Games and Adversarial Search II
Alpha-Beta Pruning (AIMA 5.3)

Some slides adapted from Richard Lathrop, USC/ISI, CS 271
Review: The Minimax Rule

Idea: Make the best move for MAX assuming that MIN always replies with the best move for MIN

1. Start with the current position as a MAX node.
2. Expand the game tree a fixed number of ply.
3. Apply the evaluation function to all leaf positions.
4. Calculate back-up values bottom-up:
   - For a MAX node, return the maximum of the values of its children (i.e. the best for MAX)
   - For a MIN node, return the minimum of the values of its children (i.e. the best for MIN)
5. Pick the move assigned to MAX at the root
6. Wait for MIN to respond and REPEAT FROM 1
2-ply Example: Backing up values

This is the move selected by minimax

New point: Actually calculated by DFS!
Minimax Algorithm

function MINIMAX-DECISION(state) returns an action

inputs: state, current state in game

\[ v \leftarrow \text{MAX-VALUE}(state) \]

return an action in SUCCESSORS(state) with value \( v \)

function MAX-VALUE(state) returns a utility value

if TERMINAL-TEST(state) then return UTILITY(state)

\[ v \leftarrow -\infty \]

for \( a, s \) in SUCCESSORS(state) do

\[ v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(s)) \]

return \( v \)

function MIN-VALUE(state) returns a utility value

if TERMINAL-TEST(state) then return UTILITY(state)

\[ v \leftarrow \infty \]

for \( a, s \) in SUCCESSORS(state) do

\[ v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(s)) \]

return \( v \)
Alpha-Beta Pruning

• A way to improve the performance of the Minimax Procedure

• Basic idea: “If you have an idea which is surely bad, don’t take the time to see how truly awful it is” ~ Pat Winston

- Assuming left-to-right tree traversal:
  - We don’t need to compute the value at this node!
  - No matter what it is it can’t effect the value of the root node.
Alpha-Beta Pruning II

- During Minimax, keep track of two additional values:
  - $\alpha$: current lower bound on MAX’s outcome
  - $\beta$: current upper bound on MIN’s outcome

- MAX will never choose a move that could lead to a worse score (for MAX) than $\alpha$
- MIN will never choose a move that could lead to a better score (for MAX) than $\beta$

- Therefore, stop evaluating a branch whenever:
  - When evaluating a MAX node: a value $v \geq \beta$ is backed-up
    — MIN will never select that MAX node
  - When evaluating a MIN node: a value $v \leq \alpha$ is found
    — MAX will never select that MIN node
Alpha-Beta Pruning IIIa

- Based on observation that for all viable paths utility value \( f(n) \) will be \( \alpha \leq f(n) \leq \beta \)

- Initially, \( \alpha = -\infty \), \( \beta = \infty \)

- As the search tree is traversed, the possible utility value window shrinks as \( \alpha \) increases, \( \beta \) decreases
Whenever the current ranges of alpha and beta no longer overlap ($\alpha \geq \beta$), it is clear that the current node is a dead end, so it can be pruned.
Alpha-beta Algorithm: In detail

- Depth first search (usually bounded, with static evaluation)
  - only considers nodes along a single path from root at any time

\[ \alpha = \text{current lower bound on MAX’s outcome} \]
(initially, \( \alpha = -\infty \))

\[ \beta = \text{current upper bound on MIN’s outcome} \]
(initially, \( \beta = +\infty \))

- Pass current values of \( \alpha \) and \( \beta \) \textit{down} to child nodes during search.
- Update values of \( \alpha \) and \( \beta \) during search:
  - MAX updates \( \alpha \) at MAX nodes
  - MIN updates \( \beta \) at MIN nodes
- Prune remaining branches at a node whenever \( \alpha \geq \beta \)
When to Prune

Prune whenever \( \alpha \geq \beta \).

- Prune below a Max node when its \( \alpha \) value becomes \( \geq \) the \( \beta \) value of its ancestors.
  - **Max nodes update** \( \alpha \) based on children’s returned values.
  - Idea: Player MIN at node above won’t pick that value anyway, since MIN can force a worse value.

