

Intelligenza Artificiale

Complementi ed Esercizi

Giochi a Due Avversari

A.A. 2010-2011

Definizioni (2/2)

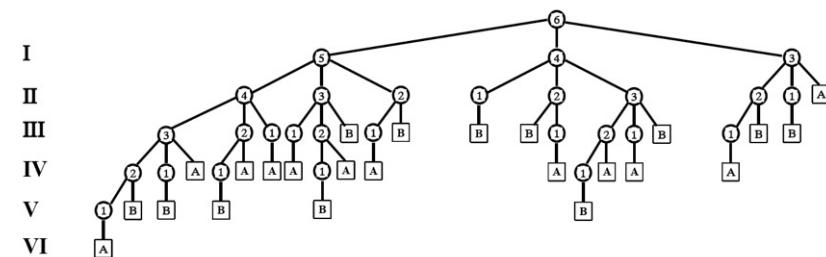
- L'insieme delle possibili istanze di un gioco finito è rappresentabile tramite un grafo orientato, in cui:
 - ogni vertice corrisponde ad una configurazione
 - ogni arco ad una mossa
- Ogni arco (mossa) congiunge i 2 vertici (configurazioni) precedente e successiva alla mossa.

Definizioni (1/2)

- Ogni gioco è definito da una serie di regole che definiscono le mosse lecite.
- Le mosse sono fatte alternativamente da 2 giocatori A e B.
- Ad ogni istante si definisce una configurazione C del gioco.
- Una configurazione terminale rappresenta una situazione di vittoria, sconfitta o pareggio per un giocatore.
- Tutte le altre configurazioni si dicono non-terminali.

Albero di gioco

- La prima mossa viene compiuta da A.
- Le configurazioni terminali contengono l'indicazione del giocatore vincente.



Tipi di giochi

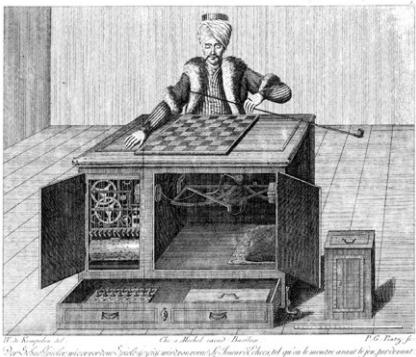
	deterministica	non deterministica
Informazione perfetta	scacchi dama othello	backgammon monopoli
Informazione imperfetta	mastermind	bridge poker scarabeo

I giochi deterministici (2/2)

- **Dama:** Chinook, 444 miliardi di posizioni memorizzate in una base di dati con otto pezzi sulla scacchiera (o meno). Chinook batte il campione in carica Marion Tinsley nel 1994.
- **Othello:** lo spazio di ricerca è più piccolo di quello degli scacchi (numero di mosse legali va da 5 a 15). Logistello nel 1997 ha sconfitto il campione del mondo per 6 volte consecutive.

I giochi deterministici

- **Scacchi:** nel 1997 il programma Deep Blue sconfisse Garry Kasparov in un incontro-esibizione della durata di sei partite. Deep Blue cerca 200 milioni di posizioni al secondo.



Game Search

- **Initial State:** tavolo di gioco, giocatore
- **Successor function:** lista di coppie (azioni, stato), indicano una mossa legale e lo stato risultante
- **Terminal test:** determina la fine del gioco
- **Utility function:** valore numerico che riporta lo stato di utilità per un dato giocatore.

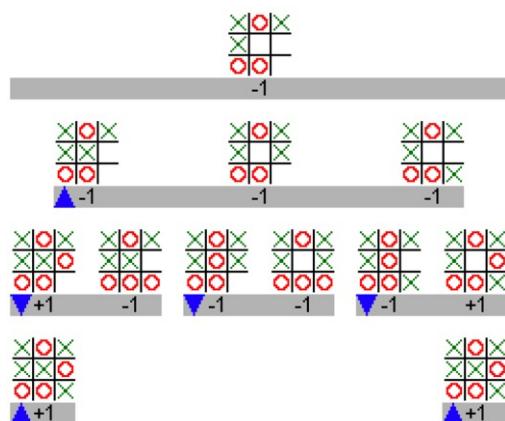
Utility Function

- Si definisce una funzione di utilità U che associa ad ogni configurazione un indice di bontà.
 - $U = +1$ in corrispondenza delle configurazioni terminali in cui A vince.
 - $U = -1$ in corrispondenza di quelle in cui A perde.
 - Per le altre configurazioni ci si pone nella condizione che ambedue i giocatori giochino al meglio.
- Espressa dal punto di vista del giocatore A.

Algoritmo Min Max

- Consiste nello scegliere ad ogni passo la mossa che è più conveniente per
 - MAX, se è il turno di MAX
 - MIN, se è il turno di MIN
- Quindi si sceglie la mossa con
 - la funzione di utilità massima, se è il turno di MAX
 - la funzione di utilità minima, se è il turno di MIN.

Tic Tac Toe

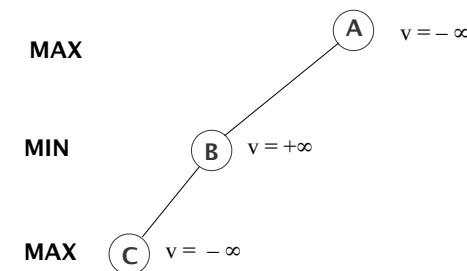


Min Max: un esempio

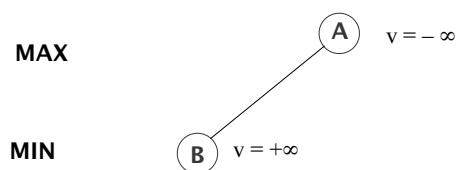
Min Max: un esempio



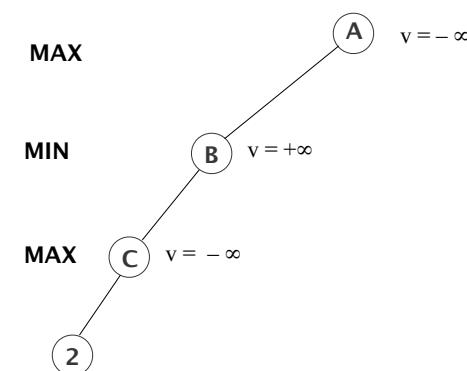
Min Max: un esempio



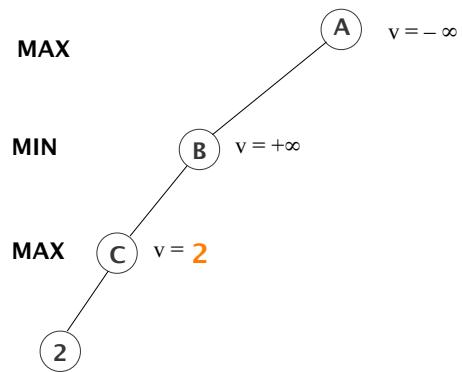
Min Max: un esempio



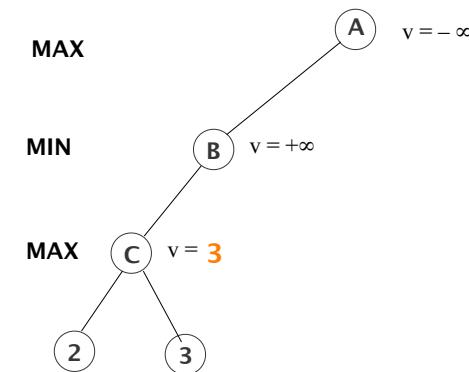
Min Max: un esempio



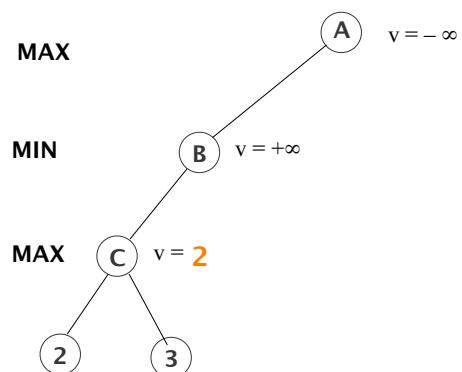
Min Max: un esempio



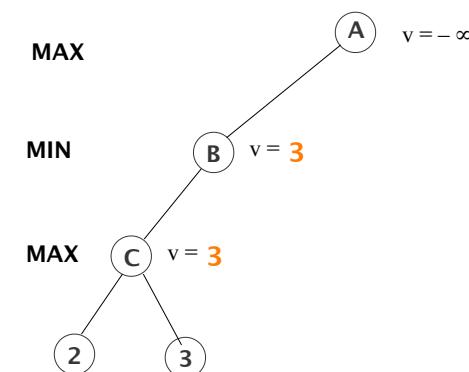
Min Max: un esempio



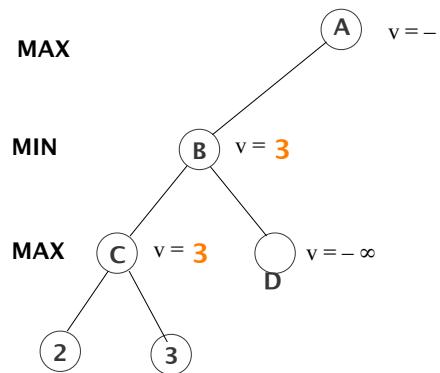
Min Max: un esempio



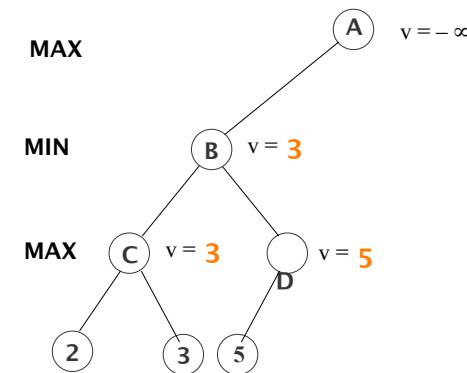
Min Max: un esempio



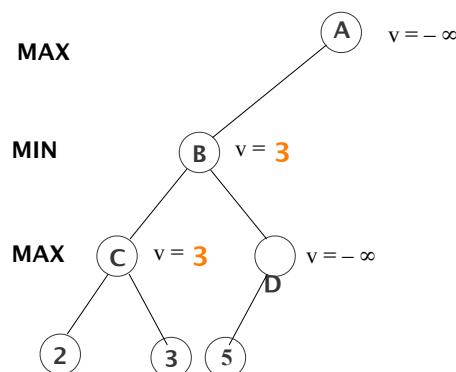
Min Max: un esempio



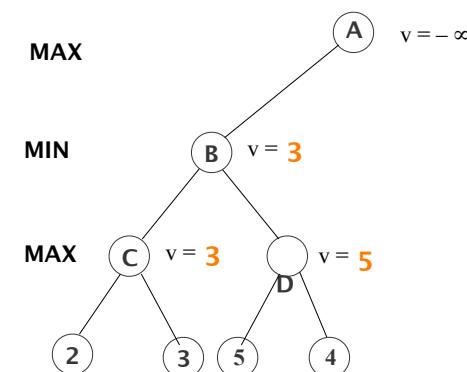
Min Max: un esempio



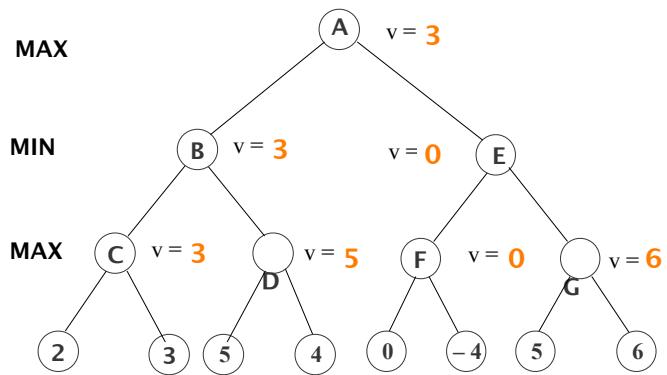
Min Max: un esempio



Min Max: un esempio



Min Max: un esempio



Pseudo Codifica Min Max (1/2)

