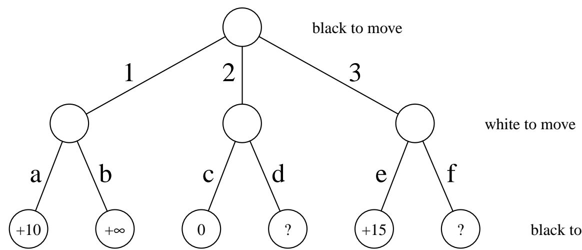
Game tree searching

- 

Game tree searching

- the diagram shows that it is blacks turn to move
  - a positive score for black indicates a good move
  - $+\infty$ indicates a win for black
  - 0 indicates a neutral position
  - any value <0 indicates a good position for white

Game tree searching

- we assume that white evaluates the board in exactly the same way as black

- if black chooses move 1, white will respond with move b
  - we can see that white is attempting to minimise the score
  - whereas black is attempting to maximise the score

Minimax algorithm

- ```
def minimax (colour, board):
    if gameOver(board) or lookedFarEnough(board):
        return evaluateScore(board)
    moves = getLegalMoves(colour, board)
    if colour == black:
       score = -infinity;
       for i in moves:
           score = max(minimax(white, makeMove(board, i)),
                       score)
    else:
       score = +infinity;
       for i in moves:
           score = min(minimax(black, makeMove(board, i)),
                       score)
    return score
```
```

Minimax algorithm

- we assume
  - `getLegalMoves` returns a list of legal moves
  - `makeMove(board, i)` applies a move, `i`, to a `board` and returns the new board
- also assume `evaluateScore` will return a reasonably accurate score based on the board position
- however there is a paradox here:
  - we need an accurate `evaluateScore` function
  - yet if it were really accurate we wouldn't need to look ahead!

Evaluation functions

- are the limitation to game tree searching
  - often an evaluation function reports a good score for a potential board position in the future
  - when we get to that position and look ahead the picture may not look as good!
- often called a local maximum, rather similar to hill climbing

Returning to Game tree searching

- examine the first game tree diagram
- notice that black calculates the score for move 1, as +10
  - when it examines move 2 (it can remember the previous score)
  - it examines move C which white might play, and it sees this score as 0
- at this point the search algorithm is able to conclude:
  - *there is no point in examining move, d, since no matter what score it might deliver, white can still make move, c*
  - *and the score for move, c, is worse that the score for move 1*
  - so there is no point black choosing move, 2, and therefore no point examining move, d

AlphaBeta algorithm

- implements the previous optimisation
  - the AlphaBeta algorithm always returns the same results as the minimax algorithm, only it is more efficient
- it is significantly more efficient than the minimax algorithm when the game has a high branching factor
  - for example chess has an approximate branching factor of 20
    - very roughly 20 different moves can be played at each board position
- Alphabeta also is more efficient if the first move it examines is a good move, as this is likely to cut off many subsequent poor moves

AlphaBeta

- remember a low value score is good for white
- remember a high value score is good for black
- the parameter, `alpha`, is the best move for white
  - initially set to ∞
  - low value is good
- the parameter, `beta`, is the best move for black
  - initially set to $-\infty$
  - high value is good

AlphaBeta

```
def alphaBeta (colour, board, alpha, beta):
    if gameOver(board) or lookedFarEnough(board):
        return evaluateScore(board)
    moves = getLegalMoves(colour, board)
    if colour == black:
      for i in moves:
        try = alphaBeta(white, makeMove(board, i), alpha, beta)
        if try > beta:
          beta = try    # found a better move
        if alpha >= beta:
          return beta   # no point searching further as WHITE would
                        # choose a different previous move
      return beta  # the best score for a move BLACK has found
    else:
      for i in moves:
        try = alphaBeta(black, makeMove(board, i), alpha, beta)
        if try < alpha:
          alpha = try     # found a better move
        if alpha >= beta:
          return alpha
      return alpha  # the best score for a move WHITE has found
```


Question

- is it necessary for the Alphabeta search algorithm to examine the board position created by move, f? Give reasons for your answer...

Tutorial

- modify the source of Othello so that it takes into consideration position advantage
  - rather than just material
- to do this, use an editor and modify the line 59 of source code from

```
#undef USE_CORNER_SCORES
```

Tutorial

- to

- ```
  #define USE_CORNER_SCORES
  ```

- save the source and rebuild the game via: `gcc o64bit.c`

- run the game by: `./a.out`

Tutorial

- now modify the game so that the function `evaluate` in `o64bit.c` awards points for taking squares: 42, 45, 18 and 21
  - hint you should be able to borrow the code which awards scores for the outer corners and adapt it
  ```