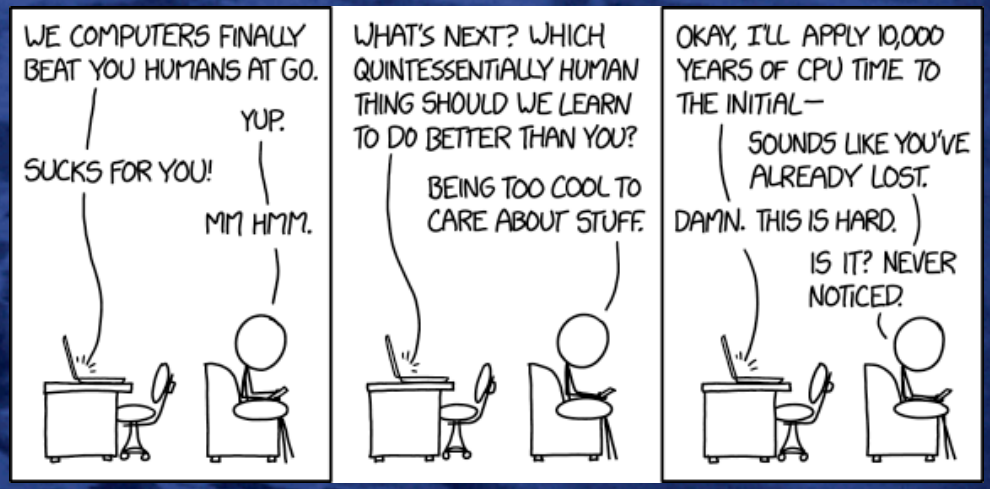
Game-playing

CITS3001 Algorithms, Agents and Artificial Intelligence

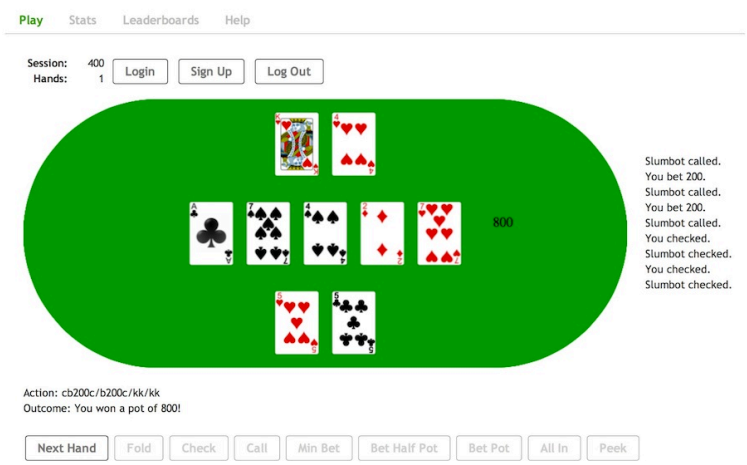


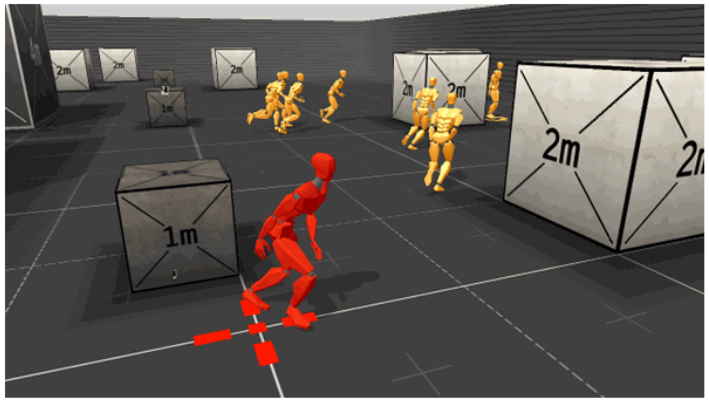
Tim French
Department of Computer Science and Software Engineering
The University of Western Australia

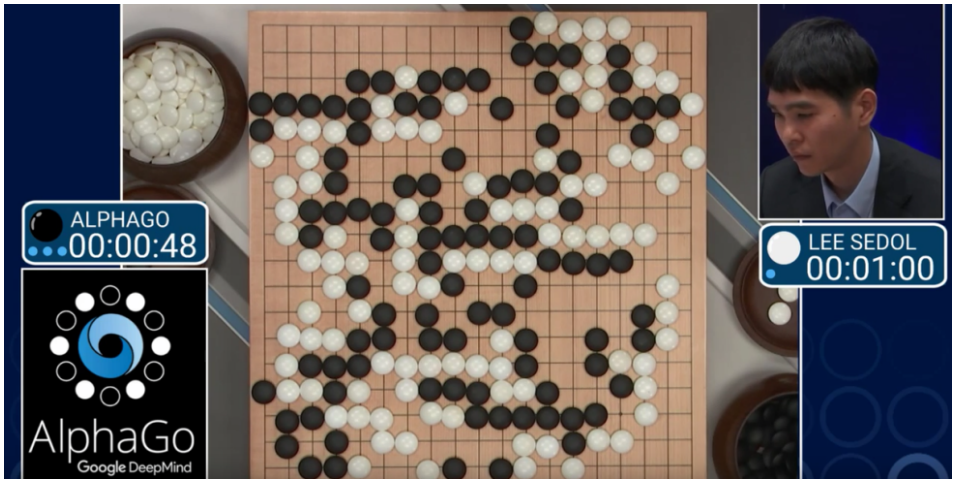
2021, Semester 2

Introduction

- We will motivate the investigation of games in AI
- We will apply our ideas on search to game trees
  - Minimax
  - Alpha-beta pruning
- We will introduce the idea of an evaluation function
  - And some concepts important to their design







Broadening our worldview

- In our discussions so far, we have assumed that world descriptions have been
  - *Complete* – all information needed to solve the problem is available to the search algorithm
  - *Deterministic* – the effects of actions are uniquely determined and predictable

- But this is rarely the case with real-world problems!