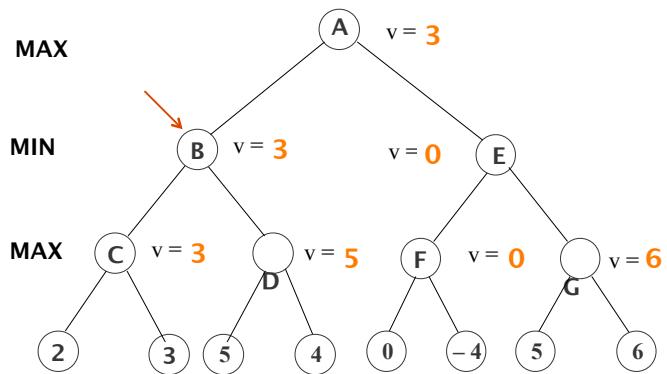
```

function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  v ← MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v

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

Min Max: un esempio



Pseudo Codifica Min Max (2/2)

```

MinMax(GamePosition game) {
  return MaxMove(game);
}

MaxMove (GamePosition game) {
  if (GameEnded(game)) {
    return EvalGameState(game);
  }
  else {
    best_move <- {};
    moves <- GeneratesMoves(game);
    ForEach moves {
      move <- MinMove(ApplyMove(game));
      if (Value(move) > Value(best_move)) {
        best_move <- move;
      }
    }
    return best_move;
  }
}

MinMove (GamePosition game) {
  best_move <- {};
  moves <- GeneratesMoves(game);
  ForEach moves {
    move <- MaxMove(ApplyMove(game));
    if (Value(move) > Value(best_move)) {
      best_move <- move;
    }
  }
  return best_move;
}
  
```

Implementazione Min Max (1/3)

```
public void makeMinMaxMove() {
    getMinMaxValue(presentState);
    GameState nextState = (GameState) presentState.get("next");
    if (nextState == null) {
        throw new RuntimeException("Mini Max Move failed");
    }
    makeMove(presentState, nextState.get("moveMade"));
}

private int getMinMaxValue(GameState state) {
    if (getPlayerToMove(state).equalsIgnoreCase("X")) {
        return maxValue(state);
    } else {
        return minValue(state);
    }
}
```

Implementazione Min Max (2/3)

```
public int maxValue(GameState state)
{
    int v = Integer.MIN_VALUE;
    if (terminalTest(state))
    { return computeUtility(state); }
    else
    {
        ArrayList successorList = getSuccessorStates(state);
        for (int i = 0; i < successorList.size(); i++)
        {
            GameState successor = (GameState) successorList.get(i);
            int minimumValueOfSuccessor = minValue(successor);
            if (minimumValueOfSuccessor > v)
            {
                v = minimumValueOfSuccessor;
                state.put("next", successor);
            }
        }
        return v;
    }
}
```

Implementazione Min Max (3/3)

```
public int minValue(GameState state) {
    int v = Integer.MAX_VALUE;

    if (terminalTest(state)) {
        return computeUtility(state);

    } else {
        ArrayList successorList = getSuccessorStates(state);
        for (int i = 0; i < successorList.size(); i++) {
            GameState successor = (GameState) successorList.get(i);
            int maximumValueOfSuccessors = maxValue(successor);
            if (maximumValueOfSuccessors < v) {
                v = maximumValueOfSuccessors;
                state.put("next", successor);
            }
        }
        return v;
    }
}
```

Albero di gioco parziale

- In molti casi la dimensione dell'albero di gioco non ne rende possibile la costruzione completa.
- In tali casi per ogni mossa si costruisce una parte di albero di gioco (a partire dalla mossa stessa) con profondità pari ad una costante k; alle foglie si associa una funzione di valutazione calcolata sulla base della bontà di quella configurazione.
- Valutazione statica ed euristica

Pseudo Codifica

MinMax con albero parziale

```

function MINMAX-DECISION(state) returns an action
  inputs: state, current state in game
  v  $\leftarrow$  MAX-VALUE(state)
  if CUTOFF-TEST(state, depth) then return EVAL(state)
  v  $\leftarrow -\infty$ 
  for at, s in SUCCESSORS(state) do
    v  $\leftarrow$  MAX(v, MIN-VALUE(s))
  return v

function MAX-VALUE(state) returns a utility value
  if CUTOFF-TEST(state, depth) then return EVAL(state)
  v  $\leftarrow -\infty$ 
  for at, s in SUCCESSORS(state) do
    v  $\leftarrow$  MAX(v, MAX-VALUE(s))
  return v

function MIN-VALUE(state) returns a utility value
  if CUTOFF-TEST(state, depth) then return EVAL(state)
  v  $\leftarrow \infty$ 
  for at, s in SUCCESSORS(state) do
    v  $\leftarrow$  MIN(v, MAX-VALUE(s))
  return v

```

Pseudo Codifica

MinMax con albero parziale

```

MinMax(GamePosition game) {
  return MaxMove(game);
}

```

```

MaxMove (GamePosition game) {
  if (GameEnded(game) || DepthLimitReached()) {
    return EvalGameState(game, MAX);
  } else {
    best_move  $\leftarrow$  {};
    moves  $\leftarrow$  GeneratesMoves(game);
    ForEach moves {
      move  $\leftarrow$  MinMove(ApplyMove(game));
      if (Value(move) > Value(best_move)) {
        best_move  $\leftarrow$  move;
      }
    }
    return best_move;
  }
}

```

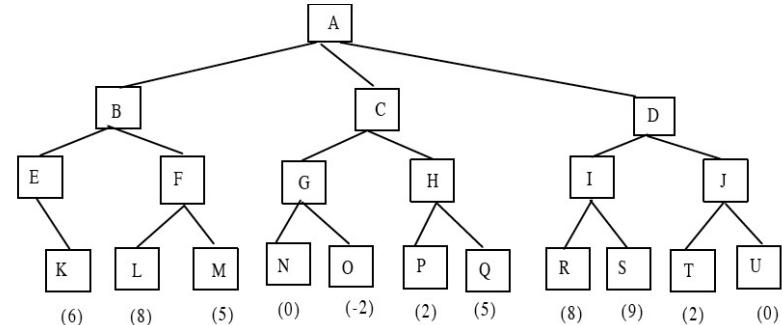
```

MinMove (GamePosition game) {
  if (GameEnded(game) || DepthLimitReached())
    return EvalGameState(game, MIN);
  } else {
    best_move  $\leftarrow$  {};
    moves  $\leftarrow$  GeneratesMoves(game);
    ForEach moves {
      move  $\leftarrow$  MaxMove(ApplyMove(game));
      if (Value(move) < Value(best_move)) {
        best_move  $\leftarrow$  move;
      }
    }
    return best_move;
}

