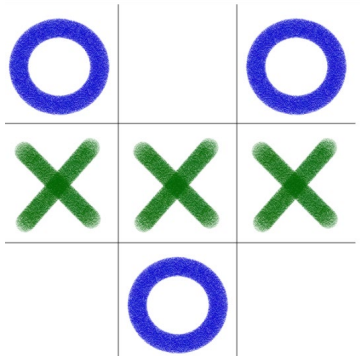
CS4386 AI Game Programming

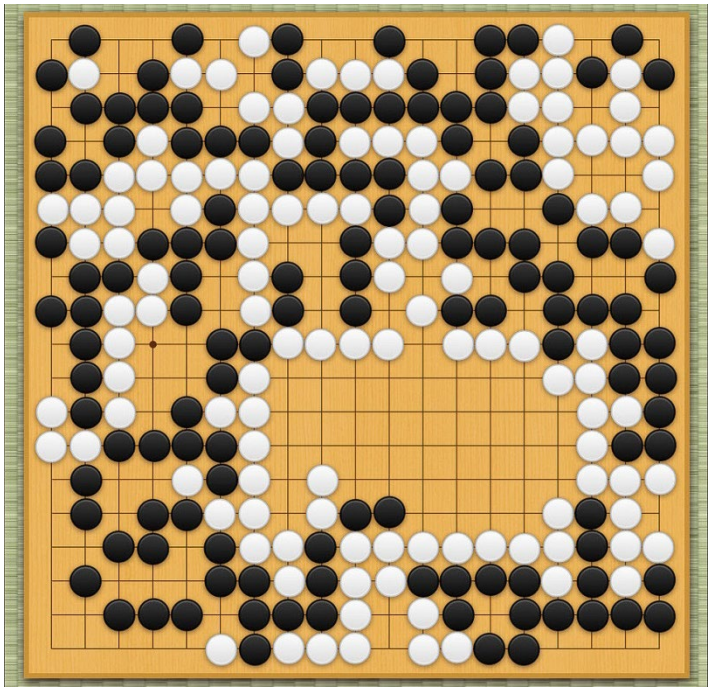
Lecture 02
Board Games

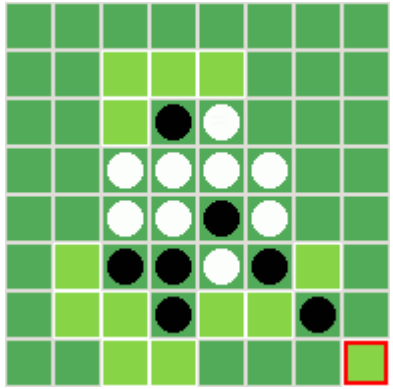
Semester B, 2020-2021
Department of Computer Science
City University of Hong Kong

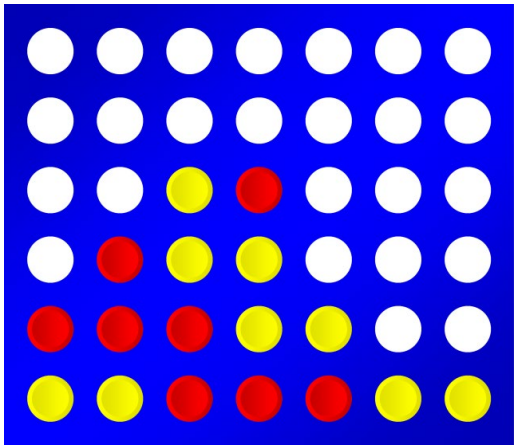
Board Games

- The earliest application of AI to computer games was as opponents in simulated version of common board games



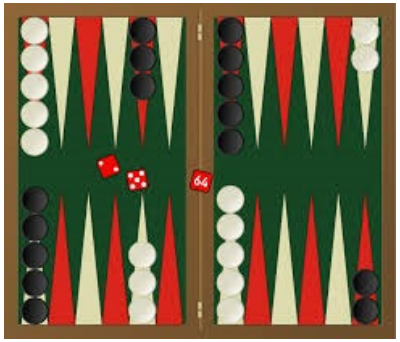






Goal of the Game

- Zero-sum game: your win is your opponent's loss
  - If you scored 1 point for winning, then it would be equivalent to scoring -1 for losing (If you gamble in a casino, is it a zero-sum game?)
  - In a zero-sum game, it does not matter if you try to win or if you try to make your opponent lose; the outcome is the same
- Information
  - Perfect information: all players know about the state of the game: the options and result of every move, e.g., chess
  - Imperfect information: there is a random element