- Prune below a Min node when its \( \beta \) value becomes \( \leq \) the \( \alpha \) value of its ancestors.
  - **Min nodes update** \( \beta \) based on children’s returned values.
  - Idea: Player MAX at node above won’t pick that value anyway; she can do better.
Pseudocode for Alpha-Beta Algorithm

function ALPHA-BETA-SEARCH(state) returns an action

inputs: state, current state in game

\[ \nu \leftarrow \text{MAX-VALUE}(state, -\infty, +\infty) \]

return an action in ACTIONS(state) with value \( \nu \)
Pseudocode for Alpha-Beta Algorithm

function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game
  \( v \leftarrow \text{MAX-VALUE}(state, -\infty, +\infty) \)
  return an action in ACTIONS(state) with value \( v \)

function MAX-VALUE(state, \( \alpha \), \( \beta \)) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  \( v \leftarrow -\infty \)
  for \( a \) in ACTIONS(state) do
    \( v \leftarrow \text{MAX}(v, \text{MIN-VALUE}({\text{Result}(s,a)}, \alpha, \beta)) \)
    if \( v \geq \beta \) then return \( v \)
    \( \alpha \leftarrow \text{MAX}(\alpha, v) \)
  return \( v \)
Alpha-Beta Algorithm II

function MIN-VALUE(state, α, β) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)

v ← +∞
for a,s in SUCCESSORS(state) do

  v ← MIN(v, MAX-VALUE(s, α, β))
  if v ≤ α then return v

β ← MIN(β, v)
return v
An Alpha-Beta Example

Do DF-search until first leaf

\[ \alpha, \beta, initial \ values \]

\[ \alpha = -\infty \]
\[ \beta = +\infty \]

\[ \alpha, \beta, passed \ to \ kids \]

\[ \alpha = -\infty \]
\[ \beta = +\infty \]
Alpha-Beta Example (continued)

\[ \alpha = -\infty \]
\[ \beta = +\infty \]

MIN updates $\beta$, based on kids

\[ \beta = 3 \]
MIN updates β, based on kids.
No change.
MAX updates $\alpha$, based on kids.

$\alpha = 3$

$\beta = +\infty$

3 is returned as node value.
Alpha-Beta Example (continued)

\[ \alpha = 3 \]
\[ \beta = +\infty \]

\[ \alpha, \beta, \text{passed to kids} \]
\[ \alpha = 3 \]
\[ \beta = +\infty \]
Alpha-Beta Example (continued)

\[ \alpha = 3 \]
\[ \beta = +\infty \]

MIN updates \( \beta \), based on kids.

\[ \alpha = 3 \]
\[ \beta = 2 \]
Alpha-Beta Example (continued)

\[ \alpha = 3 \quad \beta = \infty \]

\[ \alpha \geq \beta, \text{ so prune.} \]
Alpha-Beta Example (continued)

MAX updates $\alpha$, based on kids.
No change.

$\beta \downarrow$

$\alpha \uparrow$

2 is returned as node value.
**Alpha-Beta Example (continued)**

\[ \alpha = 3 \]
\[ \beta = +\infty \]

\( \alpha, \beta, \text{passed to kids} \)

\( \alpha = 3 \)
\[ \beta = +\infty \]
Alpha-Beta Example (continued)

MIN updates $\beta$, based on kids.

$\alpha = 3$

$\beta = +\infty$

$\alpha = 3$

$\beta = 14$
Alpha-Beta Example (continued)

\[ \alpha = 3, \quad \beta = +\infty \]

MIN updates \( \beta \), based on kids.

CIS 421/521 - Intro to AI
Alpha-Beta Example (continued)

\[ \alpha = 3 \]
\[ \beta = +\infty \]

2 is returned as node value.
Max now makes it’s best move, as computed by Alpha-Beta.
Effectiveness of Alpha-Beta Pruning

• Guaranteed to compute same root value as Minimax
• **Worst case:** no pruning, same as Minimax \(O(b^d))\)
• **Best case:** when each player’s best move is the first option examined, examines only \(O(b^{d/2})\) nodes, allowing to search twice as deep!
When best move is the first examined, examines only $O(b^{d/2})$ nodes….

