

## Lecture 4

### Two player games

Two player games are among the most well-studied areas of AI.

We will look into the basic strategies and some variations of AI game playing in this chapter.

#### Reference:

- Textbook Chapter 6.

AI(0270)

#### Why study games?

- It is clear that **most games require intelligence** to play well: good “existence proof” of AI research.
- State of many games are **easy to represent accurately** (e.g., chess, backgammon, etc), and **the number of actions are usually small**. Dynamics easy to understand.  
Less formal games like football does not attract much interest in AI community.
- **Presence of opponent** introduces a certain degree of **uncertainty**, requiring an interesting element in AI: **contingency**.  
Contingency: “What if ...” reasoning.
- **Single opponent**, so no politics to play with. Each player simply focuses on his own goal.
- **Huge trees**: game trees are usually unrealistic to search fully—force the researchers to deal with another type of **uncertainty**: we don’t have time to find the exact answer!

AI(0270)-4.1

#### Basic concept: Formalizing games as search

It is easy to see that **states of games form state spaces**.

But **how one can search** through such state spaces?

In other words, how to formalize the two-player nature?

- To represent the **end result** of a game, we use a **single number** called **utility**. This number **represents how good is the outcome**.  
E.g., who win, by how much, etc.
- One of the players makes moves which **maximize** the utility, and the other player makes moves which **minimize** the utility.
- E.g., for a simple game between A and B with a goal to win the game (without the possibility of a draw), we can assign the value to be 1 if A wins and 0 if B wins. A is the player maximizing the utility, B is the player minimizing it.
- They are called the **MAX player** and **MIN player** respectively.

AI(0270)-4.2

#### Game in computers

In searching problem, we use a **state representation**, a **successor function** and a **goal test function** to perform a search. For games, we need...

- State **representation**, as usual.
- A **successor function**, as usual.
- A **terminal test function** telling us whether the game is terminated.
- A **utility function** which, when given a state which is terminal, tells us what is the utility (for the MAX player).

AI(0270)-4.3

#### Example: Nim

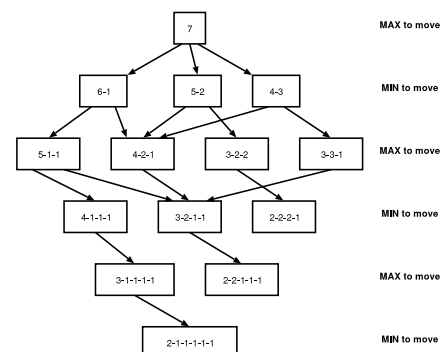
To illustrate the idea, let’s see a very simple game called **Nim**:

- There is a **pile of some matches** on the table.
- The two players move in **alternative turns**.
- When a player moves, he must **choose a pile with at least 3 matches** on the table and **split it into two piles**.
- When doing a split, the **resulting piles** must not be of the same size.  
So the smallest pile which can be split is of size 3.
- Since piles are smaller and smaller, at the end all piles have either 1 or 2 matches, when a player **can’t make a move**. That player **loses**.

AI(0270)-4.4

#### Example tree: Nim with 7 matches

The complete search space is as follow.



AI(0270)-4.5

### Searching in the tree

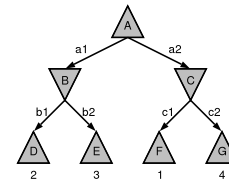
How we know whether a player should win? What should be the **ideal strategy**?

- If we are already in a state with **end-game state**, the **end-game utility tells us the result of the game**.
- What if we are at a state A **one step away from end-game**? Depending on whether we are a MAX player or a MIN player, we may want to choose the step leading to the largest or smallest utility. Let's call this utility the **utility of the state A**, since we will "normally" get that utility.
- What if we are at a state two steps away from end-game? Of course, the **next states** are one step away from end-game, with **well-defined utility**. We **should choose the one with largest or smallest utility depending on whether we are MIN or MAX player**.
- Now it seems **recursively defined**.