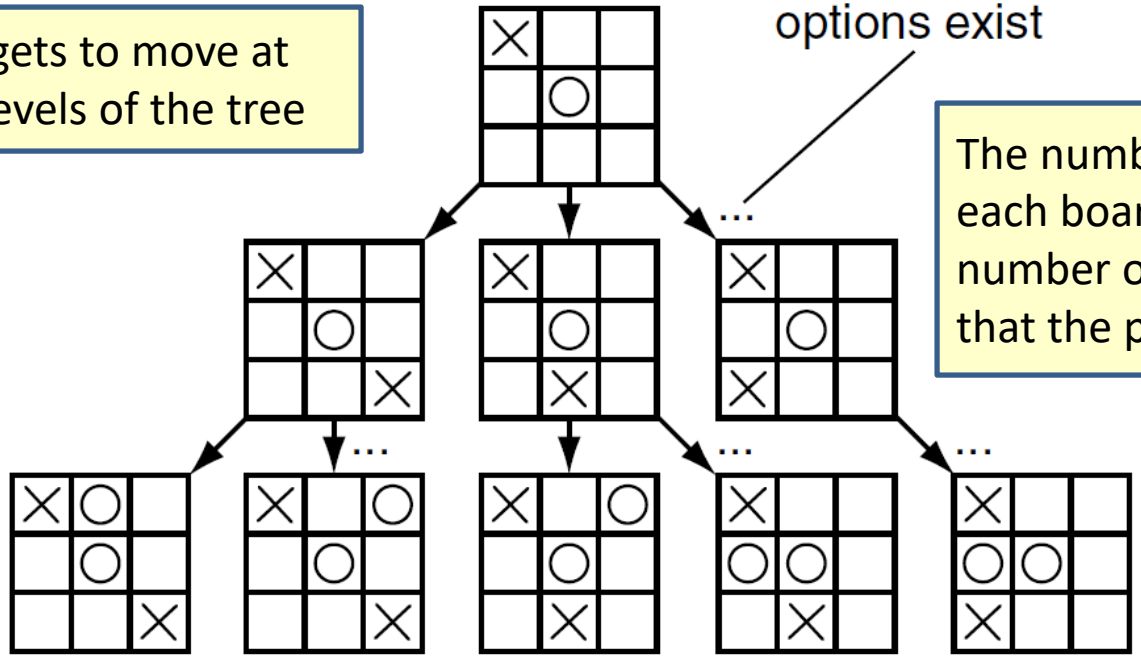


Game Tree

- Any turn-based game can be represented as a game tree

Each node represents a board position
Each branch represents a possible move

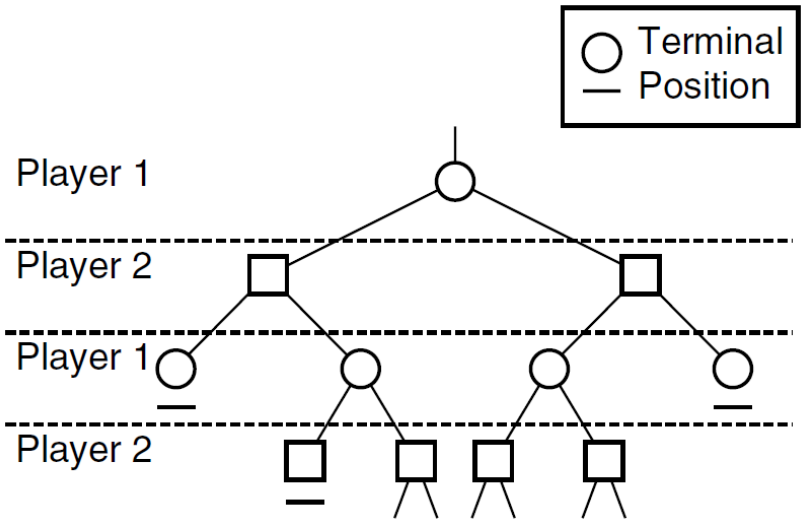
Each player gets to move at alternating levels of the tree



The number of branches from each board is equal to the number of possible moves that the player can make

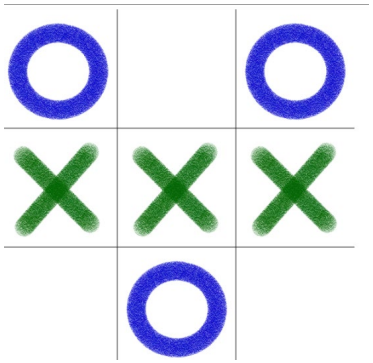
Game Tree (2)

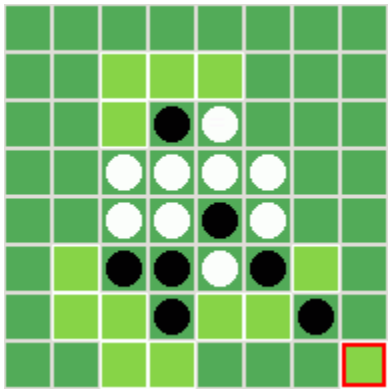
- Some board positions do not have any possible moves
  - Known as terminal positions representing the end of the game
  - For each terminal position, a final score is given to each player, e.g., +1 for a win and −1 for a loss
  - Draws are also allowed, scoring 0. In a zero-sum game, the final scores for each player will add up to zero



Branching Factor and Depth

- Branching factor: the number of branches at each branching point in the tree
  - a good indicator of how difficult a computer will find it to play the game
- Depth: maximum number of turns
  - What is the depth of a game tree for tic-tac-toe? For Reversi?

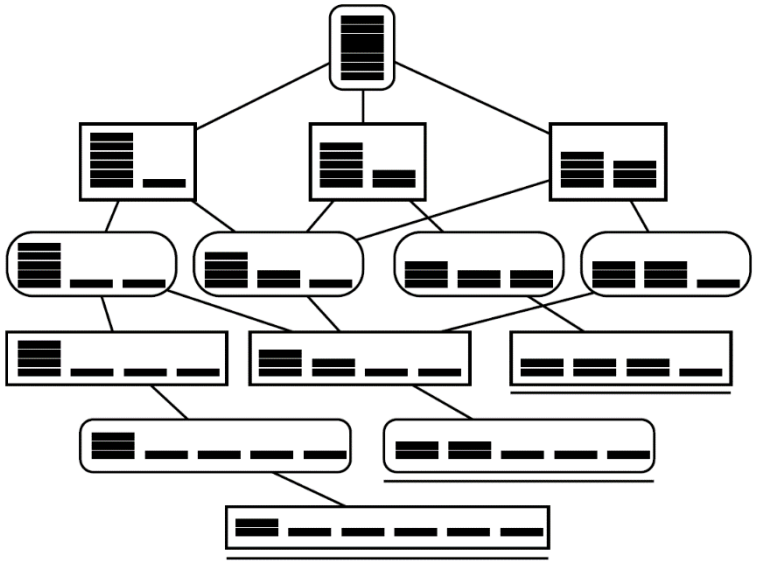




Transposition

- In many games it is possible to arrive at the same board position several times in a game
- In many more games it is possible to arrive at the same position by different combinations of moves
- Having the same board position from different sequences of moves is called transposition

