

**Backgammon: a game with chance**

**Lecture 7**

**More about games**

In this lecture we will continue to look into the design of game-playing programs, with a focus on games with choices and alternative game-playing strategies.

**Reference:**

- Textbook Chapter 5

- **Backgammon** is a classic example of a game that have an element of **chance**, in the form of 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 die, and *then* advance one man (maybe the same, maybe different) by the number on the top of another die.
- Except when the player get double, when he makes four moves.

AI(0270)

AI(0270)-7.1

**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**.

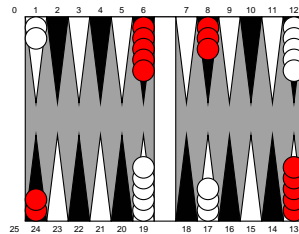
- A player leaves some moves (dice) unused if and only if there is **no possible valid move**.

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.

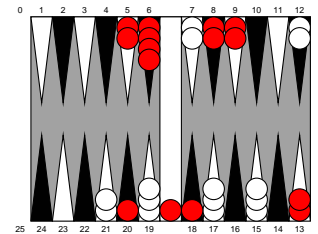
- And there are other subtleties on end-game that we won't explain.

AI(0270)-7.2

**Some game positions**



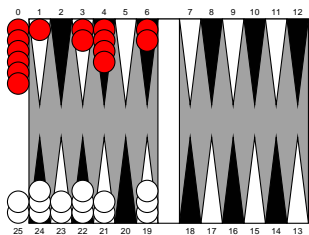
Initial position. White to move to position 25, red to position 0.



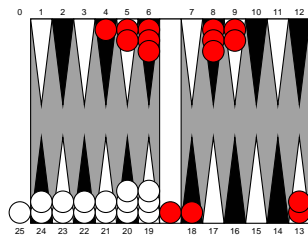
Middle of game, white better a bit: it trapped a red man in bar, moved both men at position 1 out of first quadrant, and leave no open man.

AI(0270)-7.3

**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.

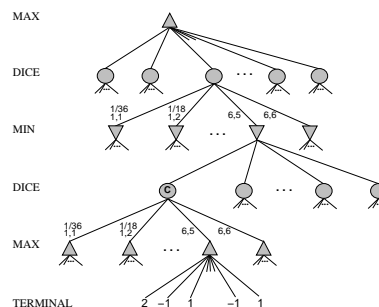
**If you don't know the game, it is perfectly okay.**

The only reason to talk so much about the game is to convince you that it needs a lot of skills.

AI(0270)-7.4

**A State Space with chance states**

Now it is time to think about how to draw a state space:



There are some **chance states** in the SS, destined as DICE. Nobody control them, and each outcome is taken at certain probability.

Under such situation, 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)-7.5

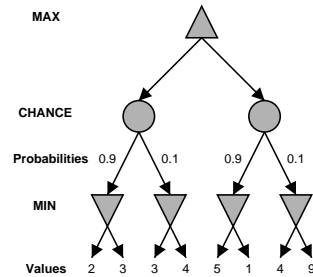
### State space evaluation with chance states

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 dispersion 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)-7.6

### Example



The values of the MIN nodes are 2, 3, 1 and 4 respectively.

The values of the CHANCE nodes 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 node is thus 2.1, choosing the left branch.

AI(0270)-7.7

### Code

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

```
double ExpectMinimax(State s, int &op_return) {
    if (Terminated(s)) return Value(s);
    int best = -1, dummy;
    double bestval = is_max(s) ? -HUGE_VAL : is_min(s) ? HUGE_VAL : 0;
    for (int i = 0; i < num_ops; ++i) {
        State next = s;
        apply(next, i); // apply operator i to the next state
        double value = ExpectMinimax(next, dummy);
        if ((max_to_move(s) && value > bestval) ||
            (min_to_move(s) && value < bestval)) {
            bestval = value;
            op_return = i;
        } else if (chance_move(s))
            bestval += value * ChanceOf(i);
    }
    return bestval;
}
```

AI(0270)-7.8

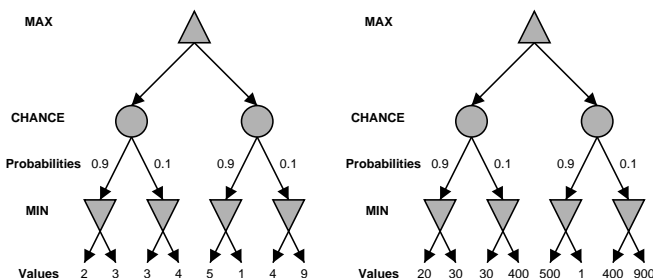
### The importance of exact utility values

- 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**.
- The problem is of course that **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.
- Example: 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.

