Smart Graphics: Methoden 3
Suche, Constraints

Vorlesung „Smart Graphics”
Andreas Butz, Otmar Hilliges
Mittwoch, 30. November 2005
Themen heute

• Suchverfahren
  – Alpha-Beta
  – Hillclimbing
  – Simulated Annealing
  – Genetische Suche

• Constraints
Domain knowledge

Design knowledge

Reflection of results

Reasoning, Planning, Inferencing, Optimization, ...

Rendering, Output, Presentation, ...

Display

User model, Preferences

input
Game tree (2-player, deterministic, turns)
Minimax

- Perfect play for deterministic games.
- Idea: choose move to position with highest minimax value = best achievable payoff against best play.
- E.g., 2-ply game:
**Minimax algorithm**

Hauptprogramm (Auszug):

```haskell
var doNext : number
dummy := maxWert ( gewünschte suchTiefe )
Zug doNext ausführen

function maxWert ( restTiefe ) returns number
var ermittelt, zugWert : number
begin
  ermittelt := - unendlich
  für alle möglichen Züge begin
    Zug simulieren
    if restTiefe <= 1 or keineFolgezügeMehrMöglich
      then zugWert := bewertungsFunktion
      else zugWert := minWert ( restTiefe - 1 )
    Zug-Simulation zurücksetzen
    if zugWert > ermittelt then begin
      ermittelt := zugWert
      doNext := nummer des Zuges /* für das Hauptprogramm */
    end
  end
end

end maxWert

function minWert ( restTiefe ) returns number
var ermittelt, zugWert : number
begin
  ermittelt := + unendlich
  für alle möglichen Züge begin
    Zug simulieren
    if restTiefe <= 1 or keineFolgezügeMehrMöglich
      then zugWert := bewertungsFunktion
      else zugWert := maxWert ( restTiefe - 1 )
    Zug-Simulation zurücksetzen
    if zugWert < ermittelt then ermittelt := zugWert
  end
end
return ermittelt
end minWert
```
Properties of minimax

- **Complete?** Yes (if tree is finite).
- **Optimal?** Yes (against an optimal opponent).
- **Time complexity?** $O(b^m)$.
- **Space complexity?** $O(bm)$ (depth-first exploration).

- For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games → exact solution completely infeasible.
$\alpha$-$\beta$ pruning example
α-β pruning example
α-β pruning example
α-β pruning example
α-β pruning example
Properties of $\alpha$-$\beta$

• Pruning does not affect final result.

• Good move ordering improves effectiveness of pruning.

• With "perfect ordering," time complexity = $O(b^{m/2})$
  $\rightarrow$ doubles depth of search.

• A simple example of the value of reasoning about which computations are relevant (a form of metareasoning).
Why is it called $\alpha$-$\beta$?

- $\alpha$ is the value of the best (i.e., highest-value) choice found so far at any choice point along the path for $\text{max}$.
- If $v$ is worse than $\alpha$, $\text{max}$ will avoid it. → prune that branch.
- Define $\beta$ similarly for $\text{min}$.
The $\alpha$-$\beta$ algorithm

```c
int Max(int tiefe, int alpha, int beta) {
    if (tiefe == 0) return Bewerten();
    GeneriereMoeglicheZuege();
    while (ZuegeUebrig()) {
        FuehreNaechstenZugAus();
        wert = Min(tiefe-1, alpha, beta);
        MacheZugRueckgaengig();
        if (wert >= beta) return wert;
        if (wert > alpha) alpha = wert;
    }
    return alpha;
}
```
The $\alpha$-$\beta$ algorithm

```c
int Min(int tiefe, int alpha, int beta) {
    if (tiefe == 0) return Bewerten();
    GeneriereMoeglicheZuege();
    while (ZuegeUebrig()) {
        FuehreNaechstenZugAus();
        wert = Max(tiefe-1, alpha, beta);
        MacheZugRueckgaengig();
        if (wert <= alpha) return wert;
        if (wert < beta) beta = wert;
    }
    return beta;
}
```
Zusammenfassung

Ausgangspunkt: Bei der Tiefensuche schon während der Expansion die Bewertungsfunktion auf Blätter anwenden und Werte nach Minimax-Verfahren nach oben verteilen.

