CE213 Artificial Intelligence – Lectures 5&6

Game Playing

Game playing is an excellent example of problem solving by state space search.

It is more complex than general problem solving due to
1) the competition from the opponent;
2) large size of search space;
3) lack of accurate heuristics or evaluation functions.

Game Playing

We will be concerned with games that, like chess or Go, have the following characteristics:

- **Two players**
- **Turn-taking**
- **Deterministic**

  The same move in the same situation always has the same effect.
- **Perfect information**

  The entire game state is always available to both players.

Other examples include: Draughts (Checkers in US), Noughts and Crosses (Tic-Tac-Toe in US). Games like poker and bridge are not considered here.

Game Playing (2)

Choosing the best move in such games has much in common with solving problems by state-space search:

- **Positions** in the game (game states) are **states** in the space of all possible positions.
- The starting arrangement is the **initial state** (e.g., empty board).
- Winning positions or winning endgames are **goal states**.
- Legal moves are the possible **transitions/operations**.

However, there is one big difference between the search strategies for game playing and for general problem solving.

Adversarial Search – Challenge 1

The opponent

- Normally has very different goals.
- Selects every other move.
- Will try to stop us reaching our goal.

This form of state space search taking the opponent into account is called ***adversarial search***.

There is usually another challenge in developing search strategies for game playing.

Large Search Space – Challenge 2

The puzzles and navigation problems we considered when discussing state space search had small state spaces:

Road map problem	11 distinct states (11 towns)
Corn goose fox problem	16 distinct states
Three jugs problem	24 distinct states (for the one we discussed)
(Eight puzzle	9! distinct states)
(Fifteen puzzle	16! distinct states)

Non-trivial games have vastly larger state spaces:

Chess has about 10^{40} distinct states.

Go has 3^{361} distinct states, and $2.08168199382 \times 10^{170}$ legal game positions.

Even noughts and crosses has 3^9=19683 distinct states (although many of these are illegal, in which '|no. of noughts –no. of crosses|≤1 ' is not true.)

Solutions

Adversarial search: Minimax search

Large state spaces: Evaluation functions (Heuristics)

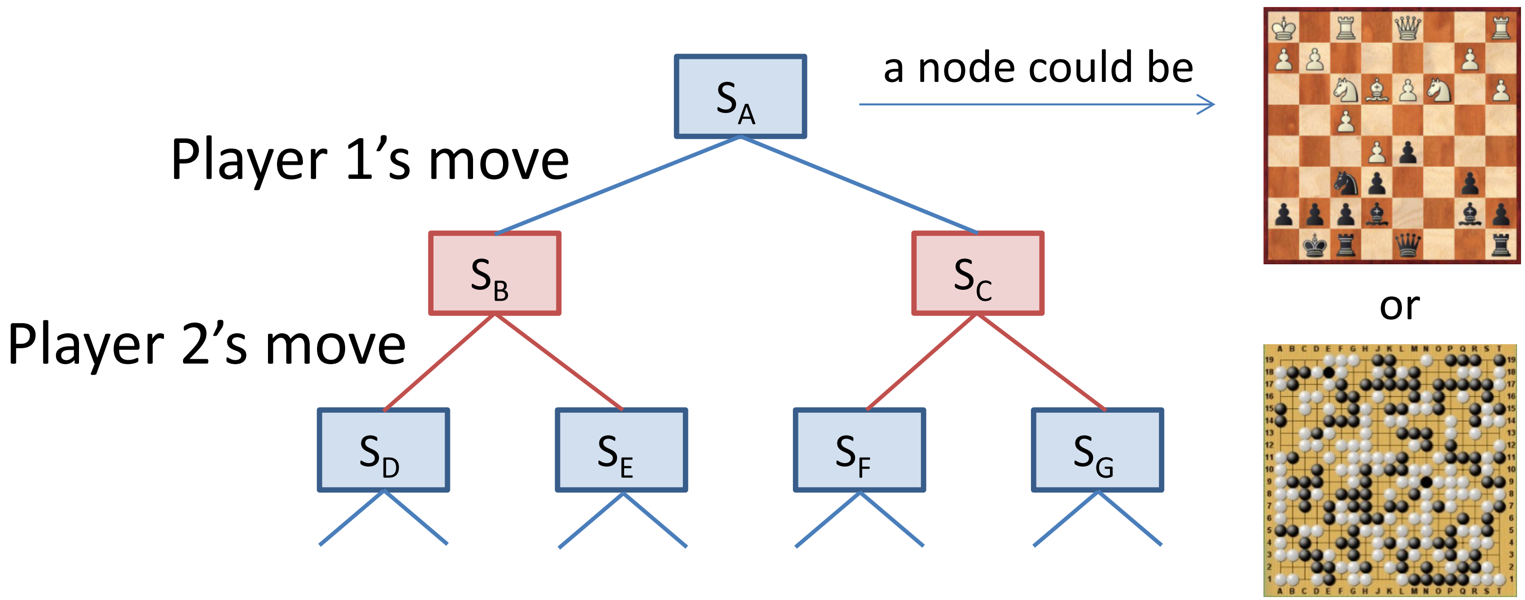
Challenge 2 is mainly about how to evaluate game states

(In this lecture we mainly address Challenge 1)

Effective and efficient approach to "generate and evaluate" is a key AI research topic.

Game Tree – Search Tree for Game Playing

A simplest game tree for the first two moves of a possible game:



There are two types of moves that aim at different goals!

(For convenience, let's assume Player 1 is AI and Player 2 is Opponent)

Minimax Search

If the opponent could be trusted to help us, then the problem would be easy:

An exhaustive search to the depth limit would reveal the best move.

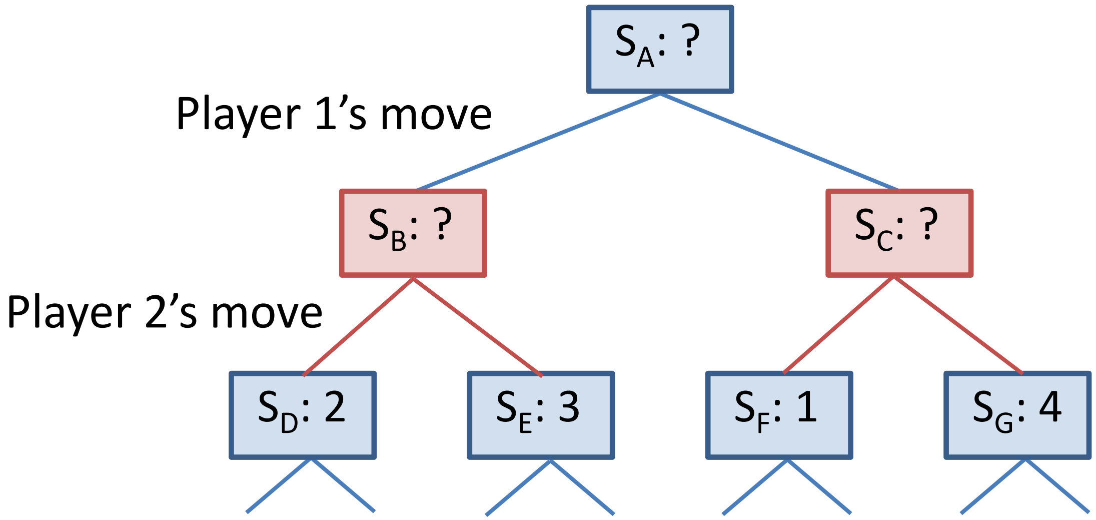
But in fact we expect the opponent to *hinder* us.