AI(0270)-7.9

### 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 correct play on the new SS is the **right** branch.

While the chance is slim, the payoff is large, making the expected value large.

AI(0270)-7.10

### How difficult is it?

Okay, so fast is Expectiminimax?

- Let's suppose that **every chance node 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  $(bn)^d$  **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 nodes, 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)-7.11

### Adapting alpha-beta: possible?

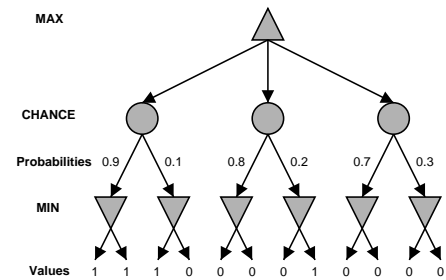
If Expectiminimax is so hard, is it possible to prune it?

- It is much more difficult: at a chance state, **you must evaluate all branches to find the value of the state.**
- It turns out that it is **possible**—given an upper or lower bound on value of the terminal states of the tree.
- Given such a bound, we can in turn **bound the value of the intermediate states** without examining the whole sub-tree.
- Using the same idea as Alpha-Beta, we can prune part of the tree.
- But pruning is somewhat more complicated: **have to find bounds** for states during the search.

AI(0270)-7.12

### Example

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



Here, the second branch of the second and third chance states (with probability 0.2 and 0.3) needs not be expanded: the subtrees won't be chosen. Exercise: What will happen if the bound of utility is between 0 and 3?

AI(0270)-7.13

### Other ways to improve the agent

- **Book move:** when the game starts or is about to terminate, the number of possibilities is usually small, so it is possible keep an **opening** or **end-game database** storing all the best moves.
- **Selective search** (heuristic pruning): don't even go into nodes that looks too bad (e.g., make a queen to be lost uncompensated, etc.)  
But this strategy can be risky: perhaps for the opponent to capture the queen, it has to lose the king. This cannot be known until we actually search the sub-tree.
- If things goes really bad (will lose in any way), **allow for mistakes made by the opponent.**  
More effective against human player than against another computer player.
- Use **special hardware** to evaluate states very quickly.  
E.g., the current chess champion (Deep Blue) uses 256 specially designed VLSI circuits to perform around 100 billion evaluations per second.

AI(0270)-7.14

### State of the art

How good are current programs?

- **Chess:** using huge amount of computational resources and good chess knowledge (i.e., good selective search), the machine of IBM consistently won the human champion.
- **Checker:** "Chinook" plays at world champion level, using alpha-beta with an end-game database of all positions containing only 6 pieces.
- **Othello:** human players fails to compete with any serious agent playing the game—even if the agent uses just a PC.
- **Backgammon:** computers play at world champion level, using just 4-ply search with a good evaluation function ("TDGammon").
- **Go** (a game on a  $19 \times 19$  board to get 5 pieces on the same line): computers play at beginner level and fails to defeat consistently with human players. (Why? branching factor is over 150!)

AI(0270)-7.15

### Meta-reasoning

- Alpha-beta is designed to **find the value** of a state.
- In many situation, the **value is irrelevant.** E.g., when there is only 1 move, or when there are many moves but all moves but one are too bad to consider.  
Alpha-beta will search in the only good subtree to find the value it provides, wasting a lot of time.
- This gives rise to the idea of **the value of information** we gain by searching: for some states, searching in it gains very little information.
- One possibility: to **search only if the search cost can be covered by the value of information.**
- To do so, one would need to **have an estimate** of the search cost (easy) and the **value of information** we get (much more difficult).

AI(0270)-7.16

### Conclusion

- The game tree and minimax formalism appear very early in AI history. This allows very little room for alternative directions.
- We cannot perform a full search due to large state space. So we have to use approximations.
- Alpha-beta reduces the number of states to evaluate, by declaring early that some states will not be chosen and hence the value does not matter.
- The environments we study are very easy to deal with: full knowledge, (usually) deterministic action, discrete nature. This makes game programs very different from those in any other AI fields.

AI(0270)-7.17