Datenstrukturen & Algorithmen 22. April, 2015

Handout 8

Sebastian Millius, Sandro Feuz, Daniel Graf

Thema: Spielbäume, Min-Max, Branch & Bound

Referenz: Russell, Norvig, *Artificial Intelligence: A Modern Approach*

Links

- Min-Max
  http://en.wikipedia.org/wiki/Minimax (http://goo.gl/oqnf)
- Branch & Bound
  http://en.wikipedia.org/wiki/Branch_and_bound (http://goo.gl/iJXCx)
- Backtracking
  http://en.wikipedia.org/wiki/Backtracking (http://goo.gl/mlGa)

Backtracking

Ein Backtracking (Exhaustive Search) Algorithmus versucht eine Lösung inkrementell aufzubauen. Wann immer der Algorithmus sich zwischen mehreren Alternativen für den nächsten Schritt entscheiden muss, probiert er *alle* möglichen Optionen (rekursiv) aus. Der Algorithmus verwirft ("backtracks") eine Teillösung, wenn er erkennt, dass keine gültige Lösung mehr erhalten werden kann. Der Backtracking Algorithmus verfährt nach einer systematischen "generation order", so dass keine Lösung übersehen oder mehrfach betrachtet wird.

Konzeptuell können die möglichen Teillösungen als Knoten eines Baumes gesehen werden (Suchbaum). Jede potentielle Teillösung ist Vater der Teillösungen, die aus dieser hervorgehen durch einen Schritt. Die Blätter dieses Baumes sind (Teil)lösungen, die nicht mehr erweitert werden können. Der Backtracking Algorithmus durchsucht diesen Suchbaum rekursiv (oft in DFS - order): Bei jedem Knoten überprüft der Algorithmus, ob die Teillösung (überhaupt noch) zu einer validen Lösung vervollständigt werden kann. Wenn nicht, wird der gesamte Teilbaum ausgelassen (*pruned*).

Beispiel: Das "n Queens" Problem

Ein klassisches Problem ist das n Damen Problem, welches als erstes von dem deutschen Schach Enthusiast Max Bezzel in 1848 für das Standard 8×8 Brett präsentiert wurde und 1850 durch Franz Nauck gelöst und verallgemeinert für grössere Spielbretter. Das Problem ist n Damen auf einem $n \times n$ Schachbrett zu positionieren, so dass keine zwei Damen sich angreifen, d.h. keine zwei Damen dürfen in der gleichen Spalte, Zeile oder Diagonale stehen.

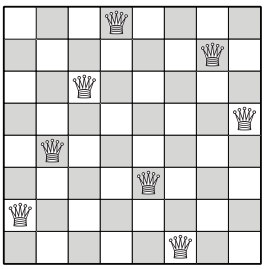

Abb.: Eine Lösung für das 8 Damen Problem $(4, 7, 3, 8, 2, 5, 1, 6)$

Da offensichtlich in jeder Lösung genau eine Dame in jeder Zeile ist, werden wir mögliche Lösungen durch ein Array $Q[1 \ldots n]$ representieren, wobei $Q[i]$ angibt, welches Feld der $i$ten Zeile eine Dame enthält.

Um eine Lösung zu finden, werden wir die Damen nun Zeile für Zeile platzieren (startend mit der ersten Zeile).

Um schnell zu überprüfen, ob eine Spalte oder Diagonale bereits belegt ist, werden wir folgende boolean Arrays betrachten:

- $column[1 \ldots n]$, dabei ist $column[i] =$ TRUE, falls in der $i$ten Spalte eine Dame platziert wurde
- $diag[1 \ldots 2n-1]$, dabei ist $diag[i] =$ TRUE, falls in der $i$ten links-rechts Diagonale eine Dame platziert wurde
- $diag2[1 \ldots 2n-1]$, dabei ist $diag2[i] =$ TRUE, falls in der $i$ten rechts-links Diagonale eine Dame platziert wurde

Der folgende rekursive Algorithmus zählt rekursiv alle möglichen Lösungen auf, die konsistent mit einer gegebenen Teillösung sind. Der Eingabeparameter k ist die erste freie Zeile.

