Outline

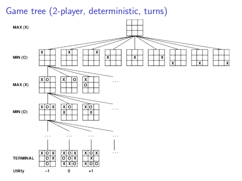
Today I will talk about games, like chess, torpedo, backgammon or poker. We will learn how can we - or our programs - play a perfect game. This includes the minimax and the alpha-beta pruning. We will examine the cases, when we do not have enough time or memory to store the whole game-tree, and how to deal with this. We will examine games where chance has impact on the next step, and the games in which there is some missing information.

Games vs. search problems

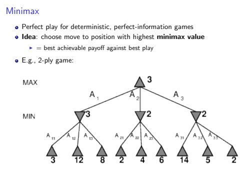
- "Unpredictable" opponent ⇒ solution is a **strategy**
  - specifying a move for every possible opponent reply
- Time limits ⇒ unlikely to find goal, must approximate
- Plan of attack:
  - Computer considers possible lines of play (Babbage, 1846)
  - Algorithm for perfect play (Zermelo, 1912; Von Neumann, 1944)
  - Finite horizon, approximate evaluation (Zuse, 1945; Wiener, 1948; Shannon, 1950)
  - First chess program (Turing, 1951)
  - Machine learning to improve evaluation accuracy (Samuel, 1952–57)
  - Pruning to allow deeper search (McCarthy, 1956)

You must be prepared for all the possible moves from the opponent, i.e. you need a strategy (what the the next step should be at any state). As for most games there is a time limit, it is not possible to calculate every scenario, so you need to decide on partial calculations.

The construction of the strategy has a long history. Even without computers many researchers have been thinking in algorithms.



We have two dimensions: a game may contain some chance and there may be some hidden information in the game. We will discuss all the cases.



Let us discuss this in a more general way. Usually the leaves denote the score of the game, and sometimes they may have other values than [-1,0,1]. The numbers at the leaves of the tree denote the prize of the first player (which is paid by the second player). The second player chooses from the possible moves so that he would have to pay as little as possible.

So from 3, 12 and 8 it selects the minimum value 3 (ans A11 step). In the other two cases he chooses the move that gives payoff 2. When the first player needs makes the first move, he can calculate these numbers, so he needs to choose A1 with 3 to get the best output.

Minimax algorithm

```
function Minimax-Decision(state) returns an action
  state: current state in game

  return a in Actions(state) maximizing Min-Value(Result(a, state))
function Max-Value(state) returns a utility value

  if Terminal-Test(state) then return Utility(state)
  v := -infinity
  for (a, s) in Successors(state) do
    v := Max(v, Min-Value(s))
  return v
function Min-Value(state) returns a utility value

  if Terminal-Test(state) then return Utility(state)
  v := infinity
  for (a, s) in Successors(state) do
    v := Min(v, Max-Value(s))
  return v
```

In the algorithm we have a function which examines all the possible moves of the first player, and selects the one which has the best (maximum) value. For the two players we have two functions. In both cases if we reach a leaf we return its value. Otherwise we take the minimum/maximum of the successor states, which is determined by the other function.

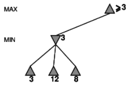
Properties of minimax

- Complete
  - Yes, if tree is finite (chess has specific rules for this)
- Optimal
  - Yes, against an optimal opponent. Otherwise??
- Time complexity
  - $O(b^m)$
- Space complexity
  - $O(bm)$ (depth-first exploration)
- For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games
  - $\Rightarrow$ exact solution completely infeasible
- But do we need to explore every path?

The four properties we examined when looking at search algorithms, we can check again:

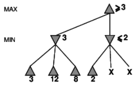
- If the game tree is finite, then this program can run in a finite time, and we are able to determine the best moves. Most of the games have rules to exclude infinite games.
- If the other player plays in an optimal way, we can get the best outcome. If the other player is not optimal, we can earn an even better payoff.
- We need to construct the whole game-tree. The branching factor and the maximal depth determines the exponential time complexity.
- As we use DFS, linear space is enough.

But for a complex game like at chess the branching factor and the depth is so big that we cannot explore the whole tree.



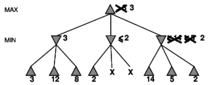
Let us see the previous game tree. If the first player calculates the effect of the first alternative, then he can realize, that he can earn at least 3 at payoff (if the other steps are better, then even more.)

α–β pruning example



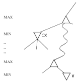
The very next alternative (2) means, that the opponent can reach at most 2, or otherwise less, if there is a leaf with a smaller payoff. Hence the first player will not select this alternative, he has a better one as discussed above. Therefore the numbers at X are not interesting, we can omit them (prune) totally.

α–β pruning example



The third alternative at first promises a value 14 (or less, if there are fewer numbers here). So we need continue the search. The next number is five. The opponent will select 5 here and not 14, so the value of the third alternative (for the first player) is 5 or less. As the last number here 2, so the value of the third alternative for player 1 is 2, therefore the value of the root is 3.

Why is it called α–β?