AI(0270)-4.6

### Example

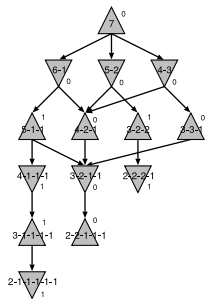
We use up triangles to mean **MAX states**, when the one to make a decision is the MAX player. Similarly, down triangles means **MIN states**.



- Here best MAX move is a1, while the best MIN reply is b1, resulting in a value of 2.  
If MAX takes a2, MIN will take c1, resulting in a smaller value 1 for MAX.
- Note that the **exact value** of the utilities are **not important**. **Only the orderings** of the utilities are.

AI(0270)-4.7

### Example tree: Nim with 7 matches



The root state has utility 0, meaning that the MIN player should always win.

Why? At the root state, the MAX player has no way to choose a next-state with utility = 1: the MIN player can always choose to move to the 4-2-1 state and win eventually (utility = 0).

Note that **the fate is fixed at the beginning** for any such game.

That is, unless the MIN player makes a mistake. In other words, to win any non-trivial deterministic two-player game, you're really seeking the mistake of the opponent.

AI(0270)-4.8

### Minimax algorithm

Once we know how the game tree works, the **optimal strategy can be found easily**—at least in principle. Simply modify **backtracking search**.

```
def MinimaxDecision(state):
    # return an action, assuming computer is MAX
    value, action = -infinity, NoAction
    for newaction, newstate in Successors(state):
        newvalue = MinValue(newstate)
        if newvalue > value:
            value, action = newvalue, newaction
    return action

def MinValue(state):
    # MaxValue(state) is similar
    if (IsTerminal(state)):
        return Utility(state)
    value = infinity
    for newaction, newstate in Successors(state):
        value = min(value, MaxValue(newstate))
    return value
```

AI(0270)-4.9

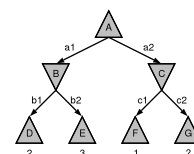
### What happens when there are multiple players?

- Utility becomes a **vector** (one value for each player), and cannot be represented by just one number.
- Each player tries to maximize his own value.
- Players might form **alliance**, e.g., to crush strong opponents.
- If alliance is formed due to a strong opponent, and the strong opponent is weakened as a result, alliance tends to be **broken**.
- Breaking of alliances might be **delayed** if the short term advantage of breaking alliance is less than the long term disadvantage of being judged as untrustworthy.
- Sum of utility might cease to be 1 (non-zero-sum game). Then players might play **cooperatively** to get to a point where both values are maximized.
- But we will not further consider such possibilities.

AI(0270)-4.10

### Avoiding complete search

Sometimes we don't need to consider all the states. E.g.,



Suppose we have searched the left subtree and know that its value is 2. Now we check the state C, and found that c1 is of value 1.

Do we need to consider c2? **No!**

Why? It will **never be chosen by the MAX player anyway!**

It already has a choice with value 2, why bother with a value of at most 1?

AI(0270)-4.11

### Alpha-Beta pruning: avoid searching useless branches

- At each stage, keep a range of “**interesting values**” for the search.
- Initially, at the top-level evaluation, **everything is interesting**, so we set the range to  $[-\infty, \infty]$ . They are called  $[\alpha, \beta]$  conventionally.
- When a **recursive call returns**, we **update** the bounds. A MAX-state updates its **lower** bound, a MIN-state updates its **upper** bound.  
E.g., the state B returns with value 2, so the max state A trims down the set of interesting values to  $[2, \infty]$ .
- When **performing a recursive call**, the bounds of interesting values are **passed down the tree**.  
E.g., when A calls C recursively, its current range  $[2, \infty]$  is given to C.
- If the range **collapses**, we declare the remaining branches **uninteresting**, and thus returns immediately. We say those branches are **pruned**.  
E.g., when the c<sub>1</sub> branch returns, the bound becomes  $[3, 2]$ —collapsed (because there is no value in that range). The search returns immediately.

