Busca Adversarial

- A Busca Adversarial se insere no contexto de jogos, onde:
  - O ambiente é **multiagente**, ou seja, onde múltiplos agentes interagem entre si
  - Como tal, é necessário pensar em **contingências**, ou seja, é preciso que os agentes consigam interagir adequadamente com os outros agentes, dado que cada agente tem seu objetivo
  - E, o mais importante, no contexto de jogos os agentes **competem entre si**.
- Na Inteligência Artificial estamos interessados em:
  - **Jogos Determinísticos**: aqueles em que a sorte não interefere. Jogos de carta são exemplos de jogos em que a sorte interefere nos resultados. Enquanto jogos como xadrez e go dependem somente da ação dos jogadores
  - **Jogos Zero-Sum**: são jogos em que a soma do estado final dos jogadores é igual a zero. Ou seja, se um jogador ganha, o outro perde - dizemos que seus **valores de utilidade** ao final do jogo são opostos
    - Valor de Utilidade: podemos dizer que o valor de utilidade é a quantificação da qualidade de satisfação de um agente segundo uma determinada métrica.
  - **Perfect Information**: o jogo é completamente transparente, ou seja, não é possível esconder nada do adversário e todos os jogadores conseguem ter acesso ao históricos de jogadas. O ambiente é **completamente observável**.
    - De forma análoga, jogos de informação imperfeita são aqueles em que nem todas as informações são visíveis aos jogadores.
  - **Jogos de turnos**
- Os jogos são problemas **difíceis de resolver**, onde muitas vezes o **branching factor** pode chegar a 35, ou seja, para cada nó existem 35 possibilidades.

    - Isto torna necessária a adoção de uma heurística em busca de uma solução subótima, dada a impossibilidade de calcular todas as possibilidades e adotar uma estratégia de força bruta.
- Temos que definir o que é um **movimento ótimo** e como encontrá-lo.
- Utilizamos **poda** para ignorar porções de uma árvore de decisão que não vão nos levar à uma progressão em direção à escolha correta.
- Vamos utilizar também uma **função de avaliação** que nos permite indicar uma melhoria ou nos permite chegar ao estado final de utilidade sem avaliarmos todo o espaço de busca.
- Primeiro, vamos considerar jogos de dois jogadores, que vamos chamar MAX e MIN.
- Um jogo pode ser definido formalmente:

  (tome s como estado)

    - **S0**: Estado Inicial - qual é a configuração de início do jogo.
    - **Player(s)**: Define qual jogador fará movimento em um determinado estado
    - **Actions(s)**: Dado um estado s, quais são ações possíveis?
    - **Result(s, a)**: Dado um **Modelo de Transição**, qual será a consequência de uma ação sobre um determinado estado.
        - **Modelo de Transição**: um sistema que é capaz de representar as propriedades principais de um ambiente, e que pode se encontrar em diversos estados análogos à realidade, sobre a qual podem ser feitas ações, e onde a ação feita sobre um determinado estado resulta em um novo estado.
    - **Terminal-Test(s)**: um teste que indica se o jogo já foi terminado ou não - retorna True se o jogo já se encerrou e False caso ainda não tenha terminado. Estados que representam o término de um jogo são chamadaos de **Terminal States**.
    - **Utility(s, p)**: define um valor numérico para um jogador que chegou a um *Terminal State*, sendo que este valor representa a performance do jogador ao final do jogo. Tipicamente, quando o jogador vence, empata ou perde,

damos, respectivamente, os valores 1, 1/2 e 0. A ideia de *zero-sum* é que a soma da função utilidade para todos os jogadores é igual a zero.

    - Existem jogos que são menos simétricos ou que envolvem pontuações, então nem sempre os jogos são *zero-sum*, podendo o *Valor de Utilidade* assumir o valor de pontuação para uma determinada partida.