- Sources of incompleteness include
  - Sensor limitations – it may be impossible to perceive the entire state of the world
  - Intractability – the full state description may be too large to store, or too large to compute

- Sources of non-determinism are everywhere
  - e.g. people, weather, mechanical failure, dice, *etc*.

- Incompleteness ↔ non-determinism?
  - Both imply uncertainty
  - Addressing them involves similar techniques

Three Approaches to Uncertainty

THE UNIVERSITY OF WESTERN AUSTRALIA

- *Contingency planning*
  - Build all possibilities into the plan
  - Often makes the tree very large
  - Can only guarantee a solution if the number of contingencies is tractable

- *Interleaving*, or *adaptive planning*
  - Alternate between planning, acting, and sensing
  - Requires extra work during execution
  - Unsuitable for offline planning

- *Strategy learning*
  - Learn, from examples, strategies that can be applied in any situation
  - Must decide on parameterisation, state-evaluation, suitable examples to study, *etc*.

Why Study Games

THE UNIVERSITY OF WESTERN AUSTRALIA

- Games provide
  - An abstraction of the real world
  - Well-defined, clear state descriptions
  - Limited operations with well-defined consequences
  - A way of making incremental, controllable changes
  - A way of including hostile agents

- So they provide a forum for investigating many of the real-world issues outlined previously
  - More like the real world than previous examples…

- The initial state and the set of actions (the moves of the game) define a *game tree* that serves as the search tree
  - But of course different players get to choose actions at various points
  - So our previous search algorithms don't work!

- Games are to AI, as F1 is to car design

Example: noughts and crosses

- Each level down represents a move by one player
  - Known as one *ply*
  - Stop when we get to a goal state (three in a line)
- What is the "size" of this problem?

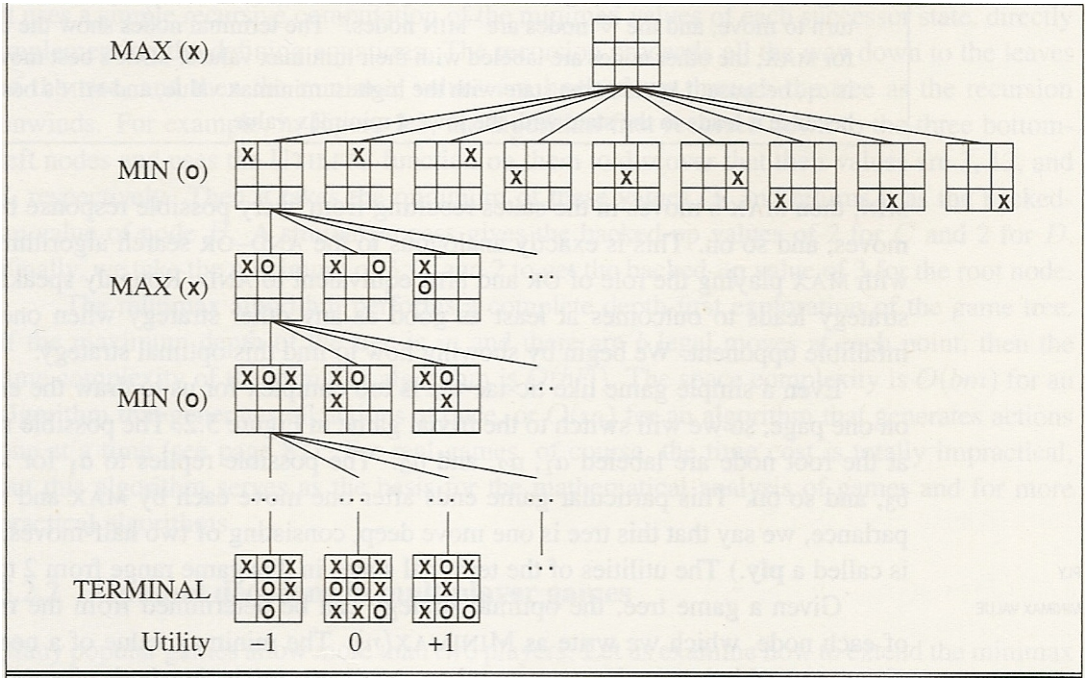
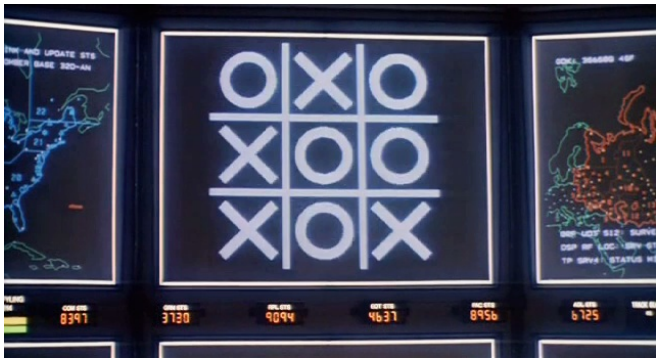