E.g., Split-Nim starts with a single pile of coins. At each turn, alternating players have to split one pile of coins into two non-equal piles. The last player to be able to make a move wins

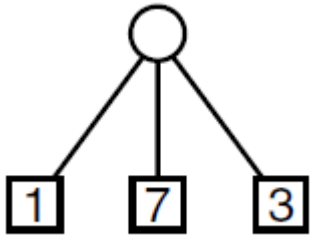


Static Evaluation Function

- A computer plays a turn-based game by looking at the actions available to it at this move and selecting one of them
  - It needs to know what moves are better than others
  - This knowledge is provided to the computer by the programmer using a heuristic called the static evaluation function
- Static evaluation function: look at the current state of the board and score it from the point of view of one player
- If the board is a terminal position in the tree, then this score will be the final score for the game
  - E.g., if the board is showing checkmate to black, then its score will be +1 to black (or whatever the winning score is set to be), while white's score will be −1
  - It is easy to score a winning position: one side will have the highest possible score and the other side will have the lowest possible score
- In the middle of the game, it is much harder to score. The score should reflect how likely a player is to win the game from that board position
  - So if the board is showing an overwhelming advantage to one player, then that player should receive a score very close to the winning score
  - In most cases the balance of winning or losing may not be clear

Simple Move Choice

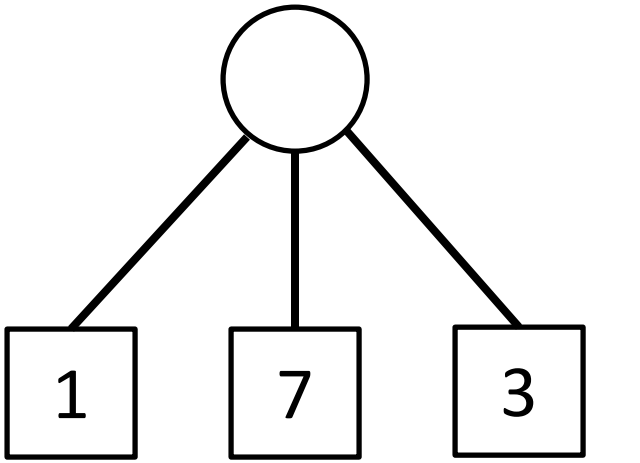
- With a good static evaluation function, the computer can select a move by scoring the positions that will result after making each possible move and choosing the highest score
- So to build the game AI for board game, don't we just need to apply the static evaluation function to evaluate each possible next move and pick the one that has the best score?



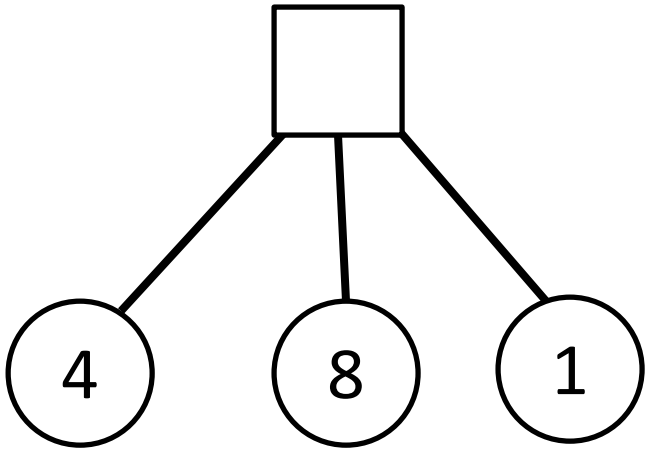
- Static evaluation function is hardly perfect for every possible board status so practical evaluation function would play poorly when used this way
- Human players in fact look ahead one or more moves. Computer heuristics are usually fairly narrow, limited an poor thus computer needs to look ahead many more moves than a person can
- The most famous search algorithm for games is minimax

Choosing a Move

- Assume that you are playing against your opponent in a zero-sum game, and the evaluation function is showing your score

