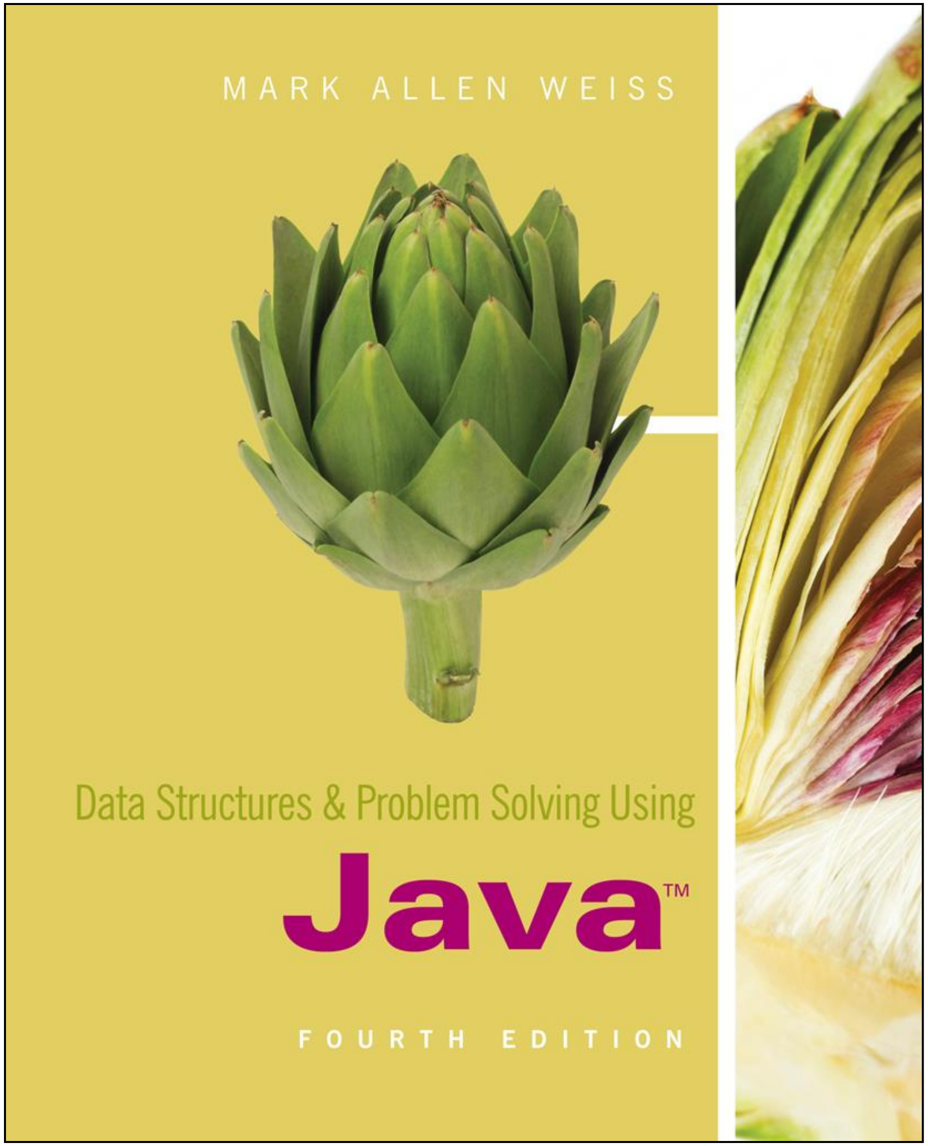
Data Structures
Lesson 12

BSc in Computer Science
University of New York, Tirana

Assoc. Prof. Marenglen Biba

Chapter 10

Fun and Games



Addison-Wesley is an imprint of PEARSON

Word search puzzle

- The input to the word search puzzle problem is a two-dimensional array of characters and a list of words, and the object is to find the words in the grid.

- These words may be horizontal, vertical, or diagonal in any direction (for a total of eight directions).

Brute force algorithm

For each word W in the word list

    for each row R

        for each column C

            for each direction D

            check if W exists at row R, column C in direction D.

figure 10.1

A sample word search grid

	0	1	2	3
0	t	h	i	s
1	w	a	t	s
2	o	a	h	g
3	f	g	d	t

Complexity

- Because there are eight directions, this algorithm requires eight word/row/column (8WRC) checks.
- Typical puzzles published in magazines feature 40 or so words and a 16 x 16 grid, which involves roughly 80,000 checks.
- That number is certainly easy to compute on any modern machine.
- Suppose, however, that we consider the variation in which only the puzzle board is given and the word list is essentially an English dictionary.
- In this case, the number of words might be 40,000 instead of 40, resulting in 80,000,000 checks.
- Doubling the grid would require 320,000,000 checks, which is no longer a trivial calculation.

Alternative algorithm

for each row R

for each column C

for each direction D

for each word length L

check if L chars starting at row R column C in direction D form a word

This algorithm rearranges the loop to avoid searching for every word in the word list.

If we assume that words are limited to 20 characters, the number of checks used by the algorithm is 160 RC. For a 32 x 32 puzzle, this number is roughly 160,000 checks.

The problem, of course, is that we must now decide whether a word is in the word list.

Finding a word in a list

- How to decide whether a word is in the word list?
- If we use a linear search, we lose.
- If we use a good data structure, we can expect an efficient search.
- If the word list is sorted, which is to be expected for an online dictionary, we can use a binary search and perform each check in roughly logW string comparisons.
  - For 40,000 words, doing so involves perhaps 16 comparisons per check
    - for a total of less than 3,000,000 string comparisons (16x160RC = 160x16x32x32=2.621.440).
    - This number of comparisons can certainly be done in a few seconds and is a factor of 100 better than the previous algorithm.