RECURSIVE-N-QUEENS(Q, k)

```
1  if k == n + 1
2       output Q
3  else for j = 1 to n // mögliche Spalte für kte Dame
4            // gültige Position?
5            if spalte[j] == False and diag[k + j] == False and diag2[n − k + j] == False
6            // setzte Dame
7                 Q[k] = j
8                 spalte[j] = True, diag[k + j] = True, diag2[n − k + j] = True
9                 Recursive-N-Queens(Q, k + 1)
10                // mache Zug rückgängig
11                spalte[j] = False, diag[k + j] = False, diag2[n − k + j] = False
```

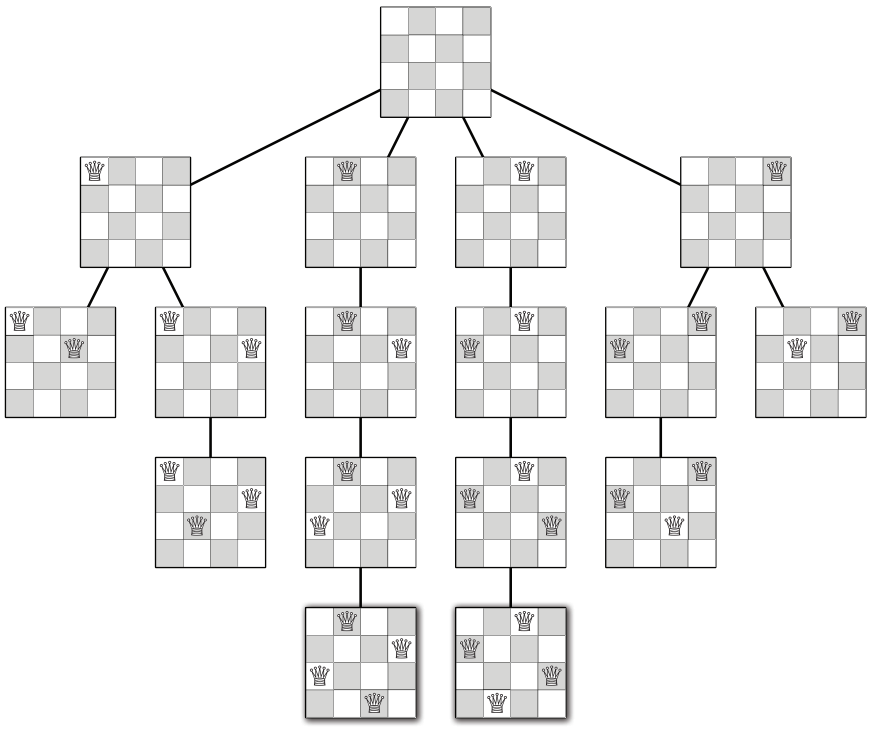

Abb.: Der vollständige Suchbaum für das 4 Dame Problem.

Anmerkung

Eine *explizite* Lösung für das n Dame Problem ist bekannt. Vergleiche hierzu *Explicit solutions to the N-queens problem for all N* (`http://goo.gl/JOQum`). Zugreifbar aus dem ETH Netz.

Branch & Bound

Branch & Bound ist eine Methode für Algorithmen für Optimierungsprobleme. Branch & Bound basiert auf *Backtracking*, das eine *erschöpfende* Suchtechnik im Raum der möglichen Lösungen ist. Das Problem dabei ist, dass die Anzahl der möglichen Zustände (Grösse des Suchbaumes) oft exponentiell oder von der Grössenordnung $2^n, n!, n^n \ldots$ in der Eingabegrösse n sein kann.

Branch & Bound ist eine Methode, um in einem grossen Entscheidungsbaum ganze Teilbäume mittels Schranken (*Bounds*) abschneiden zu kïnen. Grob gesagt ist Branch & Bound eine Technik um Backtracking zu beschleunigen, indem Teile des Suchraumes nicht durchsucht werden, weil man erkennt dass *diese Bereiche keine optimale Lösung enthalten können.*

Je nach dem wie eng die Schranken gesetzt werden, kann die Laufzeit zum Finden einer optimalen (oder genügend guten) Lösung stark verkürzt werden. Branch & Bound wird auch häufig in Zusammenhang mit Heuristiken gebraucht. Oft wird auch eine Schranke für die Kosten der optimalen Lösung vorberechnet.