How can we take account of this while carrying out the search?

Different from the search strategies learnt so far, in minimax search we need to check the nodes at deeper levels before choosing which move to make or which node to expand at the current depth in game tree construction.

Minimax Search (2)

It is well-known that it is easier to evaluate game states deeper in the game tree as they are closer to endgame. So, suppose that an evaluation function has given the following values to the nodes at depth 2, as shown in the game tree, but it is hard to evaluate the values of the nodes at depth 1 directly. We need to get the values of states S_A, S_B, and S_C by minimax search.



Note that all the values are to Player 1 (the larger the better to Player 1).

(You may ignore how these values are generated at the moment. They are game-dependent. Here, these values are hypothetical only.)

Minimax Search (3)

Player 2 will choose a move so as to ***minimise*** the value to Player 1.

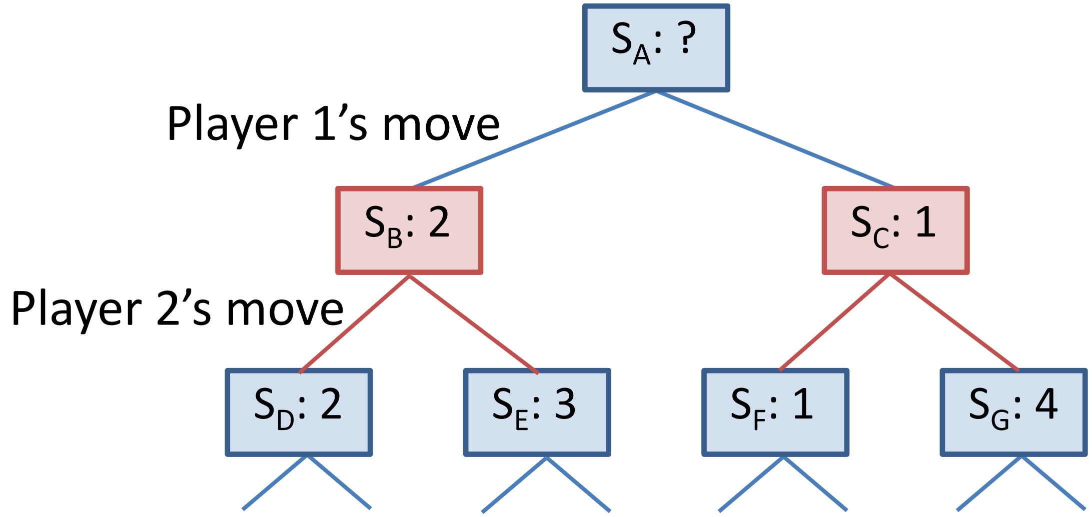
In game state S_B, Player 2 will choose to go to S_D.

Thus the potential value of S_B to Player 1 is only 2.

In game state S_C, Player 2 will choose to go to S_F.

Thus the potential value of S_C to Player 1 is only 1.

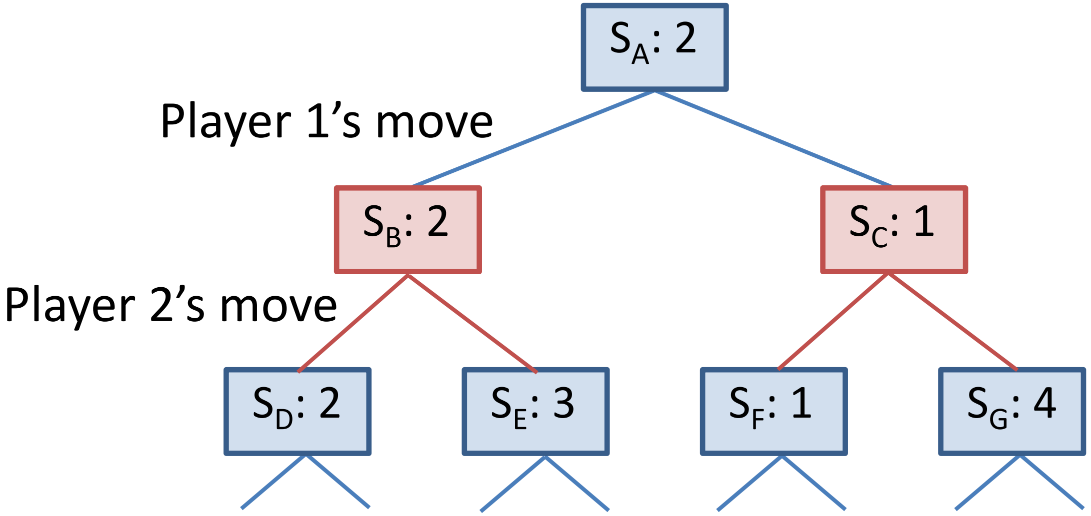
So we can include these values of S_B and S_C in the search tree:



Minimax Search (4)

Player 1 should choose a move so as to ***maximise*** the potential value of the chosen game state. Clearly in this case Player 1 should choose to go to S_B.

Thus the potential value of S_A to Player 1 is 2, as shown in the game tree now.



Minimax Search (5)

This process of passing the evaluation values of game states (nodes) back up the tree is called ***minimaxing,*** or ***minimax*** search:

For nodes where opponent makes the move, pass back the minimum value.

For nodes where AI player makes the move, pass back the maximum value.

Minimaxing can be applied to any depth, with iterations of minimax search for every 2 depths. The deeper, the better, because the evaluation of deeper nodes is more accurate in general in game playing.

Note that minimaxing places a lower bound on how badly the AI player will do, because it is assumed that the opponent would make perfect moves – the worst case scenario.

If the opponent departs from minimaxing, the AI player may do better. Therefore, AI player is guaranteed not to do worse.

Minimax Search (6)

Why not just apply the evaluation function to the directly reachable game states?

Doing this would be fine if the evaluation function is perfect.

In practice it will only give an approximate indication of how good a game state is. The evaluation would be more accurate in more depth where the game states are closer to endgame.

	?	

Which of the 9 possible first moves in Tic-Tac-Toe is better?

o	o	x
x	x	o
		x

o	o	x
x		o
x		x

Which of these 2 game positions is better for Player ‘x’?
(see next slide)

How deep should the search go?

How much time have you got? The deeper the better!