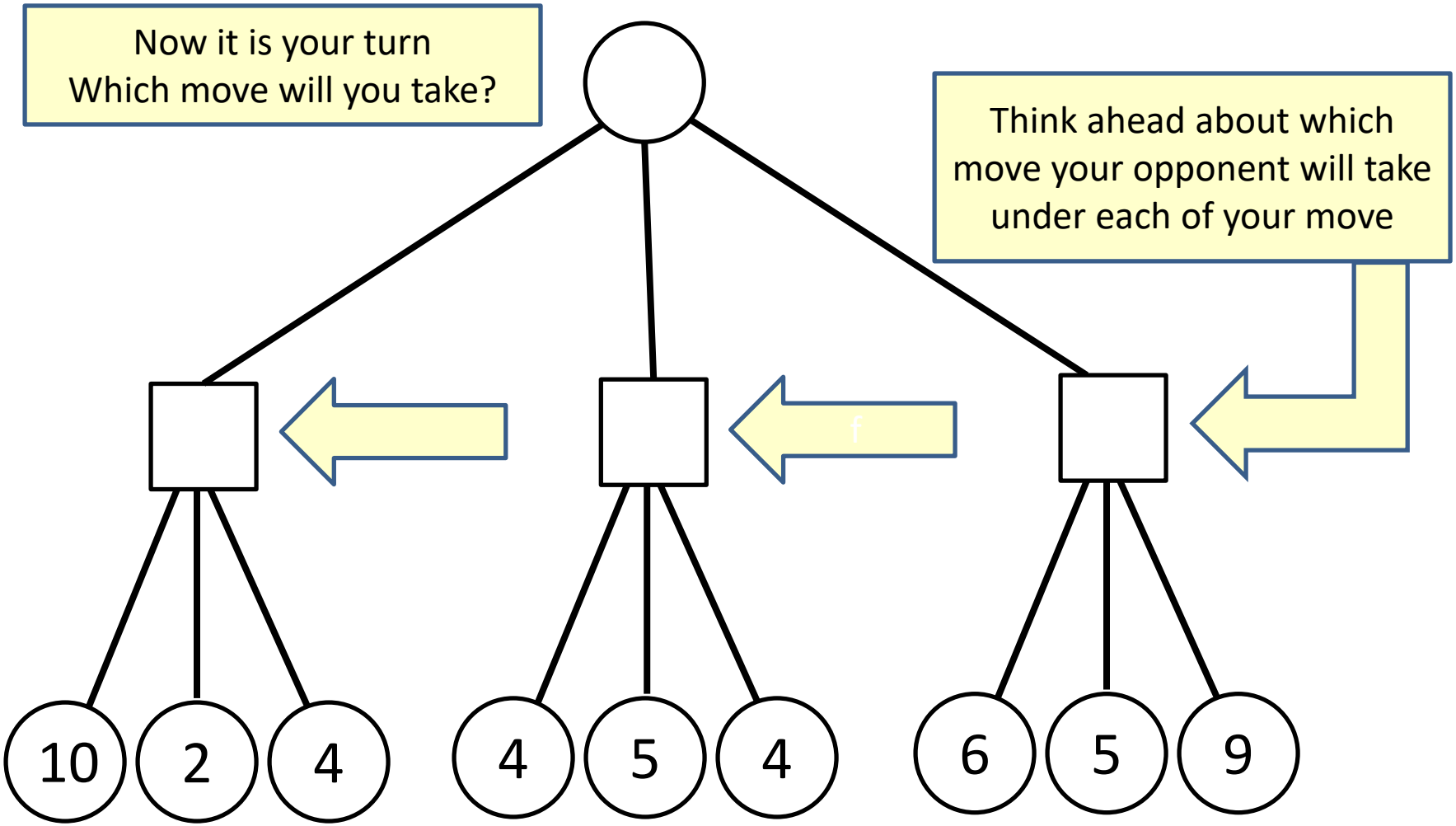


Now it is your turn
Which move will you take?



Now it is your opponent's turn
Which move do you think he/she will take?

Two-move Game Tree



Minimaxing

- Starting from the bottom of the tree, scores are bubbled up according to the minimax rule
  - On our turn, we bubble up the highest score
  - On our opponent's turn, we bubble up the lowest score

  Eventually, we have scores for the results of each available move, and then we choose the best one
- The process of bubbling scores up the tree is what the minimaxing algorithm does
  - by searching for responses, and responses to those responses, until it can search no further and then relies on the static evaluation function
  - It then bubbles the scores back up to get a score for each of its available moves

Pseudo Code - Minimax

To stop the search from going on forever (in the case where the tree is very deep), the algorithm has a maximum search depth

```
1  def minimax(board, player, maxDepth, currentDepth):
2
3    # Check if we're done recursing
4    if board.isGameOver() or currentDepth == maxDepth:
5      return board.evaluate(player), None
6
7    # Otherwise bubble up values from below
8
9    bestMove = None
10   if board.currentPlayer() == player: bestScore = -INFINITY
11   else: bestScore = INFINITY
12
13   # Go through each move
14   for move in board.getMoves():
15
16     newBoard = board.makeMove(move)
17
18     # Recurse
19     currentScore, currentMove = minimax(newBoard, player,
20                                         maxDepth, currentDepth+1)
21
22     # Update the best score
23     if board.currentPlayer() == player:
24       if currentScore > bestScore:
25         bestScore = currentScore
26         bestMove = move
27     else:
28       if currentScore < bestScore:
29         bestScore = currentScore
30         bestMove = move
31
32   # Return the score and the best move
33   return bestScore, bestMove
```

Call the static evaluation function

Look at each possible move from the current board position

Recursive

Maximization step for current player

Minimization step for opponent

Calling the minimax function

- The minimax function is assumed to return the best move and its score

```
1  def getBestMove(board, player, maxDepth):
2
3    # Get the result of a minimax run and return the move
4    score, move = minimax(board, player, maxDepth, 0)
5    return move
```

- The algorithm can be extended for a game with more than 2 players
- Performance
  - O(d) in memory
  - O(nd) in time