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

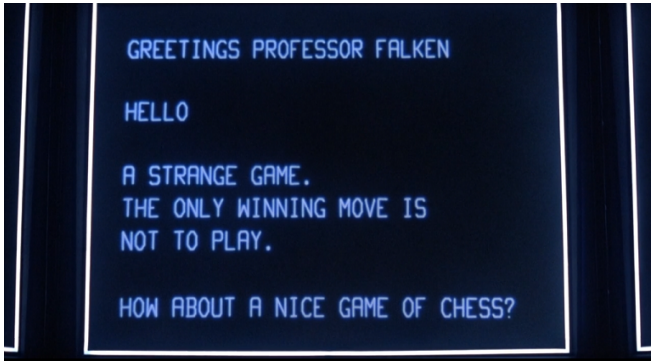
Noughts and Crosses Vital Statistics

THE UNIVERSITY OF WESTERN AUSTRALIA

- The game tree as drawn above has *9! = 362,880* edges
  - But that includes games that continue after a victory
  - Removing these gives *255,168* edges
- Combining equivalent game boards leaves *26,830* edges
  - Mostly this means resolving rotations and reflections



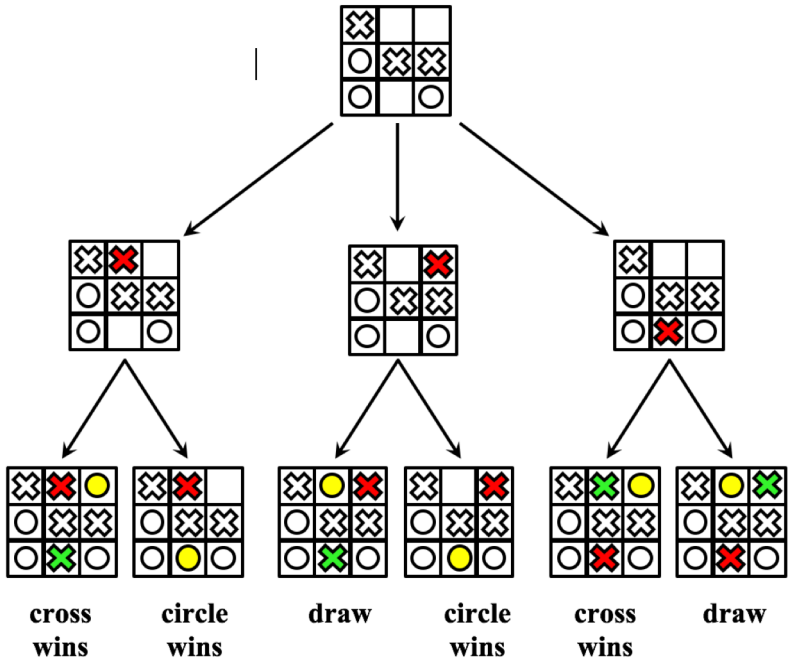
- Each square can be a cross, a circle, or empty
  - Therefore there are 3^9 = *19,683* distinct boards
  - But that includes (e.g.) boards with five crosses and two circles
  - Removing these gives *5,478* distinct legal boards
- Resolving rotations and reflections leaves *765* distinct legal boards



- The takeaway message is “think before you code”!

Noughts and Crosses Scenarios

- You get to choose your opponent's moves, and you know the goal, but you don't know what is a good move
  - Normal search works, because you control every
  - What is the best uninformed search strategy?
    - How many states does it visit?
  - What is a good heuristic for *A** here?
    - How many states does it visit?



- Your opponent plays randomly
  - Does normal search work?
  - Uninformed strategy?
  - *A** heuristic?

- Your opponent tries
  - We know it's a draw really

- One important difference with games is that we don't get to dictate all of the actions chosen, *The opponent has a say too!*

Perfect Play: the Minimax algorithm

THE UNIVERSITY OF WESTERN AUSTRALIA

- Consider a two-player game between MAX and MIN
  - Moves alternate between the players
- Assume it is a zero-sum game
  - Whatever is good for one player, is bad for the other
- Assume also that we have a *utility function* that we can apply to any game position
  - *utility(s)* returns $r \in R$
  - ∞ if *s* is a win for MAX
  - positive if *s* is good for MAX
  - 0 if *s* is even
  - negative if *s* is good for MIN
  - $-\infty$ if *s* is a win for MIN
- Whenever MAX has the move in position *s*, they choose the move that maximises the value of *utility(s)*
  - Assuming that MIN chooses optimally
- Conversely for MIN

Minimax(s)

= *utility(s)*, if *terminal(s)*

= *max{Minimax(result(s, a)) | a* $\in$ *actions(s)}*, if *player(s)* = MAX

= *min{Minimax(result(s, a)) | a* $\in$ *actions(s)}*, if *player(s)* = MIN

Minimax operation

- We imagine that the game tree is expanded to some definition of terminals
  - This will depend on the *search depth*
    - In the figure, two ply
  - This will depend on the available resources
  - In general, it won't be uniform across the tree
- The tree is generated top-down, starting from the current position
  - Then Minimax is applied bottom-up, from the leaves back to the current position
- At each of MAX's choices, they (nominally) choose the move that maximises the utility
  - Conversely for MIN