- **Game Tree:** por meio de Estado Inicial (S0), pela função Action e Result podemos dar início à Game Tree, que é uma árvore em que os nós representam estados do jogo e as arestas as possíveis movimentações dos jogadores dado um certo estado.
    - Umas das principais característica da Game Tree é ser exaustiva. Em muitos casos ela é teórica, como no xadrez, em que temos possivelmente 10^{40} nós folha (Estados Terminais). Ela deve mostrar todos os resultados possíveis, ou seja, todas as movimentações possíveis sobre todos os estados possíveis. Este é um conceito que se difere da **Search Tree**.
- **Search Tree:** é uma árvore que sobrepomos sobre a Game Tree, mas cujo crescimento é limitado através de podas pois ela deve ser factível e eficiente, utilizando heurísticas para determinar quais as melhores próximas movimentações.

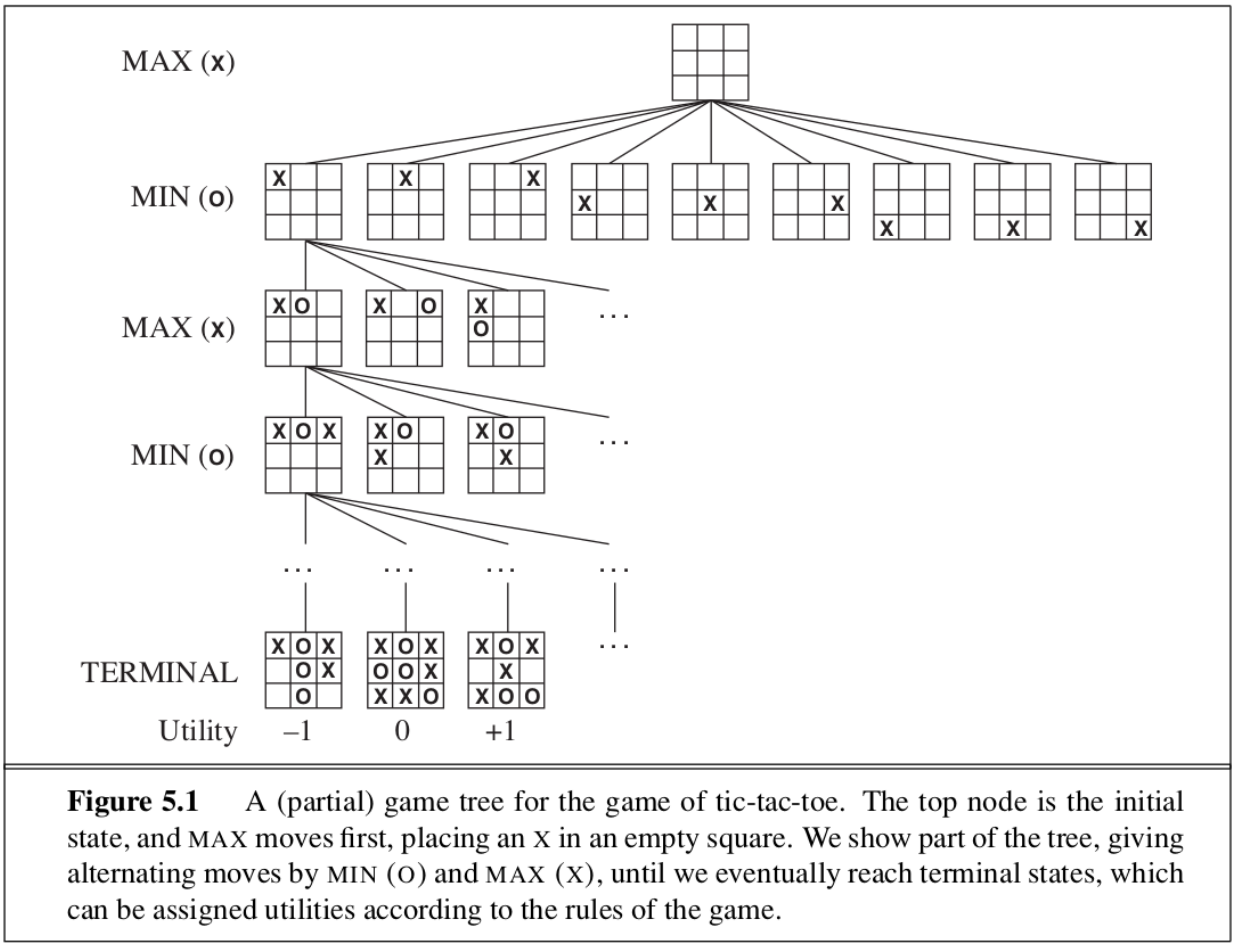

Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities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

Perceba o que é o S0, os estados e as ações. Note que um estado terminal é caracterizado pelo fato de que todos os quadradinhos foram preenchidos ou então MAX ou MIN formaram uma linha. Outra coisa interessante a ser notada é que o valor de utilidade deve ser calculado segundo a perspectiva de algum jogador. No exemplo dado este é calculado segundo a perspectiva do jogador MAX.