AI(0270)-4.12

### Code: Alpha-beta pruning

The change to Minimax is minimal...

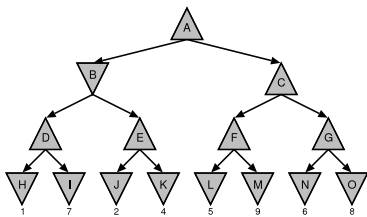
```
def AlphaBetaDecision(state): # return an action, assuming computer is MAX
    alpha, beta, action = -infinity, infinity, NoAction
    for newaction, newstate in Successors(state):
        newvalue = MinValue(newstate, alpha, beta)
        if alpha < newvalue:
            alpha, action = newvalue, newaction
    return action

def MinValue(state, alpha, beta): # MaxValue(state, alpha, beta) is similar
    if (IsTerminal(state)):
        return Utility(state)
    for newaction, newstate in Successors(state):
        beta = min(beta, MaxValue(newstate, alpha, beta))
        if alpha >= beta: # Collapsed range
            break
    return beta
```

AI(0270)-4.13

### Exploration order matters a lot!

- Just like DFS for CSP, order of exploration does not matter for the end result, but **matters a lot for performance**.  
The key assumption still holds: the path doesn't matter.
- If we **explores a good subtree first** for each state, we can prune the tree more effectively. E.g., try left to right and right to left of this:



AI(0270)-4.14

### How good is it?

- If **random ordering**: number of states examined is  $O(b^{3d/4})$ .
- If **perfect ordering**: number of states examined is  $O(b^{d/2})$ .
- But **how** to find good ordering?  
Won't be perfect, otherwise the ordering will play the game perfectly!
- Static ordering: **some moves are usually better**, e.g., capturing, queening... Try them first.
- Or, guess how good are the states (more about it later).
- For most games, a simple ordering rule makes us close to the perfect ordering case.
- So effectively, we can search **twice as far**.

AI(0270)-4.15

### Repeated states

- Like other searching problems, game trees can contain a lot of repeated states, which seriously reduces search efficiency.
- This is particularly important when various permutations, or “**transpositions**”, can lead to the same outcome.  
E.g., in chess, if moves do not affect each other.
- They can be handled by remembering each state seen, and their corresponding values.
- It is very similar to the **closed** list in the *GraphSearch* algorithm, except that it will be a **map** instead of a set.  
Usually it is implemented by a hash table or a height-balancing tree.
- Such a list is called **transposition table** in game searching.

AI(0270)-4.16

### Heuristic in games

- Minimax** and **alpha-beta** are good in the sense that it **always finds the best strategy**.
- But it **requires** us to **search to the bottom of a tree**, which is **possible only in the simplest game**.  
E.g. Tic-tac-toe.
- So what if we want to play a more complicated game, which state space is too large to be searched completely?
- We will have to **cut-off** at some point, i.e., return **even though it is not a terminal state**.
- What value to return? We will need a **guess** of the utility **without further search: evaluation function**.  
So the Minimax and alpha-beta algorithm is still usable, as long as we can find a good cut-off point and evaluation function. Of course we don't always find the best strategy: heuristic search.

AI(0270)-4.17

### Evaluation functions

- The most frequently used strategy: use some **features** of the game configuration to evaluate the board.  
E.g., in chess, number of each type of pieces, good pawn structure, king safety, etc.
- Effectively, the evaluation function **categorizes the possible configurations**, and assign values to categories rather than states.  
E.g., same number of pieces for all types => same category.
- To be a good evaluation function:
  - It should **match the utility** at end-game conditions.
  - It should reflect the **chance of winning** of each category.  
Why "chance"? Uncertainty arise when we "categorize" states.
  - It should **categorize the states properly**: a category should consist mostly of states that stands similar winning chance.