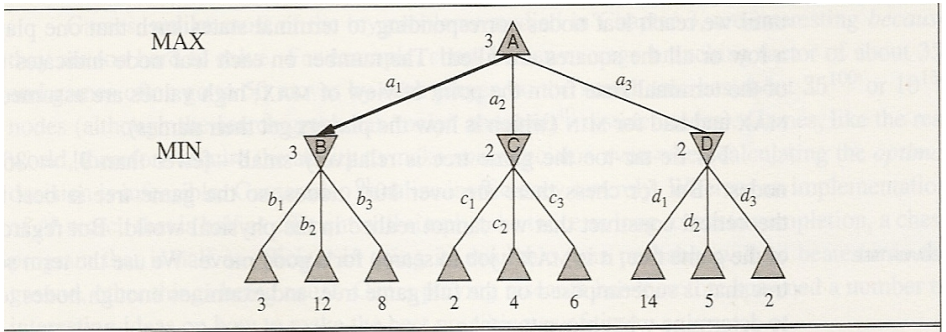


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Minimax Performance

- Complete: yes, for a finite tree
- Optimal: yes, against an optimal opponent
- Time: O(b^m), all nodes examined
- Space: O(bm), depth-first (or depth-limited) search

- Minimax can be extended straightforwardly to multi-player games
  - Section 5.2.2 of AIMA

- But for a "big" game like chess, expanding to the terminals is completely infeasible

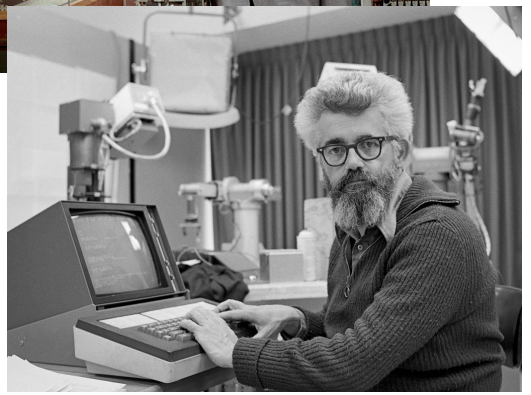
- The standard approach is to employ
  - A cut-off test, e.g. a depth limit
    - Possibly with *quiescence search*
  - An evaluation function
    - Heuristic used to estimate the desirability of a position
- This will still be perfect play.... ***If*** we have a perfect evaluation function…

Example: Chess

- Average branching factor is 35
  - Search tree has maybe 35^{100} nodes
  - Although only around 10^{40} distinct legal positions
- Clearly cannot solve by brute force
  - Intractable nature → incomplete search
  - So offline contingency planning is impossible

- Interleave time- or space-limited search with moves
  - This lecture
  - Algorithm for perfect play [Von Neumann, 1944]
  - Finite-horizon, approximate evaluation [Zuse, 1945]
  - Pruning to reduce search costs [McCarthy, 1956]

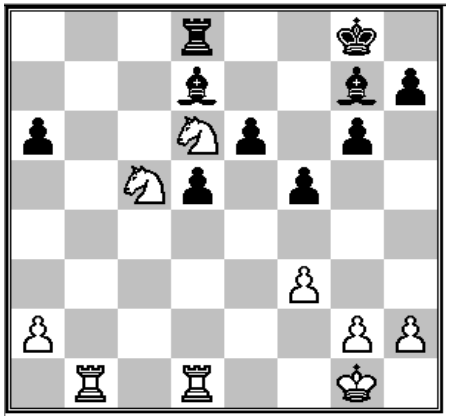
- Or use/learn strategies to facilitate move-choice based on current position
  - Later in CITS3001



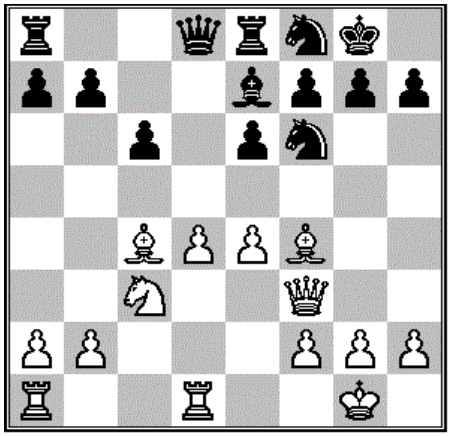
- What do humans do?