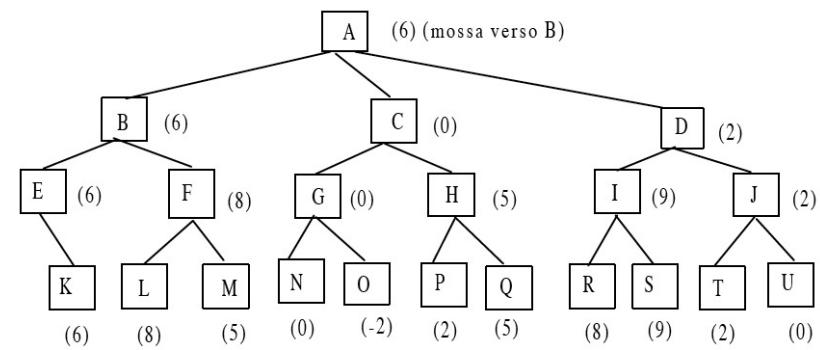
```

Esercizio

- Si consideri il seguente albero di gioco dove i punteggi sono tutti dal punto di vista del primo giocatore.
- Supponendo che il primo giocatore sia MAX, quale mossa dovrebbe scegliere?



Soluzione



Pruning

- Senza perdere in esattezza, è possibile evitare di percorrere alcune parti dell'albero, in quanto si sa preventivamente che esse non influenzano il risultato finale.
- La tecnica su cui si basa tale potatura di dice Alpha-Beta cutoff.

Limiti di Min Max

- In generale, la complessità temporale è $O(b^m)$.
- Negli scacchi $35^{100} = 2,5 \times 10^{154}$
- "*Unfortunately, the number of possible positions in the chess tree surpasses the number of atoms in the Milky Way.*" Claude Shannon

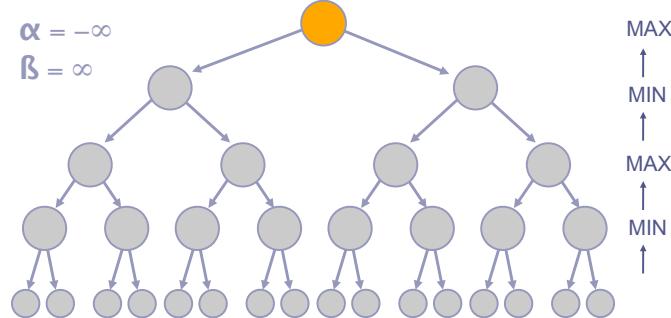
Algoritmo Alpha Beta Cutoff (1/2)

- Durante la navigazione, passare due valori:
 - Alpha = il migliore trovato per MAX
 - Beta = il migliore trovato per MIN
- Al livello MAX, prima di valutare il percorso che parte da ciascun figlio, confrontare il valore ritornato dall'esplorazione del cammino del figlio precedente con il valore Beta. Se il valore è maggiore di Beta interrompere la ricerca.

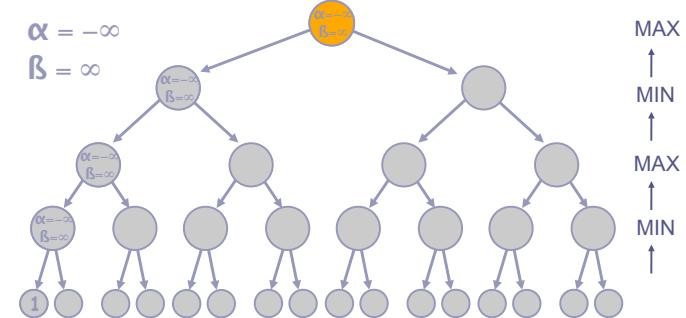
Algoritmo Alpha Beta Cutoff (2/2)

- Al livello Min, prima di valutare il percorso che parte da ciascun figlio, confrontare il valore ritornato dall'esplorazione del cammino del figlio precedente con il valore Alpha.
- Se il valore è minore di Alpha interrompere la ricerca per il nodo precedente.

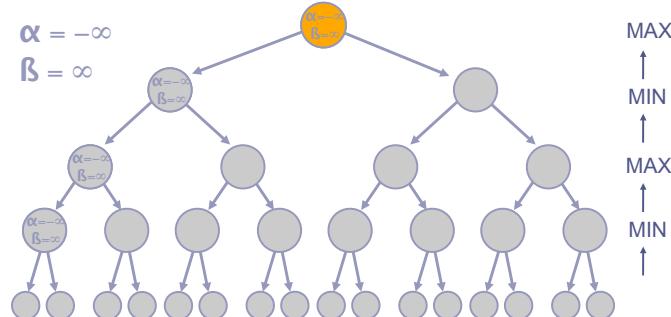
Alpha Beta Pruning: un esempio



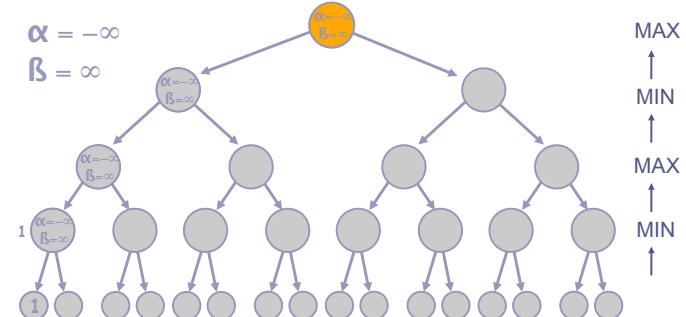
Alpha Beta Pruning: un esempio



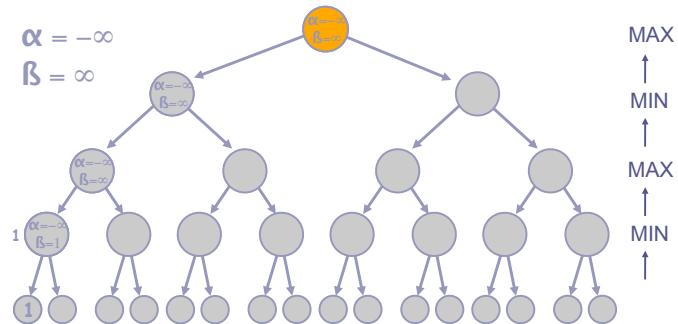
Alpha Beta Pruning: un esempio



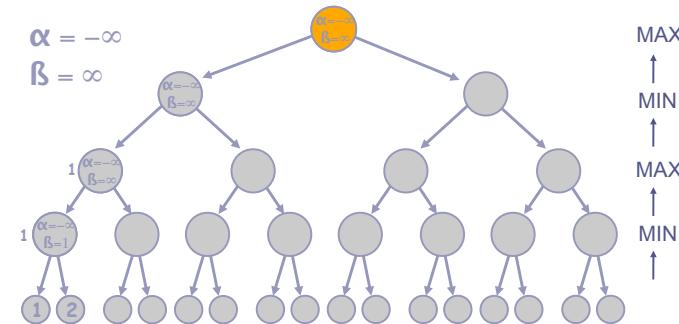
Alpha Beta Pruning: un esempio



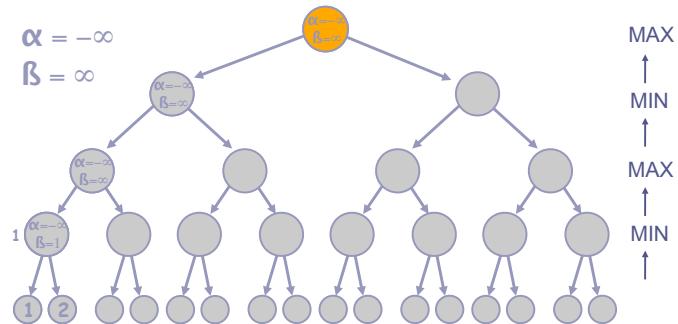
Alpha Beta Pruning: un esempio



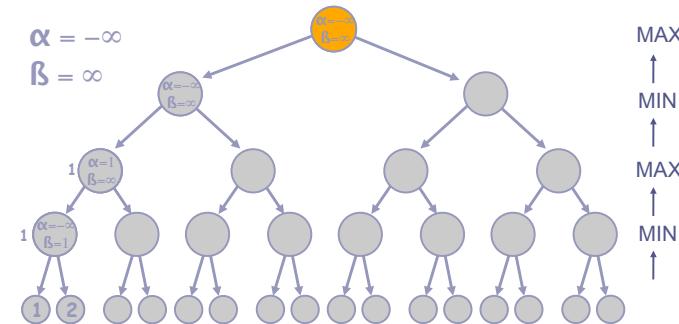
Alpha Beta Pruning: un esempio



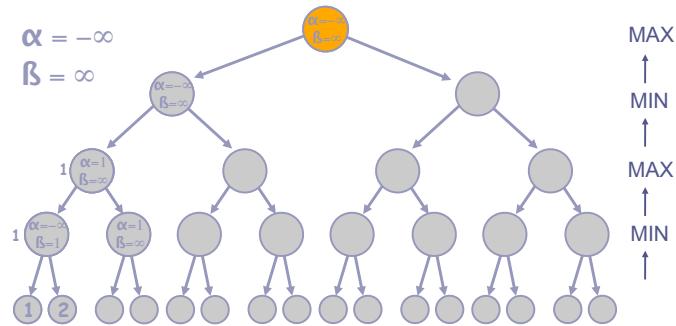
Alpha Beta Pruning: un esempio



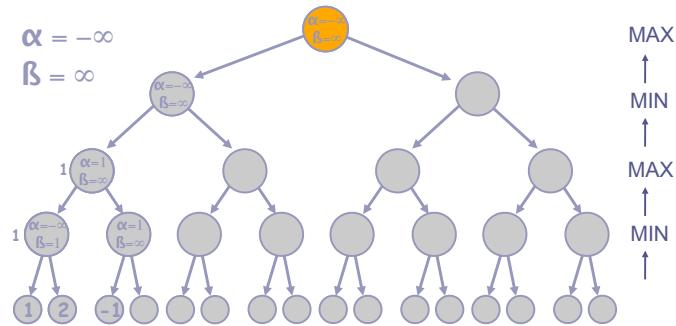
Alpha Beta Pruning: un esempio



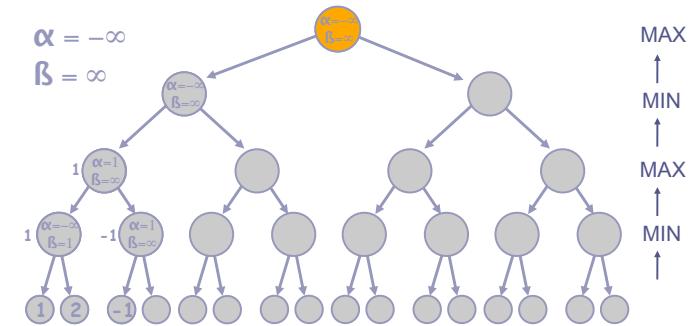
Alpha Beta Pruning: un esempio



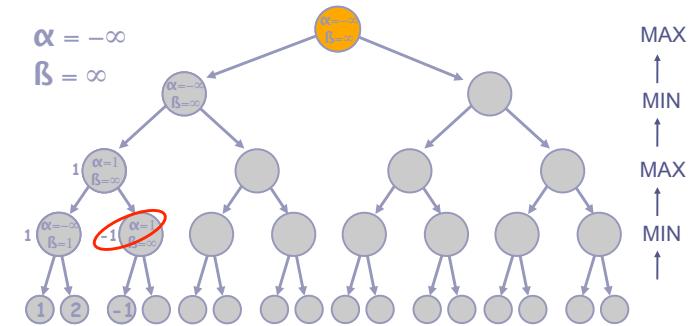
Alpha Beta Pruning: un esempio