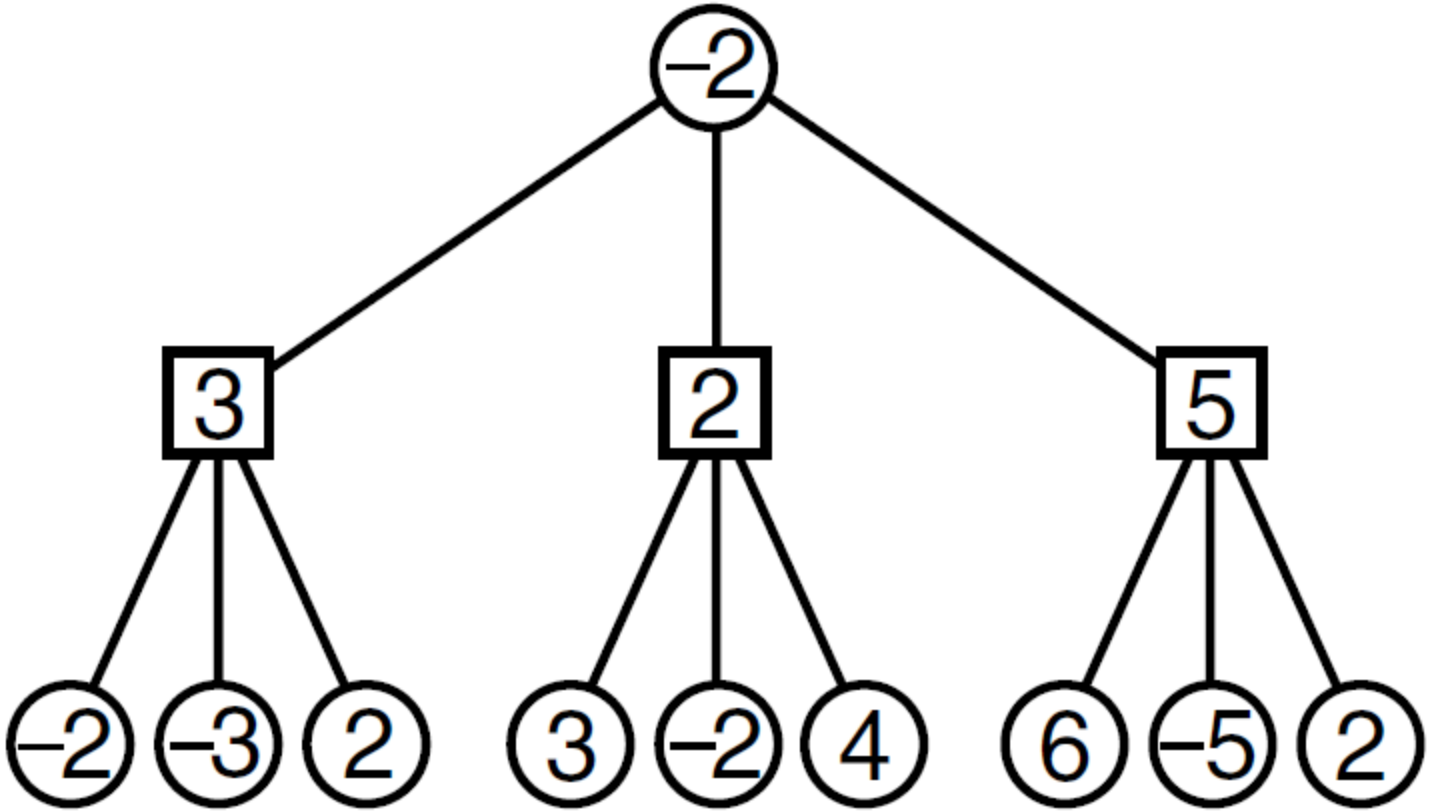
  where d: the maximum depth of the search (or the maximum depth of the tree if that is smaller)

  n: the number of possible moves at each board position

Negamaxing

- Remember that for a zero-sum game with 2 players, one player's gain is the other player's loss
- At each stage of bubbling up, rather than choosing either the smallest or largest, all the scores from the previous level have their signs changed. The scores are then correct for the player at that move
- Because at each bubbling up we invert the scores and choose the maximum, the algorithm is known as "negamax." It gives the same results as the minimax algorithm, but each level of bubbling is identical. There is no need to track whose move it is and act differently

Negamaxing Game Tree



At each node, the score gets inverted when bubbled up and the maximum score is retained

Pseudo Code - Negamax

Because negamax alternates viewpoints between players at each turn, the evaluation function always needs to score from the point of view of the player whose move it is on that board
So the point of view alternates between players at each move. To implement this, the evaluation function no longer needs to accept a point of view as input. It can simply look at whose turn it is to play

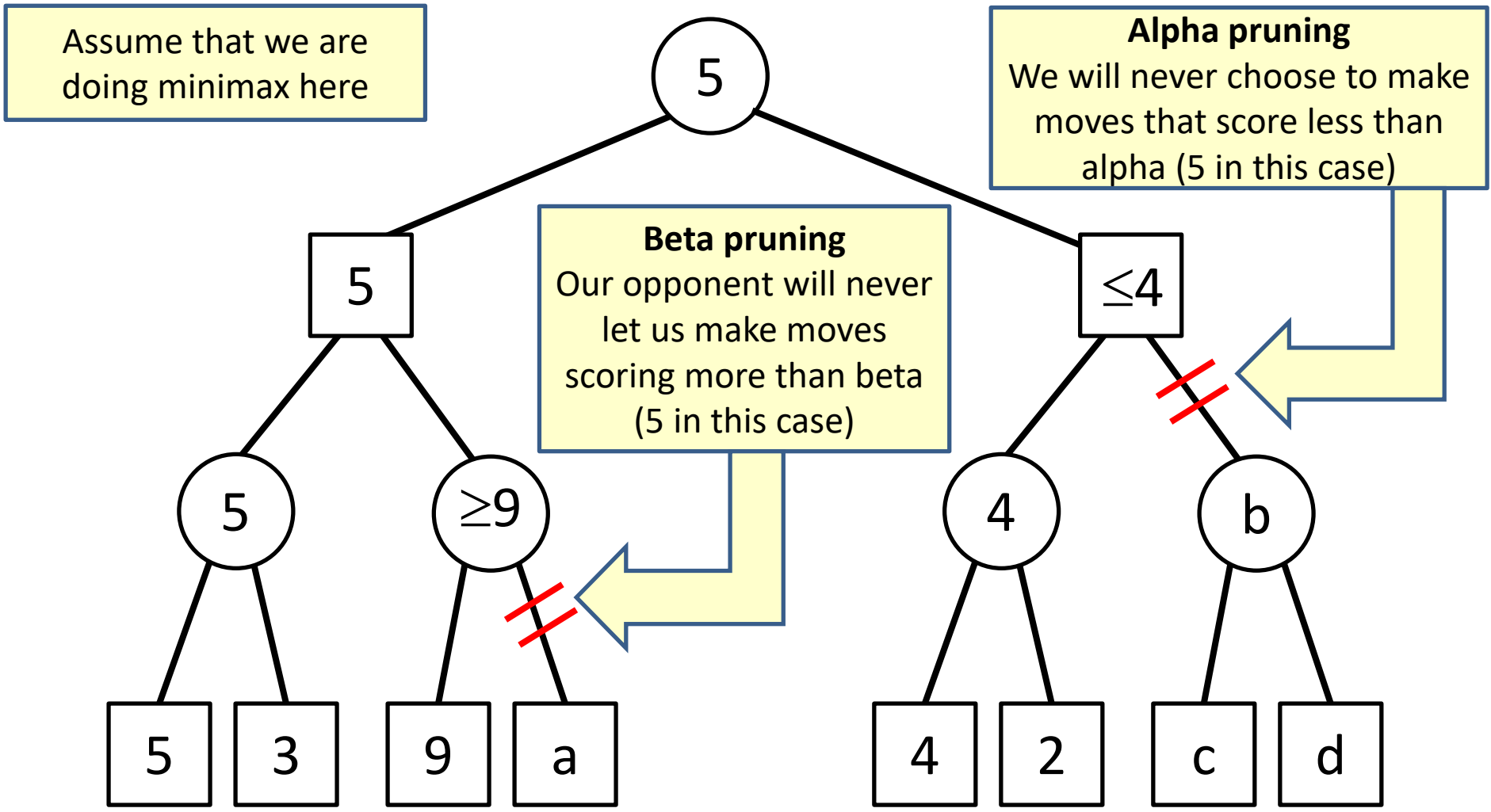
```
def negamax(board, maxDepth, currentDepth):

  # Check if we're done recursing
  if board.isGameOver() or currentDepth == maxDepth:
    return board.evaluate(), None

  # Otherwise bubble up values from below

  bestMove = None
  bestScore = -INFINITY

  # Go through each move
  for move in board.getMoves():

     newBoard = board.makeMove(move)

     # Recurse
     recursedScore, currentMove = negamax(newBoard,
                                          maxDepth, currentDepth+1)
     currentScore = -recursedScore

     # Update the best score
     if currentScore > bestScore:
       bestScore = currentScore
       bestMove = move

  # Return the score and the best move
  return bestScore, bestMove
```