Evaluation Functions

- If we cannot expand the game tree to terminal nodes, we expand "as far as we can" and apply some judgement to decide which positions are best
- A standard approach is to define a *linear weighted sum* of relevant features
  - e.g. in chess: 1 for each pawn, 3 for each knight or bishop, 5 for each rook, 9 for each queen
  - Plus positional considerations, e.g. centre control
  - Plus dynamic considerations, e.g. threats
- $eval(s) = w_1f_1(s) + w_2f_2(s) + \ldots + w_nf_n(s)$
  - e.g. $w_1 = 9$
  - *e.g.* $f_1(s)$ = number of white Qs – number of black Qs
- Non-linear combinations are also used
  - e.g. reward pairs of bishops



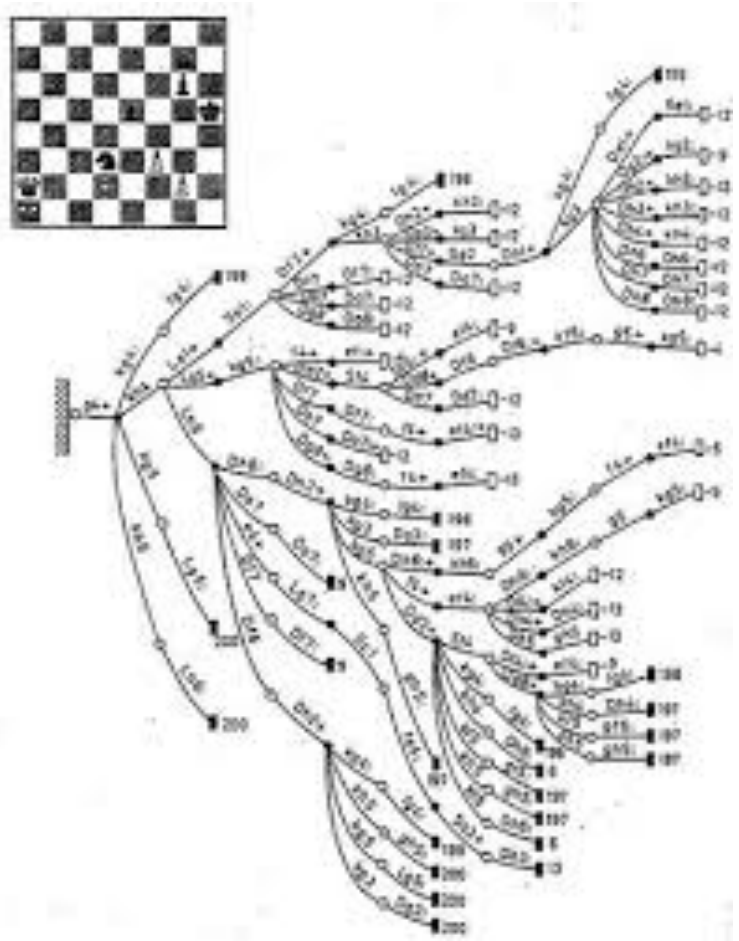
Material advantage



Positional advantage

Properties of good evaluation functions

- Usually the quality of the player depends ***critically*** on the quality of the evaluation function
- An evaluation function should
  - Agree with the utility function on terminal states
  - Reflect the probability of winning
  - Be time efficient, to allow maximum search depth
- Note that the exact values returned seldom matter
  - Only the ordering matters
- An evaluation could also be accompanied by a measure of *certainty*
  - e.g. we may prefer high certainty when we are ahead, low certainty when we are behind



Cutting off Search

THE UNIVERSITY OF WESTERN AUSTRALIA

- We can cut-off search at a fixed depth
  - Works well for simple games
  - Depth-limited search
- Often we are required to manage the time taken per move
  - Can be hard to turn time into a cut-off depth
  - Use iterative-deepening
    - An *anytime algorithm*
  - Sacrifice (some) depth for flexibility
- Sometimes we are required to manage the time taken for a series of moves
  - More complicated again
  - Sometimes we can anticipate changes in the branching factor
- Seldom want cut-off depth to be uniform across the tree
  - Two particular issues that arise often are *quiescence* and the *horizon effect*

Quiescence

- A *quiescent* situation is one where values from the evaluation function are unlikely to change much in the near future
- Using a fixed search-depth can mean relying on the evaluations of non-quiescent situations
  - Can avoid this by e.g. extending the search to the end of a series of captures

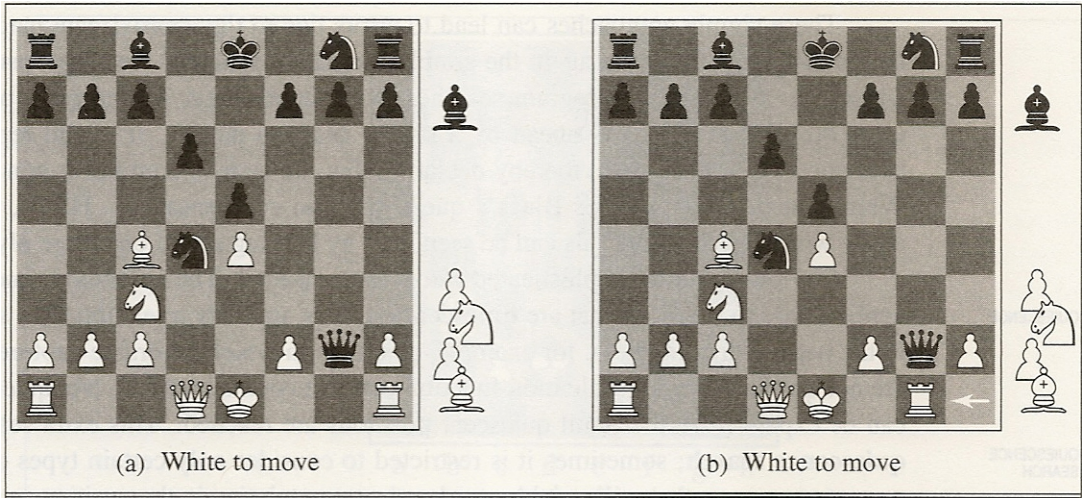


Figure 5.8 Two chess positions that differ only in the position of the rook at lower right. In (a), Black has an advantage of a knight and two pawns, which should be enough to win the game. In (b), White will capture the queen, giving it an advantage that should be strong enough to win.