Optimal Decisions in Games

- Em um problema de busca convencional, a solução ótima é constituída de uma sequência de ações que levam ao **Estado Objetivo**.
    - Na *Busca Adversarial* MIN tentará dificultar a situação, forçando MAX a adotar estratégias de *Contingência* para dar uma resposta às ações de MIN.
- MAX inicia o jogo já levando em consideração as dificuldades que MIN pode impor, e então executa uma ação. Esta ação então é respondidada por MIN

através de outra ação. E então MAX deve novamente levar MIN em consideração, escolhendo uma ação que consiga contornar as ações de MIN.

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
  OR-SEARCH(problem.INITIAL-STATE, problem, [])

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
  if problem.GOAL-TEST(state) then return the empty plan
  if state is on path then return failure
  for each action in problem.ACTIONS(state) do
      plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
      if plan ≠ failure then return [action | plan]
  return failure

function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
  for each s_i in states do
      plan_i ← OR-SEARCH(s_i, problem, path)
      if plan_i = failure then return failure
  return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```

Figure 4.11 An algorithm for searching AND–OR graphs generated by nondeterministic environments. It returns a conditional plan that reaches a goal state in all circumstances. (The notation $[x \mid l]$ refers to the list formed by adding object x to the front of list l.)

- Uma estratégia ótima leva a resultados tão bons quanto qualquer outra estratégia quando estamos jogando contra um **oponente infalível**.
- **Ply**: turno de um jogo. Two-ply é um jogos de dois turnos, onde dois jogadores alternam turnos.
- **Minimax value**: é o valor de utilidade para um dado nó da Game Tree.
  - MAX está interessado em aumentar o valor de utilidade.
  - MIN está interessado em que o valor de utilidade seja mínimo.

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \end{cases}$$

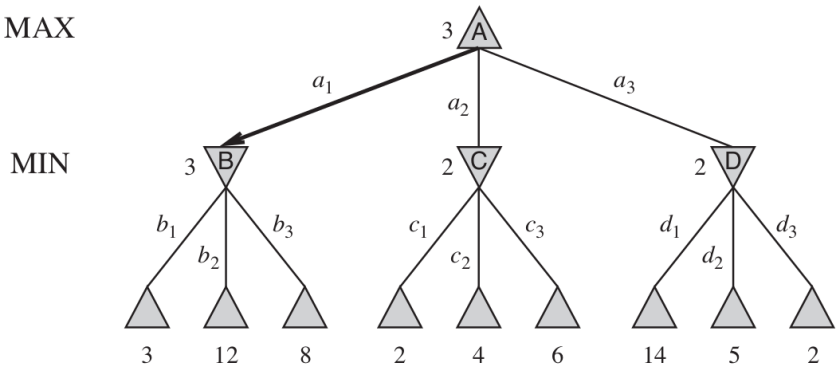

Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state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

- Exemplo do cálculo de Minimax (fig. 5.2):
  - Os nós folha representam os estados terminais, e o Minimax é igual ao Valor de Utilidade.
  - Os nós BCD são MIN, e através de ações cada nó expande em mais três. Para o nó B o valor de Minimax é 3, para C e para D é 2.
  - O nó raiz é MAX, e seu valor Minimax é 3.
- **Minimax Decision:** é a decisão por uma determinada ação guiada pelo valor de Minimax. Por exemplo, a decisão minimax para o nó raiz no exemplo da figura 5.2 é a_1.
- **Estratégia Pessimista**: a estratégia ótima é definida partindo do pressuposto que o oponente de MAX vai tomar todas as melhores decisões possíveis. Desta forma, se o oponente for pior é fácil mostrar que a estratégia funciona, e caso ele tome as melhores decisões estamos preparados.