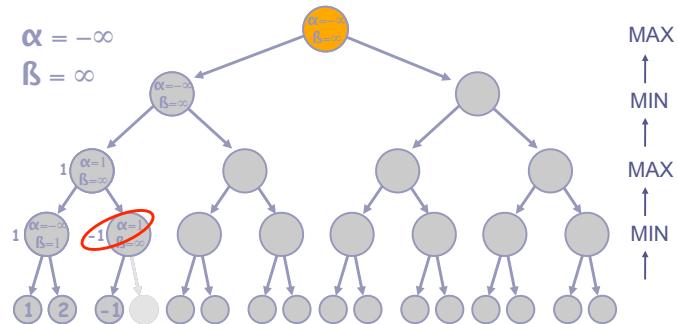
Alpha Beta Pruning: un esempio



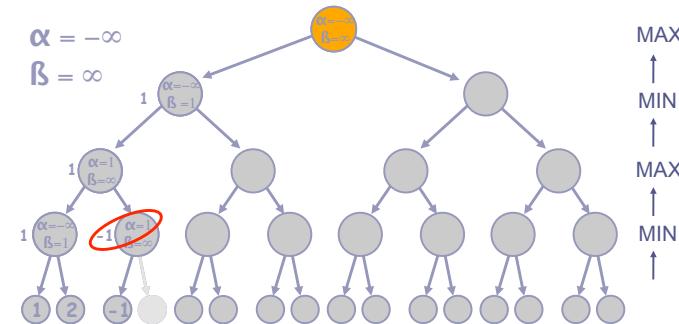
Alpha Beta Pruning: un esempio



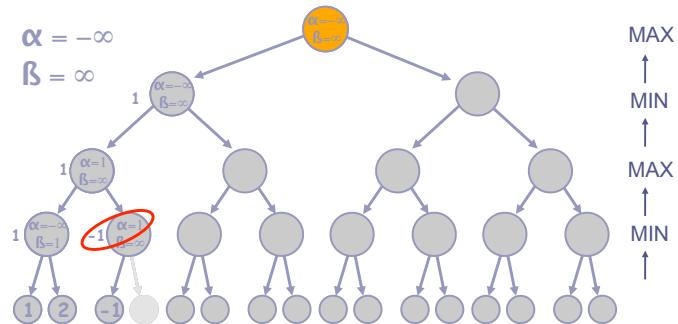
Alpha Beta Pruning: un esempio



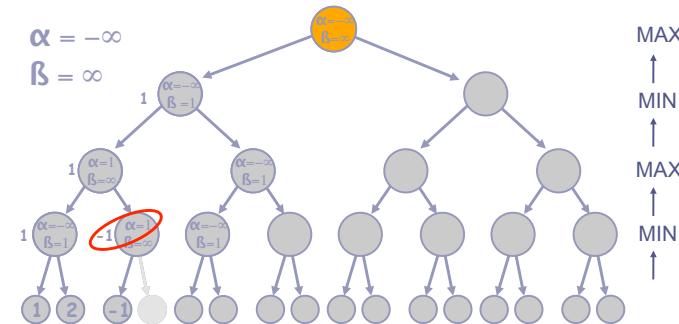
Alpha Beta Pruning: un esempio



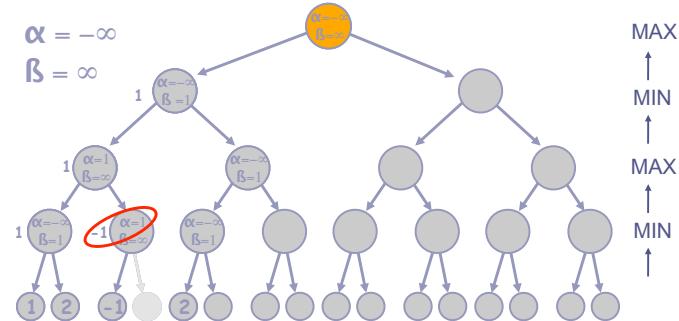
Alpha Beta Pruning: un esempio



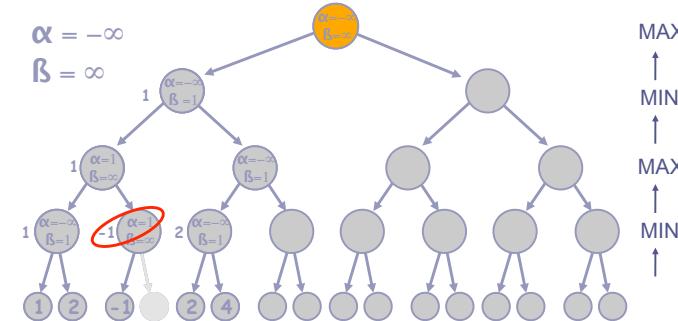
Alpha Beta Pruning: un esempio



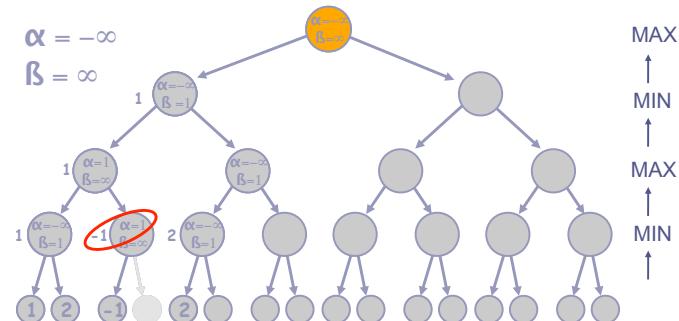
Alpha Beta Pruning: un esempio



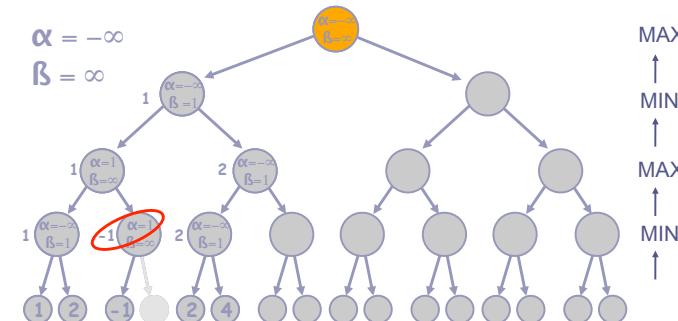
Alpha Beta Pruning: un esempio



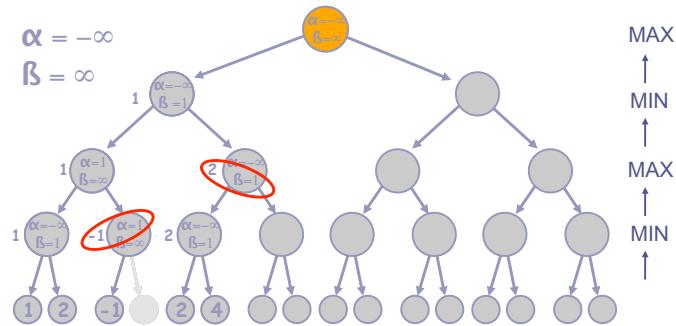
Alpha Beta Pruning: un esempio



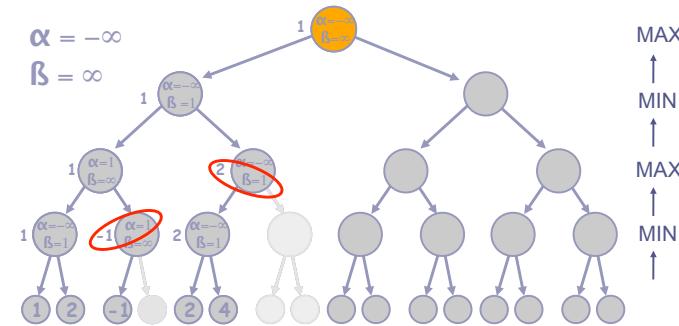
Alpha Beta Pruning: un esempio



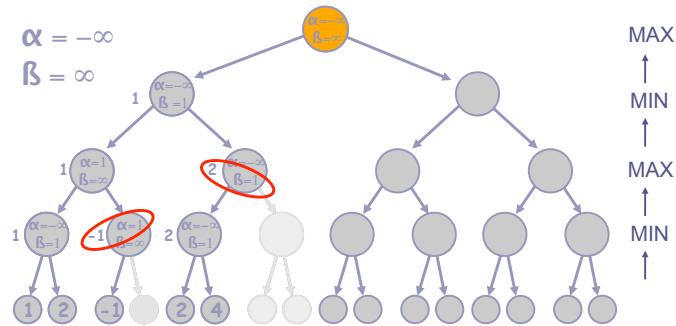
Alpha Beta Pruning: un esempio



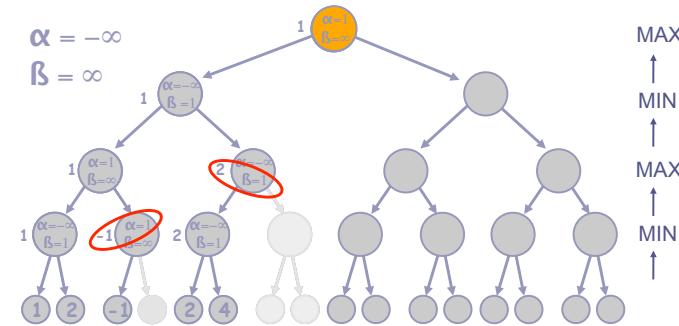
Alpha Beta Pruning: un esempio



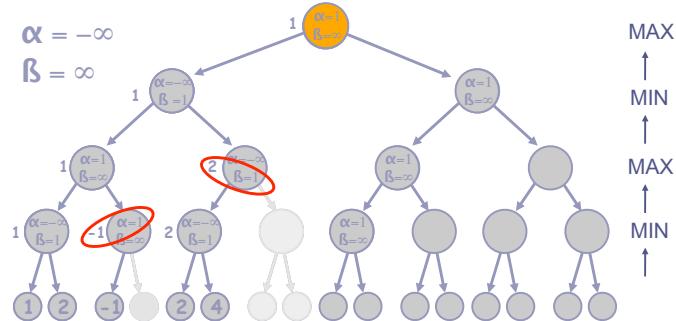
Alpha Beta Pruning: un esempio



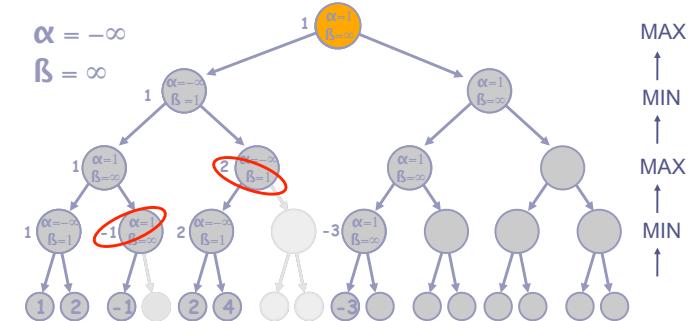
Alpha Beta Pruning: un esempio



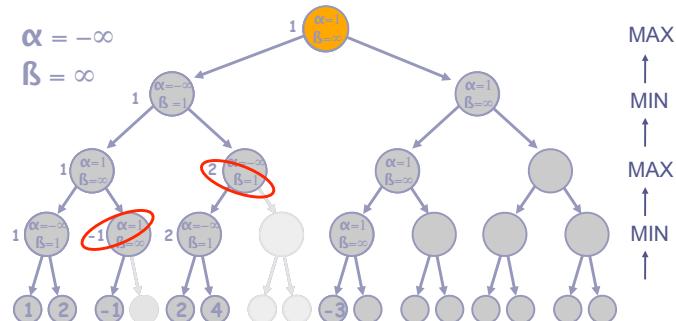
Alpha Beta Pruning: un esempio



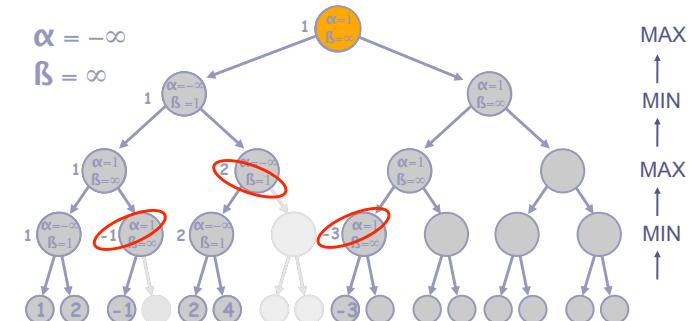
Alpha Beta Pruning: un esempio



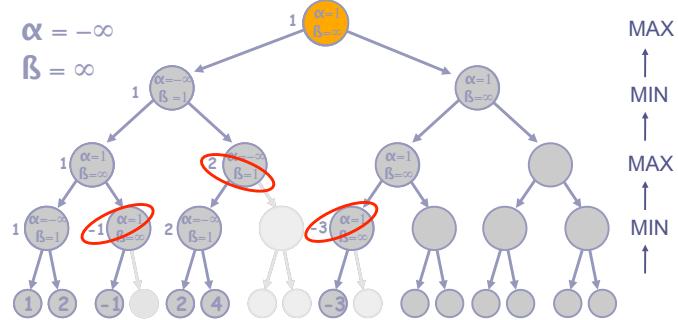
Alpha Beta Pruning: un esempio



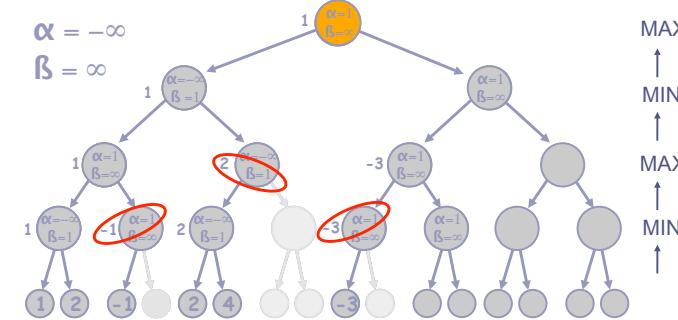
Alpha Beta Pruning: un esempio



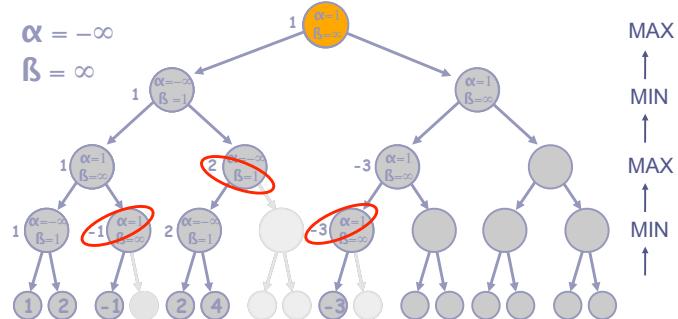
Alpha Beta Pruning: un esempio



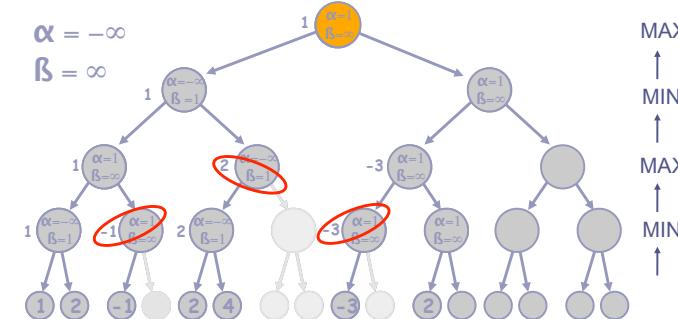
Alpha Beta Pruning: un esempio



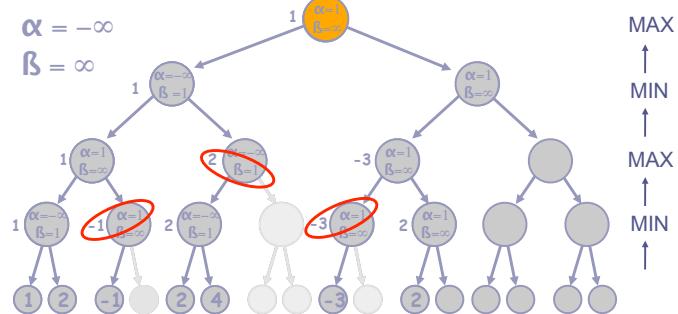
Alpha Beta Pruning: un esempio



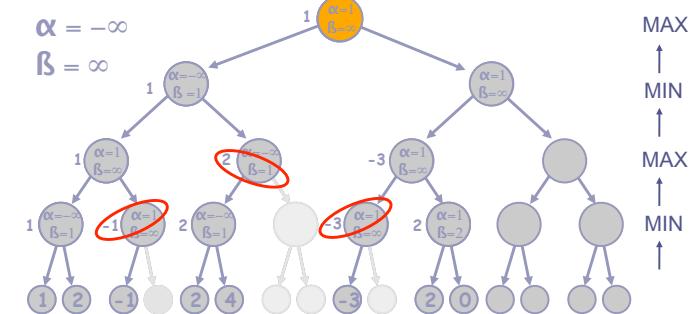
Alpha Beta Pruning: un esempio



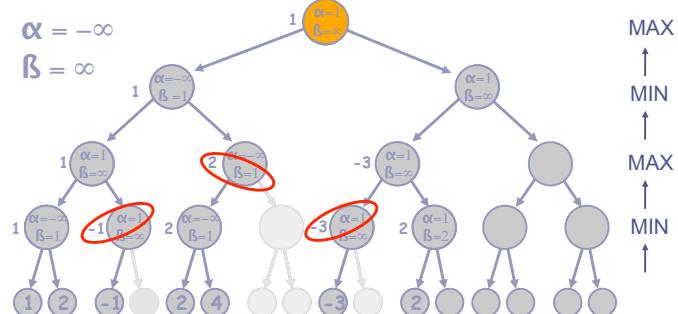
Alpha Beta Pruning: un esempio



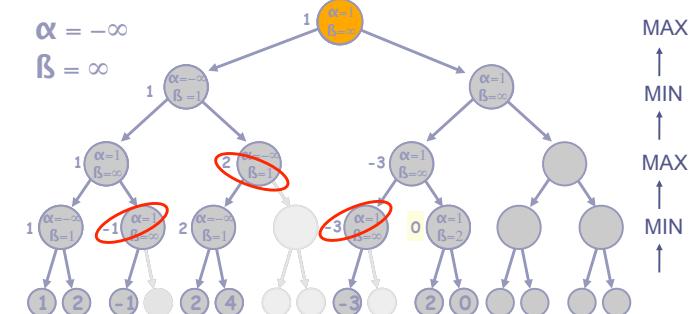
Alpha Beta Pruning: un esempio



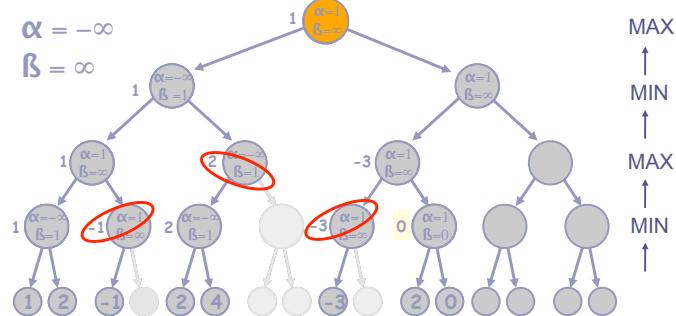
Alpha Beta Pruning: un esempio



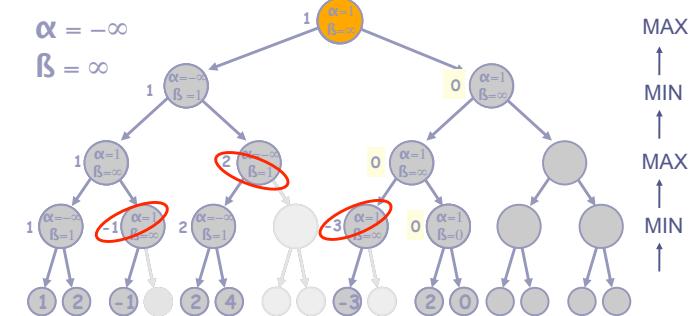
Alpha Beta Pruning: un esempio



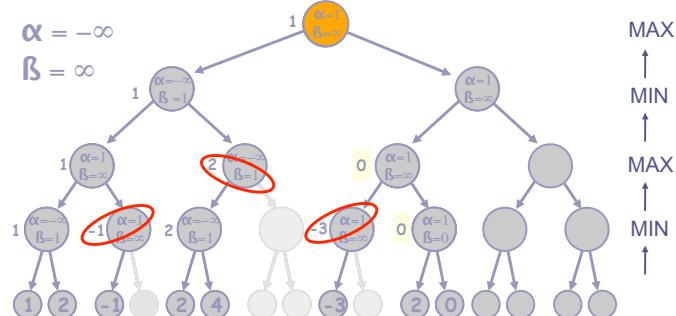
Alpha Beta Pruning: un esempio



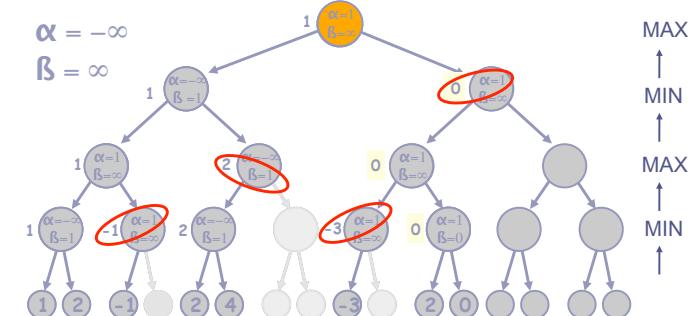
Alpha Beta Pruning: un esempio



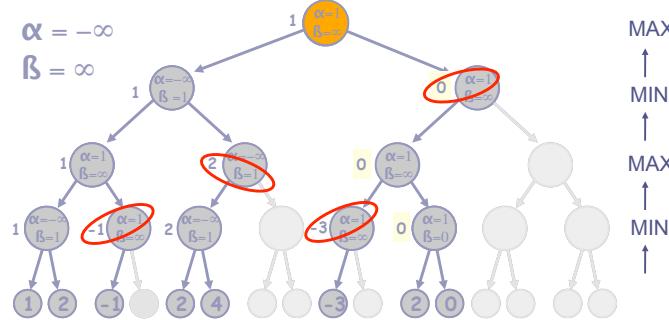
Alpha Beta Pruning: un esempio



Alpha Beta Pruning: un esempio



Alpha Beta Pruning: un esempio



Pseudo Codifica Alpha Beta Cutoff (2/3)