The Horizon Effect

- If we are searching to *k* ply, something bad that will happen on the *k+1*th ply (or later) will be invisible
- In extreme cases, we may even select bad moves, simply to postpone the inevitable
  - If "the inevitable" scores *–x*, any move that scores better than *–x* in the search window looks good
  - Even if the inevitable is still guaranteed to happen later!
- No general solution to this problem
  - It is fundamentally a problem with lack of depth

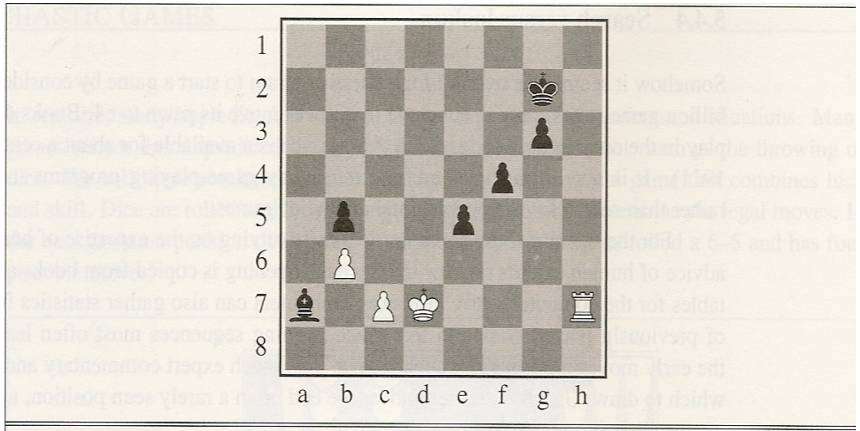


Figure 5.9 The horizon effect. With Black to move, the black bishop is surely doomed. But Black can forestall that event by checking the white king with its pawns, forcing the king to capture the pawns. This pushes the inevitable loss of the bishop over the horizon, and thus the pawn sacrifices are seen by the search algorithm as good moves rather than bad ones.

Alpha-Beta Pruning

- One way we can reduce the number of nodes examined by Minimax is to identify nodes that cannot be better than those that we have already seen
  - This will enable a deeper search in the same time
- Consider again Fig. 5.2
- Minimax(A) = max(min(_,_,_), min(_,_,_), min(_,_,_))
  - Working from left-to-right
  - First we inspect the 3, 12, and 8
- Minimax(A) = max(3, min(_, _, _), min(_, _, _))
  - Next we inspect the first 2
- Minimax(A) = max(3, min(2, _, _), min(_, _, _))
  - This is less than the 3
  - The next two leaves are immediately irrelevant
- Minimax(A) = max(3, min(_, _, _)) = max(3, 2) = 3
- We ***do not*** need to inspect the 5th and 6th leaves
  - But we do need to inspect the 8th and 9th…

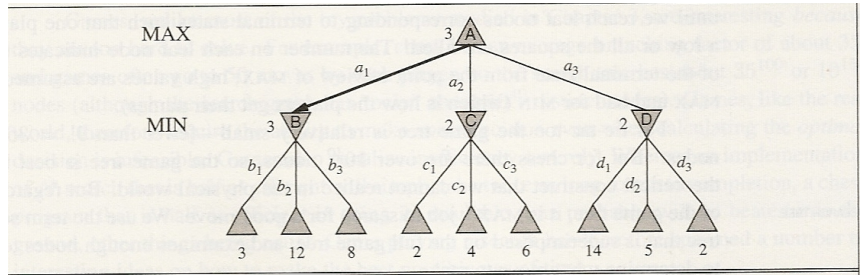


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Alpha-Beta Operation

- We need to keep track of the range of possible values for each internal node
- In Fig. 5.6, if
  - On the left sub-tree, we know definitely that we can choose a move that gives score *m*, and
  - On the right sub-tree, we know that the opponent can choose a move that limits the score to $n \leq m$
- Then we will never (rationally) choose the move that leads to the right sub-tree

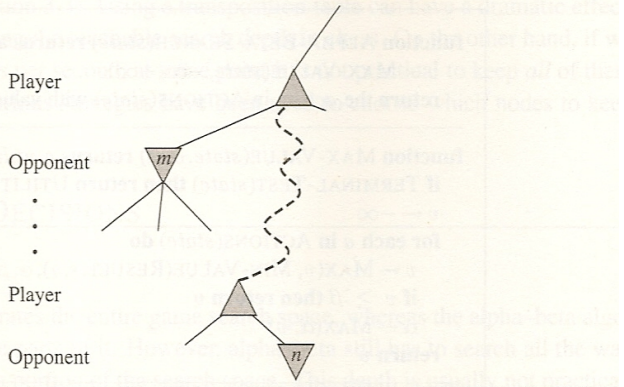


Figure 5.6 The general case for alpha–beta pruning. If m is better than n for Player, we will never get to n in play.

