CSC 421: Algorithm Design & Analysis

Spring 2017

Backtracking

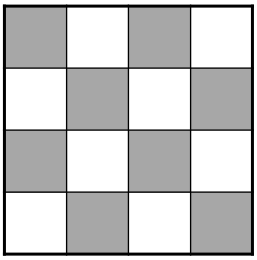
- greedy vs. backtracking (DFS)
- greedy vs. generate & test
- examples:

  N-queens, 2-D gels, Boggle
- branch & bound
- problem characteristics
- aside: game trees

Greed is good?

IMPORTANT: the greedy approach is not applicable to all problems

- but when applicable, it is very effective (no planning or coordination necessary)



GREEDY approach for N-Queens: start with first row, find a valid position in current row, place a queen in that position then move on to the next row

since queen placements are not independent, local choices do not necessarily lead to a global solution

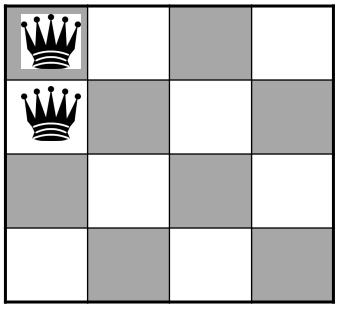
GREEDY does not work – need a more holistic approach

Generate & test?

recall the generate & test solution to N-queens

- systematically generate every possible arrangement
- test each one to see if it is a valid solution

$\binom{16}{4}$ = 1,820 arrangements



fortunately, we can do better if we recognize that choices can constrain future choices

- e.g., any board arrangement with a queen at (1,1) and (2,1) is invalid
- no point in looking at the other queens, so can eliminate 16 boards from consideration

- similarly, queen at (1,1) and (2,2) is invalid, so eliminate another 16 boards

3

Backtracking

backtracking is a smart way of doing generate & test

- view a solution as a sequence of choices/actions (*similar to* GREEDY)
- when presented with a choice, pick one (*similar to* GREEDY)
- however, reserve the right to change your mind and backtrack to a previous choice (*unlike* GREEDY)

- you must remember alternatives:
  if a choice does not lead to a solution, back up and try an alternative

- eventually, backtracking will find a solution or exhaust all alternatives

backtracking is essentially depth first search

- add ability to prune a path as soon as we know it can't succeed
- when that happens, back up and try another path

4

N-Queens psuedocode

```
/**
 * Fills the board with queens starting at specified row
 * (Queens have already been placed in rows 0 to row-1)
 */
private boolean placeQueens(int row) {
    if (ROW EXTENDS BEYOND BOARD) {
        return true;
    }
    else {
        for (EACH COL IN ROW) {
            if ([ROW][COL] IS NOT IN JEOPARDY FROM EXISTING QUEENS) {
                ADD QUEEN AT [ROW][COL]

                if (this.placeQueens(row+1)) {
                    return true;
                }
                else {
                    REMOVE QUEEN FROM [ROW][COL]
                }
            }
        }
        return false;
    }
}
```

if row > board size, then all queens have been placed already – return true

place a queen in available column

if can recursively place the remaining queens, then done

if not, remove the queen just placed and continue looping to try other columns

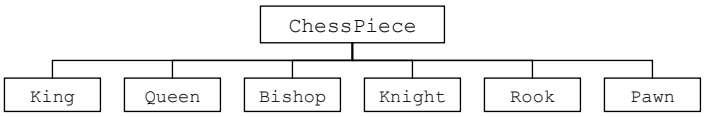
return false if cannot place remaining queens

5

Chessboard class

we could define a class hierarchy for chess pieces

- `ChessPiece` is an abstract class that specifies the common behaviors of pieces
- `Queen`, `Knight`, `Pawn`, … are derived from `ChessPiece` and implement specific behaviors




```
public class ChessBoard {
  private ChessPiece[][] board;                    // 2-D array of chess pieces
  private int pieceCount;                          // number of pieces on the board

  public ChessBoard(int size) {…}                  // constructs size-by-size board
  public ChessPiece get(int row, int col) {…}      // returns piece at (row,col)
  public void remove(int row, int col) {…}         // removes piece at (row,col)
  public void add(int row, int col, ChessPiece p) {…}  // places a piece, e.g., a queen,
                                                   //   at (row,col)
  public boolean inJeopardy(int row, int col) {..}     // returns true if (row,col) is
                                                   //   under attack by any piece
  public int numPieces() {…}                       // returns number of pieces on board
  public int size() {…}                            // returns the board size
  public String toString() {…}                     // converts board to String
}
```