Negamax Performance

- Performance
  - $O(d)$ in memory
  - $O(nd)$ in time

  Identical to minimax algorithm
- When developers talk about minimaxing, they often use a negamax-based algorithm in practice so minimax is often used as a generic term

Illustration of AB Pruning with Minimax



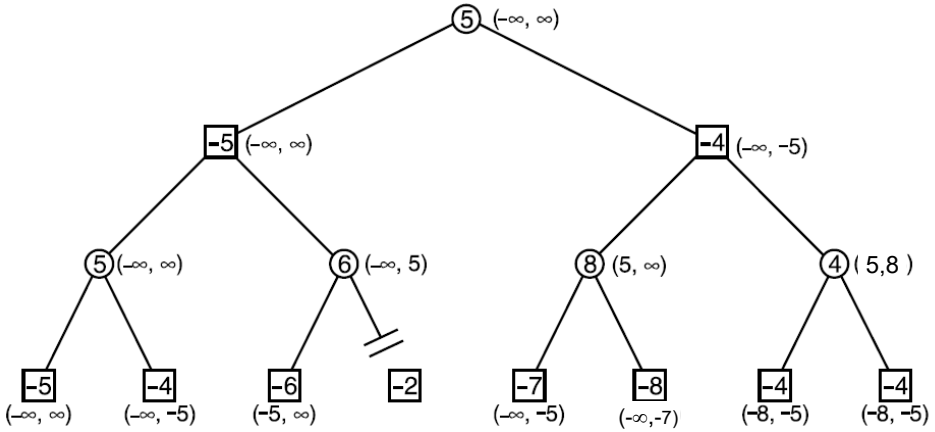
AB Negamax

- Although it is simpler to see the difference between alpha and beta prunes in the minimax algorithm, they are most commonly used with negamax
- Rather than alternating checks against alpha and beta at each successive turn, the AB negamax swaps and inverts the alpha and beta values (in the same way that it inverts the scores from the next level). It checks and prunes against just the beta value
- Using AB pruning with negamaxing, we have the simplest practical board game AI algorithm

Pseudo Code – AB Negamax

```
def abNegamax(board, maxDepth, currentDepth, alpha, beta):

  # Check if we're done recursing
  if board.isGameOver() or currentDepth == maxDepth:
    return board.evaluate(player), None

  # Otherwise bubble up values from below

  bestMove = None
  bestScore = -INFINITY

  # Go through each move
  for move in board.getMoves():

     newBoard = board.makeMove(move)

     # Recurse
     recursedScore, currentMove = abNegamax(newBoard,
                                            maxDepth,
                                            currentDepth+1
                                            -beta,
                                            -max(alpha, bestScore))
     currentScore = -recursedScore

     # Update the best score
     if currentScore > bestScore:
       bestScore = currentScore
       bestMove = move

       # If we're outside the bounds, then prune: exit immediately
       if bestScore >= beta:
         return bestScore, bestMove

  return bestScore, bestMove
```

```
def getBestMove(board, maxDepth):

  # Get the result of a minimax run and return the move
  score, move = abNegamax(board, maxDepth, 0, -INFINITY, INFINITY)
  return move
```



Performance of AB Negamax

- Performance
  - $O(d)$ in memory
  - $O(nd)$ in time
- Does it mean that AB negamax has the same efficiency as negamax?
- In fact, AB negamax will outperform regular negamax in almost all cases
- The only situation in which it will not is if the moves are ordered so that no pruning is possible so the algorithm will have extra comparison making it slower in this case
  - This situation is however very rare

AB Search Window

- Search window: the interval between the alpha and beta values in an AB algorithm
- Only new move sequences with scores in this window are considered. All others are pruned
- The smaller the search window, the more likely a branch is to be pruned, thus speeding up the algorithm

Move Order

- If the most likely moves are considered first, then
  - the search window will contract more quickly
  - the less likely moves will be considered later and are more likely to be pruned
- Determining which moves are better is the whole point of AI
  - If we knew the best moves, then we would not need the algorithm
  - In the simplest case it is possible to use the static evaluation function on the moves to determine the correct order, as it gives an approximate indication of how good a board position is. However, repeatedly calling the evaluation function slows down the algorithm
  - An even more effective ordering technique is to use the results of previous minimax searches at previous depths (e.g., iterative deepening) or on previous turns (e.g., stored in transposition table)