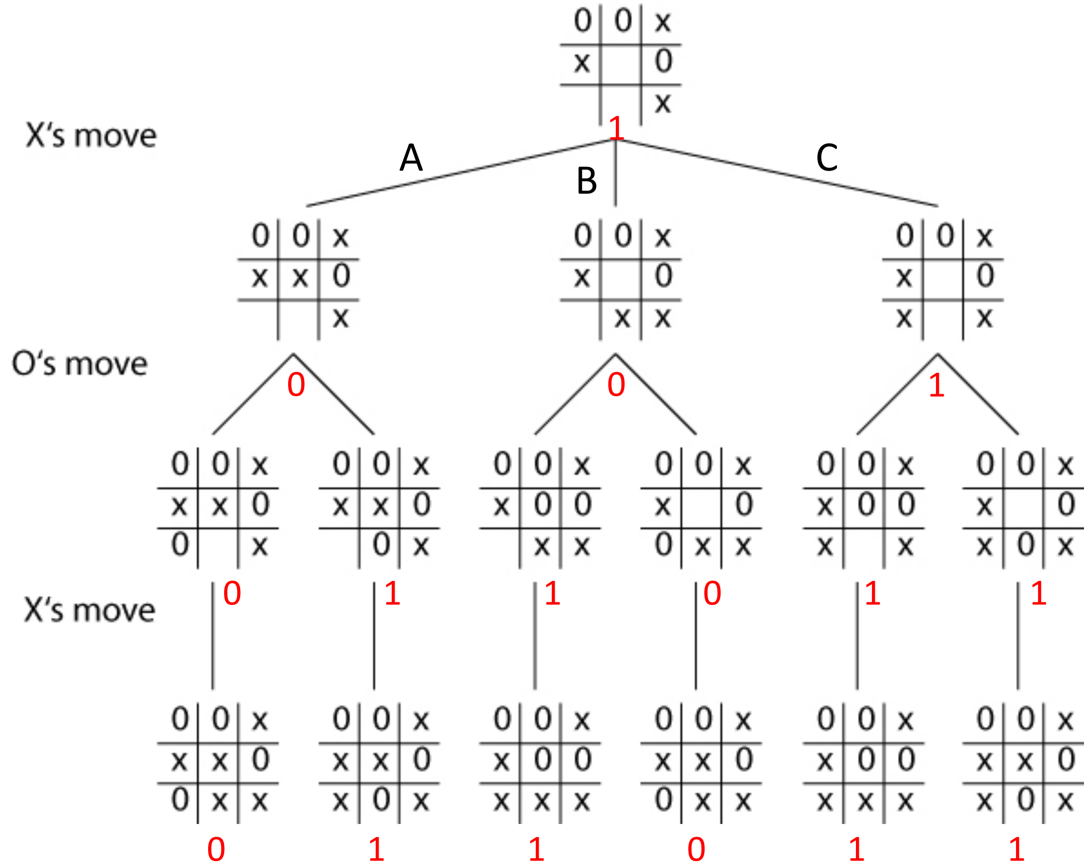
(It is easy to understand this depth requirement if you play chess.)

An Example of Minimax Search

There are 3 possible moves for player X (player O is the opponent here). Which move is the best?

We can expand nodes to reach endgame and evaluate endgame states: 1 for win, 0 for draw, and -1 for loss.

Then conduct the minimaxing procedure: maximising when it is X's move; minimising when it is O's move.



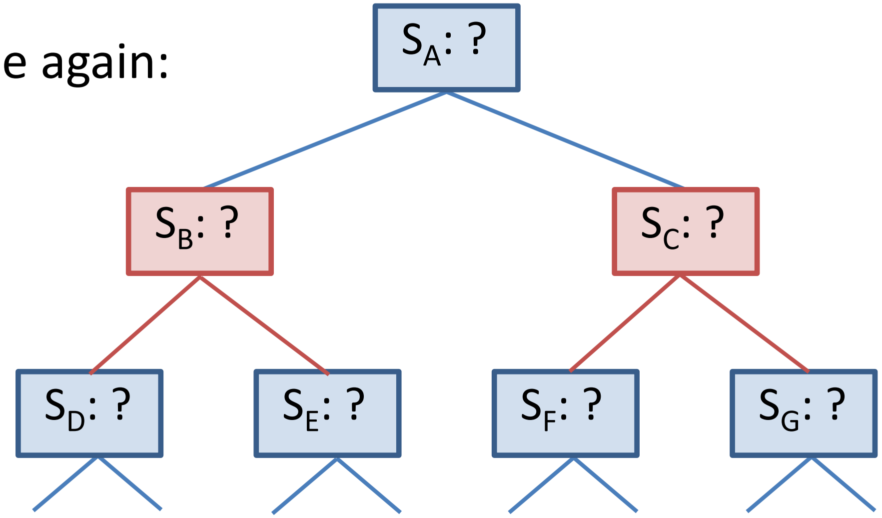
You may try to build up a game tree by starting with an empty board. How many nodes would be there in such a tree? $\sum_{n=0}^{9} \frac{9!}{(9-n)!}$

Some nodes may represent same game states!

alpha-beta Pruning
(how to avoid unnecessary evaluation?)

For many practical games, there are a large number of game states, and minimax search may be too slow in finding best moves. However, it is not always necessary to consider every node in the game tree. Taking this into account, the alpha-beta pruning algorithm can speed up minimax search.

Consider this simple game tree again:



Assume we will work from left to right.

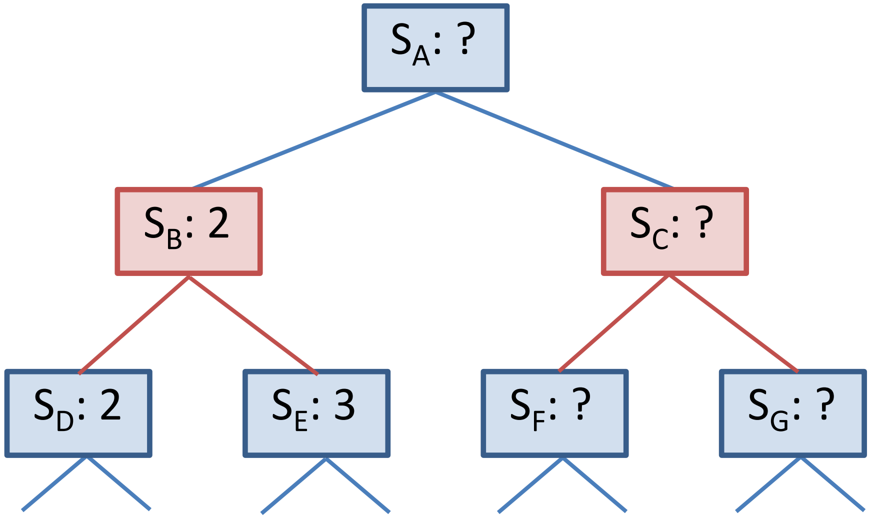
Which nodes at the bottom (at depth 2) need to be evaluated in order to get values returned to nodes S_B and S_C?

alpha-beta Pruning (2)

First we back up the appropriate value to node S_B by **minimising**.

This means we must evaluate nodes S_D and S_E.

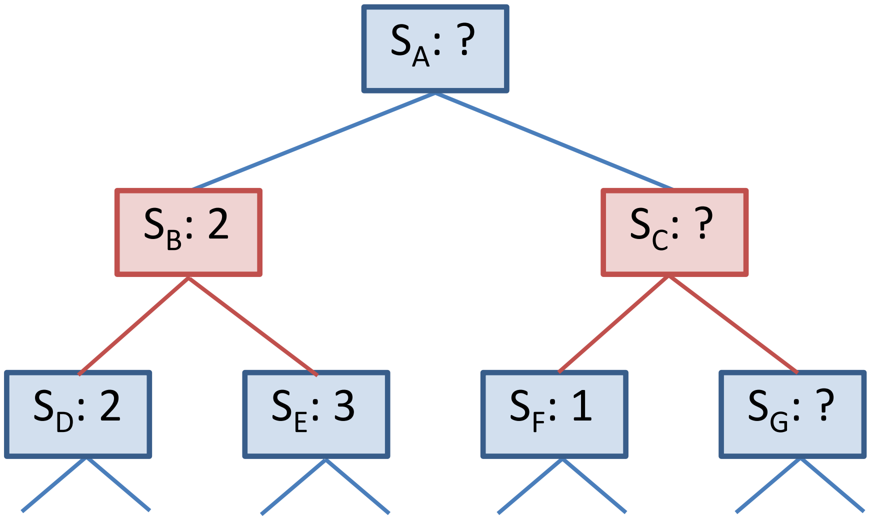
Let us **assume** this leads to the situation, as shown in the following figure. So, the value returned to S_B is 2.