Algoritmo Minimax

- O Algoritmo Minimax determina qual é a decisão minimax para o estado atual.
- É um algoritmo recursivo que calcula o valor minimax de todos os estados sucessores.

- O Algoritmo desce até os nós folha, e então na volta vai retornando os valores minimax
- **Função de Utilidade**: o algoritmo utiliza a função de utilidade para descobrir o valor minimax de cada nó.
    - Cada nível representa um turno de um jogador, se um nível é MAX, o próximo é MIN, e aquele dois turnos posterior é MAX novamente.
    - O algoritmo vai retorna o valor de utilidade de acordo com o turno do jogador. Vamos tomar como exemplo a árvore da figura 5.2:

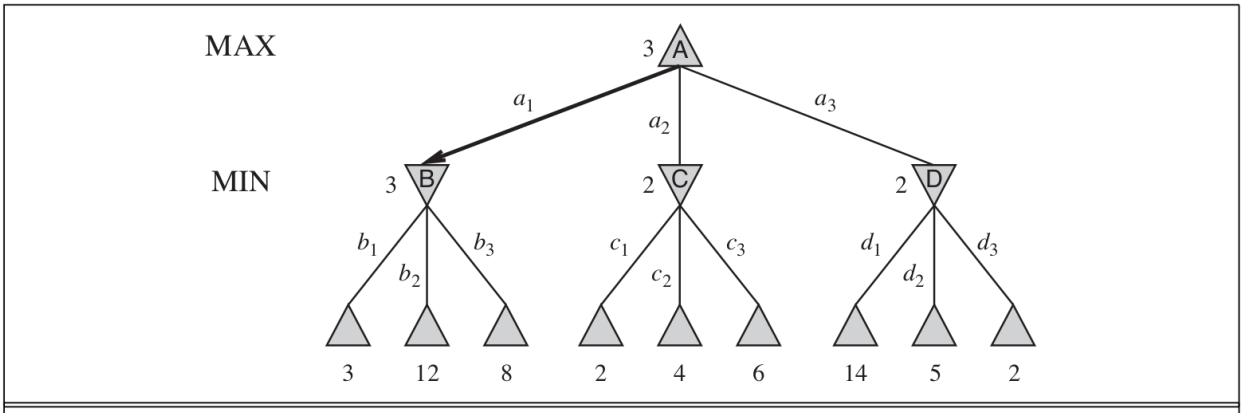

Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state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

A recursão chega até os nós folha, então para o nó B calculamos o valor de utilidade para os sucessores (3, 12 e 8) e escolhemos aquele de menor valor, que seria o 3. Fazemos o mesmo para B e D. Então num próximo nível da árvore temos os valores 3, 2, 2 - dentre esses escolhemos o maior, que é 3.

- O algoritmo desce até o nó folha mais a esquerda e vai voltando, revisitando os nós que faltaram nos níveis anterior
    - É um algoritmo **depth-first**, ou seja, ele primeiro vai fazer a pesquisa em profundidade.

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Figure 5.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation $\text{argmax}_{a \in S} f(a)$ computes the element a of set S that has the maximum value of $f(a)$.

- A complexidade de tempo para o algoritmo minimax é $O(b^d)$, onde:
  - b é o fator de branching, ou seja, qual é a quantidade de nós sucessores que partem de apenas um nó
  - d é a profundidade da árvore

Alpha-Beta Pruning

- O problema do algoritmo Minimax é que o tamanho da árvore cresce exponencialmente com a profundidade, ou seja, possui uma complexidade proibitiva para qualquer jogo real.
- Contudo, existe uma forma de reduzirmos pela metade a quantidade de cálculos, permitindo que entremos a decisão minimax ideal sem termos que verificar todos os nós da Game Tree.

- Podemos empregar o conceito de **poda**, mais especificamente **Alpha-Beta Pruning**:
  - A idéia da poda é eliminarmos trechos da árvore que sabemos não levar à decisão Minimax ideal.