AI(0270)-4.18

### Weighted linear functions

- Most game-playing programs use evaluation functions that are **linear combinations of features**. I.e., each feature is given a **weight**, and the weighted sum is used as the evaluation.
- **Finding the optimal value of the "multipliers"** can be treated as a **search problem** of continuous values.  
Use local search strategies, e.g., hill climbing, simulated annealing, etc.
- **How to know one version plays better than another?** Although not completely accurate, letting the two versions play against each other usually works well.
- Sometimes **non-linear** functions are useful. E.g., 2 knights can be more valuable than 2 times one knight. But hard to learn such rules.
- Sometimes **time-dependent** function is also useful. E.g., a knight is more valuable at end-game than at beginning of game.

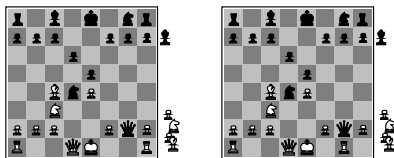
AI(0270)-4.19

### Example: Chess evaluation functions

From elementary chess books...

1. **Material value**: each pawn worths 1, knights and bishops worth 3, rooks worth 5, queens worth 9.
2. **Other structures**: e.g., king safety, good pawn structure ("center control"), etc., might worth 0.5.

They claim that a 1 point advantage leads to good chance to win, while a 3 points advantage nearly surely leads to win. Good and bad examples...



AI(0270)-4.20

### When to cut off?

When to stop generating new states and call the evaluation function?

- Simplest way: **after a fixed depth**.
- Some games with **clock**: we must make a move after some time. Then we can **use iterative deepening**. When time is up, **return the move in last completed iteration**.
- Problem: cut-off right before a **big swing in game value**: gives seriously wrong evaluation!  
E.g., In RHS of last figure: we really should not have tried evaluating at such state.

AI(0270)-4.21

### Quiescence search

- **Fix**: apply evaluation function only when the game is **quiescent**, i.e., unlikely to exhibit wild swing in value quickly.  
E.g., in chess, no favourable capture is possible, no possible queening of pawn, etc.
- What to do at a state that is **deep enough**, but is not quiescent? We can just continue the search...  
We usually call this part of the search a "quiescence search": we look for a state that is quiescent so that we can call evaluation function reliably.
- But then we might end up searching for too long (since quiescence can stay there if it is not removed).  
The fast evaluation function suddenly becomes a infinitely long computation.
- Fix: only try moves that will **remove the non-quiescence**.

AI(0270)-4.22

### Other strategies

- **Singular extension**: when confronting a long sequence of non-quiescent states, only consider the most promising move.  
So branching factor is 1, and we can afford doing it deeper (inaccurately, but so what: evaluation function is not accurate anyway).
- **Forward pruning**: for moves that looks too bad, just ignore the possibility rather than searching to the desired depth.  
Can be dangerous: that completely misses some possibilities—even though that is exactly what's actually done by most people.
- **Move databases**: hard-code a table of moves for some configurations, mostly at the start of the game ("book moves"), or near the close of the game ("end-game database").  
Book moves makes initial move fast, while end-game database allows evaluations near the end-game to be perfect.

AI(0270)-4.23

**Probabilistic Games**

- So far our environment is completely deterministic. What to do if the game is stochastic, i.e., has some sort of probability?
- Many games are probabilistic, e.g., **throwing of dice, turning of dials, or shuffling of cards.**
- The **search spaces** of such games are a little bit different from those of deterministic games.
- The **strategies** in such games are very different from deterministic games.

AI(0270)-4.24

**Example: Backgammon**

- **Backgammon** is a classic example of probabilistic games, where the uncertainty comes from two **dice**.  
Classic, probably because backgammon is considered to depend heavily on skills rather than just luck, and because a good program had been written for it.
- The board consists of 26 *positions* which can be named 0 to 25.
- Each player has 15 **men** on the board, and the aim is to quickly move all men to the other end of the board: one of them towards 0, the other towards 25.
- The players play in turns. In the beginning of each turn, **two dice are rolled to determine the possible moves.**
- The player advances one man by the number on the top of one dice, and *then* advance one man (maybe the same, maybe different) by the number on the top of the other dice.  
Except when the player get double, when he makes four moves.