- α is the best value (to *MAX*) found so far off the current path
- If *V* is worse than α, *MAX* will avoid it
  - $\Rightarrow$ prune that branch
- Define β similarly for *MIN*

We need to handle two numbers, alpha and beta. alpha can change at Max nodes (nodes at Max level), and beta at Min level. They denote the value of the best alternatives of the players. If for the actual Max node we found a successor whose value is less than alpha, then it is not interesting, we can go back.

The α–β algorithm

```
function Alpha-Beta-Decision(state) returns an action
  return a in Actions(state) maximizing
             Min-Value(Result(a, state), -infinity, infinity)

function Max-Value(state, alpha, beta) returns a utility value
  state: current state in game
  alpha: the value of the best alternative for MAX along the path to state
  beta:  the value of the best alternative for MIN along the path to state

  if Terminal-Test(state) then return Utility(state)
  v:= -infinity
  for (a, s) in Successors(state) do
    v := Max(v, Min-Value(s, alpha, beta))
    if v >= beta then return v
    alpha := Max(alpha, v)
  return v
```

function `Min-Value(state, alpha, beta)` same as `Max-Value` but with roles of alpha and beta reversed

We start with alpha set to $-\infty$, and beta set to ∞. We check all the successors of the root.

At terminal nodes (leaves) we send back the value assigned to the node. Otherwise we take the values of the successors, and the value of the actual nodes will be the maximum of the known such values. If it possible we update (raise) the value of alpha. As the previous slides demonstrated, if the value of the node is bigger than beta, we need to escape immediately.

Properties of α–β

- Pruning *does not* affect final result
- Good move ordering improves effectiveness of pruning
- With "perfect ordering," time complexity = $O(b^{m/2})$}
  - $\Rightarrow$ *doubles* solvable depth
- A simple example of the value of reasoning about which computations are relevant (a form of **metareasoning**)
- Unfortunately, 35^{50} is still impossible!

The most important property is that we do not delete any important nodes: the result with and without deletion has to be the same.

We can increase the number of deleted nodes. For this we need to order the moves carefully. We can half the exponent in the complexity. So if we have a fixed time to search, we can double the height of the (visited) search tree. This means we get a lot without any extra work, simply based on logical reasoning. But the number of remaining nodes is still huge.

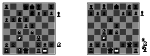
Resource limits

- Standard approach:
  - Use *Cutoff-Test* instead of *Terminal-Test*
  - e.g., depth limit (perhaps add **quiescence search**)
- Use *Eval* instead of *Utility*
  - i.e., **evaluation function** that estimates desirability of position
- Suppose we have 100 seconds, explore 10^4 nodes/second
  - $\Rightarrow 10^6$ nodes per move $\approx 35^{8/2}$
  - $\Rightarrow \alpha$–β reaches depth 8 $\Rightarrow$ pretty good chess program

We cannot visit the whole search tree, so lets cut off its upper part and focus on that instead. Instead of testing terminality we can check the cut property, e.g. we can add a depth limit (cut at a given depth level), or cut, when the values of the nodes doesnt change too much. So we have no leaves – i.e. final states – just inner states. Here it is unclear who is the winner, or more precisely who can be a winner later. Therefore we will use an evaluation function, which estimates the desirability of the state/position.

Could this help? A small calculation shows if we have slower computer, within 100 second it can search the game tree of the chess 8 levels deep, which the state of the art of the eighties.

Evaluation functions



Black to move
White slightly better

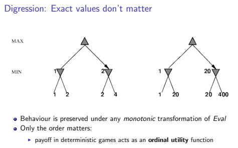
White to move
Black winning

For chess, typically **linear** weighted sum of **features**

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

e.g., $w_1 = 9$ with $f_1(s) =$ (number of white queens) – (number of black queens), etc.

If you are not a novice in chess, you may know that a queen equals 9 pawns, a rook equals 5 pawns, etc. (chess piece relative value: `https://en.wikipedia.org/wiki/Chess_piece_relative_value`)
We can add to these values the value of the right to move and their positional advantages. Usually we assign a weight to such properties, and the evaluation function (`https://en.wikipedia.org/wiki/Evaluation_function`) is their sum. The minimax and the alpha-beta pruning will use these values when the cut-test holds, and we cannot go further.



If we use a monotone transformation, we get back the same result, the Maxs step will be the same.

Deterministic games in practice

Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.

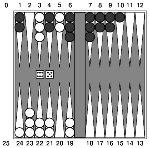
Chess: Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.

Othello: human champions refuse to compete against computers, who are too good.

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. (*from 2004*)

In the nineties, computers became fast enough to be good opponents in games. Usually the programmers included databases of openings and endgames. For chess the Deep Blue was a computer-monster. Fifteen years ago there was no hope to write good Go programs. But four years ago, *deep learning* (`https://en.wikipedia.org/wiki/Deep_learning`) became good enough.

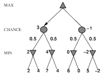
Nondeterministic games: backgammon



In backgammon (`https://en.wikipedia.org/wiki/Backgammon`) we have to throw dice and they determine the possible moves.

