Chapter 6

Adversarial Search
Outlines

• Games
• Optimal Decision in Games
• Alpha-Beta Pruning
• Imperfect Real-time Decision
• Games that include an element of chance
• State-of-Art game problem
6-1 Games

The board set for play

Head to play

deterministic

perfect information

chess, checkers, go, othello

imperfect information

backgammon, monopoly

chance

bridge, poker, scrabble, nuclear war

彰化師大
Game

• **Multiagent environments**, in which any given agent will need to consider the actions of other agents and how they affect its own welfare.
• The unpredictability of these other agents can introduce many possible *contingencies* into the agent's problem-solving process.
• cooperative and competitive multiagent environments

• **Competitive environments**, in which the agents' goals are in conflict, give rise to adversarial search problems—often known as *Games*.
Game

• Deterministic
• Turn-taking
• Two-player
• Zero-sum games of perfect information
What kind of games?

- **Abstraction**: To describe a game we must capture every relevant aspect of the game. Such as:
  - Chess
  - Tic-tac-toe
  - ...

- **Accessible environments**: Such games are characterized by perfect information

- **Search**: game-playing then consists of a search through possible game positions

- **Unpredictable opponent**: introduces uncertainty thus game-playing must deal with contingency problems
Searching for the next move

- **Complexity:** many games have a huge search space
  - **Chess:** $b = 35, m=100 \Rightarrow nodes = 35^{100}$
    - if each node takes about 1 ns to explore
    - then each move will take about $10^{50}$ millennia to calculate.

- **Resource (e.g., time, memory) limit:** optimal solution not feasible/possible, thus must approximate

1. **Pruning:** makes the search more efficient by discarding portions of the search tree that cannot improve quality result.

2. **Evaluation functions:** heuristics to evaluate utility of a state without exhaustive search.
Two-player games

- A game formulated as a search problem:
  - Initial state: ?
  - Operators: ?
  - Terminal state: ?
  - Utility function: ?
Two-player games

• A game formulated as a search problem:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial state:</td>
<td>board position and turn</td>
</tr>
<tr>
<td>Operators:</td>
<td>definition of legal moves</td>
</tr>
<tr>
<td>Terminal state:</td>
<td>conditions for when game is over</td>
</tr>
<tr>
<td>Utility function:</td>
<td>a numeric value that describes the outcome of the game. E.g., -1, 0, 1 for loss, draw, win. (AKA payoff function)</td>
</tr>
</tbody>
</table>
Game vs. search problem

“Unpredictable” opponent ⇒ solution is a contingency plan

Time limits ⇒ unlikely to find goal, must approximate

Plan of attack:

• algorithm for perfect play (Von Neumann, 1944)
• finite horizon, approximate evaluation (Zuse, 1945; Shannon, 1950; Samuel, 1952–57)
• pruning to reduce costs (McCarthy, 1956)
Example: Tic-Tac-Toe
# Type of games

<table>
<thead>
<tr>
<th></th>
<th>Deterministic</th>
<th>Chance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Perfect information</strong></td>
<td>chess, checkers, go, othello</td>
<td>backgammon monopoly</td>
</tr>
<tr>
<td><strong>Imperfect information</strong></td>
<td></td>
<td>bridge, poker, scrabble, nuclear war</td>
</tr>
</tbody>
</table>
Type of games

- Deterministic
  - Perfect information
    - Chess, checkers, go, othello
  - Imperfect information
    - Backgammon, monopoly
- Chance
  - Bridge, poker, scrabble, nuclear war
The minimax algorithm

- Perfect play for deterministic environments with perfect information
- **Basic idea:** choose move with highest minimax value
  = best achievable payoff against best play
- **Algorithm:**
  1. Generate game tree completely
  2. Determine utility of each terminal state
  3. Propagate the utility values upward in the three by applying MIN and MAX operators on the nodes in the current level
  4. At the root node use minimax decision to select the move with the max (of the min) utility value

- Steps 2 and 3 in the algorithm assume that the opponent will play perfectly.
Generate Game Tree
Generate Game Tree
Generate Game Tree
Generate Game Tree