6

Backtracking N-queens

```
public class NQueens {
  private ChessBoard board;

  . . .

  /**
   * Fills the board with queens.
   */
  public boolean placeQueens() {
    return this.placeQueens(0);
  }


  /**
   * Fills the board with queens starting at specified row
   * (Queens have already been placed in rows 0 to row-1)
   */
  private boolean placeQueens(int row) {
    if (row >= this.board.size()) {
        return true;
    }
    else {
        for (int col = 0; col < this.board.size(); col++) {
            if (!this.board.inJeopardy(row, col)) {
                this.board.add(row, col, new Queen());

                if (this.placeQueens(row+1)) {
                    return true;
                }
                else {
                    this.board.remove(row, col);
                }
            }
        }
        return false;
    }
  }
}
```

in an NQueens class, will have a ChessBoard field and a method for placing the queens

- placeQueens calls a helper method with a row # parameter

BASE CASE: if all queens have been placed, then done.

OTHERWISE: try placing queen in the row and recurse to place the rest

note: if recursion fails, must remove the queen in order to backtrack

Why does backtracking work?

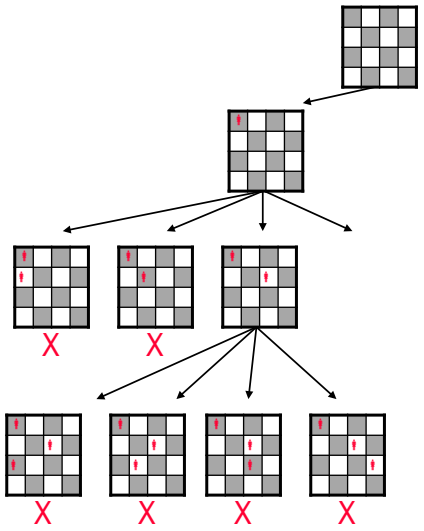
backtracking burns no bridges – all choices are reversible

backtracking provides a systematic way of trying all paths (sequences of choices) until a solution is found

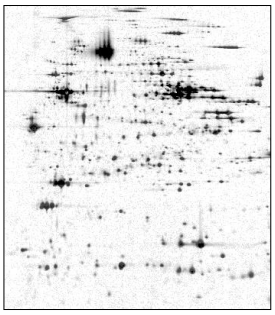
- assuming the search tree is finite, will eventually find a solution or exhaust the entire search space

backtracking is different from generate & test in that choices are made sequentially

- earlier choices constrain later ones
- can avoid searching entire branches



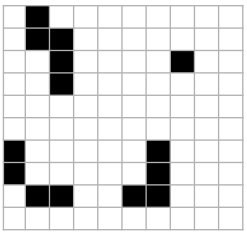
Another example: blob count



application: 2-D gel electrophoresis

- biologists use electrophoresis to produce a gel image of cellular material
- each "blob" (contiguous collection of dark pixels) represents a protein
- identify proteins by matching the blobs up with another known gel image

we would like to identify each blob, its location and size



- location is highest & leftmost pixel in the blob
- size is the number of contiguous pixels in the blob
- in this small image: Blob at [0][1]: size 5
  Blob at [2][7]: size 1
  Blob at [6][0]: size 4
  Blob at [6][6]: size 4
- can use backtracking to locate & measure blobs

9

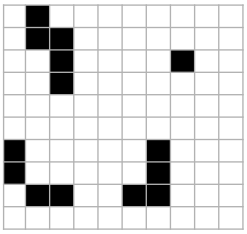
Blob count (cont.)

can use recursive backtracking to get a blob's size

when find a spot:

1 (for the spot) +

size of all connected subblobs (adjacent to spot)



note: we must not double count any spots

- when a spot has been counted, must "erase" it
- keep it erased until all blobs have been counted

pseudocode:

```
private int blobSize(int row, int col) {
    if (OFF THE GRID || NOT A SPOT) {
        return 0;
    }
    else {
        ERASE SPOT;
        return 1 + this.blobSize(row-1, col-1)
                 + this.blobSize(row-1,   col)
                 + this.blobSize(row-1, col+1)
                 + this.blobSize(  row, col-1)
                 + this.blobSize(  row, col+1)
                 + this.blobSize(row+1, col-1)
                 + this.blobSize(row+1,   col)
                 + this.blobSize(row+1, col+1);
    }
}
```

10

Blob count (cont.)