```

function MIN-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
  inputs: 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 CUTOFF-TEST(state, depth) then return EVAL(state)
   $v \leftarrow +\infty$ 
  for  $s$  in SUCCESSORS(state) do
     $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(s, \alpha, \beta))$ 
    if  $v \leq \alpha$  then return  $v$     $\leftarrow$  taglio  $\beta$ 
     $\beta \leftarrow \text{MIN}(\beta, v)$ 
  return  $v$ 

```

Pseudo Codifica Alpha Beta Cutoff (1/3)

```

function MAX-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
  inputs: 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 CUTOFF-TEST(state, depth) then return EVAL(state)
   $v \leftarrow -\infty$ 
  for  $s$  in SUCCESSORS(state) do
     $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(s, \alpha, \beta))$ 
    if  $v \geq \beta$  then return  $v$     $\leftarrow$  taglio  $\alpha$ 
     $\alpha \leftarrow \text{MAX}(\alpha, v)$ 
  return  $v$ 

```

Pseudo Codifica Alpha Beta Cutoff (3/3)

```

AlphaBeta(GamePosition game) {
  return MaxMove(game, -Inf, Inf);
}

```

<pre> MaxMove (GamePosition game, Integer alpha, Integer beta) { if (GameEnded(game) DepthLimitReached()) { return EvalGameState(game, MAX); } else { best_move <- {}; moves <- GeneratesMoves(game); ForEach moves { move <- MinMove(ApplyMove(game), alpha, beta); if (Value(move) > Value(best_move)) { best_move <- move; alpha <- Value(move); } if (beta>alpha) return best_move; } return best_move; } } </pre>	<pre> MinMove (GamePosition game, Integer alpha, Integer beta) { if (GameEnded(game) DepthLimitReached()) { return EvalGameState(game, MIN); } else { best_move <- {}; moves <- GeneratesMoves(game); ForEach moves { move <- MaxMove(ApplyMove(game), alpha, beta); if (Value(move) < Value(best_move)) { best_move <- move; beta <- Value(move); } if (beta<alpha) return best_move; } return best_move; } } </pre>
---	--

Implementazione AlphaBeta (1/3)

```

public void makeAlphaBetaMove() {
    getAlphaBetaValue(presentState);

    GameState nextState = (GameState) presentState.get("next");
    if (nextState == null) {
        throw new RuntimeException("Alpha Beta Move failed");
    }
    makeMove(presentState, nextState.get("moveMade"));
}

public int getAlphaBetaValue(GameState state) {
    if (getPlayerToMove(state).equalsIgnoreCase("X")) {
        AlphaBeta initial = new AlphaBeta(Integer.MIN_VALUE, Integer.MAX_VALUE);
        int max = maxValue(state, initial);
        return max;
    } else {
        AlphaBeta initial = new AlphaBeta(Integer.MIN_VALUE, Integer.MAX_VALUE);
        return minValue(state, initial);
    }
}

```

Implementazione AlphaBeta (2/3)