1 ply

1 move

X

O

X

O

X

O

X

O

O
A subtree
What is a good move?
Minimax

- Minimize opponent’s chance
- Maximize your chance
Minimax

- Minimize opponent’s chance
- Maximize your chance
Minimax

MAX

MIN

- Minimize opponent’s chance
- Maximize your chance
Minimax

- Minimize opponent’s chance
- Maximize your chance
minimax = maximum of the minimum
Minimax: Recursive implementation

function MINIMAX-DECISION(game) returns an operator

   for each op in OPERATORS[game] do
      VALUE[op] ← MINIMAX-VALUE(APPLY(op, game), game)
   end

   return the op with the highest VALUE[op]

function MINIMAX-VALUE(state, game) returns a utility value

   if TERMINAL-TEST(game)(state) then
      return UTILITY(game)(state)
   else if max is to move in state then
      return the highest MINIMAX-VALUE of SUCCESSORS(state)
   else
      return the lowest MINIMAX-VALUE of SUCCESSORS(state)

Complete: ?
Optimal: ?
Time complexity: ?
Space complexity: ?
Minimax: Recursive implementation

function Minimax-Decision(game) returns an operator

    for each op in Operators[game] do
        Value[op] ← Minimax-Value(Apply(op, game), game)
    end

    return the op with the highest Value[op]

function Minimax-Value(state, game) returns a utility value

    if Terminal-Test(game)(state) then
        return Utility[game](state)
    else if max is to move in state then
        return the highest Minimax-Value of Successors(state)
    else
        return the lowest Minimax-Value of Successors(state)

Complete: Yes, for finite state-space
Optimal: Yes

Time complexity: O(b^m)
Space complexity: O(bm) (= DFS
Does not keep all nodes in memory.)
Optimal decisions in multiplayer games

- Allow more than two players.
- First replace the single value for each node with a vector of values.
- For terminal states, this vector gives the utility of the state from each player's viewpoint.
- For the nonterminal states,

---

**Figure 6.4** The first three ply of a game tree with three players \((A, B, C)\). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.
Alliance in multiplayer games

• Multiplayer games usually involve affiances, whether formal or informal, among the players.
• Alliances are made and broken as the game proceeds.
• For example suppose $A$ and $B$ are in weak positions and $C$ is in a stronger position. Then it is often optimal for both $A$ and $B$ to attack $C$ rather than each other, lest $C$ destroy each of them individually.
• In this way, collaboration emerges from purely selfish behavior.
• See Section 17.6 for more on these complications.
Do We Have To Do All That Work?

MAX

MIN

3  12  8
Do We Have To Do All That Work?

\[
\begin{array}{c}
\text{MAX} \\
\text{MIN} \\
3 \quad 12 \quad 8
\end{array}
\]
Do We Have To Do All That Work?

Since 2 is smaller than 3, then there is no need for further search
Do We Have To Do All That Work?

\[
\text{MINIMAX-VALUE}(\text{root}) = \max(\min(3, 12, 8), \min(2, x, y), \min(14, 5, 2))
\]
\[
= \max(3, \min(2, x, y), 2)
\]
\[
= \max(3, z, 2) \quad \text{where } z \leq 2
\]
\[
= 3.
\]
2. **α-β pruning: search cutoff**

- **Pruning:** eliminating a branch of the search tree from consideration without exhaustive examination of each node.

- **α-β pruning:** the basic idea is to prune portions of the search tree that cannot improve the utility value of the max or min node, by just considering the values of nodes seen so far.

- Does it work? Yes, in roughly cuts the branching factor from $b$ to $\sqrt{b}$ resulting in double as far look-ahead than pure minimax.
\( \alpha - \beta \) pruning: example

\[
\begin{align*}
\text{MAX} & \quad \geq 6 \\
\text{MIN} & \quad 6
\end{align*}
\]
\( \alpha - \beta \) pruning: example

MAX

MIN

\( \geq 6 \)

\( \leq 2 \)
\( \alpha-\beta \) pruning: example

\[
\begin{array}{c}
\text{MAX} \\
\begin{array}{c}
\text{MIN} \\
\begin{array}{c}
6 \\
6 \quad 12 \quad 8 \\
\times \quad \times \\
2 \\
5 \\
\times \quad \times \\
\end{array}
\end{array}
\end{array}
\]

\[ \geq 6 \quad \leq 2 \quad \leq 5 \]
\(\alpha - \beta\) pruning: example

MAX

MIN

Selected move

\[\begin{array}{c}
\geq 6 \\
\leq 2 \\
\leq 5
\end{array}\]
α-β pruning: general principle