Aspiration Search

- Having a small search window can lead to massive speed up so we may want to artificially limiting the window using an estimated range (instead of the initial range of -∞ to ∞)
- This estimated range is called an aspiration, and the AB algorithm in this way is called aspiration search
- The smaller range will speed up the algorithm, but there may be no suitable move sequences within the given range of values and the algorithm will return with failure: no best move found. The search can then be repeated with wider window
- The aspiration for the search is often based on the results of a previous search, e.g., as aspiration search can be performed using (5 - window size, 5 + window size), where 5 is the score from previous search, and window size may depend on the range of scores that can be returned by the evaluation function

Pseudo Code – Aspiration Search

```
1  def aspiration(board, maxDepth, previous):
2    alpha = previous - WINDOW_SIZE
3    beta = previous + WINDOW_SIZE
4
5    while True:
6
7      result, move = abNegamax(board, maxDepth, 0, alpha, beta);
8      if (result <= alpha) alpha = -NEAR_INFINITY;
9      else if (result >= beta) beta = NEAR_INFINITY;
10     else return move;
```

Transposition Table

- As mentioned previously on Transposition, the same board position can occur as a result of different combinations of moves
- In many games the same board position can even occur multiple times within the same game (e.g., chess)
- To avoid doing extra work searching the same board position several times, algorithms can make use of a transposition table that keeps a record of board positions and the results of a search from that position
- When an algorithm is given a board position, it first checks if the board is in the memory and uses the stored value if it is
- How can we represent a transposition table?
  - As an array such that each time we evaluate a board position, we add a new element to the array? What is the problem?
  - As a lookup table by first converting each possible board position as a unique index and then use the resulting index to check whether the given board position has already been evaluated? What is the problem?

Hashing Game States

- In principle any hash algorithm will work
- A common algorithm for transposition table hashing is Zobrist keys
- A Zobrist key is a set of fixed-length random bit patterns stored for each possible state of each possible location on the board
  - E.g., Chess has 64 squares, and each square can be empty or 1 of 6 different pieces on it, each of two possible colors. The Zobrist key for a game of Chess needs to be 64×2×6=768 entries long
- For each non-empty square, the Zobrist key is looked up and XORed with a running hash total
- The length of the hash value in the Zobrist key will depend on the number of different states for the board
  - Chess games can make do with 32 bits but are best with 64-bit key
  - Checkers works comfortably with 32 bits
- The Zobrist keys need to be initialized with random bit-strings of the appropriate size

Hash Implementation for Tic-Tac-Toe

```
# The Zobrist key.
zobristKey[9*2]

# Initialize the key.
def initZobristKey():
  for i in 0..9*2:
    zobristKey[i] = rand32()
```

```
# Calculate a hash value.
def hash(ticTacToeBoard):
    # Start with a clear bitstring
    result = 0

    # XOR each occupied location in turn
    for i in 0..9:
      # Find what piece we have
      piece = board.getPieceAtLocation(i)

      # If its unoccupied, lookup the hash value and xor it
      if piece != UNOCCUPIED:
        result = result xor zobristKey[i*2+piece]

    return result
```

Incremental Zobrist Hashing

- One particularly nice feature of Zobrist keys is that they can be incrementally updated
- Adding an element is as simple as XORing a value
- Removing a piece can be done by XORing a value as well
- Incrementally hashing is much faster than calculating the hash from first principles, i.e., XORing all the pieces on board, especially when there are many pieces in play at once

Hash Table Implementation – General Form

```
1  struct Bucket:
2
3    # The table entry at this location
4    TableEntry entry;
5
6    # The next item in the bucket
7    Bucket *next;
8
9    # Returns a matching entry from this bucket, even
10   # if it comes further down the list
11   def getElement(hashValue):
12     if entry.hashValue == hashValue: return entry;
13     if next: return next->getElement(hashValue);
14     return None
15
16 class HashTable:
17   # Holds the contents of the table
18   buckets[MAX_BUCKETS]
19
20   # Finds the bucket in which the value is stored
21   def getBucket(hashValue):
22     return buckets[hashValue \% MAX_BUCKETS]
23
24   # Retrieves an entry from the table
25   def getEntry(hashValue):
26     return getBucket(hashValue).getElement(hashValue)
```

A general hash table has an array of lists; the arrays are often called “buckets”

When an element is hashed, the hash value looks up the correct bucket
Each item in the bucket is then examined to see if it matches the hash value

The key undergoes a modular multiplication by the number of buckets, and the new value is the index of the bucket to examine

Hash Table Implementation – Hash Array

- In searching for moves, it is more important that the hash lookup is fast, rather than guaranteeing that the contents of the hash table are permanent (i.e., replaceable)
- For this reason a hash array implementation is used, where each bucket has a size of one

```
1  class HashArray:
2    # Holds the entries
3    entries[MAX_BUCKETS]
4
5    # Retrieves an entry from the table
6    def getEntry(hashValue):
7      entry = entries[hashValue \% MAX_BUCKETS];
8      if entry.hashValue == hashValue: return entry
9      else: return None
```

Replacement Strategy

- Since there can be only one stored entry for each bucket, there needs to be some mechanism for deciding how and when to replace a stored value when a clash occurs
- The simplest technique is to always overwrite
  - The contents of a table entry are replaced whenever a clashing entry wants to be stored
  - This is easy to implement and is often perfectly sufficient
- Another common heuristic is to replace whenever the clashing node is for a later move
  - So if a board at move 6 clashes with a board at move 10, the board at move 10 is used
  - This is based on the assumption that the board at move 10 will be useful for longer than the board at move 6
- There are many more complex replacement strategies, but there is no general agreement as to which is the best
  - It seems likely that different strategies will be optimal for different games
  - Experimentation is probably required
- Several programs have had success by keeping multiple transposition tables using a range of strategies
  - Each transposition table is checked in turn for a match
  - This seems to offset the weakness of each approach against others