`findBlobs` traverses the image, checks each grid pixel for a blob

`blobSize` uses backtracking to expand in all directions once a blob is found

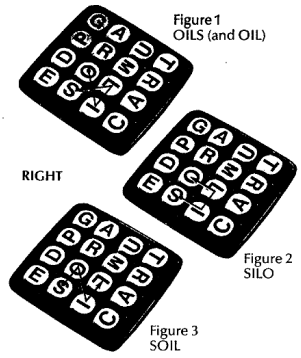
each pixel is "erased" after it is processed in `blobSize` to avoid double-counting (& infinite recursion)

the image is restored at the end of `findBlobs`

```
public class Gel{
  private char[][] grid;

  . . .

  public void findBlobs() {
    for (int row = 0; row < this.grid.length; row++) {
        for (int col = 0; col < this.grid.length; col++) {
            if (this.grid[row][col] == '*') {
                System.out.println("Blob at [" + row + "][" +
                                   col + "] : size " +
                                   this.blobSize(row, col));
            }
        }
    }

    for (int row = 0; row < this.grid.length; row++) {
        for (int col = 0; col < this.grid.length; col++) {
            if (this.grid[row][col] == 'O') {
                this.grid[row][col] = '*';
            }
        }
    }
  }

  private int blobSize(int row, int col) {
    if (row < 0 || row >= this.grid.length ||
            col < 0 || col >= this.grid.length ||
            this.grid[row][col] != '*') {
        return 0;
    }
    else {
        this.grid[row][col] = 'O';
        return 1 + this.blobSize(row-1, col-1)
                 + this.blobSize(row-1,   col)
                 + this.blobSize(row-1, col+1)
                 + this.blobSize(  row, col-1)
                 + this.blobSize(  row, col+1)
                 + this.blobSize(row+1, col-1)
                 + this.blobSize(row+1,   col)
                 + this.blobSize(row+1, col+1);
    }
  }
}
```

11

Another example: Boggle



recall the game

- random letters are placed in a 4x4 grid
- want to find words by connecting adjacent letters (cannot reuse the same letter)
- for each word found, the player earns points = length of the word
- the player who earns the most points after 3 minutes wins

how do we automate the search for words?

12

Boggle (cont.)

can use recursive backtracking to search for a word

when the first letter is found:

remove first letter & recursively search for remaining letters

again, we must not double count any letters

- must "erase" a used letter, but then restore for later searches

G	A	U	T
P	R	M	R
D	O	L	A
E	S	I	C

pseudocode:

```
private boolean findWord(String word, int row, int col) {
    if (WORD IS EMPTY) {
        return true;
    }
    else if (OFF_THE_GRID || GRID LETTER != FIRST LETTER OF WORD) {
        return false;
    }
    else {
        ERASE LETTER;
        String rest = word.substring(1, word.length());
        boolean result = this.findWord(rest, row-1, col-1) ||
                         this.findWord(rest, row-1,   col) ||
                         this.findWord(rest, row-1, col+1) ||
                         this.findWord(rest,   row, col-1) ||
                         this.findWord(rest,   row, col+1) ||
                         this.findWord(rest, row+1, col-1) ||
                         this.findWord(rest, row+1,   col) ||
                         this.findWord(rest, row+1, col+1);
        RESTORE LETTER;
        return result;
    }
}
```