If $\alpha > v$ then MAX will choose $m$ so prune tree under $n$

Similar for $\beta$ for MIN
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

$\Rightarrow$ can easily reach depth 8 and play good chess

A simple example of the value of reasoning about which computations are relevant (a form of *metareasoning*)
The $\alpha$-$\beta$ algorithm:

Basically MINIMAX + keep track of $\alpha$, $\beta$ + prune

```
function Max-Value(state, game, $\alpha$, $\beta$) returns the minimax value of state
    inputs: state, current state in game
             game, game description
             $\alpha$, the best score for MAX along the path to state
             $\beta$, the best score for MIN along the path to state
    if Cutoff-Test(state) then return Eval(state)
    for each s in Successors(state) do
        $\alpha$ ← Max($\alpha$, Min-Value(s, game, $\alpha$, $\beta$))
        if $\alpha$ ≥ $\beta$ then return $\beta$
    end
    return $\alpha$

function Min-Value(state, game, $\alpha$, $\beta$) returns the minimax value of state
    if Cutoff-Test(state) then return Eval(state)
    for each s in Successors(state) do
        $\beta$ ← Min($\beta$, Max-Value(s, game, $\alpha$, $\beta$))
        if $\beta$ ≤ $\alpha$ then return $\alpha$
    end
    return $\beta$
```
More on the $\alpha$-$\beta$ algorithm

• Same basic idea as minimax, but prune (cut away) branches of the tree that we know will not contain the solution.
More on the $\alpha$-$\beta$ algorithm

• Same basic idea as minimax, but prune (cut away) branches of the tree that we know will not contain the solution.

• Because minimax is depth-first, let’s consider nodes along a given path in the tree. Then, as we go along this path, we keep track of:
  • $\alpha$: Best choice so far for MAX
  • $\beta$: Best choice so far for MIN
function \textsc{Alpha-Beta-Search}(state) \textbf{return}s an action
inputs: state, current state in game

\(v \leftarrow \textsc{Max-Value}(state, -\infty, +\infty)\)
\textbf{return} the action in \textsc{Successors}(state) with value \(v\)

function \textsc{Max-Value}(state, \(\alpha, \beta\)) \textbf{return}s a utility value
inputs: state, current state in game
\(\alpha\), the value of the best alternative for \textsc{Max} along the path to state
\(\beta\), the value of the best alternative for \textsc{Min} along the path to state

\textbf{if} \textsc{Terminal-Test}(state) \textbf{then return} \textsc{Utility}(state)
\(v \leftarrow -\infty\)
\textbf{for} \(a, s\) in \textsc{Successors}(state) \textbf{do}
\(v \leftarrow \textsc{Max}(v, \textsc{Min-Value}(s, \alpha, \beta))\)
\textbf{if} \(v \geq \beta\) \textbf{then return} \(v\)
\(\alpha \leftarrow \textsc{Max}(\alpha, v)\)
\textbf{return} \(v\)

function \textsc{Min-Value}(state, \(\alpha, \beta\)) \textbf{return}s a utility value
inputs: state, current state in game
\(\alpha\), the value of the best alternative for \textsc{Max} along the path to state
\(\beta\), the value of the best alternative for \textsc{Min} along the path to state

\textbf{if} \textsc{Terminal-Test}(state) \textbf{then return} \textsc{Utility}(state)
\(v \leftarrow +\infty\)
\textbf{for} \(a, s\) in \textsc{Successors}(state) \textbf{do}
\(v \leftarrow \textsc{Min}(v, \textsc{Max-Value}(s, \alpha, \beta))\)
\textbf{if} \(v \leq \alpha\) \textbf{then return} \(v\)
\(\beta \leftarrow \textsc{Min}(\beta, v)\)
\textbf{return} \(v\)
More on the \( \alpha - \beta \) algorithm: start from Minimax