```

protected int maxValue(GameState state, AlphaBeta ab) {
    int v = Integer.MIN_VALUE;
    if (terminalTest(state)) {
        return computeUtility(state);
    } else {
        ArrayList successorList = getSuccessorStates(state);
        for (int i = 0; i < successorList.size(); i++) {
            GameState successor = (GameState) successorList.get(i);
            int minimumValueOfSuccessor = minValue(successor, ab.copy());
            if (minimumValueOfSuccessor > v) {
                v = minimumValueOfSuccessor;
                state.put("next", successor);
            }
            if (v >= ab.beta()) {
                return v;
            }
            ab.setAlpha(Math.max(ab.alpha(), v));
        }
    }
    return v;
}

```

Implementazione AlphaBeta (3/3)

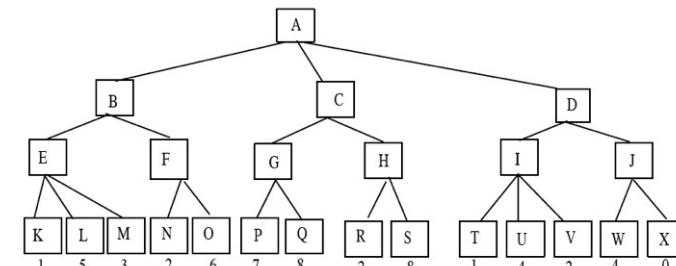
```

public int minValue(GameState state, AlphaBeta ab) {
    int v = Integer.MAX_VALUE;
    if (terminalTest(state)) {
        return computeUtility(state);
    } else {
        ArrayList successorList = getSuccessorStates(state);
        for (int i = 0; i < successorList.size(); i++) {
            GameState successor = (GameState) successorList.get(i);
            int maximumValueOfSuccessor = maxValue(successor, ab.copy());
            if (maximumValueOfSuccessor < v) {
                v = maximumValueOfSuccessor;
                state.put("next", successor);
            }
            if (v <= ab.alpha()) {return v;}
            ab.setBeta(Math.min(ab.beta(), v));
        }
    }
    return v;
}

```

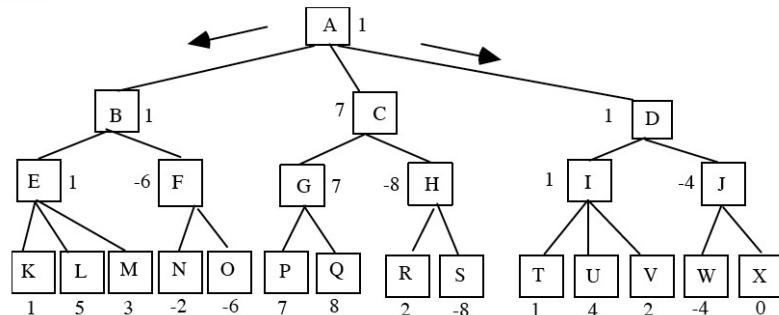
Esercizio 1 Alpha Beta

- Si consideri il seguente albero di gioco dove i punteggi sono tutti dal punto di vista del primo giocatore.
- Supponendo che il primo giocatore sia MIN, quale mossa dovrebbe scegliere? (applicare MIN-MAX e poi Alpha-Beta)



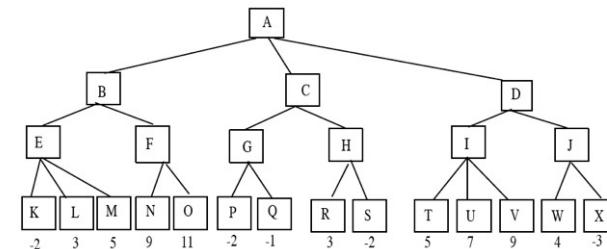
Soluzione (1/2)

Min-Max



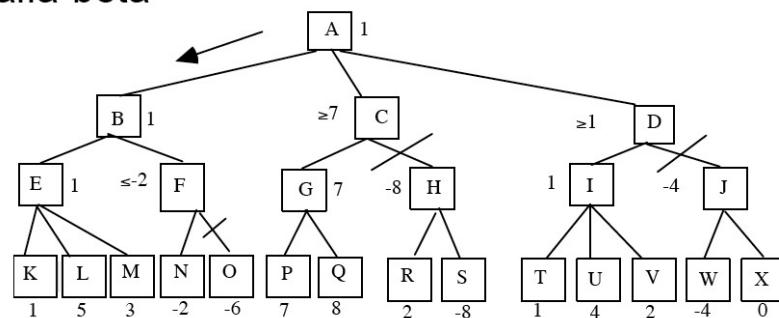
Esercizio 2 Alpha Beta

- Si consideri il seguente albero di gioco dove i punteggi sono tutti dal punto di vista del primo giocatore.
- Supponendo che il primo giocatore sia MAX, quale mossa dovrebbe scegliere? (applicare MIN-MAX e poi Alpha-Beta)

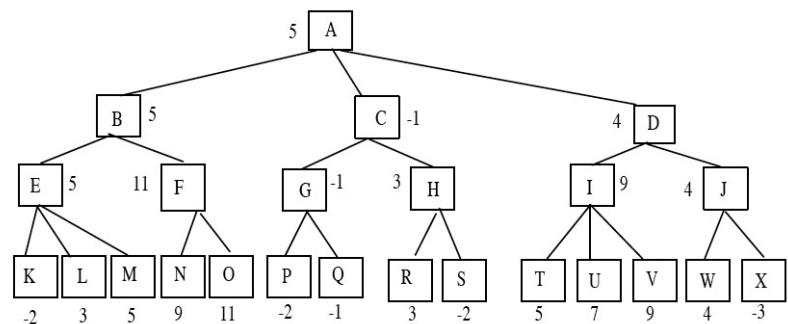


Soluzione (2/2)

Tagli alfa-beta

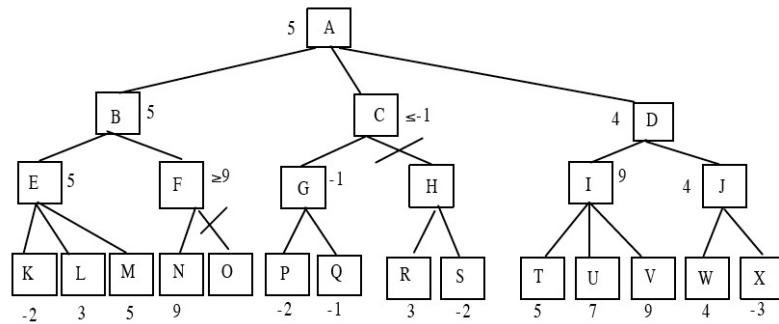


Min-Max



Soluzione (2/2)

Tagli alfa beta:



Tic Tac Toe Alpha-Beta

```
int chooseMove(Side s, int& bestRow, int& bestColumn, int alpha, int beta)
{
    Side opp(s==COMPUTER ? HUMAN : COMPUTER);

    int value(s==COMPUTER ? alpha : beta);
    int simpleEval(positionValue());

    if(simpleEval==UNCLEAR)
        return simpleEval;

    for (int r(0); r<board.numrows(); ++r)
        for (int c(0); c<board.numcols(); ++c)
            if (squareIsEmpty(r,c))
            {
                place(r,c,s);
                int dc;
                int reply(chooseMove(opp, dc, dc, alpha, beta));
                place(r,c,EMPTY);
                if (s==COMPUTER && reply>value || s==HUMAN && reply<value)
                {
                    value = reply;
                    if (s==COMPUTER) alpha = value;
                    else beta = value;
                    bestRow = r;
                    bestColumn=c;
                    if (alpha>beta) return value;
                }
            }
    return value;
}
```

Tic Tac Toe Min Max

```
int chooseMove(Side s, int& bestRow, int& bestColumn)
{
    Side opp(s==COMPUTER ? HUMAN : COMPUTER);

    int value(s==COMPUTER ? HUMAN_WIN : COMPUTER_WIN);
    int simpleEval(positionValue());

    if(simpleEval!=UNCLEAR)
        return simpleEval;