13

BoggleBoard class

can define a `BoggleBoard` class that represents a board

- has public method for finding a word
- it calls the private method that implements recursive backtracking
- also needs a constructor for initializing the board with random letters
- also needs a `toString` method for easily displaying the board

```
public class BoggleBoard {
  private char[][] board;

  . . .

  public boolean findWord(String word) {
    for (int row = 0; row < this.board.length; row++) {
        for (int col = 0; col < this.board.length; col++) {
            if (this.findWord(word, row, col)) {
                return true;
            }
        }
    }
    return false;
  }

  private boolean findWord(String word, int row, int col) {
    if (word.equals("")) {
        return true;
    }
    else if (row < 0 || row >= this.board.length ||
             col < 0 || col >= this.board.length ||
             this.board[row][col] != word.charAt(0)) {
        return false;
    }
    else {
        char safe = this.board[row][col];
        this.board[row][col] = '*';
        String rest = word.substring(1, word.length());
        boolean result = this.findWord(rest, row-1, col-1) ||
                         this.findWord(rest, row-1,   col) ||
                         this.findWord(rest, row-1, col+1) ||
                         this.findWord(rest,   row, col-1) ||
                         this.findWord(rest,   row, col+1) ||
                         this.findWord(rest, row+1, col-1) ||
                         this.findWord(rest, row+1,   col) ||
                         this.findWord(rest, row+1, col+1);
        this.board[row][col] = safe;
        return result;
    }
  }

  . . .
}
```

14

BoggleGame class

a separate class can implement the game functionality

- constructor creates the board and fills `unguessedWords` with all found words
- `makeGuess` checks to see if the word is valid and has not been guessed, updates the sets accordingly
- also need methods for accessing the `guessedWords`, `unguessedWords`, and the board (for display)

see `BoggleGUI`

```
public class BoggleGame {
  private final static String DICT_FILE = "dictionary.txt";
  private BoggleBoard board;
  private Set<String> guessedWords;
  private Set<String> unguessedWords;

  public BoggleGame() {
      this.board = new BoggleBoard();
      this.guessedWords = new TreeSet<String>();
      this.unguessedWords = new TreeSet<String>();

      try {
          Scanner dictFile = new Scanner(new File(DICT_FILE));
          while (dictFile.hasNext()) {
              String nextWord = dictFile.next();
              if (this.board.findWord(nextWord)) {
                  this.unguessedWords.add(nextWord);
              }
          }
      }
      catch (java.io.FileNotFoundException e) {
          System.out.println("DICTIONARY FILE NOT FOUND");
      }
  }

  public boolean makeGuess(String word) {
      if (this.unguessedWords.contains(word)) {
          this.unguessedWords.remove(word);
          this.guessedWords.add(word);
          return true;
      }
      return false;
  }

  . . .
}
```

15

Branch & bound

the central idea of backtracking is cutting off a branch of the search as soon as we see that it can't lead to a solution

- then, backtrack and try a different branch

e.g., for the shortest path problem

- we cut off a branch if the vertex was a dead end
- we also cut it off if its length exceeded that of an already found path

what if we also had the ability to look ahead?

- i.e., if we could tell ahead of time (using some deduction) that a branch was not going to work, then we could preemptively cut
- this variant of backtracking is known as *branch & bound*

16

B & B example

suppose you have four jobs and 4 contractors (with bids), and want to assign the jobs to the contractors to minimize cost

	job 1	job 2	job 3	job 4
contractor a	$9K	$2K	$7K	$8K
contractor b	$6K	$4K	$3K	$7K
contractor c	$5K	$8K	$1K	$8K
contractor d	$7K	$6K	$9K	$4K

- e.g., a → 1, b → 2, c → 3, d → 4 9 + 4 + 1 + 4 = $18K total
- e.g., a → 2, b → 3, c → 1, d → 4 2 + 3 + 5 + 4 = $14K total

generate & test?

17

B & B example (cont.)

	job 1	job 2	job 3	job 4
contractor a	$9K	$2K	$7K	$8K
contractor b	$6K	$4K	$3K	$7K
contractor c	$5K	$8K	$1K	$8K
contractor d	$7K	$6K	$9K	$4K

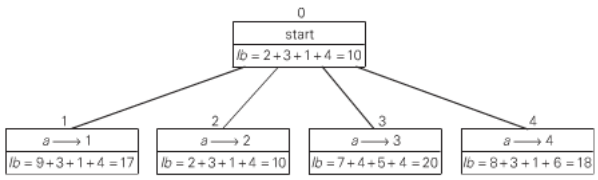
note that there is a (possibly unobtainable) lower bound on the bid total