Basically \texttt{MINIMAX} + keep track of \( \alpha, \beta \) + prune

```python
function \texttt{Max-Value}(state, game, \alpha, \beta) \texttt{returns} the minimax value of state
    \texttt{inputs: state, current state in game}
        \texttt{game, game description}
        \texttt{\alpha, the best score for MAX along the path to state}
        \texttt{\beta, the best score for MIN along the path to state}
    \texttt{if Cutoff-Test(state) then return Eval(state)}
    \texttt{for each s in Successors(state) do}
        \texttt{\alpha \leftarrow Max(\alpha, Min-Value(s, game, \alpha, \beta))}
        \texttt{if \alpha \geq \beta then return \beta}
    \texttt{end}
    \texttt{return \alpha}

function \texttt{Min-Value}(state, game, \alpha, \beta) \texttt{returns} the minimax value of state
    \texttt{if Cutoff-Test(state) then return Eval(state)}
    \texttt{for each s in Successors(state) do}
        \texttt{\beta \leftarrow Min(\beta, Max-Value(s, game, \alpha, \beta))}
        \texttt{if \beta \leq \alpha then return \alpha}
    \texttt{end}
    \texttt{return \beta}
```

Note: These are both Local variables. At the Start of the algorithm, We initialize them to \( \alpha = -\infty \) and \( \beta = +\infty \)
More on the $\alpha$-$\beta$ algorithm

In Min-Value:

```
for each $s$ in Successors(state) do
    $\beta \leftarrow \text{MIN}(\beta, \text{MAX-VALUE}(s, game, \alpha, \beta))$
    if $\beta \leq \alpha$ then return $\alpha$
end
return $\beta$
```

MAX

$\alpha = -\infty$
$\beta = +\infty$

MIN

Max-Value loops over these

MIN-Value loops over these

MAX

$\alpha = -\infty$
$\beta = 5$

$\alpha = -\infty$
$\beta = 5$

$\alpha = -\infty$
$\beta = 5$

5 10 6 2 8 7
More on the $\alpha$-$\beta$ algorithm

In Max-Value:

\[
\text{for each } s \text{ in } \text{Successors}(state) \text{ do}
\]
\[
\alpha \leftarrow \text{MAX}(\alpha, \text{MIN-Value}(s, game, \alpha, \beta))
\]
\[
\text{if } \alpha \geq \beta \text{ then return } \beta
\]
\[
\text{end}
\]
\[
\text{return } \alpha
\]

\[
\begin{array}{c}
\alpha = -\infty \\
\beta = +\infty
\end{array}
\]

Max-Value loops over these

\[
\begin{array}{c}
\alpha = 5 \\
\beta = +\infty
\end{array}
\]

\[
\begin{array}{c}
5 \\
10 \\
6 \\
2 \\
8 \\
7
\end{array}
\]

\[
\begin{array}{c}
\alpha = -\infty \\
\beta = 5
\end{array}
\]

\[
\begin{array}{c}
\alpha = -\infty \\
\beta = 5
\end{array}
\]

\[
\begin{array}{c}
\alpha = -\infty \\
\beta = 5
\end{array}
\]
More on the $\alpha$-$\beta$ algorithm

In Min-Value:

```python
for each $s$ in Successors(state) do
    $\beta \leftarrow \min(\beta, \text{MAX-VALUE}(s, game, \alpha, \beta))$
    if $\beta \leq \alpha$ then return $\alpha$
end
return $\beta$
```

MAX

MIN

Min-Value loops over these

MAX

<table>
<thead>
<tr>
<th>Value</th>
<th>$\alpha$</th>
<th>$\beta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>$-\infty$</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>$-\infty$</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>$-\infty$</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

End loop and return 5
More on the $\alpha\beta$ algorithm

In Max-Value:

\[
\text{for each } s \text{ in } \text{Successors(state) do}
\]
\[
\alpha \leftarrow \text{MAX}(\alpha, \text{MIN-Value}(s, \text{game}, \alpha, \beta))
\]
\[
\text{if } \alpha \geq \beta \text{ then return } \beta
\]
\[
\text{end}
\]
\[
\text{return } \alpha
\]

Max-Value loops over these

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

\[
\alpha = 5, \quad \beta = +\infty
\]

\[
\alpha = 5
\]
\[
\beta = 2
\]

End loop and return 5
Another way to understand the algorithm

• From:
  

• For a given node $N$,

  $\alpha$ is the value of $N$ to MAX
  $\beta$ is the value of $N$ to MIN
Alpha-Beta Pruning

- Explore the game tree to depth $h$ in **depth-first** manner
- Back up alpha and beta values whenever possible
- Prune branches that can’t lead to changing the final decision
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
6-4 Imperfect Real time Decisions