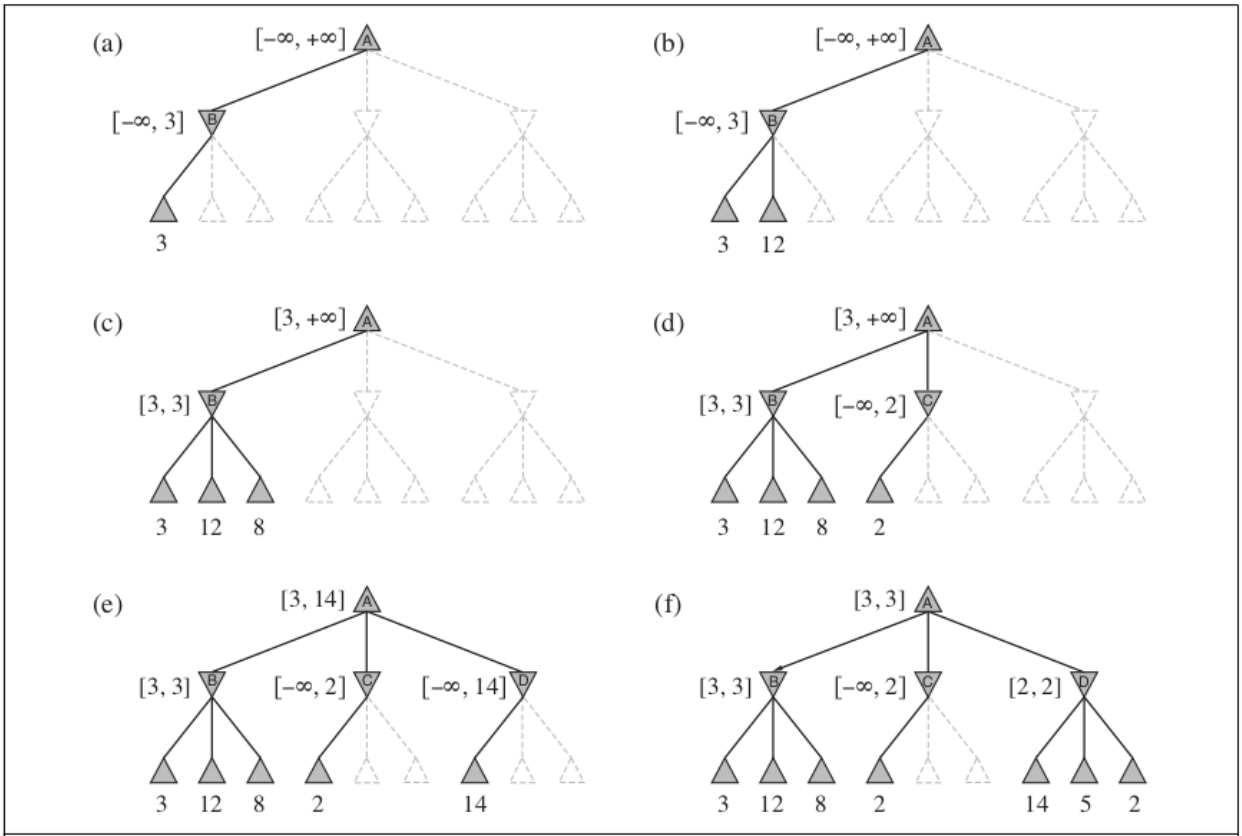

Figure 5.5 Stages in the calculation of the optimal decision for the game tree in Figure 5.2. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value 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.

- O princípio geral da poda é: dado um nó n, caso haja uma outro nó m que seja irmão, esteja no mesmo nível, seja pai, ascendente ou esteja em algum nível superior ao nível de n (superior → mais próximo da raiz), e m possui um valor de utilidade maior que n, então podemos podar n e toda a sua subárvore.
- **Alpha**:
  - Em um dado momento, **Alpha** é o melhor valor que o jogador **MAX** consegue atingir em face das limitações impostas pelas escolhas de **MIN**. Em tese, MAX gostaria de atingir um valor maior que Alpha, mas isto não foi possível graças às escolhas de MIN.
- **Beta**:
  - Em um dado momento, **Beta** é o melhor valor que o jogador **MIN** consegue atingir em face das limitações impostas pelas escolhas de **MAX**. Em tese, MIN gostaria de atingir valores ainda menores de Beta, mas isto não foi possível graças às escolhas de MAX.

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).