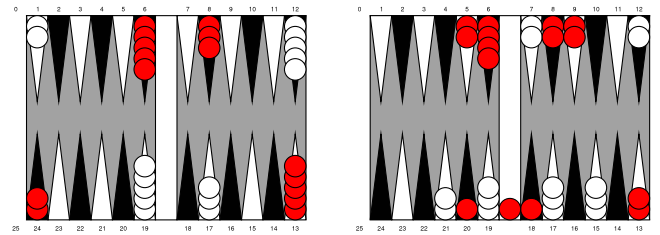
AI(0270)-4.25

**Backgammon: cont'd**

- When a man advances and stops at a position occupied by **one other opponent man** (an open man), the opponent's man is captured and must **make the complete journey again** from the "bar".  
So it is bad to leave man open. In actual plays, captured men that are never moved are placed at the middle bar, rather than cell 0 or 25.
- When a player has captured men on the bar, all other men **must not move** until all of them started moving.
- When a man is moved, it cannot stop at a position occupied by **two or more other opponent pieces** (called a **point**). Such move is **invalid**.
- A player leaves some moves (dice) unused if and only if there is **no possible valid move**.  
With skill (and luck), it is possible to have many points and form a "wall" to block captured men from getting out of the bar, stopping the opponent for a long time.
- And there are other subtleties on end-game that we won't explain.

AI(0270)-4.26

**Some game positions**

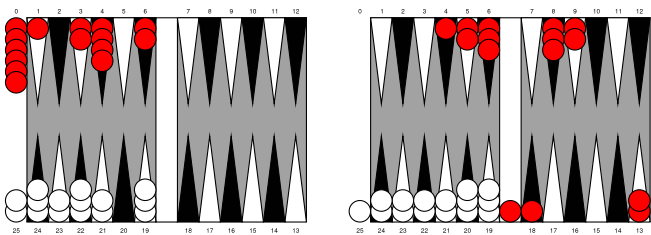


Initial position. White to move to position 25, red to position 0.

Middle of game, white better a bit: it trapped a red man in bar, moved both men at position 1 out of first quadrant, and leave no open man.

AI(0270)-4.27

**Some more game positions**



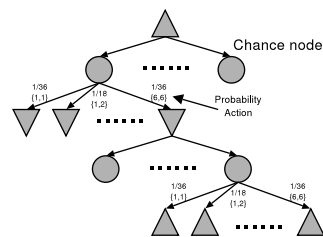
Game nearly ends. Red is ahead of white a bit.

A really horrible position for red: it has a man in the bar, but there is no way to get out at all—until white start clearing out the board.

AI(0270)-4.28

**A State Space with chance states**

How to draw a state space?



There are some **chance states** in the SS (the circles). Nobody controls them, and each outcome is taken at certain probability.

Under such situations, the **outcome** of a game is **not known at the beginning**.

Target of our agent: **best expected outcome!**

i.e., averaging out the values of all possibilities by their probabilities.

AI(0270)-4.29

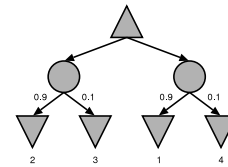
**State space evaluation with chance states**

Let's go back to basics and see how the SS can be used to find the **best move** for a player—given the power to completely look at the tree.

- In the past, we define the **value of a state** to be the value that one can expect **if both players make the best move**.
- But now, even if both players make the best move, **the value is unknown**. So we have to change the definition of "value of a state".
- The fix: the value of each state is **the expected value** obtained if we start from that state and every player does the best she can.  
Those familiar with risk theories would definitely object to this definition: it does not take variance into account at all. But let's keep things simple.
- So we assume that **the players seek to maximize and minimize the expected value** of the game.