N.B. Whenever possible, return min or max value back to parent node before evaluating new nodes at the deeper level.

alpha-beta Pruning (3)

Next we consider node S_C, which depends on the values of nodes S_F and S_G. Suppose node S_F has a value of 1:



N.B. The value of S_F is compared to the value of S_B (not the values of the nodes at the same depth).

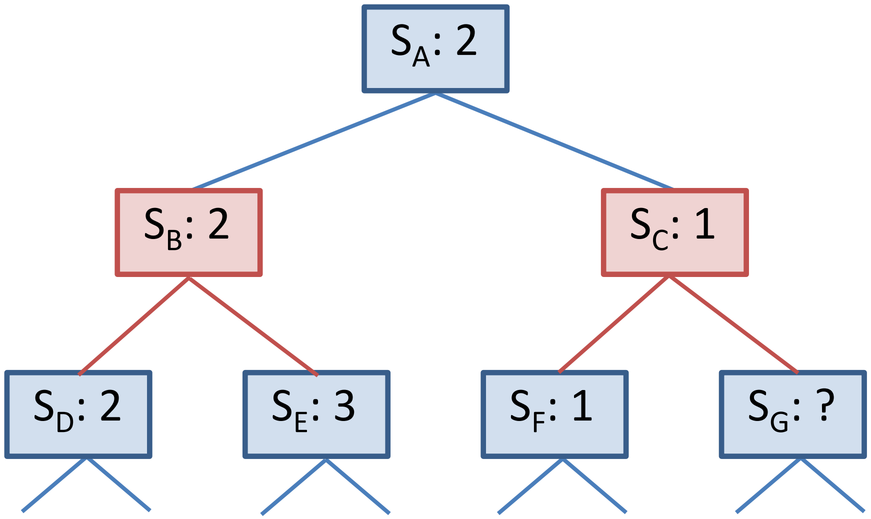
Since we are minimizing, we now know that the value of S_C is equal to or smaller than 1.

This means that S_B (with value of 2) is bound to be a better choice when we come to maximise for the value of S_A.

So we do not need to know the value of S_G.

alpha-beta Pruning (4)

Hence, we can get the value of S_A without evaluating S_G.



This process of skipping unnecessary evaluations is called ***alpha-beta pruning***. (alpha, beta correspond to minimising and maximising processes)

It can be performed at every level of the tree and thus may save considerable time, especially when the branching factor is large.

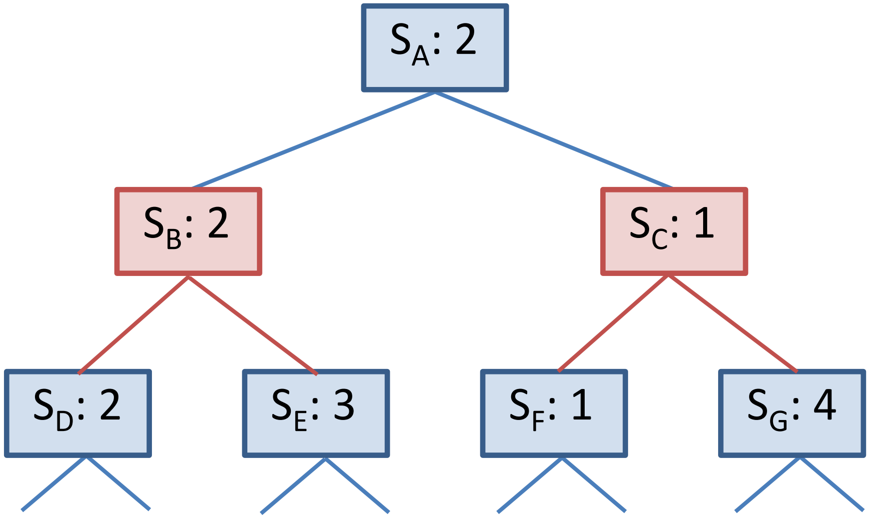
alpha-beta Pruning (5)

How much effort does alpha-beta pruning save?

It depends upon the order in which the nodes are considered.

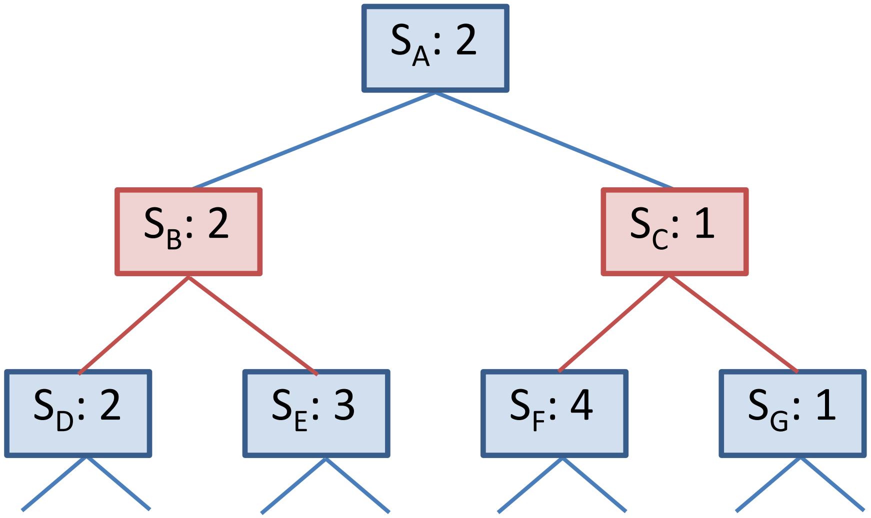
Suppose that S_G actually has a value of 4.

As we have seen, in this case there would be no need to evaluate it.



alpha-beta Pruning (6)

However, if the values of S_F and S_G are transposed, then there would be no saving through alpha-beta pruning in this simple case, because both S_F and S_G would have to be evaluated in order to return the correct value to S_C (Because the value of S_F is larger than the value of S_B, it cannot be returned to S_C without knowing the value of S_G).