    for (int r(0); r<board.numrows(); ++r)
        for (int c(0); c<board.numcols(); ++c)
            if (squareIsEmpty(r,c))
            {
                place(r,c,s);
                int dc;
                int reply(chooseMove(opp, dc, dc));
                place(r,c,EMPTY);
                if (s==COMPUTER && reply>value || s==HUMAN && reply<value)
                {
                    value = reply;
                    bestRow = r;
                    bestColumn=c;
                }
            }
    return value;
}
```

Esercizio NIM (1/2)

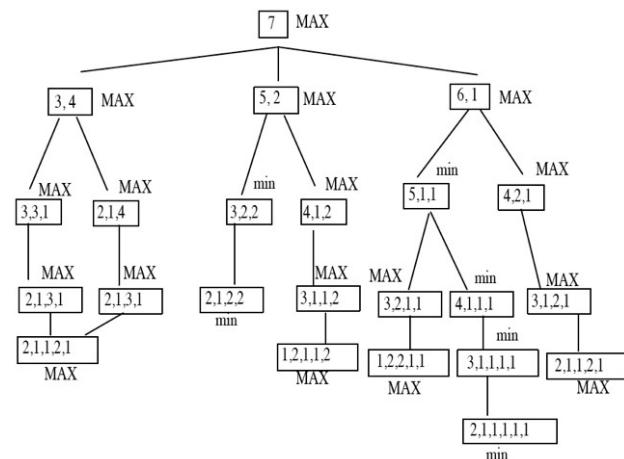
- ➊ Due giocatori hanno davanti una singola pila di oggetti (ad esempio una pila di monete).
- ➋ Il primo giocatore divide la pila in due pile separate che devono essere diverse. Poi continua il secondo lavorando su una a scelta delle due pile e così via.
- ➌ Il gioco termina solo quando in ogni pila ci sono una o due monete. A questo punto il primo giocatore che non può muovere ha perso.
- ➍ Si applichi la procedura MIN-MAX e si mostri lo spazio di ricerca nel caso in cui cominci MIN con una pila di 7 monete.

Esercizio NIM (2/2)

- Si commenti lo spazio risultante.
- Ogni nodo dello spazio di ricerca rappresenta una configurazione: in particolare X,Y,Z indicano che si hanno tre pile corrispondenti ciascuna ad un numero, che indica il numero di monete contenute.
- Esempio di un possibile stato:



Soluzione



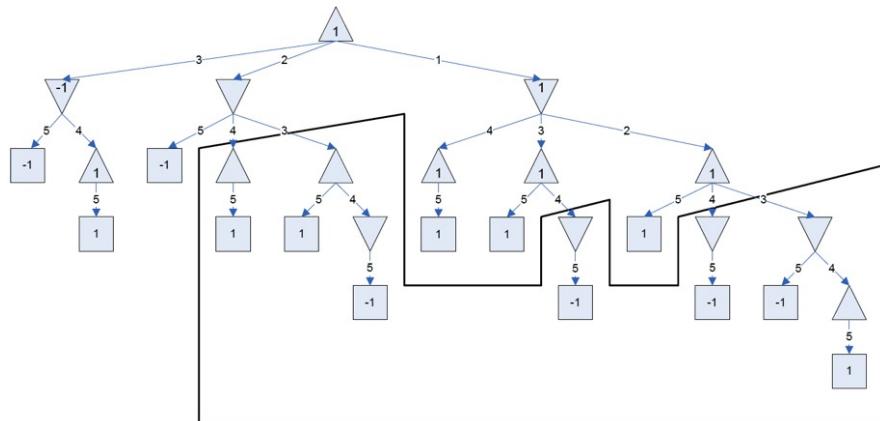
Esercizio: gioco del 5 (1/2)

- Ci sono due giocatori che, alternandosi, scelgono un numero. Il giocatore che gioca al turno iniziale può scegliere un numero compreso tra {1,2,3}. Dal secondo turno in avanti, il giocatore che gioca deve scegliere un numero strettamente maggiore di quello scelto al turno precedente dal suo avversario, ma distante non più di 3.
- Costruire l'albero di gioco assegnando il nodo max al giocatore che gioca al turno iniziale e applicando le azioni dei giocatori dal numero più grande che possono scegliere al numero più piccolo.

Esercizio: gioco del 5 (2/2)

- La funzione di utilità assegna +1 alle foglie che corrispondono alla vittoria di max e valore -1 a quelle che corrispondono alla vittoria di min.
- Risolvere il problema utilizzando l'algoritmo min max con potatura alpha-beta. Si indichi la porzione di albero non visitata dall'algoritmo. Non si richiede di specificare i valori di alpha e beta lungo l'albero.
- Indicare infine il numero scelto da max nel turno iniziale del gioco.

Soluzione



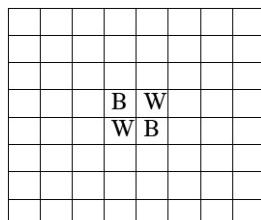
Il giocatore max sceglie 1

Othello

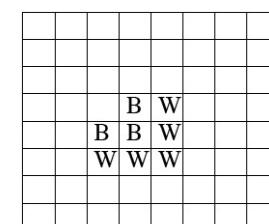
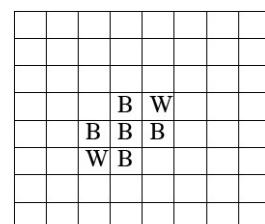
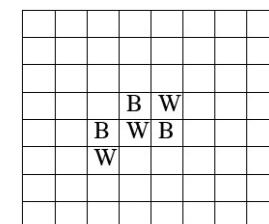
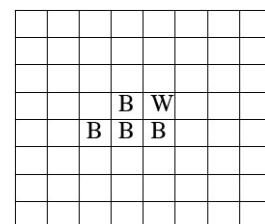
- E' possibile incastrare pedine in orizzontale, in verticale e in diagonale e, a ogni mossa, si possono girare pedine in una o più direzioni.
- Sono ammesse solo mosse con le quali si gira almeno una pedina, se non è possibile farlo si salta il turno.
- Quando nessuno dei giocatori ha la possibilità di muovere, si contano le pedine e si assegna la vittoria a chi ne ha il maggior numero

Othello

- Si muove alternativamente (inizia il nero) appoggiando una nuova pedina in una casella vuota in modo da imprigionare, tra la pedina che si sta giocando e quelle del proprio colore già presenti sulla scacchiera, una o più pedine avversarie. A questo punto le pedine imprigionate devono essere rovesciate e diventano di proprietà di chi ha eseguito la mossa.



Una possibile evoluzione



Othello

Utility Function

- Fondamentale avere il maggior numero di dischi: **un punto per ogni disco del proprio colore**

- Matrice di utilità:

- Gli angoli sono molto importanti (una volta conquistati non possono cambiare): **+45 punti per ogni angolo**
- I lati sono 'abbastanza' importanti....

```
45, 36, 25, 4, 4, 25, 36, 45,
36, 34, 23, 3, 3, 23, 34, 36,
25, 23, 18, 2, 2, 18, 23, 25,
12, 10, 6, 1, 1, 6, 10, 12,
12, 10, 6, 1, 1, 6, 10, 12,
25, 23, 18, 2, 2, 18, 23, 25,
36, 34, 23, 3, 3, 23, 34, 36,
45, 36, 25, 4, 4, 25, 36, 45
```

```
#define TRUE 1
#define FALSE 0

#define EMPTY 0
#define BLACK 1
#define WHITE 2
#define OTHER(x) (3-(x))

#define CONV_21(x,y) (((y) << 3)+(x))
#define ON_BOARD(x,y) ((x) >= 0 && (x) < 8 && (y) >= 0 && (y) < 8)

const int DIRECTION[8][2] =
{ {1, 0}, {1, 1}, {0, 1}, {-1, 1}, {-1, 0}, {-1, -1}, {0, -1}, {1, -1} };
```

```
void copyArray (int *a,int *b)
{
    int i;

    for (i = 0; i < 64; i++)
        b[i] = a[i];
}
```

Implementazione Othello

Funzioni di supporto

- void makeMove (int *b, int *move, int wt)
- int legalMove (int *b, int *move, int wt)
- int countFlippedPieces (int *b, char x, char y, int wt)
- int findLegalMoves (int *b, int *lm, int wt)
- int findBestMove (int *b, int wt, int *move, int(*boardEvalFunc)())
- int checkWinner (int *board)
- int countPiecesWeighted (int *board, int player)
- int countPieces (int *board, int player)
- int countPiecesWeightedPlus (int *board, int player)
- void *getBoardEvalFunc (int id)

Othello (2/10)

makeMove

```
void makeMove (int *b, int *move, int wt)
{
    int x = move[0];
    int y = move[1];
    int place = CONV_21 (x, y);
    int i, dir;
    for (dir = 0; dir < 8; dir++)
    {
        int dx = DIRECTION[dir][0];
        int dy = DIRECTION[dir][1];
        int tx = x + 2 * dx;
        int ty = y + 2 * dy;
        if (!ON_BOARD (tx, ty) || b[CONV_21 (x + dx, y + dy)] != OTHER (wt))
            continue;
        while (ON_BOARD (tx, ty) && b[CONV_21 (tx, ty)] == OTHER (wt)) {
            tx += dx;
            ty += dy;
        }
        if (ON_BOARD (tx, ty) && b[CONV_21 (tx, ty)] == wt) {
            tx -= dx;
            ty -= dy;
            while (b[CONV_21 (tx, ty)] == OTHER (wt)) {
                b[CONV_21 (tx, ty)] = wt;
                tx -= dx;
                ty -= dy;
            }
            b[place] = wt;
        }
    }
}
```

Othello (3/10) legalMove

```
int legalMove (int *b, int *move, int wt)
{
    int x = move[0];
    int y = move[1];
    int place = CONV_21 (x, y);
    if (x < 0 || x > 7 || y < 0 || y > 7)
        return FALSE;
    if (b[place] != EMPTY)
        return FALSE;
    int result = FALSE;
    int dir;
    for (dir = 0; dir < 8; dir++)
    {
        int dx = DIRECTION[dir][0];
        int dy = DIRECTION[dir][1];
        int tx = x + 2 * dx;
        int ty = y + 2 * dy;
        if (!ON_BOARD (tx, ty)
            || b[CONV_21 (x + dx, y + dy)] != OTHER (wt))
            continue;
        while (ON_BOARD (tx, ty) && b[CONV_21 (tx, ty)] == OTHER (wt))
        {
            tx += dx;
            ty += dy;
        }
        if (ON_BOARD (tx, ty) && b[CONV_21 (tx, ty)] == wt)
        {
            result = TRUE;
            break;
        }
    }
    return result;
}
```

Othello (4/10) countFlippedPieces

```
int countFlippedPieces (int *b, char x, char y, int wt)
{
    char place = CONV_21 (x, y);
    if (x < 0 || x > 7 || y < 0 || y > 7)
        return 0;
    if (b[place] != EMPTY)
        return 0;
    int dir;
    int k = 1;
    int sum = 0;

    for (dir = 0; dir < 8; dir++)
    {
        int dx = DIRECTION[dir][0];
        int dy = DIRECTION[dir][1];
        int tx = x + 2 * dx;
        int ty = y + 2 * dy;

        if (!ON_BOARD (tx, ty)
            || b[CONV_21 (x + dx, y + dy)] != OTHER (wt))
            continue;
        while (ON_BOARD (tx, ty) && b[CONV_21 (tx, ty)] == OTHER (wt))
        {
            tx += dx;
            ty += dy;
            k++;
        }
        if (ON_BOARD (tx, ty) && b[CONV_21 (tx, ty)] == wt)
            sum += k;
    }
    return sum;
}
```

Versione C

Othello (5/10) findLegalMoves

```
int findLegalMoves (int *b, int *lm, int wt)
{
    int result = FALSE;
    int x, y;
    int *move = (int *) malloc (2 * sizeof (int));
    for (y = 0; y < 8; y++)
    {
        for (x = 0; x < 8; x++)
        {
            move[0] = x;
            move[1] = y;
            lm[CONV_21 (x, y)] = legalMove (b, move, wt);
            if (lm[CONV_21 (x, y)])
                result = TRUE;
        }
    }
    free(move);
    return result;
}
```

Versione C

Othello (6/10) findBestMove

```
int findBestMove (int *b, int wt, int *move, int (*boardEvalFunc)())
{
    int x, y;
    int resultMax = -1;
    int resultTemp;
    move[0] = -1;
    move[1] = -1;

    int* lm = (int*)malloc(64*sizeof(int));
    findLegalMoves(b, lm, wt);
    int *a = (int *) malloc (64 * sizeof (int));
    int *tempMove = (int*)malloc(2*sizeof(int));
    for (y = 0; y < 8; y++) {
        for (x = 0; x < 8; x++) {
            if(lm[CONV_21(x,y)]){

copyArray(b,a);

tempMove[0]=x;
tempMove[1]=y;

makeMove(a,tempMove,wt);

resultTemp = boardEvalFunc(a,wt);

if (resultTemp > 0
    && resultTemp > resultMax)

{
    resultMax = resultTemp;
    move[0] = x;
    move[1] = y;
}

}
}
free(lm);
free(a);
free(tempMove);
return resultMax;
}
```

Versione C

Othello (7/10) checkWinner

```
int checkWinner (int *board)
{
    int x, y, black, white;
    black = white = 0;

    for (x = 0; x < 8; x++)
        for (y = 0; y < 8; y++)
    {
        if (board[CONV_21 (x, y)] == BLACK)
            black++;
        else if (board[CONV_21 (x, y)] == WHITE)
            white++;
    }

    if (black > white)
        return BLACK;
    else if (white > black)
        return WHITE;

    else
        return EMPTY;
}
```

Versione C

Othello (8/10) countPiecesWeighted

```
int countPiecesWeighted (int *board, int player)
{
    int weight[64] = { 45, 36, 25, 4, 4, 25, 36, 45,
                      36, 34, 23, 3, 3, 23, 34, 36,
                      25, 23, 18, 2, 2, 18, 23, 25,
                      12, 10, 6, 1, 1, 6, 10, 12,
                      12, 10, 6, 1, 1, 6, 10, 12,
                      25, 23, 18, 2, 2, 18, 23, 25,
                      36, 34, 23, 3, 3, 23, 34, 36,
                      45, 36, 25, 4, 4, 25, 36, 45 };

    int pieces = 0;
    int i;
    for (i = 0; i < 64; i++)
        if (board[i] == player)
            pieces += weight[i];
    return pieces;
}
```

Versione C

Othello (9/10) countPiecesWeightedPlus