- The minimax algorithm generates the entire game search space, whereas the alpha-beta algorithm allows us to prune large parts of it.
- However, alpha-beta still has to search all the way to terminal states for at least a portion of the search space.
- **This depth is usually not practical, because moves must be made in a reasonable amount of time.**
- Shannon's 1950 paper, *Programming a computer for playing chess*, proposed instead that programs should **cut off** the search earlier and apply a heuristic **evaluation function** to states in the search, effectively turning nonterminal nodes into terminal leaves.
Suggestion

• the utility function is replaced by a **heuristic evaluation** function EVAL, which gives an estimate of the position's utility,

• the terminal test is replaced by a **cutoff test** that decides when to apply Eval.
1. Move evaluation without complete search

- Complete search is too complex and impractical
- **Evaluation function**: evaluates value of state using heuristics and cuts off search
- Chess players have developed ways of **judging the value of a position**, because humans are even more limited in the amount of search they can do than are computer programs.
- The performance of a game-playing program is dependent on the quality of its evaluation function.
- An inaccurate evaluation function will guide an agent toward positions that turn out to be lost.
How exactly do we design good evaluation functions?

- The evaluation function should order the terminal states in the same way as the true utility function; otherwise, an agent using it might select suboptimal moves even if it can see ahead all the way to the end of the game.
- The computation must not take too long! (The evaluation function could call MINIMAX-DECISION as a subroutine and calculate the exact value of the position, but that would defeat the whole purpose: to save time.)
- For nonterminal states, the evaluation function should be strongly correlated with the actual chances of winning.
Feature

• Most evaluation functions work by calculating various features of the
  • the number of pawns (兵) possessed by each side in a game of chess.
• The features, taken together, define various categories or equivalence classes of states: the states in each category have the same values for all the features. Any given category, generally speaking, will contain some states that lead to wins, some to draws, and some that lead to losses.
• The evaluation function cannot know which states are which, but it can return a single value that reflects the proportion of states with each-outcome.
• For example, suppose our experience suggests that 72% of the states encounter in the category lead to a win (utility +1); 20% to a loss (-1), and 8% to a draw (0). Then reasonable evaluation for states in the category is the weighted average or expected value; $(0.72 \times +1) + (0.20 \times -1) + (0.08 \times 0) = 0.52$.
• In principle, the expected value can determined for each category, resulting in an evaluation function that works for any state.
Material value

- Feature:
  - requires too many categories and hence too much experience to estimate all the probabilities of Winning.
- Instead, most evaluation function compute separate numerical contributions from each feature and then *combine* them to find the total value.
- Chess books give an approximate **material value** for each piece:
  - each pawn is worth 1, a knight or bishop(主教) is worth 3, a rook(車) 5, and the queen 9. Other features such as "**good pawn structure**" and "**king safety**", might be worth half a pawn, say.
- These feature values are then simply added up to obtain the evaluation of the position.
- Mathematically, this kind of evaluation function is called a **weighted linear function**.
Chess
Chess

- 棋盤分成黑白兩色，共64格，且player的最底層一列的最右邊一格為白色，每次必由白棋先開始。
- 棋子分為：king(國王)，queen(皇后)，rook(城堡)，bishop(主教)，knight(騎士)，pawn(士兵) 六種。
- king: 只能走一格，但可以直向，橫向，和斜向。
- queen: 能走三個方向，不限格數。
- rook: 能走直向，橫向，不限格數。
- bishop: 只能走斜向，不限格數。
- knight: 能直向或橫向走兩格之後，置於左或右邊一格。
- pawn: 只能前進一格或兩格，不能後退，可吃左前或右前一格之子。
- 輸贏即以是否擄獲對方之 king，若不行，就為平手
Features

- Evaluation functions

- **Weighted linear evaluation function:** to combine $n$ heuristics
  \[ f = w_1f_1 + w_2f_2 + \ldots + w_nf_n \]

E.g., $w$’s could be the values of pieces (1 for prawn, 3 for bishop etc.)
$f$’s could be the number of type of pieces on the board
Cutting off Search

- The next step is to modify ALPHA-BETA-SEARCH so that it will call the heuristic EvAL function when it is appropriate to cut off the search.
- Replace the two lines in Figure 6.7 that mention TERMINAL-TEST with the following line:
- if CUTOFF-TEST(state, depth) then return EvAL(state)
Minimax with cutoff: viable algorithm?