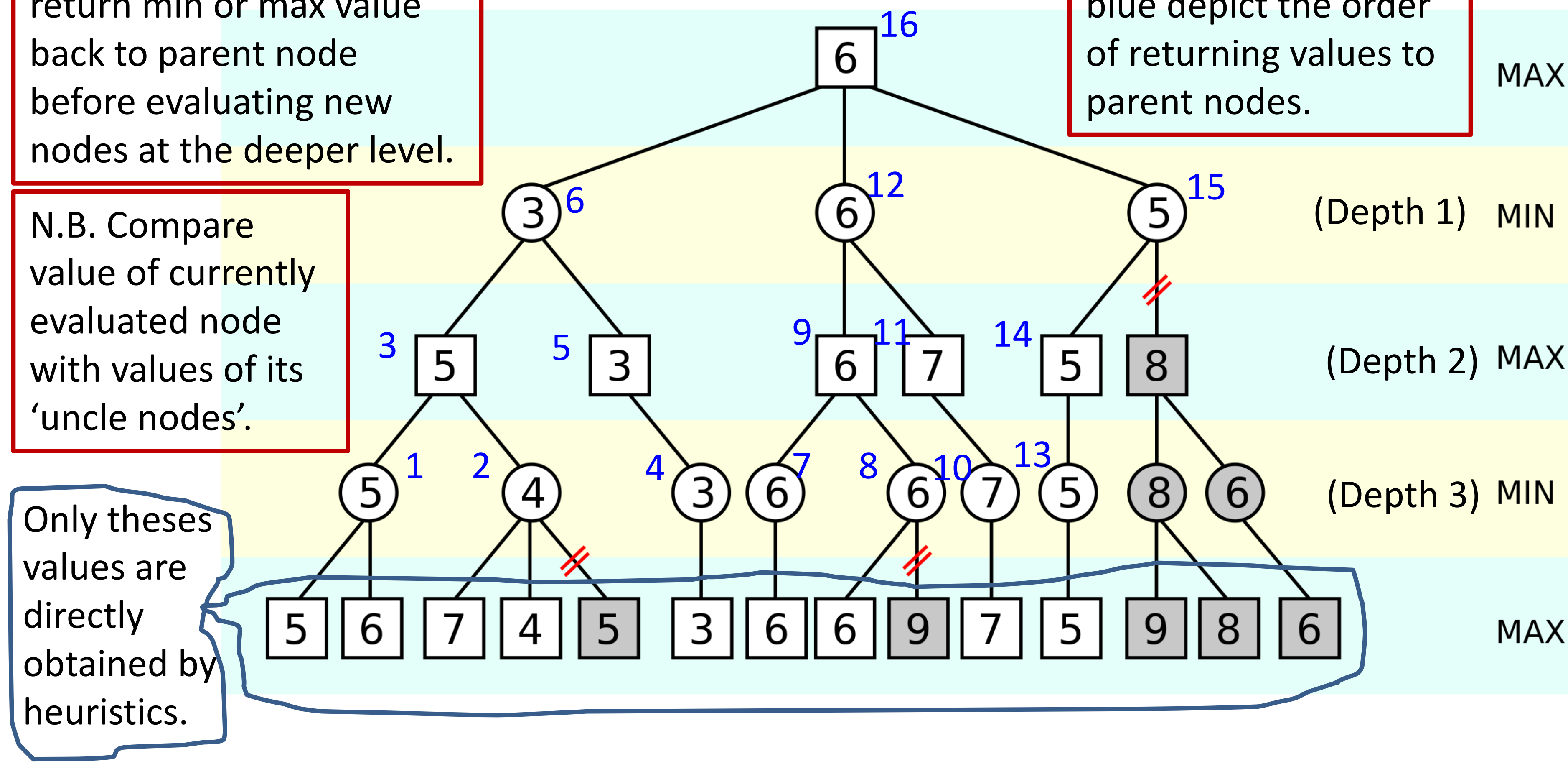


An Example of Minimax Search with alpha-beta Pruning

N.B. Whenever possible, return min or max value back to parent node before evaluating new nodes at the deeper level.

N.B. Compare value of currently evaluated node with values of its 'uncle nodes'.

N.B. The numbers in blue depict the order of returning values to parent nodes.



The effectiveness and efficiency of minimax search with alpha-beta pruning heavily depend on how good the game state evaluations are. Monte-Carlo tree search provides game state evaluation by running simulated games.

Monte-Carlo Tree Search (MCTS) – Basic Ideas

- Updating values of states by running simulated games

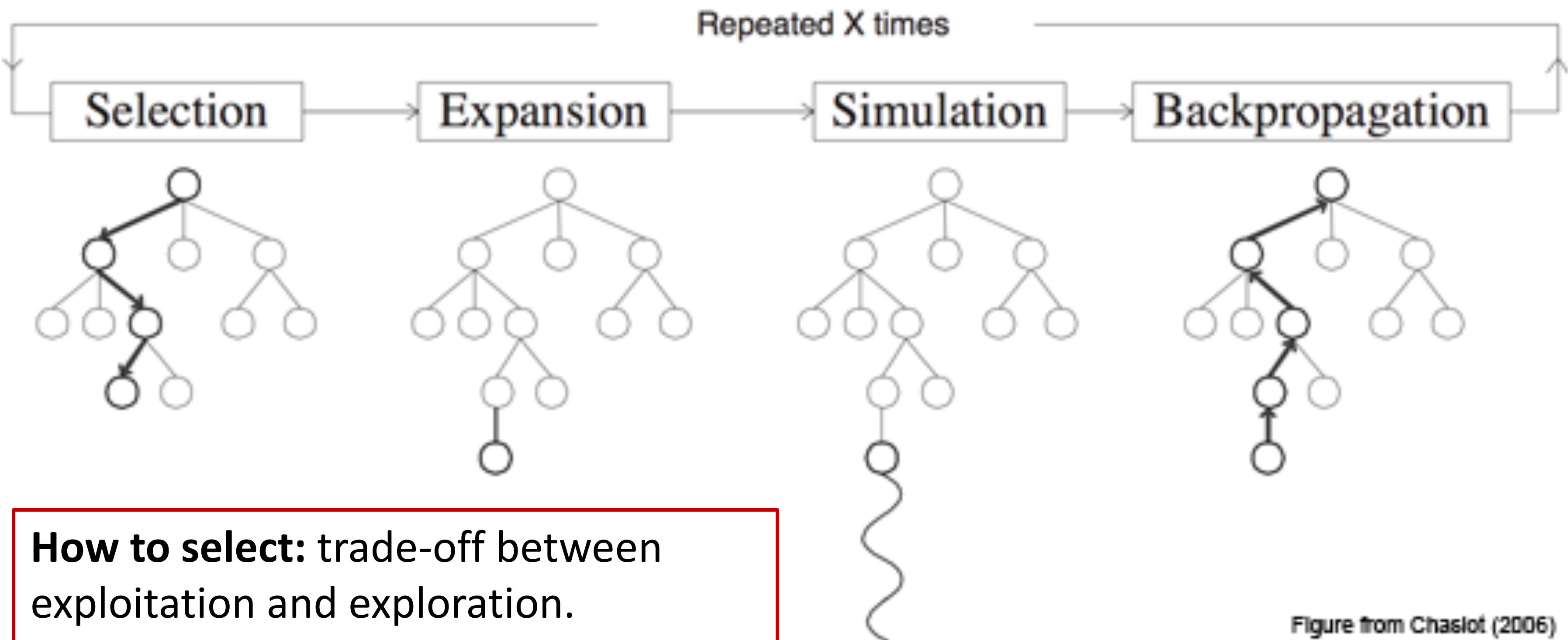


Figure from Chaslot (2006)