- **Alpha-Wert eines Max-Knotens** ist die jeweils größte geerbte Bewertung seiner Nachfolger. Der Wert den Spieler Max mindestens erhält.
- **Beta-Wert eines Min-Knotens** ist der jeweils kleinste geerbte Wert seiner Nachfolger. Der Wert den Spieler Min maximal erhält.

2 Typen von Beschneidungen des Suchbaums:
- **Alpha-Schnitt**: Suche wird abgebrochen an einem Min-Knoten, dessen Beta-Wert > Alpha-Wert von irgendeinem seiner Max-Vorgänger. Der Beta-Wert bleibt der vererbte Wert des Min-Knotens, der gemäß Minimax weiterverarbeitet wird.
- **Beta-Schnitt**: Suche wird abgebrochen an einem Max-Knoten, dessen Alpha-Wert < Beta-Wert von irgendeinem seiner Min-Vorgänger. Der Alpha-Wert bleibt der vererbte Wert des Max-Knotens.
Cutting off search

$MinimaxCutoff$ is identical to $MinimaxValue$ except

1. $Terminal$ is replaced by $Cutoff$
2. $Utility$ is replaced by $Eval$.

Does it work in practice?.
$$b^m = 10^6, \ b=35 \rightarrow m=4.$$  

4-ply lookahead is a hopeless chess player!

– 4-ply $\approx$ human novice.
– 8-ply $\approx$ typical PC, human master.
– 12-ply $\approx$ Deep Blue, Kasparov.
Deterministic games in practice

• Checkers:
  – Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used a pre-computed endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 444 billion positions.
Deterministic games in practice

• Chess:
  – Deep Fritz achieved remis twice against Kasparov and Kramnik in 2002/03
  – Hydra defeated Adams (ranked No. 7) in 2005 with 1:5
  – Strategy vs. Nr. of computed positions.

• Go:
  – human champions refuse to compete against computers, who are too bad. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.
Local search algorithms

• In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution.

• State space = set of "complete" configurations.
• Find configuration satisfying constraints, e.g., n-queens.

• In such cases, we can use local search algorithms.
• keep a single "current" state, try to improve it.
Example: *n*-queens

- Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal
Hill-climbing search

• "Like climbing Everest in thick fog with amnesia"

```plaintext
function HILL-CLIMBING(problem) returns a state that is a local maximum
inputs: problem, a problem
local variables: current, a node
               neighbor, a node

current ← MAKE-NODE(INITIAL-STATE[problem])
loop do
    neighbor ← a highest-valued successor of current
    if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
    current ← neighbor
```
Hill-climbing search

- Problem: depending on initial state, can get stuck in local maxima
Hill-climbing search: 8-queens problem

- $h =$ number of pairs of queens that are attacking each other, either directly or indirectly
- $h = 17$ for the above state
Hill-climbing search: 8-queens problem

• A local minimum with $h = 1$
Simulated annealing search

• Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency

• The algorithm employs a random search which not only accepts changes that decrease objective function $f$, but also some changes that increase it. The latter are accepted with a probability $p = \exp\left(\frac{-\delta f}{T}\right)$
Properties of simulated annealing search

• One can prove: If $T$ decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1

• Widely used in VLSI layout, airline scheduling, etc.

• Adaptation of values for $T$ is application driven.
Local beam search

• Keep track of \( k \) states rather than just one

• Start with \( k \) randomly generated states

• At each iteration, all the successors of all \( k \) states are generated

• If any one is a goal state, stop; else select the \( k \) best successors from the complete list and repeat.
Genetic algorithms

• A successor state is generated by combining two parent states

• Start with $k$ randomly generated states (population)

• A state is represented as a string over a finite alphabet (often a string of 0s and 1s)

• Evaluation function (fitness function). Higher values for better states.