- you can't possibly do better than assigning every contractor their lowest bid
- here, 2 + 3 + 1 + 4 = $10K is a lower bound
- (it is not even achievable, since b & c are assigned the same job)

the lower bound gives us a basis for choosing one branch over another

- i.e., use a greedy approach to select the branch with smallest lower bound

  lb = cost of bids assigned so far + minimal bids possible for remaining contractors



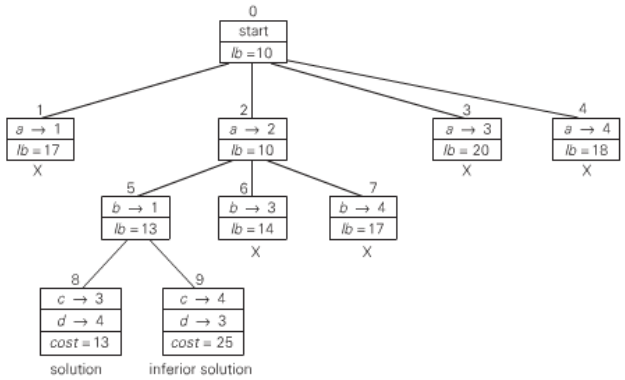
18

B & B example (cont.)

	job 1	job 2	job 3	job 4
contractor a	$9K	$2K	$7K	$8K
contractor b	$6K	$4K	$3K	$7K
contractor c	$5K	$8K	$1K	$8K
contractor d	$7K	$6K	$9K	$4K

at each step, choose the vertex/state with smallest lower bound, and extend

- can cut off a branch if its lb exceeds the cost of a found solution



Algorithmic approaches summary (so far)

brute force: sometimes the straightforward approach suffices

transform & conquer: sometimes the solution to a simpler variant suffices

divide/decrease & conquer: tackles a complex problem by breaking it into smaller piece(s), solving each piece (often w/ recursion), and combining into an overall solution

- applicable for any application that can be divided into smaller or independent parts

greedy: makes a sequence of choices/actions, choose whichever looks best at the moment

- applicable when a solution is a sequence of moves & perfect knowledge is available

backtracking: makes a sequence of choices/actions (similar to greedy), but stores alternatives so that they can be attempted if the current choices lead to failure

- more costly in terms of time and memory than greedy, but general-purpose
- branch & bound variant cuts off search at some level and backtracks

Interesting aside: B & B search in game playing

consider games involving:

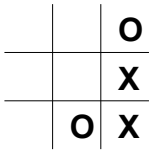
- 2 players
- perfect information
- zero-sum (player's gain is opponent's loss)

examples: tic-tac-toe, checkers, chess, othello, …

non-examples: poker, backgammon, prisoner's dilemma, …

von Neumann (the father of game theory) showed that for such games, there is always a "rational" strategy

- that is, can always determine a best move, assuming the opponent is equally rational

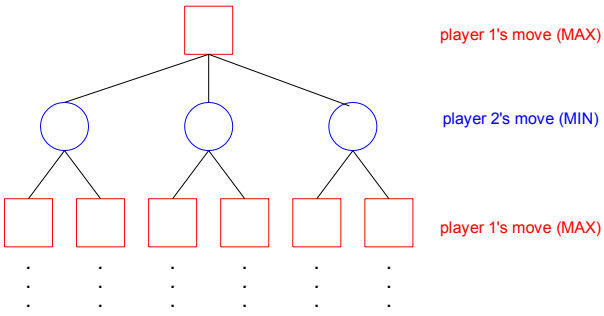


what is X's rational move?

Game trees

idea: model the game as a search tree

- associate a value with each game state (possible since zero-sum)
  player 1 wants to maximize the state value (call him/her MAX)
  player 2 wants to minimize the state value (call him/her MIN)
- players alternate turns, so differentiate MAX and MIN levels in the tree

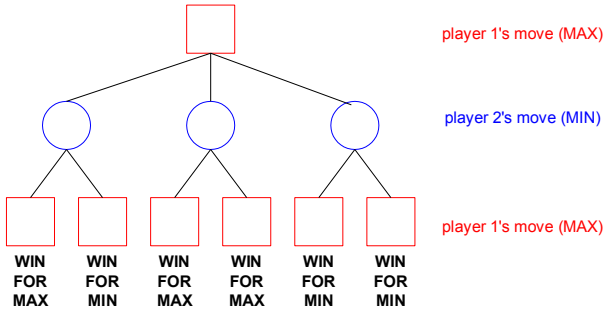


the leaves of the tree will be end-of-game states

Minimax search

minimax search:

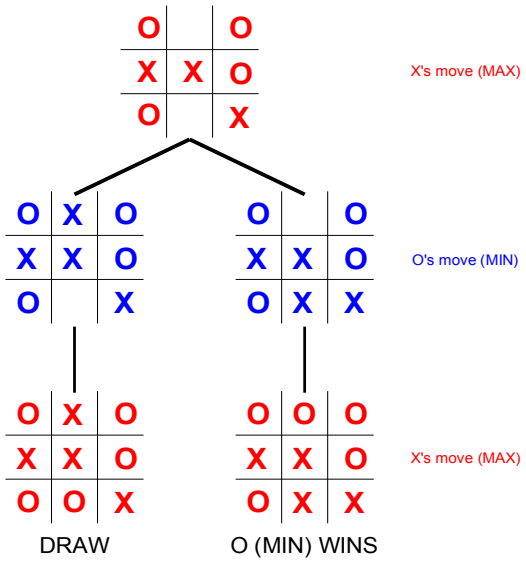
- at a MAX level, take the maximum of all possible moves
- at a MIN level, take the minimum of all possible moves



can visualize the search bottom-up (start at leaves, work up to root)
likewise, can search top-down using recursion

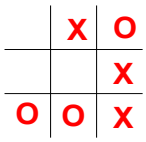
23

Minimax example



24

In-class exercise



X's move (MAX)

25

Minimax in practice

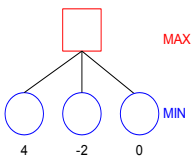
while Minimax Principle holds for all 2-party, perfect info, zero-sum games, an exhaustive search to find best move may be infeasible

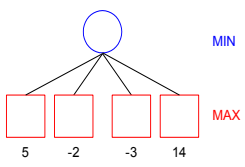
EXAMPLE: in an average chess game, ~100 moves with ~35 options/move
→ ~35^{100} states in the search tree!

practical alternative: limit the search depth and use heuristics

- expand the search tree a limited number of levels (limited look-ahead)
- evaluate the "pseudo-leaves" using a heuristic
  high value → good for MAX low value → good for MIN

back up the heuristic estimates to determine the best-looking move
at MAX level, take maximum at MIN level, take minimum



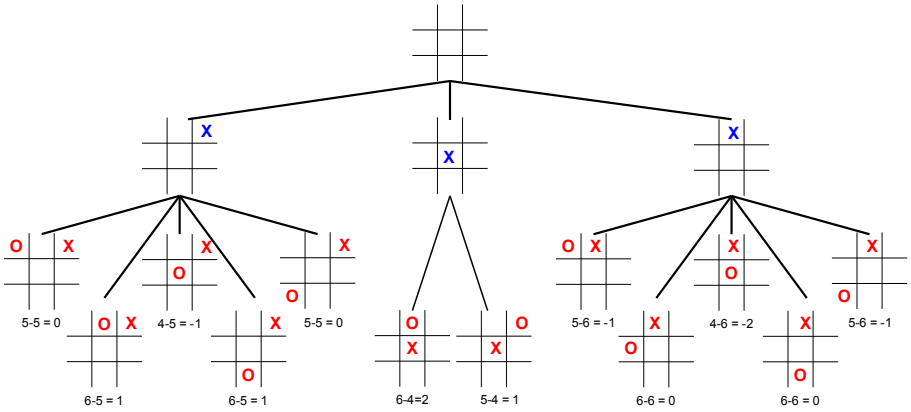


26

Tic-tac-toe example

heuristic(State) = { 1000 *if win for MAX (X)*; -1000 *if win for MIN (O)*; (#rows/cols/diags open for MAX – #rows/cols/diags open for MIN) *otherwise* }