MinimaxCutoff is identical to MinimaxValue except
1. Terminal? is replaced by Cutoff?
2. Utility is replaced by Eval

Does it work in practice?

\[ b^m = 10^6, \quad b = 35 \quad \Rightarrow \quad m = 4 \]

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

Assume we have 100 seconds, evaluate \( 10^4 \) nodes/s; can evaluate \( 10^6 \) nodes/move
6.5 State-of-the-art for deterministic games

- **Chess:**
  - In 1957, Herbert Simon predicted that within 10 years computers would beat the human world champion.
  - Forty years later, the Deep Blue program defeated Garry Kasparov in a six-game exhibition match.
  - Simon was wrong, but only by a factor of 4.
Deep Blue

- Developed by Murray Campbell, Feng-Hsiung Hsu, and Joseph Hoane at IBM.
- The winning machine was a parallel computer with
  - 30 IBM RS/6000 processors running the "software search" and
  - 480 custom VLSI chess processors that performed move generation (including move ordering),
  - the "hardware search" for the last few levels of the tree, and the evaluation of leaf nodes.
- Deep Blue searched 126 million nodes per second on average, with a peak speed of 330 million nodes per second.
- It generated up to 30 billion positions per move, reaching depth 14 routinely.
Deep Blue

- The heart of the machine is a standard iterative-deepening alpha–beta search with a transposition table,
- but the key to its success seems to have been its ability to generate extensions beyond the depth limit for sufficiently interesting lines of forcing/forced moves.
- In some cases the search reached a depth of 40 plies.
- The evaluation function had over 8000 features, many of them describing highly specific patterns of pieces.
- An "opening book" of about 4000 positions was used, as well as a database of 700,000 grandmaster games from which consensus recommendations could be extracted.
- The system also used a large endgame database of solved positions, containing all positions with five pieces and many with six pieces.
- This database has the effect of substantially extending the effective search depth, allowing Deep Blue to play perfectly in some cases even when it is many moves away from checkmate.
雙陸棋 Backgammon
Checkers: Tinsley vs. Chinook

Name: Marion Tinsley
Profession: Teach mathematics
Hobby: Checkers
Record: Over 42 years
loses only 3 games
of checkers
World champion for over 40 years

Mr. Tinsley suffered his 4th and 5th losses against Chinook
Chinook

First computer to become official world champion of Checkers!
Othello: Murakami vs. Logistello

1997: The Logistello software crushed Murakami by 6 games to 0
• 在「Backgammon」棋盤上下總共有24列的棋格，玩家遊戲時一次擲兩粒骰子，取得兩個數字，再決定由哪一顆棋子移動骰子的步數，把棋子往目的地移動，玩家可以一次讓兩顆棋子分別移動，也可以只讓一顆棋子移動總合的步數，最後先將所有棋子完成搬運的即算獲勝。

但是，在移動棋子的過程中，一旦對方的棋子在某個棋格上只有單一棋子，而玩家剛好擲出可讓己方棋子到達對方棋格位置的數字時，可做出攻擊的動作，把己方的棋子移至對方棋格上，並將對方棋子送至中間的空白區。這樣可以減慢對手搬運的速度。

丟骰子後.若點數為4..5..則兩子各走4.與5步.或1子走9步.. 若2顆一樣如5..5..則可double走.即2子都10步或單子走20步. 每子落點位置必須無對方的棋子.或對方只有單一個子.此情形可以拿掉對方的棋子讓對方該子從頭走起.直到你的棋子全部走到終點.如圖所示. 對方剩下幾子未入終點你就贏幾子.
Go 圍棋

• Go is the most popular board game in Asia, requiring at least as much discipline from its professionals as chess.
• Board is 19 x 19, the branching factor starts at 361, which is too daunting for regular search methods.
• Up to 1997 there were no competent programs at all, but now programs often play respectable moves.
• Most of the best programs combine pattern recognition techniques with limited search.
• The strongest programs at the time of writing are probably Chen Zhixing's Goemate and Michael Reiss' Go4++, each rated somewhere around 10 kyu (weak amateur).
• Go is an area that is likely to benefit from intensive investigation using more sophisticated reasoning methods. Success may come from finding ways to integrate several lines of local reasoning about each of the many, loosely connected "subgames" in which Go can be decomposed. Such techniques would be of enormous value for intelligent systems in general.
Go: Goemate vs. ??