- So: run Iterative Deepening search, sort by value returned on last iteration.
- So: expand captures first, then threats, then forward moves, etc.

- $O(b^{d/2})$ is the same as having a branching factor of $\sqrt{b}$,
  - Since $(\sqrt{b})^d = b^{d/2}$
  - e.g., in chess go from $b \sim 35$ to $b \sim 6$

- For Deep Blue, alpha-beta pruning reduced the average branching factor from 35-40 to 6, as expected, doubling search depth
Chinook and Deep Blue

- **Chinook**
  - the World Man-Made Checkers Champion, developed at the University of Alberta.
  - Competed in human tournaments, earning the right to play for the human world championship, and defeated the best players in the world.

- **Deep Blue**
  - Defeated world champion Gary Kasparov 3.5-2.5 in 1997 after losing 4-2 in 1996.
  - Uses a parallel array of 256 special chess-specific processors
  - Evaluates 200 billion moves every 3 minutes; 12-ply search depth
  - Expert knowledge from an international grandmaster.
  - 8000 factor evaluation function tuned from hundreds of thousands of grandmaster games
  - Tends to play for tiny positional advantages.
FOR STUDY....
Example

-which nodes can be pruned?
Answer to Example

-which nodes can be pruned?

Answer: NONE! Because the most favorable nodes for both are explored last (i.e., in the diagram, are on the right-hand side).
Second Example
(the exact mirror image of the first example)

- which nodes can be pruned?
Answer to Second Example (the exact mirror image of the first example)

-which nodes can be pruned?

Answer: **LOTS!** Because the most favorable nodes for both are explored **first** (i.e., in the diagram, are on the left-hand side).
Constraint Satisfaction Problems

AIMA: Chapter 6
Big idea

- Represent the *constraints* that solutions must satisfy in a uniform *declarative* language
- Find solutions by *GENERAL PURPOSE* search algorithms with no changes from problem to problem
  - No hand built transition functions
  - No hand built heuristics
- Just specify the problem in a formal declarative language, and a general purpose algorithm does everything else!
Constraint Satisfaction Problems

A CSP consists of:

• **Finite set of variables** $X_1, X_2, \ldots, X_n$

• **Nonempty *domain of possible values** for each variable $D_1, D_2, \ldots, D_n$ where $D_i = \{v_1, \ldots, v_k\}$

• **Finite set of constraints** $C_1, C_2, \ldots, C_m$

  — Each *constraint* $C_i$ limits the values that variables can take, e.g., $X_1 \neq X_2$ A *state* is defined as an *assignment* of values to some or all variables.

• A **consistent** assignment does not violate the constraints.

• Example problem: Sudoku
Constraint satisfaction problems

- An assignment is *complete* when every variable is assigned a value.
- A *solution* to a CSP is a *complete, consistent* assignment.
- Solutions to CSPs can be found by a completely *general purpose* algorithm, given only the formal specification of the CSP.

- Beyond our scope: CSPs that require a solution that maximizes an *objective function*.
Applications

- Map coloring
- Line Drawing Interpretation
- Scheduling problems
  - Job shop scheduling
  - Scheduling the Hubble Space Telescope
- Floor planning for VLSI
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.,  \( WA \neq NT \)
    - So (\( WA, NT \)) must be in \{\( (\text{red, green}), (\text{red, blue}), (\text{green, red}) \), \( \ldots \)\}
Example: Map-coloring

Solutions: complete and consistent assignments

- e.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green
Example: Cryptarithmetic

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

• Clean specification of many problems, generic goal, successor function & heuristics
  • Just represent problem as a CSP & solve with general package

• CSP “knows” which variables violate a constraint
  • And hence where to focus the search

• CSPs: Automatically prune off all branches that violate constraints
  • (State space search could do this only by hand-building constraints into the successor function)
CSP Representations

- **Constraint graph:**
  - nodes are variables
  - arcs are (binary) constraints

- **Standard representation pattern:**
  - variables with values

- **Constraint graph** simplifies search.
  - e.g. Tasmania is an independent subproblem.

