen des Gegners verkleinern
  - Bedrohung durch Gegner verkleinern

- Mögliche Zugaufwertung falls eine Blockade geschlagen wird
  - Seitliche Züge möglicherweise wertvoller als zielorientierte
  - Umwege möglicherweise günstiger als Blockadebeseitigung

Blockadezug als eigener Knoten im Spielbaum

- Berechnung der Blockade als eigene Ebene im Spielbaum
- Berechnung der Bewertungsdifferenz aller Blockadepositionen