Nondeterministic games in general

- In nondeterministic games, chance introduced by dice, card-shuffling
- Simplified example with coin-flipping:



In these games chance has a role: coin, dice, card-shuffling. We can treat chance as a player, it has its own moves. Here, after the first player makes a move, we toss a coin, and based on that result the second player may make his move too. We can see the values of the leaves of the game tree, and we can calculate the previous nodes. For the nodes that correspond to chance, we need to take into account the probability and the value of the successor nodes.

Algorithm for nondeterministic games

- *Expectiminimax* gives perfect play
- Just like *Minimax*, except we must also handle chance nodes:

```
...
if state is a MAX node then
   return the highest ExpectiMinimax-Value of Successors(state)
if state is a MIN node then
   return the lowest ExpectiMinimax-Value of Successors(state)
if state is a chance node then
   return average of ExpectiMinimax-Value of Successors}(state)
...
```

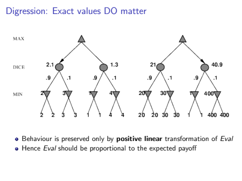
We need to multiply the values and probabilities (which gives the expected value) to get the value of a node that corresponds to chance. This is the only modification of the minimax method.

Nondeterministic games in practice

- Dice rolls increase b: 21 possible rolls with 2 dice
  - Backgammon $\approx$ 20 legal moves (can be 6,000 with 1-1 roll)
  - $$\text{depth } 4 = 20 \times (21 \times 20)^3 \approx 1.2 \times 10^9$$
- As depth increases, probability of reaching a given node shrinks
  - $\Rightarrow$ value of lookahead is diminished
- α–β pruning is much less effective
  - *TDGammon* uses depth-2 search + very good *Eval*
  - $\approx$ world-champion level

Let us take backgammon! We have 2 dice (36 possibilities), but the order does not matter (-15 possibilities). As in general around 20 moves are possible, we get huge numbers, even if the depth is just 4. So we have a bulky tree, hence the probability of a given path is extremely small. There is no reason to search in depth. As we need to calculate with every possible events, we cannot prune too much.

The best program of the last century made a shallow search, but used a very good evaluation function. The program used artificial neural network to construct this function, and played millions of matches against itself, to improve it.

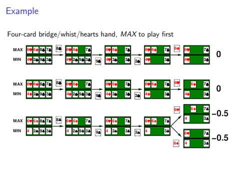


If we calculate the values of two similar trees, where the transition is monotone, we get different answers. So we need a linear transformation, and at the cutoff we need to use the payoffs.

Games of imperfect information

- E.g., card games, where opponent's initial cards are unknown
- Typically we can calculate a probability for each possible deal
- Seems just like having one big dice roll at the beginning of the game
- **Idea**: compute the minimax value of each action in each deal then choose the action with highest expected value over all deals
- Special case: if an action is optimal for all deals, it's optimal.
- GIB, current best bridge program, approximates this idea by
  1. generating 100 deals consistent with bidding information
  2. picking the action that wins most tricks on average

At a typical card game you dont know the cards of the opponents and the order in the deck. But you can calculate the possibility that your opponent has two pairs in poker at the beginning. There are too many possible deals, we cannot take into account all of them. Hence typically we generate a sample which is suited to yours cards (and any other visible cards), and test all kind of steps on this sample. Then we choose the step with the highest payoff. GIB (`https://en.wikipedia.org/wiki/Computer_bridge`) used the same approach around 2000.



Let us see a simple card game where the players see all the cards and must follow the suit (if possible). Here the payoff is zero. If the second player has $\diamondsuit 4$ instead of $\heartsuit 4$, the payoff is the same. If the first player does not know that the opponent has $\diamondsuit 4$ or $\heartsuit 4$, it cannot choose the right card in the last step, so in both cases the average payoff is -0.5

Commonsense example

- Road A leads to a small heap of gold pieces
- Road B leads to a fork:
  - take the left fork and you'll find a mound of jewels;
  - take the right fork and you'll be run over by a bus.
- Road A leads to a small heap of gold pieces
- Road B leads to a fork:
  - take the left fork and you'll be run over by a bus;
  - take the right fork and you'll find a mound of jewels.
- Road A leads to a small heap of gold pieces
- Road B leads to a fork:
  - guess correctly and you'll find a mound of jewels;
  - guess incorrectly and you'll be run over by a bus.

If you dont like card games, we can give the same problem using different terminology.

Proper analysis

- Intuition that the value of an action is the average of its values
  - in all actual states is *WRONG*
- With partial observability, value of an action depends on the **information state** or **belief state** the agent is in
- Can generate and search a tree of information states
- Leads to rational behaviors such as
  - Acting to obtain information
  - Signalling to one's partner
  - Acting randomly to minimize information disclosure

Our intuition is wrong. If you have missing information you decide on your information/belief state. We need to work with beliefs, and based on this we can construct a game-tree. To write a good opponent we need to acquire as much information as possible, and confuse the opponents.

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
  - optimal decisions depend on information state, not real state
- Games are to AI as grand prix racing is to automobile design

Everybody likes to play games, many final thesis were written about AI methods in concrete games. Here we can tests new ideas, and the limits enforce the construction of new methods. Most of the games have a contest: comparing the programs.