• Produce the next generation of states by selection, crossover, and mutation
Genetic algorithms

- Fitness function: number of non-attacking pairs of queens (min = 0, max = $8 \times 7/2 = 28$)
- $24/(24+23+20+11) = 31\%$
- $23/(24+23+20+11) = 29\%$ etc
Genetic algorithms
Constraint Satisfaction Problems

• Constraint Satisfaction Problems (CSP)
• Backtracking search for CSPs
• Local search for CSPs
Constraint satisfaction problems (CSPs)

• Standard search problem:
  – **state** is a "black box“ – any data structure that supports successor function, heuristic function, and goal test

• CSP:
  – **state** is defined by variables $X_i$ with values from domain $D_i$
  – **goal test** is a set of constraints specifying allowable combinations of values for subsets of variables

• Simple example of a **formal representation language**

• Allows useful **general-purpose** algorithms with more power than standard search algorithms
Example: Map-Coloring

- Variables: $WA, NT, Q, NSW, V, SA, T$
- Domains: $D_i = \{\text{red, green, blue}\}$
- Constraints: adjacent regions must have different colors
- e.g., $WA \neq NT$, or $(WA, NT)$ in $\{(\text{red, green}), (\text{red, blue}), (\text{green, red}), (\text{green, blue}), (\text{blue, red}), (\text{blue, green})\}$
Example: Map-Coloring

- **Solutions** are complete and consistent assignments, e.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green
Constraint graph

- **Binary CSP**: each constraint relates two variables
- **Constraint graph**: nodes are variables, arcs are constraints
Varieties of CSPs

• Discrete variables
  – finite domains:
    • $n$ variables, domain size $d \rightarrow O(d^n)$ complete assignments
    • e.g., Boolean CSPs, incl.~Boolean satisfiability (NP-complete)
  – infinite domains:
    • integers, strings, etc.
    • e.g., job scheduling, variables are start/end days for each job
    • need a constraint language, e.g., $\text{StartJob}_1 + 5 \leq \text{StartJob}_3$

• Continuous variables
  – e.g., start/end times for Hubble Space Telescope observations
  – linear constraints solvable in polynomial time by linear programming
Varieties of constraints

- **Unary** constraints involve a single variable,
  - e.g., $SA \neq \text{green}$

- **Binary** constraints involve pairs of variables,
  - e.g., $SA \neq WA$

- **Higher-order** constraints involve 3 or more variables,
  - e.g., cryptarithmetical column constraints
Example: Cryptarithmetic

- Variables: $F, T, U, W, R, O, X_1, X_2, X_3$
- Domains: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- Constraints: $\text{Alldiff}(F, T, U, W, R, O)$
  - $O + O = R + 10 \cdot X_1$
  - $X_1 + W + W = U + 10 \cdot X_2$
  - $X_2 + T + T = O + 10 \cdot X_3$
  - $X_3 = F, T \neq 0, F \neq 0$
Real-world CSPs

• Assignment problems
  – e.g., who teaches what class
• Timetabling problems
  – e.g., which class is offered when and where?
• Transportation scheduling
• Factory scheduling

• Notice that many real-world problems involve real-valued variables
• Constraints may be preferred constraints rather than absolute
Standard search formulation (incremental)

Let's start with the straightforward approach, then fix it

States are defined by the values assigned so far

- **Initial state**: the empty assignment \{ \}
- **Successor function**: assign a value to an unassigned variable that does not conflict with current assignment → fail if no legal assignments
- **Goal test**: the current assignment is complete

1. This is the same for all CSPs
2. Every solution appears at depth \( n \) with \( n \) variables → use depth-first search
3. Path is irrelevant, so can also use complete-state formulation
4. \( b = (n - \ell) d \) at depth \( \ell \), hence \( n! \cdot d^n \) leaves
5. But only \( d^n \) complete assignments!
**Backtracking search**

- Variable assignments are commutative, i.e.,
  \[
  \text{[ WA = red then NT = green ] same as [ NT = green then WA = red ]}
  \]

- Only need to consider assignments to a single variable at each node
  \[\text{Æ } b = d \text{ and there are } d^n \text{ leaves}\]

- Depth-first search for CSPs with single-variable assignments is called **backtracking search**

- Backtracking search is the basic uninformed algorithm for CSPs

- Can solve \(n\)-queens for \(n \approx 25\)
Backtracking search

function BACKTRACKING-SEARCH(csp) returns a solution, or failure
    return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns a solution, or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(Variables[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment according to Constraints[csp] then
            add { var = value } to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove { var = value } from assignment
    return failure
Backtracking example
Backtracking example
Backtracking example
Backtracking example
Improving backtracking efficiency

• **General-purpose** methods can give huge gains in speed:
  – Which variable should be assigned next?
  – In what order should its values be tried?
  – Can we detect inevitable failure early?
Most constrained variable