Min-Max-Algorithmus

Spielbaum

Jedes Strategiespiel (2 Spieler, die abwechselnd ziehen; vollständiger Information; alle Partien endlich) hat einen endlichen *Spielbaum*. In einem Spielbaum entspricht die Wurzel der Ausgangsposition des Spiels und die inneren Knoten sind momentane Zustände einer Partie. Die Blätter repräsentieren die Endpositionen einer Partie. Folglich entsprechen die Kanten auf Level $2n+1$ ($n \in \mathbb{N}$) Zügen von Spieler A und die Kanten auf Level $2n$ Zügen von Spieler B. Eine Partie entspricht einem Pfad von der Wurzel zu einem Blatt.

Strategie

Ein Spiel kann nun wie folgt gelöst werden: Jedem Blatt wird zugeordnet wer gewonnen hat (resp. ob unentschieden ist). Nun kann der Baum von den Blättern her ausgefüllt werden, so dass in jedem Knoten (jedem möglichen Spielzustand) steht wer gewinnen wird. Dabei wird wie folgt ausgefüllt:

- Falls Spieler A als nächstes zieht, dann wird in der aktuellen Konstellation Spieler A gewinnen, falls mindestens ein Zug von A zu einer Gewinnsituation für A führt.
- Falls kein Zug zu einer Gewinnsituation aber mind. ein Zug zu einer Remisituation führt, wird die aktuelle Konstellation auch eine Remisituation.
- Andernfalls wird es eine Gewinnsituation für Spieler B (= Verlustsituation für Spieler A).

Für Spieler B erfolgt die Einteilung analog.

Min-Max

Wir benennen nun die Spieler A und B um in Max und Min. Desweiteren sagen wir nicht mehr Gewinnsituation, sondern bewerten die Situation X mit

$$bew(X) = \begin{cases} 1 & \text{Gewinnsituation für Max (A)} \\ 0 & \text{Remisituation} \\ -1 & \text{Gewinnsituation für Min (B)} \end{cases}$$

Unsere Strategie-Regeln vereinfachen sich dann zu:
Max ist am Zug: wähle den Zug, der in einer Situation mit **maximaler Bewertung** endet.
Min ist am Zug: wähle den Zug, der in einer Situation mit **minimaler Bewertung** endet.

Feldbewertung

Dieses Vorgehen erlaubt es, jedes strat. Spiel optimal zu spielen. Jedoch ist in realen Spielen der Spielbaum zu gross (er wächst exponentiell in der Länge der Partien). Darum wird nicht der ganze Spielbaum generiert, sondern (ausgehend von der aktuellen Spielposition) bis zu einer gewissen Tiefe simuliert. Dann wird die Simulation abgebrochen und die aktuelle Spielsituation lokal betrachtet und bewertet (kleine Werte sind gut für Min, grosse gut für Max). Danach wird ausgehend von diesen Werten von unten her der jeweils beste Zug für Min und für Max in jeder simulierten Situation bestimmt (vergleiche Abbildung).

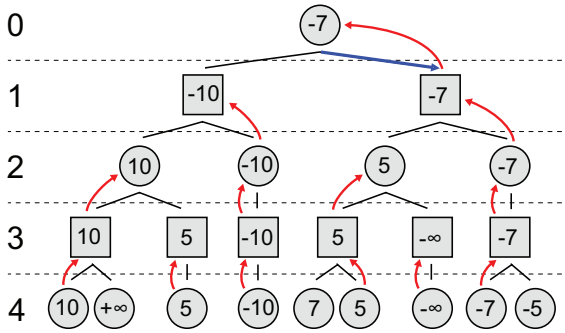

Abb.: Min-Max Algorithmus. Max ist am Zug.
Der Min-Max Algorithmus schlägt den zweiten Zug vor.