Name: Chen Zhixing
Profession: Retired
Computer skills:
  - self-taught programmer
Author of Goemate (arguably the best Go program available today)

Gave Goemate a 9 stone handicap and still easily beat the program, thereby winning $15,000

Jonathan Schaeffer
Bridge 橋牌

- Bridge is a game of **imperfect information**: a player's cards are hidden from the other players.
- Bridge is also a **multiplayer game** with four players, although the players are paired into two teams.
- Optimal play in bridge can include elements of
  - information-gathering,
  - communication,
  - bluffing, and
  - careful weighing of probabilities.
- Many of these techniques are used in the Bridge Baron TM program (Smith *et al.*, 1998), which won the 1997 computer bridge championship.
- While it does not play optimally, Bridge Baron is one of the few successful game-playing systems to use **complex, hierarchical plans** (see Chapter 12) involving high-level ideas such as **finessing** and **squeezing** that are familiar to bridge players.
Bridge

- The GIB program (Ginsberg, 1999) won the 2000 championship quite decisively.
- GIB uses the "averaging over clairvoyancy" method, with two crucial modifications.
  - First, rather than examining how well each choice works for every possible arrangement of the hidden cards—of which there can be up to 10 million—it examines a random sample of 100 arrangements.
  - Second, GIB uses explanation-based generalization to compute and cache general rules for optimal play in various standard classes of situations. This enables it to solve each deal exactly. GIB's tactical accuracy makes up for its inability to reason about information.
- It finished 12th in a field of 35 in the par contest (involving just play of the hand) at the 1998 human world championship, far exceeding the expectations of many human experts.
Summary

Games are fun to work on! (and dangerous)
They illustrate several important points about AI
◊ perfection is unattainable ⇒ must approximate
◊ good idea to think about what to think about
◊ uncertainty constrains the assignment of values to states
Games are to AI as grand prix racing is to automobile design
Exercise: Game Playing

Consider the following game tree in which the evaluation function values are shown below each leaf node. Assume that the root node corresponds to the maximizing player. Assume the search always visits children left-to-right.

(a) Compute the backed-up values computed by the minimax algorithm. Show your answer by writing values at the appropriate nodes in the above tree.
(b) Compute the backed-up values computed by the alpha-beta algorithm. What nodes will not be examined by the alpha-beta pruning algorithm?
(c) What move should Max choose once the values have been backed-up all the way?
### Chess: Kasparov vs. Deep Blue

<table>
<thead>
<tr>
<th>Kasparov</th>
<th>Deep Blue</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Height</strong></td>
<td>6’ 5”</td>
</tr>
<tr>
<td>5’10”</td>
<td></td>
</tr>
<tr>
<td><strong>Weight</strong></td>
<td>2,400 lbs</td>
</tr>
<tr>
<td>176 lbs</td>
<td></td>
</tr>
<tr>
<td><strong>Age</strong></td>
<td>4 years</td>
</tr>
<tr>
<td>34 years</td>
<td></td>
</tr>
<tr>
<td><strong>Computers</strong></td>
<td>32 RISC processors + 256 VLSI chess engines</td>
</tr>
<tr>
<td>50 billion neurons</td>
<td></td>
</tr>
<tr>
<td><strong>Speed</strong></td>
<td>200,000,000 pos/sec</td>
</tr>
<tr>
<td>2 pos/sec</td>
<td></td>
</tr>
<tr>
<td><strong>Knowledge</strong></td>
<td>Primitive</td>
</tr>
<tr>
<td>Extensive</td>
<td></td>
</tr>
<tr>
<td><strong>Power Source</strong></td>
<td>Electrical</td>
</tr>
<tr>
<td>Electrical/chemical</td>
<td></td>
</tr>
<tr>
<td>Enormous</td>
<td>None</td>
</tr>
<tr>
<td><strong>Ego</strong></td>
<td></td>
</tr>
</tbody>
</table>

1997: Deep Blue wins by 3 wins, 1 loss, and 2 draws

Jonathan Schaeffer
Chess: Kasparov vs. Deep Junior

Deep Junior

8 CPU, 8 GB RAM, Win 2000
2,000,000 pos/sec
Available at $100

August 2, 2003: Match ends in a 3/3 tie!