- Most constrained variable: choose the variable with the fewest legal values

- a.k.a. minimum remaining values (MRV) heuristic or fail first

- Magnitude of 3 to 3000 times faster than BT
Most constraining variable

- Tie-breaker among most constrained variables
- Most constraining variable:
  - choose the variable with the most constraints on remaining variables
Least constraining value

• Given a variable, choose the least constraining value:
  – the one that rules out the fewest values in the remaining variables

• Combining these heuristics makes 1000 queens feasible
Forward checking

- **Idea:**
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values
Forward checking

• **Idea:**
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values
Forward checking

• **Idea:**
  – Keep track of remaining legal values for unassigned variables
  – Terminate search when any variable has no legal values
Forward checking

- **Idea:**
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values
Constraint propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

- NT and SA cannot both be blue!
- **Constraint propagation** repeatedly enforces constraints locally
Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent if for every value $x$ of $X$ there is some allowed $y$
Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent if for every value $x$ of $X$ there is some allowed $y$
Arc consistency

- Simplest form of propagation makes each arc consistent
- \( X \rightarrow Y \) is consistent if
  for every value \( x \) of \( X \) there is some allowed \( y \)

- If \( X \) loses a value, neighbors of \( X \) need to be rechecked
Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent if
  - For every value $x$ of $X$ there is some allowed $y$

- If $X$ loses a value, neighbors of $X$ need to be rechecked
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment
Arc consistency algorithm AC-3

```plaintext
function AC-3(csp) returns the CSP, possibly with reduced domains
inputs: csp, a binary CSP with variables \{X_1, X_2, \ldots, X_n\}
local variables: queue, a queue of arcs, initially all the arcs in csp

while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if RM-INCONSISTENT-VALUES(X_i, X_j) then
        for each X_k in NEIGHBORS[X_i] do
            add (X_k, X_i) to queue

function RM-INCONSISTENT-VALUES(X_i, X_j) returns true iff remove a value
removed ← false
for each x in DOMAIN[X_i] do
    if no value y in DOMAIN[X_j] allows (x, y) to satisfy constraint(X_i, X_j) then delete x from DOMAIN[X_i]; removed ← true
return removed
```

- Time complexity: O(n^2d^3)
Local search for CSPs

• Hill-climbing, simulated annealing typically work with "complete" states, i.e., all variables assigned

• To apply to CSPs:
  – allow states with unsatisfied constraints
  – operators reassign variable values

• Variable selection: randomly select any conflicted variable

• Value selection by min-conflicts heuristic:
  – choose value that violates the fewest constraints
  – i.e., hill-climb with $h(n) = \text{total number of violated constraints}$
Example: 4-Queens

- **States**: 4 queens in 4 columns \((4^4 = 256 \text{ states})\)
- **Actions**: move queen in column
- **Goal test**: no attacks
- **Evaluation**: \(h(n) = \text{number of attacks}\)

- Given random initial state, can solve \(n\)-queens in almost constant time for arbitrary \(n\) with high probability (e.g., \(n = 10,000,000\))
Summary

• CSPs are a special kind of problem:
  – states defined by values of a fixed set of variables
  – goal test defined by constraints on variable values

• Backtracking = depth-first search with one variable assigned per node

• Variable ordering and value selection heuristics help significantly

• Forward checking prevents assignments that guarantee later failure

• Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies

• Iterative min-conflicts is usually effective in practice
Literatur, Links

  - (daraus auch wesentliche Teile der heutigen Vorlesung)
  - [http://aima.cs.berkeley.edu/newchap05.pdf](http://aima.cs.berkeley.edu/newchap05.pdf)
Wissenschaftliches Experiment zu (il)legalem P2P

• Testpersonen gesucht: Untersuchung einer Idee für eine legale Online-Musiktauschbörse
• Es gibt etwas zu essen und zu trinken! :-)
• Gruppen zu je ca. 25 Personen können nach einer Einweisung auf einer Beispiel-Website beliebig Songs herunterladen und tauschen
• Kleiner Fragebogen vor und nach dem Experiment
• Termin: Mittwoch, **7.12.2005**, fünf Termine von 10.00 bis 17.00 Uhr, Dauer ca. 1 Stunde
Anmeldung: [www.intermedia.lmu.de/experiment](http://www.intermedia.lmu.de/experiment)