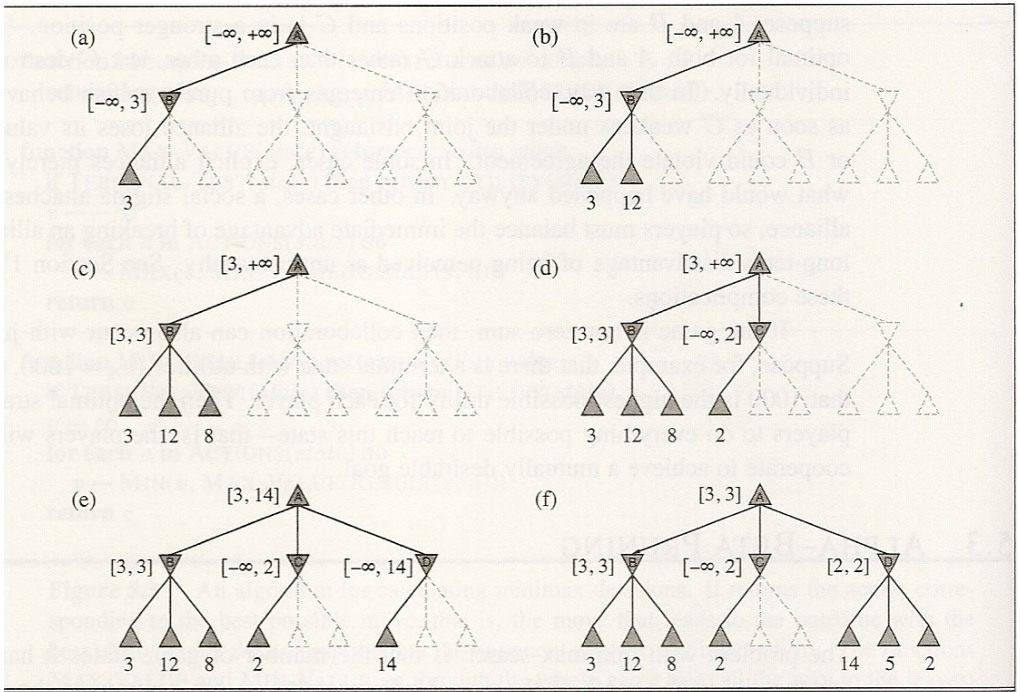


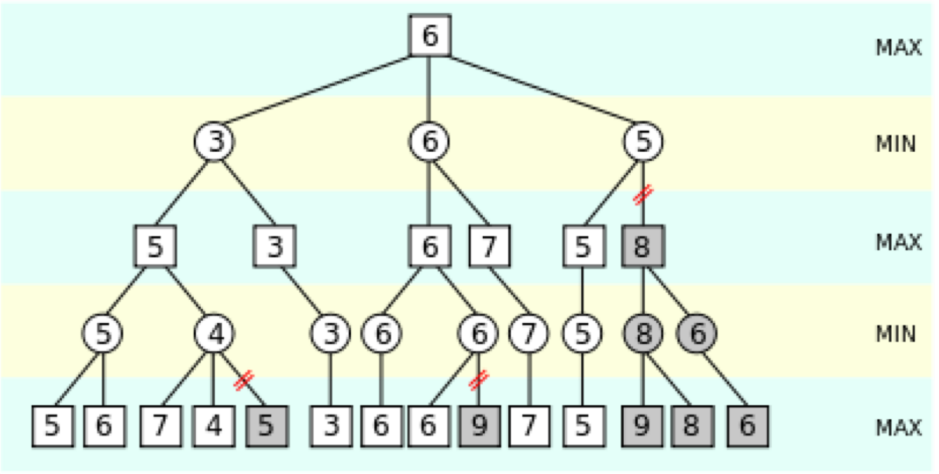
Figure 5.5 Stages in the calculation of the optimal decision for the game tree in Figure 5.2. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of *at most* 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. This is an example of alpha–beta pruning. (e) The first leaf below D has the value 14, so D is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

Alpha-beta pseudo-code

```
αβsearch(s):
     return a ∈ actions(s) with value maxvalue(s, −∞, +∞)

maxvalue(s, α, β):
  if terminal(s) return utility(s)
  else
    v = −∞
    for a in actions(s)
      v = max(v, minvalue(result(s, a), α, β))
      if v ≥ β return v
      α = max(α, v)
    return v

minvalue(s, α, β):
  if terminal(s) return utility(s)
  else
    w = +∞
    for a in actions(s)
      w = min(w, maxvalue(result(s, a), α, β))
      if w ≤ α return w
      β = min(β, w)
```



Alpha-beta in action

- αβsearch(A) = maxvalue(A, $-\infty$, $+\infty$), $v = -\infty$
- call minvalue(B, $-\infty$, $+\infty$), $w = +\infty$
  - call maxvalue(B_1, $-\infty$, $+\infty$)
    - returns 3, $w = 3$, $\beta = 3$
  - call maxvalue(B_2, $-\infty$, 3)
    - returns 12
  - call maxvalue(B_3, $-\infty$, 3)
    - returns 8
  - returns 3, $v = 3$, $\alpha = 3$
- call minvalue(C, 3, $+\infty$), $w = +\infty$
  - call maxvalue(C_1, 3, $+\infty$)
    - returns 2, $w = 2$
  - returns 2
- call minvalue(D, 3, $+\infty$), $w = +\infty$
  - call maxvalue(D_1, 3, $+\infty$)
    - returns 14, $w = 14$, $\beta = 14$
  - call maxvalue(D_2, 3, 14)
    - returns 5, $w = 5$, $\beta = 5$
  - call maxvalue(D_3, 3, 5)
    - returns 2, $w = 2$
  - returns 2
- returns 3