Alpha-Beta-Pruning

Im Min-Max Algorithmus tritt oft eine Situation analog zu folgender Abbildung unten links auf: Spieler Max lässt gerade seine zweite Zugsmöglichkeit simulieren (die in 4 enden wird). Er weiss dabei schon, dass der erste Zug auf eine Spielsituation mit der Bewertung 5 fführen wird. Nachdem Spieler Max seinen zweiten Zug simuliert hat, wird Spieler Min seine Antwort-Züge simulieren. Der zweite mögliche Zug für Min gibt hier 4 zurück

Ab diesem Zeitpunkt ist es Max egal, was die anderen Züge von Min zurückgeben werden! Denn auch wenn diese extrem gross wären, so wird Min (der ja die Werte minimiert) höchstens eine Spielposition mit der Bewertung 4 wählen (potentiell wird dieser Wert noch kleiner). Max hat aber schon einen Zug zur Verfügung, der den Wert 5 zurück gab (sein erster Zug), folglich wird er den zweiten Zug sowieso nicht wählen und ist nicht an seiner weiteren Evaluierung interessiert.

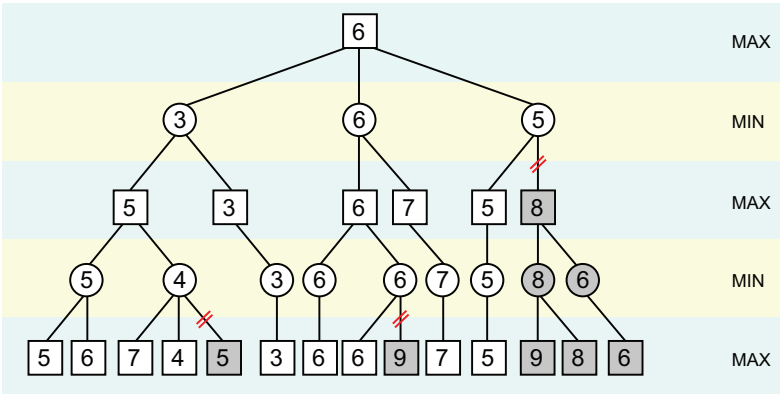

Abb.: $\alpha - \beta$ pruning

In solchen Situationen können ganze Teilsuchbäume übersprungen werden, die Suche wird effizienter. Dabei ist zu bemerken, dass der Algorithmus immer noch optimal ist. Er wird in jedem Fall das gleiche Ergebnis liefern, wie der nicht-optimierte Min-Max-Alogorithmus.

```
/*
 * Aufruf: alphabeta(0, MAX oder MIN, -INFINITY, INFINITY);
 * Die Zugauswahl wird in der Implementation weggelassen
 * (also das Speichern des besten Zuges auf Tiefe 0)
 */
double alphabeta (int tiefe, char spieler, double alpha, double beta)
{
        // die Bewertung der aktuellen Situation
        double bewertung;

        if (tiefe >= MAXTIEFE)
                return bewerteSituation();

        // betrachte alle Zuege
        foreach (Zug z) {
                z.simuliere;
                if (gewonnen)
                        bewertung = SIEG - tiefe;
                else
                        bewertung = alphabeta (tiefe+1, gegner(spieler), alpha, beta);
                z.rueckgangig;

                if (spieler == MAX) { // MAX maximiert alpha
                        alpha >?= bewertung; // neuer Zug besser?
                        if (alpha >= beta) // Beta-Cut?
                                break;
                }
                else { // MIN minimiert beta
                        beta <?= bewertung; // neuer Zug besser?
                        if (beta <= alpha) // Alpha-Cut?
                                break;
                }
        }

        if (spieler == MAX)
                return alpha;
        else
                return beta;
}
```

Anmerkungen

- *Dame*: *Chinook* (http://de.wikipedia.org/wiki/Chinook_(Programm)) beendete 1994 die 40-jährige Vorherrschaft von World-Champion Marian Tinsley. Das Programm benutzte eine Endspieldatenbank für alle Positionen mit 8 oder weniger Steinen auf dem Feld. Im Ganzen beinhaltete die Endspieldatenbank deshalb 443,748,401,247 Positionen.
- *Schach*: *Deep Blue* (http://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)) besiegte 1997 World Champion Gary Kasparov in einem 6 Spiele-Match. Deep Blue durchsucht 200 Millionen Positionen pro Sekunde, benutzt eine äusserst durchdachte Bewertefunktionen und beinhaltet (nicht veröffentlichte) Methoden die es erlauben bis zu einer Tiefe von 40 Zügen zu suchen.
- *Othello*: Menschliche Champions verweigern Spiele gegen Computerprogramme (die anscheinend zu gut sind)
- *Go*: Menschliche Champions verweigern Spiele gegen Computerprogramme. Diese sind zu schlecht. In Go gibt es über 300 Möglichkeiten pro Zug. Deshalb verwenden die meisten Programme vorprogrammierte Muster oder Ähnliches.