- **This problem: A binary CSP:**
  - each constraint relates two variables
Varieties of CSPs

• **Discrete variables**
  - finite domains:
    - \( n \) variables, domain size \( d \) \( \rightarrow O(d^n) \) complete assignments
    - e.g., Boolean CSPs, includes 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 ≠ green

- **Binary** constraints involve pairs of variables,
  - e.g., SA ≠ WA

- **Higher-order** constraints involve 3 or more variables
  - e.g., crypt-arithmetic column constraints

- **Preference** (soft constraints) e.g. *red* is better than *green* can be represented by a cost for each variable assignment
  - Constrained optimization problems.
Idea 1: CSP as a search problem

- A CSP can easily be expressed as a search problem
  - *Initial State:* the empty assignment {}.
  - *Successor function:* Assign value to any unassigned variable *provided that there is not a constraint conflict.*
  - *Goal test:* the current assignment is complete.
  - *Path cost:* a constant cost for every step.

- Solution is always found at depth \( n \), for \( n \) variables
  - Hence Depth First Search can be used
Backtracking search

- Note that variable assignments are *commutative*
  - Eg [step 1: \( WA = \text{red} \); step 2: \( NT = \text{green} \)]
    equivalent to [step 1: \( NT = \text{green} \); step 2: \( WA = \text{red} \)]
  - Therefore, a *tree search*, not a *graph search*

- Only need to consider assignments to a single variable at each node
  - \( b = d \) and there are \( d^n \) leaves (\( n \) variables, domain size \( d \))

- 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 example
Backtracking example

And so on....
Idea 2: Improving backtracking efficiency

- **General-purpose** methods & **general-purpose** heuristics can give huge gains in speed, *on average*

- **Heuristics:**
  - Q: Which variable should be assigned next?
    1. Most constrained variable
    2. (if ties:) Most constraining variable
  - Q: In what order should that variable’s values be tried?
    3. Least constraining value
  - Q: Can we detect inevitable failure early?
    4. Forward checking
Heuristic 1: Most constrained variable

- Choose a variable with the fewest legal values

  a.k.a. minimum remaining values (MRV) heuristic
Heuristic 2: Most constraining variable

- Tie-breaker among most constrained variables
- Choose the variable with the most constraints on remaining variables

These two heuristics together lead to immediate solution of our example problem.
Heuristic 3: Least constraining value

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

Note: demonstrated here independent of the other heuristics
Heuristic 4: Forward checking

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

(A first step towards Arc Consistency & AC-3)
Forward checking

**Idea:**

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

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

<table>
<thead>
<tr>
<th>WA</th>
<th>NT</th>
<th>Q</th>
<th>NSW</th>
<th>V</th>
<th>SA</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
</tr>
<tr>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
</tr>
<tr>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
</tr>
</tbody>
</table>

![Map of Australia](image)
Forward checking

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

Terminate! No possible value for SA
Example: 4-Queens Problem

(From Bonnie Dorr, U of Md, CMSC 421)
Example: 4-Queens Problem

Assign value to unassigned variable
Example: 4-Queens Problem

Forward check!
Example: 4-Queens Problem

Assign value to unassigned variable
Example: 4-Queens Problem

Backtrack!!!

Forward check!
Example: 4-Queens Problem

Picking up a little later after two steps of backtracking....

Assign value to unassigned variable
Example: 4-Queens Problem

Forward check!
Example: 4-Queens Problem

Assign value to unassigned variable
Example: 4-Queens Problem

Forward check!
Example: 4-Queens Problem

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assign value to unassigned variable
Example: 4-Queens Problem

```
  1  2  3  4
1  ,  ,  ,  
2  ,  ,  ,  
3  ,  ,  ,  
4  ,  ,  ,  
```

```
X1
{1,2,3,4}

X2
{ , , , 4}

X3
{1, , , }

X4
{ , , 3, }
```

Forward check!
Example: 4-Queens Problem

Assign value to unassigned variable
Towards 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** goes beyond forward checking & repeatedly enforces constraints locally