```
/* Array contenente le mosse consentite */
int *lm = (int *) malloc (64 * sizeof (int));

/* Verifica se ci sono mosse da effettuare */
int areLegal = findLegalMoves (board, lm, player);

float availableMoves = 0.0;
for(i=0;i<64;i++)
    if(lm[i])
        availableMoves+=weight[i];

int result2 = (int)availableMoves*0.45;

free(lm);

return pieces + result2;
```

Versione C

Othello (10/10) countPieces, getBoardEvalFunc

```
void *getBoardEvalFunc (int id)
{
    int (*boardEvalFunc) ();
    switch (id)
    {
        case 1:
            boardEvalFunc = countPiecesWeighted;
            break;
        case 2:
            boardEvalFunc = countPieces;
            break;
        case 3:
            boardEvalFunc = countPiecesWeightedPlus;
            break;
    }
    return (void *) boardEvalFunc;
}
```

Versione C

Implementazione Othello Min Max con potatura alpha-beta

```

■ void makeMove (int *b, int *move, int wt)
■ int minMaxMove (int *b, int turnPassed, int wt, int *move, int (*f) ())
■ int getMax (Board * b, int depth, int *move, int beta, int (*f) ())
■ int getMin (Board * b, int depth, int *move, int alpha, int (*f) ())

```

Othello Min Max (1/8)

```

#define DEPTH 3
#define MAX_VALUE 1000000
#define MIN_VALUE -1000000

```

```

typedef struct Board
{
    int *board;
    int emptyPlaces;
    int turnPassed;
    int wt;
} Board;

```

Versione C

Othello Min Max (2/8) minMaxMove

```

int minMaxMove (int *b, int turnPassed, int wt, int *move, int (*f) ())
{
    move[0] = -1;
    move[1] = -1;

    /* Conta le caselle vuote */
    int x, y;
    int emptyPlaces = 0;
    for (x = 0; x < 8; x++)
        for (y = 0; y < 8; y++)
            if (b[CONV_21 (x, y)] == EMPTY)
                emptyPlaces++;

    /* Inizializza la struttura corrispondente all'othelliera */
    Board *struct_board = (Board *) malloc (sizeof (Board));
    struct_board->board = b;
    struct_board->turnPassed = turnPassed;
    struct_board->emptyPlaces = emptyPlaces;
    struct_board->wt = wt;

    return getMax (struct_board, DEPTH, move, MAX_VALUE, f);
}

```

Versione C

Othello Min Max (3/8) getMax (continua)

```

int getMax (Board * b, int depth, int *move, int beta, int (*f) ())
{
    int alpha;
    int result = -1;
    Board *minBoard;
    int *maxMoveItem = (int *) malloc (2 * sizeof (int));
    /* Indica se precedentemente e' stato saltato un turno */
    int turnPassed = b->turnPassed;
    /* Ricava la scacchiera */
    int *board = b->board;
    /* Ricava il giocatore corrente */
    int currentMaxPlayer = b->wt;
    /* Ricava il numero di caselle libere sulla scacchiera */
    int emptyPlaces = b->emptyPlaces;
    /* Array contenente le mosse consentite */
    int *lm = (int *) malloc (64 * sizeof (int));
    /* Verifica se ci sono mosse da effettuare */
    int areLegal = findLegalMoves (board, lm, currentMaxPlayer);
    int winner;
    int *minMove = (int *) malloc (2 * sizeof (int));

    /* Se non ci sono piu' caselle libere */
    if (emptyPlaces == 0)
    {
        /* Determina il vincitore */
        winner = checkWinner (board);
        if (winner == currentMaxPlayer) //Vince Max
            result = 10000;
        else if (winner == OTHER (currentMaxPlayer)) // Vince Min
            result = -10000;
        else
            result = 0; //Max e Min pareggiano
    }
    else if (depth == 0)
    {
        /* Se si e' raggiunto il massimo livello di profondita' */
        result = f (b->board, currentMaxPlayer);
    }
}

```

Versione C

Othello Min Max (4/8) getMax (continua)

Versione C

```

else
{
    /* Ci sono ancora caselle libere */
    if (areLegal)
    {
        /* Se e' possibile effettuare almeno una mossa */

        int maxMin = alpha = MIN_VALUE;
        int maxMinTemp;
        int x, y;

        for (x = 0; x < 8 && alpha < beta; x++)
            for (y = 0; y < 8 && alpha < beta; y++)
                if (lm[CONV_21 (x, y)])
                {
                    minBoard = copyBoard (b);
                    maxMoveTemp[0] = x;
                    maxMoveTemp[1] = y;
                    /* Min esegue la mossa */
                    makeMove (minBoard->board,
                              maxMoveTemp,
                              currentMaxPlayer);
                    /* Decrementa il numero di caselle vuote */
                    minBoard->emptyPlaces =
                        emptyPlaces - 1;
                    result = maxMin;
                }
}

```

Othello Min Max (5/8) getMax

Versione C

```

else
{
    /* Se non ci sono piu' mosse disponibili salta il turno */
    if (turnPassed == 1)
    {
        /* Determina il vincitore */
        winner = checkWinner (board);

        if (winner == currentMaxPlayer) //Vince Max
            result = 10000;
        else if (winner == OTHER (currentMaxPlayer)) // Vince Min
            result = -10000;
        else
            result = 0; //Max e Min pareggiano
    }
}

```

```

else
{
    /* Ci sono ancora caselle libere */
    if (areLegal)
    {
        /* Salta il turno */
        minBoard->turnPassed = 1;

        /* Tocca al giocatore Min */
        minBoard->wt = OTHER (currentMaxPlayer);

        int *minMove =
            (int *) malloc (2 * sizeof (int));

        result = getMin (minBoard, depth, minMove,
                         MIN_VALUE, f);
    }
}

```

```

free(maxMoveTemp);
free(lm);
free(minMove);
free(b);
return result;
}

```

Othello Min Max (6/8) getMin (continua)

Versione C

```

int getMin (Board * b, int depth, int *move, int alpha, int (*f) ())
{
    int beta;
    int result;
    Board *maxBoard;
    int *minMoveTemp = (int *) malloc (2 * sizeof (int));
    int *maxMove = (int *) malloc (2 * sizeof (int));
    /* Indica se precedentemente e' stato saltato un turno */
    int turnPassed = b->turnPassed;
    /* Ricava la scacchiera */
    int *board = b->board;
    /* Ricava il giocatore corrente */
    int currentMinPlayer = b->wt;
    /* Ricava il numero di caselle libere sulla scacchiera */
    int emptyPlaces = b->emptyPlaces;
    /* Array contenente le mosse consentite */
    int *lm = (int *) malloc (64 * sizeof (int));
    /* Verifica se ci sono mosse da effettuare */
    int areLegal = findLegalMoves (board, lm, currentMinPlayer);
    int winner;
}

/* Se non ci sono piu' caselle libere */
if (emptyPlaces == 0)
{
    /* Determina il vincitore */
    winner = checkWinner (board);

    if (winner == currentMinPlayer) //Vince Max
        result = -1000;
    else if (winner == OTHER (currentMinPlayer)) // Vince Min
        result = 1000;
    else
        result = 0; //Max e Min pareggiano
}
else if (depth == 0)
{
    /* Se si e' raggiunto il massimo livello di profondita' */
    result = result = f (b->board, OTHER (currentMinPlayer));
}

```

Othello Min Max (7/8) getMin (continua)

Versione C

```

else
{
    /* Tocca a Max */
    maxBoard->wt =
        OTHER
        (currentMinPlayer);
    minMaxTemp =
        getMax (maxBoard,
                depth - 1,
                maxMove, beta,
                f);
    /* Salva la mossa e' piu' vantaggiosa delle precedenti */
    if (minMaxTemp < minMax)
    {
        beta = minMax =
            minMaxTemp;
        move[0] =
            minMoveTemp
            [0];
        move[1] =
            minMoveTemp
            [1];
    }
}

```

```

result = minMax;
}

```

Othello Min Max (8/8)

getMin

Versione C

```

else
{
    /* Se non ci sono piu' mossa disponibili salta il turno */

    /* Verifica se e' stato gia' saltato il turno precedente */
    if (turnPassed == 1)
    {
        /* Determina il vincitore */
        winner = checkWinner (board);

        if (winner == currentMinPlayer)    //Vince Min
            result = -1000;
        else if (winner == OTHER (currentMinPlayer)) // Vince Max
            result = 1000;
        else
            result = 0; //Max e Min pareggiano
    }
    else
    {
        //printf("getMin() - salta il turno\n");

        maxBoard = copyBoard (b);
        /* Salta il turno */
        maxBoard->turnPassed = 1;

        /* Tocca al giocatore Max */
        maxBoard->wt = OTHER (currentMinPlayer);
        result = getMax (maxBoard, depth - 1, move,
                         MAX_VALUE, f);

        free(minMoveTemp);
        free(l1m);
        free(maxMove);
        free(b);
    }
}

return result;
}

```

Esercizio A* (1/3)

Si formuli il seguente rompicapo come un problema di ricerca.

Si abbiano a disposizione due vasi (inizialmente vuoti) rispettivamente da 3 e 5 litri e, riempiendoli d'acqua, svuotandoli e travasando acqua da un vaso all'altro, si vuole ottenere 1 litro d'acqua in uno dei due vasi.

Esercizio A* (2/3)

■ Azioni:

- travasare l'acqua dal vaso 1 al vaso 2 fino al riempimento di quest'ultimo.
- travasare l'acqua dal vaso 2 al vaso 1 fino al riempimento di quest'ultimo.
- svuotare completamente il vaso 1.
- svuotare completamente il vaso 2.
- riempire completamente il vaso 1
- riempire completamente il vaso 2.

■ Ogni azione ha costo unitario. Lo stato è rappresentato da una coppia $\langle l_1, l_2 \rangle$ dove l_1 è la quantità di acqua nel vaso 1 e l_2 è la quantità di acqua nel vaso 2.

Esercizio A* (3/3)

A partire dalla rappresentazione dello stato, si definisca il test obiettivo.

(1) Si applichi A* con eliminazione degli stati ripetuti, considerando $h(\langle l_1, l_2 \rangle) = \min\{ |l_1 - 1|, |l_2 - 1| \}$. Si riporti l'albero di ricerca costruito applicando le azioni nell'ordine riportato sopra e preferendo, a parità di altro, il nodo generato prima. Per ogni nodo, si indichi l'ordine di espansione, il valore della funzione euristica e quello della funzione di valutazione e se il nodo è stato eliminato. Si applichino a un nodo solo le azioni che modificano lo stato corrispondente al nodo (per esempio, non si applichi l'azione che svuota il vaso 1 al nodo radice).

(2) h è ammissibile?

Soluzione

- (1). $l_1 = 1 \quad || \quad l_2 = 1$
(3). h non ammissibile