AI(0270)-4.30

**Example**



The values of the MIN states are 2, 3, 1 and 4 respectively.  
The values of the chance states are  $2 \times 0.9 + 3 \times 0.1 = 2.1$  and  $1 \times 0.9 + 4 \times 0.1 = 1.3$  respectively.  
The value of the MAX state is thus 2.1, choosing the left branch.

AI(0270)-4.31

**Code: ExpectiMinimax**

Again, once we get the idea, the code is simple.

```
# Minimax, MaxValue and MinValue are the same as before,
# but call ExpectiValue instead of MaxValue or MinValue

def ExpectiValue(state):
    if IsTerminal(state):
        return Utility(state)
    if IsMax(state):
        return MaxValue(state)
    if IsMin(state):
        return MinValue(state)
    # Not MIN or MAX, so it must be a chance state
    value = 0
    for newaction, newstate, chance in Successors(state):
        value = value + ExpectiValue(newstate) * chance
    return value
```

AI(0270)-4.32

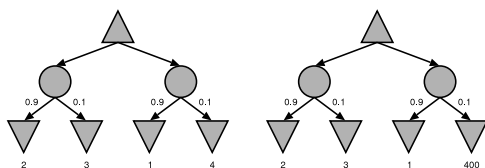
**The importance of exact utility values**

- In **deterministic games**, the **exact utilities** of the end-game positions are **not important**, only the **order** matters.
- But this is **not the case for games with chance**: chance states **adds up** the product of probability and **values**, instead of just finding min and max.
- In other words, we **must be much more careful** when defining the **utility** of the end-game, and also the **value** of the intermediate states.
- In particular, it **must reflect** how much the players like the end-game.
- E.g., the MAX player must like having two games ended as -1 and 4 respectively more than having two games ended as 1 and 1 respectively. Expectiminimax will choose the former branch.

AI(0270)-4.33

**Example**

Let's see how the game play can be different when the exact values are changed, even though the ordering doesn't change.



The better value comes from left branch (2.1) on the left SS, but comes from the right branch (40.9) on the right SS, even though the two SS's have the same **ordering** of game values.

In the right, while the chance is slim, the payoff of getting a value of 400 is large, making the expected value large.

AI(0270)-4.34

**How difficult is it?**

Okay, so fast is Expectiminimax?

- Let's suppose that **every chance state leads to n distinct results**.
- If we want to search to depth n, and the branching factor is b ...
- Each "ply" consists of n chance results, each with b different moves. So **bn** things to evaluate in each move, and **(bn)<sup>d</sup> time cost**.
- E.g.: Backgammon—n=21, b is typically around 20. So... the effective branching factor is 420!
- Depth 4 is about the best we can do. In other words, **with chance states, the search is much more costly!**
- To make a good agent **depends heavily on a good evaluation function**—looking at the board, one have to know pretty much how good is it without searching.

AI(0270)-4.35

### Adapting alpha-beta: possible?

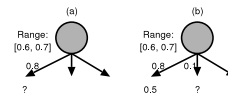
If Expectiminimax is so hard, is it possible to prune it?

- It seems much more difficult: at a chance state, **you must evaluate all branches** before knowing the value of a chance state!
- But what if the values of terminal states are bounded? Then it is possible to adapt Alpha-Beta to some extent.
- Given such a bound, we can turn **bound the value of a chance state** when some sub-trees are evaluated.
- Using the same idea as Alpha-Beta, we can prune part of the tree.
- “Interesting range” management can be complicated at chance states.
- Not as effective as that for deterministic games.

AI(0270)-4.36

### Examples: passing interesting range from chance states

Suppose we know the utility is always between 0 and 1...



Situation (a): if left subtree has value  $< 0.5$ , then even if other trees contributes 1, it can't exceed 0.4, so uninteresting. Similarly, uninteresting if value  $> 0.875$ . So range to pass:  $[0.5, 0.875]$ .