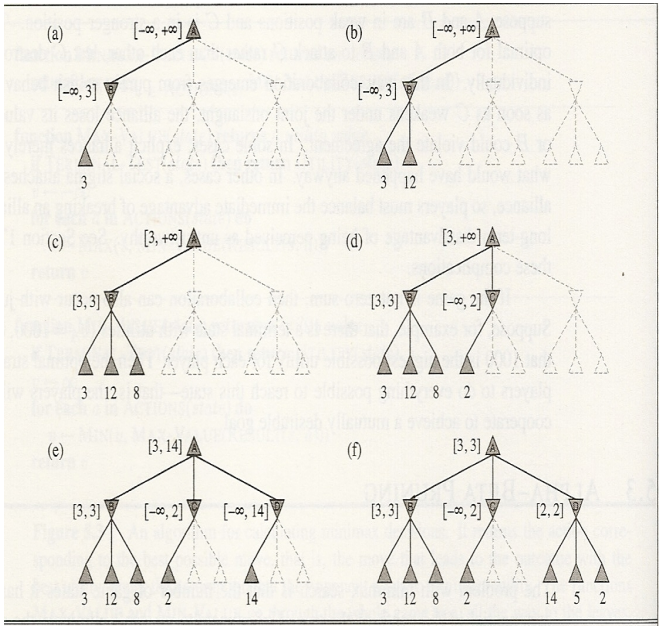


Figure 5.5 Stages in the calculation of the optimal decision for the game tree in Figure 5.2. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of *at most* 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. This is an example of alpha–beta pruning. (e) The first leaf below D has the value 14, so D is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

Alpha-beta discussion

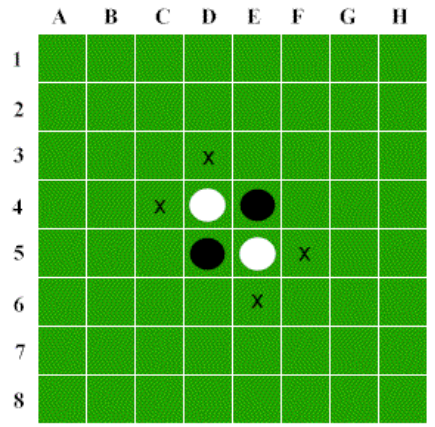
THE UNIVERSITY OF WESTERN AUSTRALIA

- *Pruning does **not** affect the final result*
  - It simply gets us there sooner
- A good move ordering means we can prune more
  - e.g. if we had inspected D_3 first, we could have pruned D_1 and D_2
- We want to test *expected good moves first*
  - "Good" from the POV of that node's player
- "Perfect ordering" can double our search depth
  - Obviously perfection is unattainable, but e.g. in chess we might test
    - Captures
    - Threats
    - Forward moves
    - Backward moves
- Sometimes we can learn good orderings
  - Known as *speedup learning*
  - Can play either faster at the same standard, or better in the same time

Game playing agents….

- **Checkers (Draughts)**
  - Marion Tinsley ruled Checkers for forty years, losing only seven games in that time
  - In 1994 Tinsley's health forced him to resign from a match against *Chinook*, which was crowned world champion shortly afterwards
  - At that time, *Chinook* used a database of 443,748,401,247 endgame positions
  - Checkers has since been proved to be a draw with perfect play
    - The proof was announced at UWA
    - Chinook now plays perfectly, using αβ search and a database of 39,000,000,000,000 positions

- **Chess**
  - *Deep Blue* defeated Gary Kasparov in a six-game match in 1997
  - Deep Blue searches 200,000,000 positions/second, up to 40 ply deep



- **Othello**
  - Look-ahead is very difficult for humans in Othello
  - *The Moor* became world champion in 1980
  - These days computers are banned from championship play

Game playing agents….

- **Go**
  - 19x19 Go has a branching factor of over 300, making look-ahead very difficult for programs
    - Play at a “good amateur” level, although still improving
  - They are much better at 9x9
  - DeepMind AlphaGo defeated Go champion Lee Sedol in March 2016.

- **Backgammon**
  - Dice rolls increase the branching factor
    - 21 possible rolls with two dice
  - About 20 legal moves with most positions and rolls
    - Although approx 6,000 sometimes with a 1+1!
    - Depth 4 means $20 \times (21 \times 20)^3 \approx 1{,}500{,}000{,}000$ possibilities
  - Obviously most of this search is “wasted”
    - Value of look-ahead is much diminished
  - *TDGammon* (1992) used depth 2 search plus a very good evaluation function to reach
    “almost” world champion level
    - Players have since copied its style!
  - Modern programs based on neural networks are believed to better than the best humans

- **Poker**
  - *Pluribus* from Facebook and CMU recently beat a group of professional poker players.