Performance

- Performance
  - O(1) in both time and memory in getting and storing an entry
  - O(n) in memory where n is the number of entries in the table, which depends on the number of checked board positions
- Path Dependency
  - Some games need to have scores that depend on the sequence of moves, e.g., repeating the same set of board positions three times in Chess results in a draw. Holding transposition tables will mean that such repetitions will always be scored identically. This can mean that the AI mistakenly throws away a winning position by repeating the sequence
  - In this instance the problem can be solved by incorporating a Zobrist key for "number of repeats" in the hash function. In this way successive repeats have different hash values and are recorded separately
  - In general, however, games that require sequence-dependent scoring need to have either more complex hashing or special code in the search algorithm to detect this situation

Performance (2)

- Instability
  - The stored values may fluctuate during the same search
  - Because each table entry may be overwritten at different times, there is no guarantee that the same value will be returned each time a position is looked up
  - For example, the first time a node is considered in a search, it is found in the transposition table, and its value is looked up. Later in the same search that location in the table is overwritten by a new board position. Even later in the search the board position is returned to (say by a different sequence of moves)
  - Although it is very rare, it is possible to have a situation where the score for a board oscillates between two values, causing some versions of a re-searching algorithm to loop infinitely

Using Opponent's Time

- A transposition table can be used to allow the AI to improve its searches while the human player is thinking
- On the player's turn, the computer can search for the move it would make if it were playing
- As results of this search are processed, they are stored in the transposition table. When the AI comes to take its turn, its searches will be faster because a lot of the board positions will already be considered and stored

Opening Books

- In many games, over many years, expert players have built up a body of experience about which moves are better than others at the start of the game
- An opening book is a list of move sequences, along with some indication of how good the average outcome will be using those sequences
  - Using these sets of rules, the computer does not need to search using minimaxing to work out what the best move is to play
  - It can simply choose the next move from the sequence, as long as the end point of the sequence is beneficial to it
- Opening books are implemented as a hash table very similar to a transposition table, but there may be more than one recommended move from each board position
  - Board positions can often belong to many different opening lines
  - Openings, like the rest of the game, branch out in the form of a tree
- Some programs use an initial opening book library and add a learning layer
  - The learning layer updates the scores assigned to each opening sequence so that better openings can be selected by updating the scores of each opening and/or learning new opening sequences

Play Books

- Many games have set combinations of moves that occur during the game and especially at the end of the game
- For almost all games, however, the range of possible board positions in the game is staggering so it is unlikely that any particular board position will be exactly the same as one in the database
  - More sophisticated pattern matching is required: looking for particular patterns among the overall board structure
  - The most common application of this type of database is for subsections of the board, e.g., the four edge configurations in Reversi
- Ending database
  - Very late in some games (like Chess, Backgammon, or Checkers) the board simplifies down. Often, it is possible to pick up an opening book-style lookup at this stage

Further Optimizations: Iterative Deepening

- For games with a large branching factor, it can take a very long time to look even a few moves ahead. Pruning cuts down a lot of the search, but most board positions still need to be considered
- For most games the computer does not have the luxury of being able to think for as long as it wants
  - Board games such as Chess use timing mechanisms, and modern computer games may allow players to play at their own speed
  - Because the minimaxing algorithms search to a fixed depth, there is no guarantee that the search will be complete by the time the computer needs to make its move
- Iterative deepening minimax search performs a regular minimax with gradually increasing depths
  - Initially, the algorithm searches one move ahead, then if it has time it searches two moves ahead, and so on until its time runs out.
  - If time runs out before a search has been completed, it uses the result of the search from the previous depth

Further Optimizations: Variable Depth Approach

- AB pruning is an example of a variable depth algorithm
  - Not all branches are searched to the same depth
  - Some branches are pruned if the computer decides it no longer needs to consider them
- In general, however, the searches are fixed depth
  - A condition in the search checks if the maximum depth has been reached and terminates that part of the algorithm
- The algorithms can be altered to allow variable depth searches on any number of grounds, and different techniques for pruning the search have different names
  - They are not new algorithms, but simply guidelines for when to stop searching a branch

Further Optimizations: Variable Depth Approach - Extensions

- The major weakness of computer players for turn-based games is the horizon effect. The horizon effect occurs when a fixed sequence of moves ends up with what appears to be an excellent position, but one additional move will show that that position is, in fact, terrible
  - Regardless of how deep the computer looks, this effect may still be present. If the search is very deep, however, the computer will have enough time to select a better move when the trouble is eventually seen
  - If the search cannot continue to a great depth because of high branching, and if the horizon effect is noticeable, then the minimax algorithm can use a technique called extensions
- Extensions are a variable depth technique, where the few most promising move sequences are searched to a much greater depth
  - By only selecting the most likely moves to consider at each turn, the extension can be many levels deep
  - It is not uncommon for extensions of 10 to 20 moves to be considered on a basic search depth of 8 or 9 moves
- Extensions are often searched using an iterative deepening approach, where only the most promising moves from the previous iteration are extended further
  - While this can often solve horizon effect problems, it relies heavily on the static evaluation function, and poor evaluation can lead the computer to extend along a useless set of options

Further Optimizations: Variable Depth Approach - Quiescence Pruning

- When a period of relative calm occurs, searching deeper often provides no additional information
  - It may be better to use the computer time to search another area of the tree or to search for extensions on the most promising lines
  - Pruning the search based on the board's stability is called quiescence pruning
- A branch will be pruned if its heuristic value does not change much over successive depths of search
  - This probably means that the heuristic value is accurate, and there is little point in continuing to search there
  - Combined with extensions, quiescent pruning allows most of the search effort to be focused on the areas of the tree that are the most critical for good play

Reference

- Artificial Intelligence for Games
  - Chapter 8