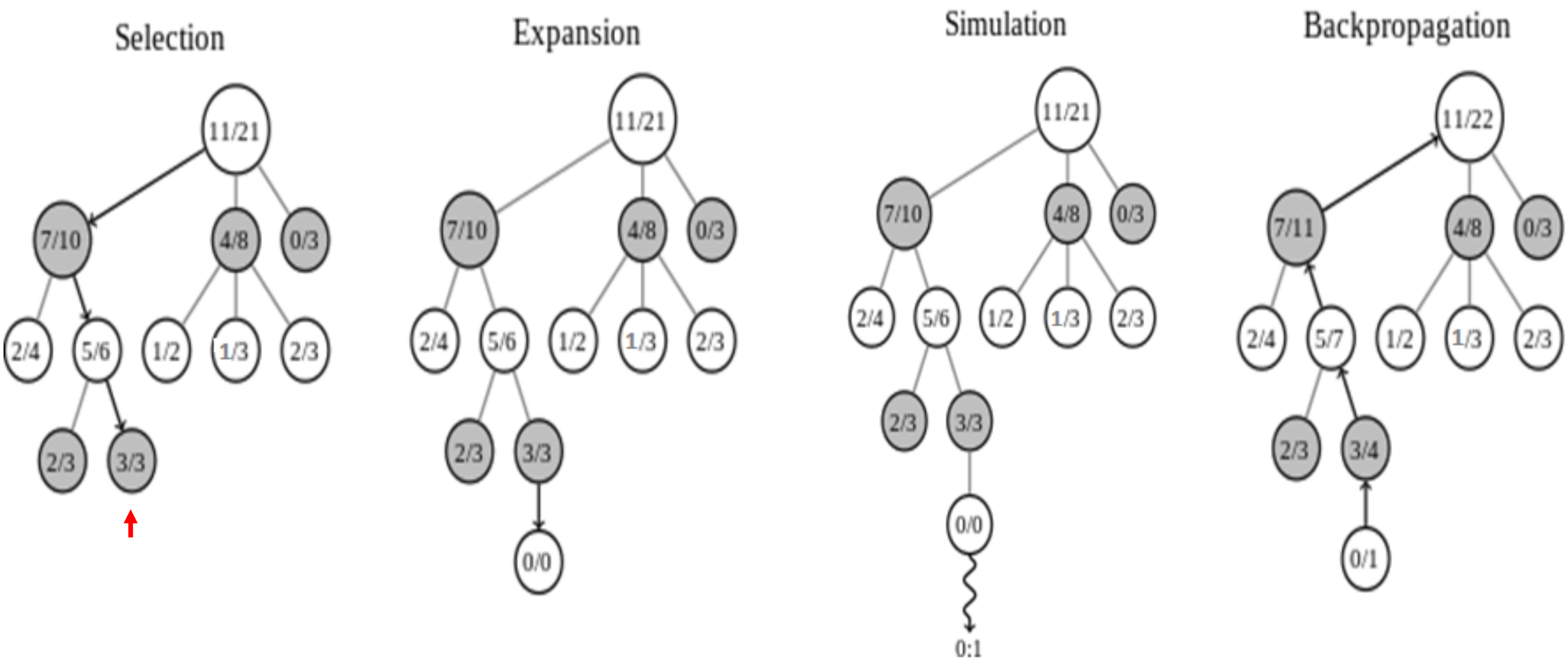
How to select: trade-off between exploitation and exploration.

How to expand: possible new legal moves.

How to simulate: randomly making moves from the selected node to end of game, resulting in win, loss or draw.

How to back-propagate: updating number of wins and number of visits for each node that has been visited in the simulated game.

An Example of a Round of MCTS



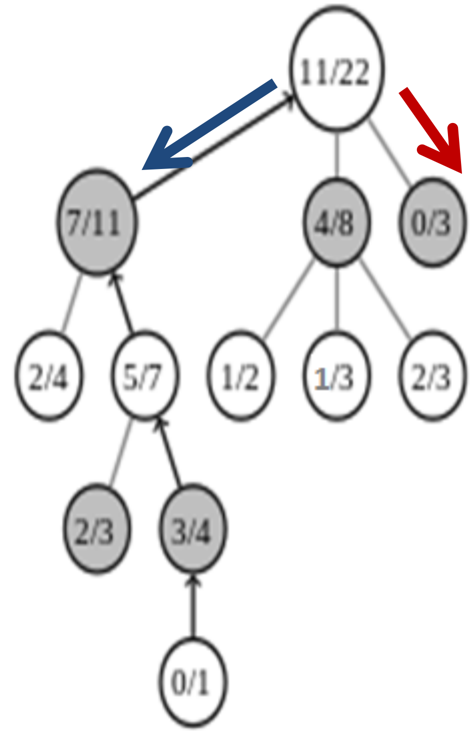
One of the key issues is still which node to **select** for expansion. Two methods:
1. Select the child node of the root node, which has the largest **value** (The node's value may be defined in various ways. See next slide.), then select the child node of the previously selected node, which has the largest value, and repeat this until a leaf node is selected.
2. Select the node that has the largest value among the unexpanded nodes.
The second approach is simpler, but may not work well with some game trees!

Selection in MCTS Using Upper Confidence Bound

Exploration-Exploitation Tradeoff:

$$UCB = \frac{w_i}{n_i} + C\sqrt{\frac{\ln(t)}{n_i}}$$

Exploitation **Exploration**



$\ln$: natural logarithm.

w_i : no. of wins after visiting node *i*

n_i : no. of times node *i* has been visited.

C : exploration factor. In theory, $C = \sqrt{2}$.

t : no. of times the parent of node *i* has been visited,

i.e., t is equal to the sum of n_i.

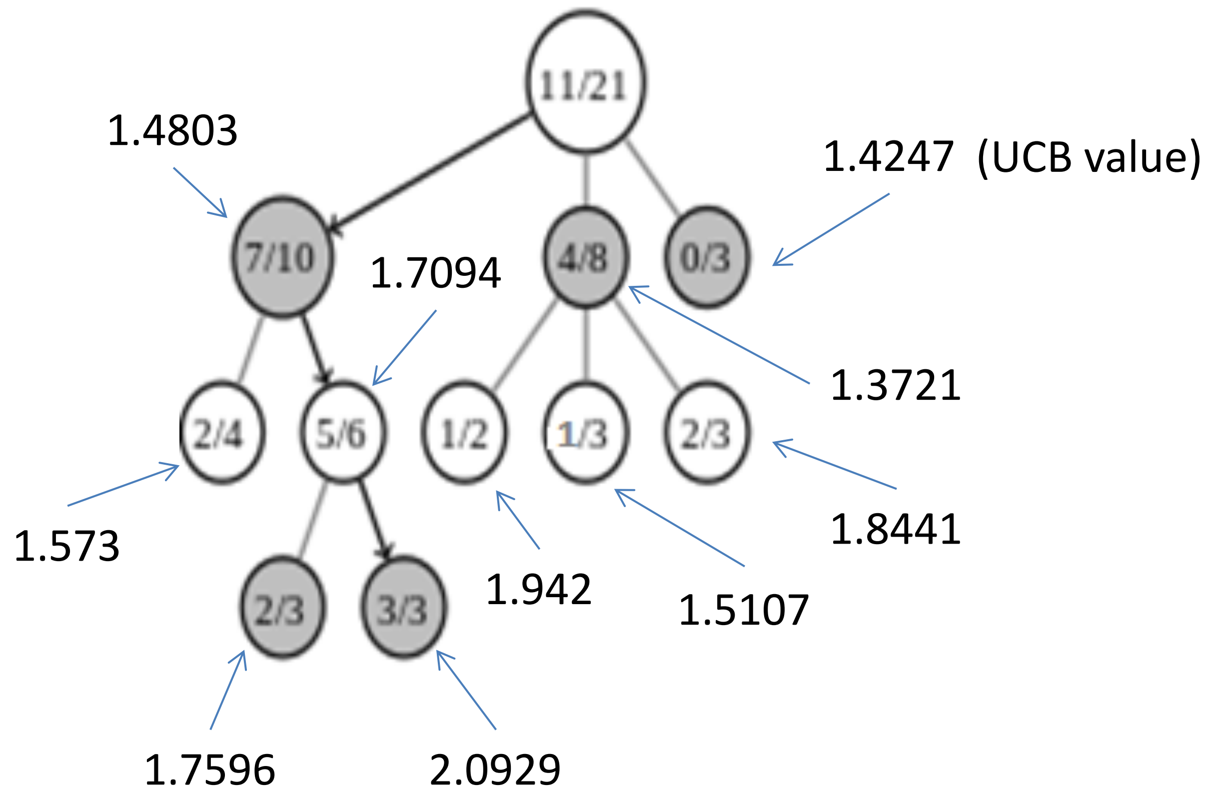
(*e.g.*, for the nodes in depth 1: $w_1=7, n_1=11, w_2=4, n_2=8, w_3=0, n_3=3, t=22$)