Verbesserungen

- *Best First*: Um die Anzahl der Cutoffs zu erhöhen, können in jedem Level die Nachfolger bewertet werden und sie nach Bewertung abarbeiten (*best-first*-Strategie). Wenn der (lokal)

beste Zug zuerst gewählt wird, erwartet man eine erhöhte Anzahl Cutoffs wenn die anderen Nachfolger durchsucht werden (think about it)

- *Iterative Deepening*: Die iterative Tiefensuche ist ähnlich wie die normale Tiefensuche: Iterativ wird eine beschränkte Suche durchgeführt (man sucht bis zu einer fixen Tiefe), und dabei wird das Level, bis zu welchem die Suche das Spiel erkundet, bei jeder Iteration um eins erhöht. Im ersten Schritt wird also ein Zug vorausgeschaut. Im nächsten Schritt wird dann 2 Züege weit erkundet. Dabei kann man nun zum Beispiel die Resultate einer Iteration brauchen um die Züge in der nächsten Iteration zu ordnen (vgl. Best First).

- *Zobrist-Hash*: Insbesondere bei Brettspielen ist es möglich (und in den meisten Szenarien sehr wahrscheinlich), dass es viele verschiedene Sequenzen von Zügen auf die gleiche Spielsituation führen. Dies führt dazu, dass wir in der Spielbaumsuche äquivalente Knoten haben. Überlege dir dies beispielsweise für das Tic-Tac-Toe-Spiel. Um eine Spielsituation/Knoten nicht mehr als einmal zu erkunden/durchsuchen, kann das Result gehasht werden. Dazu kann der *Zobrist-Hash* verwendet werden: Zobrist Hashing startet durch zufällig generierte Bitstrings für jedes mögliche Element eines Brettspiels. Für eine gegeben Spielsitutation werden die Bitstrings der einzelnen Steine/Elemente zusammengefasst mit einem bitweisen XOR. Wenn die Bitstrings lang genug sind, werden verschiedene Brettpositionen mit hoher Wahrscheinlichkeit auf unterschiedliche Werte gehasht.

  Für das Tic-Tac-Toe Spiel kann dies beispielsweise folgendermassen aussehen (think about it!)

```
#define EMPTY 0
#define CROSS 1
#define CIRCLE 2

int R[3][3][3];

void init() {
   srand(time(0));
   for (int x = 0; x < 3; ++x) {
      for (int y = 0; y < 3; ++y) {
         R[x][y][EMPTY] = rand();
         R[x][y][CROSS] = rand();
         R[x][y][CIRCLE] = rand();
      }
   }
}

int hash(int** game) {
   int key = 0;
   for (int x = 0; x < 3; ++x) {
      for (int y = 0; y < 3; ++y) {
         int whatIsAtXY = game[x][y];
         key ^= R[x][y][whatIsAtXY];
      }
   }
   return key;
}
```

Der Zobrist-Hash hat nun den Vorteil, dass wir Spielzüge leicht updaten können. Wenn wir den Hash der aktuellen Situation kennen und wir machen einen Zug (bei Tic-Tac-Toe: ein Kreuz/Kreis an eine leere Stelle schreiben) können wir den Hash-Wert der resultierenden Spielsituation inkrementell bestimmen.

Beispielsweise (Tic-Tac-Toe): das linke obere Feld ist leer und wir machen einen “Kreis” in dieses Feld. Wir mössen dann zwei Änderungen vornehmen. Zunächst müssen wir die “das linke obere Feld ist leer” - Information aus dem Schlüssel entfernen und dann dann die ”Kreis im linken oberen Feld” Information hinzufügen. Hier nutzen wir die XOR-Operation um die Information zu updaten (again think about it)

```
/* ... */
// Strip the information of the empty tile.
int newKey = oldKey ^ R[0][0][EMPTY];

// Now encode the information on the circle being placed there.
newKey ^= R[0][0][CIRCLE];
/* ... */
```

vgl. http://en.wikipedia.org/wiki/Zobrist_hashing (http://goo.gl/w8Pn2)