Situation (b): We already have 0.4. The interesting range to pass can be calculated to be  $[1, 3]$ , but this is completely outside legitimate utility range  $[0, 1]$ . We conclude that the range collapsed.

Exercise: What value should we return?

AI(0270)-4.37

### Card games

- Card games are **very** challenging to play well (either by human or machine), due to the uncertainty about where are the cards.  
Uncertainty comes not from “randomness”, which has none after the initial shuffling. Instead it comes from “inaccessible environment”.
- Given that the source of randomness is at the beginning, one might consider the following strategy, called **Averaging over clairvoyance**:
  1. Starting: a chance state, with an **edge for each possible deal**.
  2. Each branch is a **regular deterministic game** without randomness.
  3. Value of an action: **average** of values taking that action in each branch. The best action value is chosen.
- **Good** in that at least the computer can play a **reasonable** game. **Bad** in that the strategy is **consistently wrong**.

AI(0270)-4.38

### Problem of averaging over clairvoyance

Why wrong? It assumes we **see everything after the first action**. Let's see an example. Suppose two players hold these cards...

Assume we are playing a 2-persons version of bridge with no “trump” suit, so the one who cannot follow the suit of leader will lose the trick.

1. MAX: ♥6, ♦6, ♣8, 7; MIN: ♥4, ♠4, ♣9 5. MAX to lead.

Best play: MAX can play any card now. One can cash the good hearts and diamonds now, or later after MIN takes one club and one spade.

2. MAX: ♥6, ♦6, ♣8, 7; MIN: ♦4, ♠4, ♣9 5. MAX to lead.

Best play: MAX can play any card now—completely symmetric.

But **what if we don't know which of the two hands we are given?** Can we say “all plays are just as good”?

No! If we play ♣8, taken by MIN, who returns his spade, what to do?!

Should we discard a diamond or a heart? We can only guess!

AI(0270)-4.39

### What we need to do better...

Consider about what **information** we have at each point of the game.

- Try to **communicate** information you have to your partner.  
That is needed in real bridge play, where the cards of the defendants are in two hands, played by two players. E.g., on the first lead, many players agree to play a high card if he want the same suit to be lead again.
- Try to **avoid** releasing information to your opponents.  
E.g., some declarers purposely doesn't give out his least card on a losing trick to void the above signal.
- Try to **mislead** your opponents about your current situation.  
E.g., sometimes good bridge defenders will drop a high card to mislead the declarer to think that he has no more card of that suit.
- Try to play **unpredictably** at times to avoid one's tactics being seen.  
E.g., second trick above won't work if declarer always lead the 2nd lowest card.

But all is **hard to compute**, and no program play at that expertise.

AI(0270)-4.40

### State of the art

How good are current programs?

- **Chess**: “Deep-Blue” won the world champion using huge amount of computational resources and good chess knowledge (i.e., good forward pruning). A normal PC match with human champion is going on.
- **Checker**: “Chinook” is the world champion, using an end-game database of all positions containing only 8 pieces (444 billion configs).  
Checker might get “completely solved”: winning declared at start.
- **Othello**: “Logistello” won human champion (Takeshi Murakami) 6-0. Othello is easy for computers—even if the computer uses just a PC.
- **Backgammon**: “TDGammon” is consistently top 3 at world champion level, using just 4-ply search with a good evaluation function.

AI(0270)-4.41

**State of the art (cont'd)**

- **Go** (a game on a  $19 \times 19$  board to earn largest "region"): computers play at beginner level. (Why? branching factor is over 150...)  
Human plays much better because the game involves "real" reasoning.
- **Bridge**: "GIB" once ranked at 12th in a field of 35 players at championship level. Much better than what many experts expects. It uses averaging over clairvoyance by sampling only 100 arrangements, with each deterministic deal solved optimally.