Emphasising exploitation: selection in favour of nodes with higher average win ratio, e.g., as shown by the blue arrow in the figure.

Emphasising exploration: selection in favour of nodes with fewer visits, e.g., as shown by the red arrow .

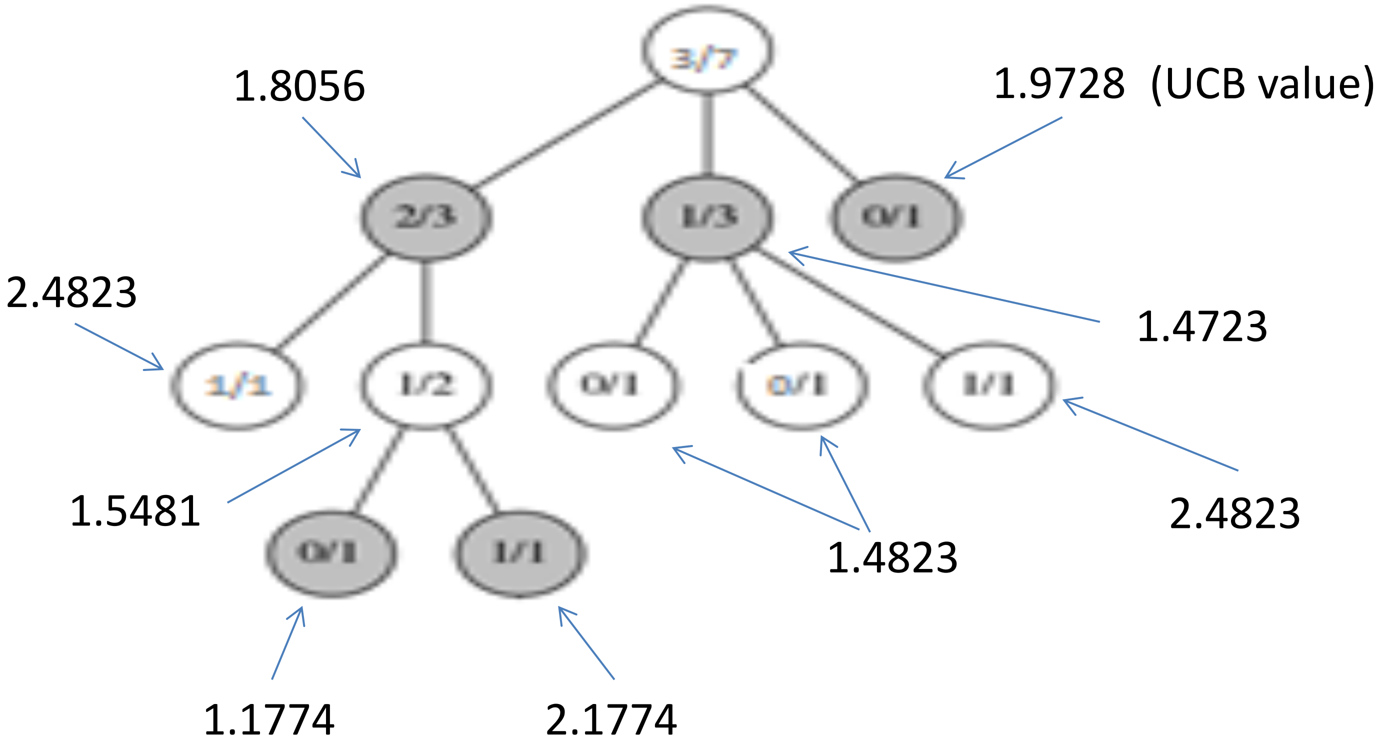
An Example of Node Selection in MCTS

Why is the node with win ratio of 3/3 selected in slide 23?



Both methods described in slide 23 obtain the same result:
The node with win ratio of 3/3 is selected.

Another Example of Node Selection in MCTS

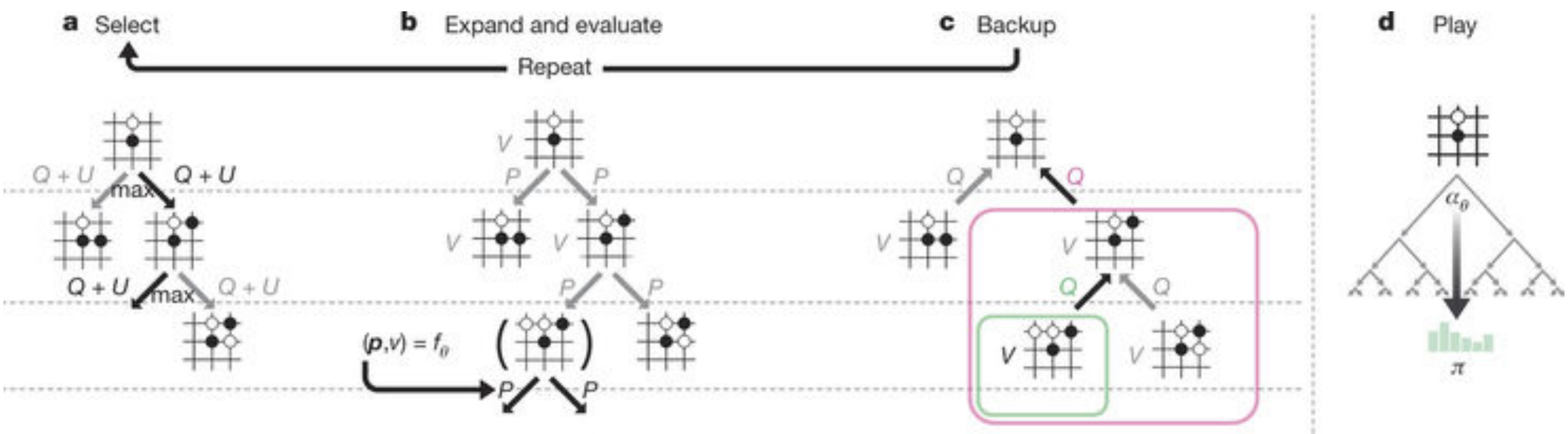


If we select among the child nodes of the root node first, the child node with UCB=1.9728 should be selected for expansion. It is unnecessary to consider the other nodes because this child node of the root node is unexpanded.