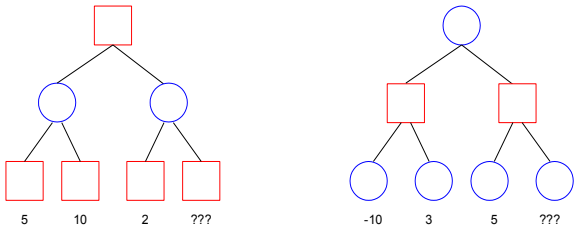
suppose look-ahead of 2 moves



27

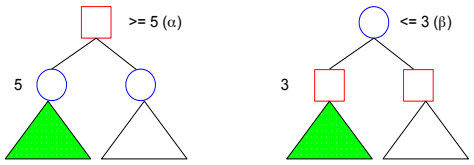
α-β bounds

sometimes, it isn't necessary to search the entire tree



α-β technique: associate bonds with state in the search

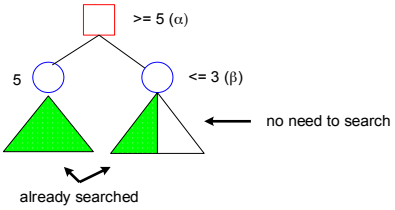
- associate lower bound α with MAX: can increase
- associate upper bound β with MIN: can decrease

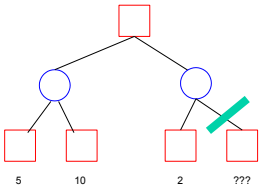


28

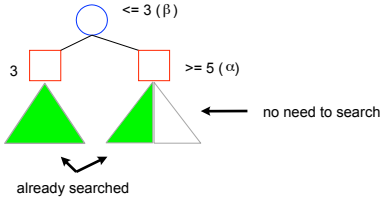
α-β pruning

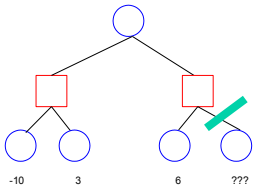
discontinue search below a MIN node if β value <= α value of ancestor





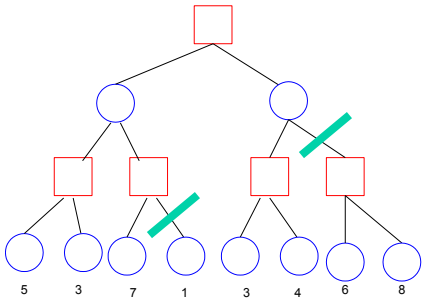
discontinue search below a MAX node if α value >= β value of ancestor





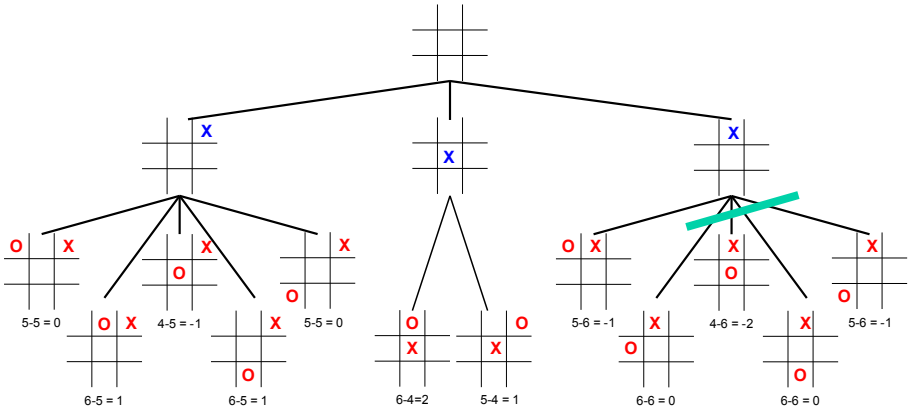
29

larger example



30

tic-tac-toe example



α-β vs. minimax:

worst case: α-β examines as many states as minimax

best case: assuming branching factor B and depth D, α-β examines ~$2b^{d/2}$ states

(i.e., as many as minimax on a tree with half the depth)

31

Iterative deepening

a common approach in game search is to set a lookahead range

- e.g., in chess, lookahead 4 moves (by each player) and rate boards at that stage
- this catches wins/losses within that range
- presumably, you can better judge the state of the game in the future

if decisions are timed,

- you can pick a conservative lookahead range to ensure a choice is made
- if time remains, extend the lookahead range and try again
- each iteration looks deeper and so makes a more informed choice

clearly, there is redundancy with iterative deepening

- lookead search of (n+1) levels must reproduce the search for n levels
- HOW COSTLY IS THIS?

32