However, if we select the node that has the largest UCB value among the unexpanded nodes, then either of the two nodes with UCB=2.4823 can be selected for expansion next. The two methods select different nodes in this case.

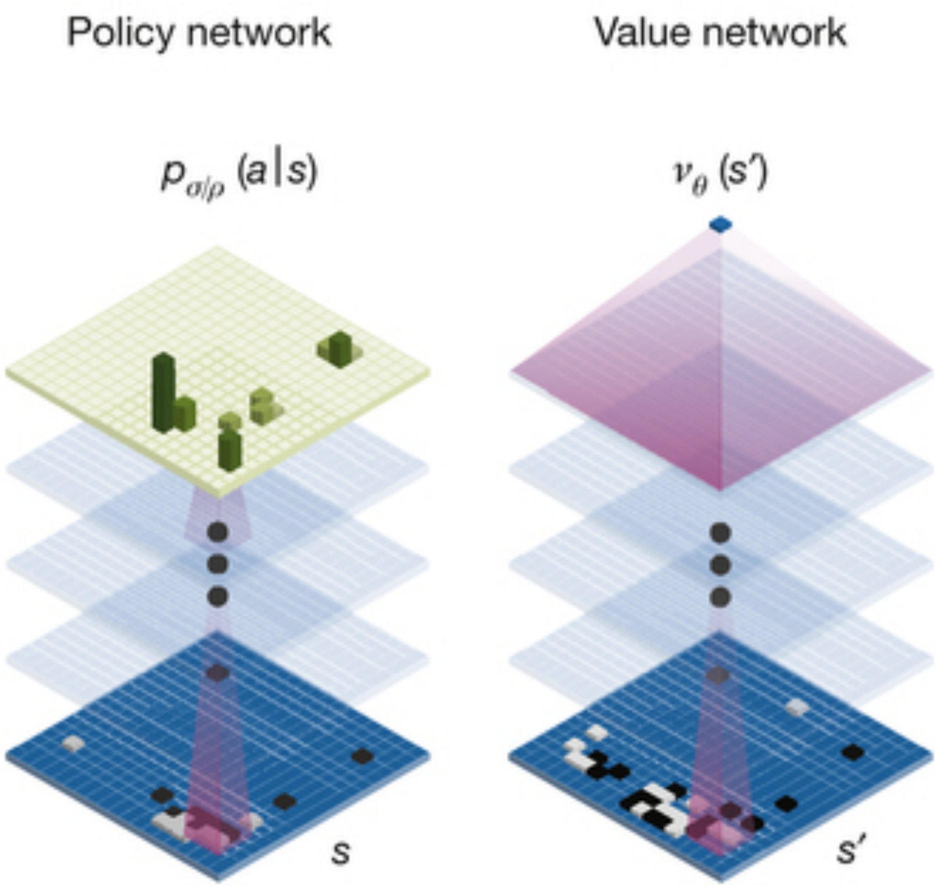
Application of MCTS in AlphaGo Zero

David Silver et al, Mastering the game of Go without human knowledge
Nature, **550**, 354–359, 19 October 2017



Value neural network (for selection) and policy neural network (for expansion and play) are combined with MCTS in AlphaGo, making MCTS more powerful.

Success of AlphaGo: MCTS + Machine Learning



Reinforcement learning, deep learning, Mote-Carlo Tree Search are combined for game playing in AlphaGo.

They are based on huge amount of data/experience from Go experts and self-playing.

Powerful cloud computing makes the complex machine learning feasible.

After learning, the value network could outperform world-leading Go experts in evaluating game positions.

The learnt policy network can choose optimal moves given current game position.

Further Reading on MCTS

MCTS has been a hot research topic in the past decade. There have been many modified MCTS methods, such as new node selection criteria and game state evaluation methods, especially those in the development of AI player for the game of Go.

Here are two survey articles on MCTS:

https://moodle.essex.ac.uk/pluginfile.php/797117/mod_resource/content/1/Monte-Carlo_Tree_Search_-_A_New_Framework_for_Game_AI_2008.pdf (short)

https://moodle.essex.ac.uk/pluginfile.php/796529/mod_resource/content/2/mcts-survey.pdf (long)

How to program a computer to beat (almost) anyone at chess/go?

Three key ideas in traditional AI can be useful:

Minimax search

Alpha-beta pruning

Evaluation functions for evaluating game positions

Modern approaches focus on game position evaluation:

Monte-Carlo Tree Search and/or machine learning

(no need of explicit evaluation function).

Grandmaster Chess Programs (for self study)

In 1997, IBM's Deep Blue beat Kasparov, the then world champion, in an exhibition match.

How was this possible?

Like all chess playing programs, Deep Blue is based upon the ideas we have already discussed.

The rules of chess tournaments impose time limits, so such programs do their minimaxing using an iterative deepening technique.

They keep going deeper until time runs out.

They examine the whole search tree to their depth limit.

[This slide and the next three are for information only.]

Grandmaster Chess Programs (2)

Deep Blue made very effective use of these basic techniques in two ways:

Evaluation function: (the most difficult part)

- Very sophisticated
- 8000 features
- (indicating what are good or bad game positions / states)

Special Purpose Hardware (huge speed and memory)

- 30 RS/6000 processors doing the alpha-beta search
- 480 custom VLSI processors doing move generation and ordering.
  - Searched 30 billion positions per move
  - Typical search depth: 14 moves
  - In some circumstances the depth reached 40 moves

Grandmaster Chess Programs (3)

Deep Blue also used another technique that dates back to the 1950s:

Databases

Deep Blue had:

- An opening book of 4000 positions
- An endgame database containing solutions for all positions with 5 or fewer pieces
- Copies of 700,000 grandmaster games

All of these enable much searching to be eliminated by immediately indicating the best move for a given position.

Making good use of information could greatly improve search efficiency!

Grandmaster Chess Programs (4)

More recent programs running on special purpose hardware have continued to beat grandmasters:

Hydra (https://en.wikipedia.org/wiki/Hydra_(chess))

Deep Fritz (https://en.wikipedia.org/wiki/Fritz_(chess))

Rybka (http://www.rybkachess.com/)

Rybka has even beaten grandmasters after giving a pawn advantage.

All of this has led to the view that success at computer chess requires special purpose hardware.

But, programs written for standard PC's have won the World Computer Chess Championship.

How? – more efficient search strategies!

Summary

Minimax search:

Most popular method for adversarial search

alpha-beta pruning (efficiency is essential for a huge search tree):

Reduces effort required in state space search

Monte-Carlo tree search – basic ideas and steps:

Evaluation of game positions by running simulated games

Application of MCTS in AlphaGo

Evaluation functions (most difficult part):

Heuristics for evaluating game states (game positions)

Improve adversarial search in large state space