Improved algorithm

- We can further improve the algorithm based on the following observation.
- Suppose that we are searching in some direction and see the character sequence qx. An English dictionary will not contain any words beginning with qx.
- So is it worth continuing the innermost loop (over all word lengths)?
- The answer obviously is no: If we detect a character sequence that is not a prefix of any word in the dictionary, we can immediately look in another direction.

Improved algorithm

```
for each row R
   for each column C
      for each direction D
         for each word length L
            check if L chars starting at row R column C in
                        direction D form a word
            if they do not form a prefix,
            break; // the innermost
```

figure 10.2

The `WordSearch` class skeleton

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.InputStreamReader;
import java.io.IOException;

import java.util.Arrays;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;


// WordSearch class interface: solve word search puzzle
//
// CONSTRUCTION: with no initializer
// ******************PUBLIC OPERATIONS******************
// int solvePuzzle( )   --> Print all words found in the
//                          puzzle; return number of matches

public class WordSearch
{
    public WordSearch( ) throws IOException
      { /* Figure 10.3 */ }
    public int solvePuzzle( )
      { /* Figure 10.7 */ }

    private int rows;
    private int columns;
    private char theBoard[ ][ ];
    private String [ ] theWords;
    private BufferedReader puzzleStream;
    private BufferedReader wordStream;
    private BufferedReader in = new
            BufferedReader( new InputStreamReader( System.in ) );

    private static int prefixSearch( String [ ] a, String x )
      { /* Figure 10.8 */ }
    private BufferedReader openFile( String message )
      { /* Figure 10.4 */ }
    private void readWords( ) throws IOException
      { /* Figure 10.5 */ }
    private void readPuzzle( ) throws IOException
      { /* Figure 10.6 */ }
    private int solveDirection( int baseRow, int baseCol,
                                int rowDelta, int colDelta )
      { /* Figure 10.8 */ }
}
```

```
/**
 * Constructor for WordSearch class.
 * Prompts for and reads puzzle and dictionary files.
 */
public WordSearch( ) throws IOException
{
    puzzleStream = openFile( "Enter puzzle file" );
    wordStream   = openFile( "Enter dictionary name" );
    System.out.println( "Reading files..." );
    readPuzzle( );
    readWords( );
}
```

figure 10.3

The `WordSearch` class constructor

```java
/**
 * Print a prompt and open a file.
 * Retry until open is successful.
 * Program exits if end of file is hit.
 */
private BufferedReader openFile( String message )
{
    String fileName = "";
    FileReader theFile;
    BufferedReader fileIn = null;

    do
    {
        System.out.println( message + ": " );

        try
        {
            fileName = in.readLine( );
            if( fileName == null )
                System.exit( 0 );
            theFile = new FileReader( fileName );
            fileIn  = new BufferedReader( theFile );
        }
        catch( IOException e )
          { System.err.println( "Cannot open " + fileName ); }
    } while( fileIn == null );

    System.out.println( "Opened " + fileName );
    return fileIn;
}
```

figure 10.4

The `openFile` routine for opening either the grid or word list file

```java
/**
 * Routine to read the dictionary.
 * Error message is printed if dictionary is not sorted.
 */
private void readWords( ) throws IOException
{
    List<String> words = new ArrayList<String>( );

    String lastWord = null;
    String thisWord;

    while( ( thisWord = wordStream.readLine( ) ) != null )
    {
        if( lastWord != null && thisWord.compareTo( lastWord ) < 0 )
        {
            System.err.println( "Dictionary is not sorted... skipping" );
            continue;
        }
        words.add( thisWord );
        lastWord = thisWord;
    }

    theWords = new String[ words.size( ) ];
    theWords = words.toArray( theWords );
}
```

If not sorted, it does not add the word

figure 10.5

The `readWords` routine for reading the word list

```java
/**
 * Routine to read the grid.
 * Checks to ensure that the grid is rectangular.
 * Checks to make sure that capacity is not exceeded is omitted.
 */
private void readPuzzle( ) throws IOException
{
    String oneLine;
    List<String> puzzleLines = new ArrayList<String>( );

    if( ( oneLine = puzzleStream.readLine( ) ) == null )
        throw new IOException( "No lines in puzzle file" );

    columns = oneLine.length( );
    puzzleLines.add( oneLine );

    while( ( oneLine = puzzleStream.readLine( ) ) != null )
    {
        if( oneLine.length( ) != columns )
            System.err.println( "Puzzle is not rectangular; skipping row" );
        else
            puzzleLines.add( oneLine );
    }

    rows = puzzleLines.size( );
    theBoard = new char[ rows ][ columns ];

    int r = 0;
    for( String theLine : puzzleLines )
        theBoard[ r++ ] = theLine.toCharArray( );
}
```

Put in every row the array of characters

figure 10.6

The `readPuzzle` routine for reading the grid

figure 10.7

The solvePuzzle routine for searching in all directions from all starting points

```
/**
 * Routine to solve the word search puzzle.
 * Performs checks in all eight directions.
 * @return number of matches
 */
public int solvePuzzle( )
{
    int matches = 0;

    for( int r = 0; r < rows; r++ )
        for( int c = 0; c < columns; c++ )
            for( int rd = -1; rd <= 1; rd++ )
                for( int cd = -1; cd <= 1; cd++ )
                    if( rd != 0 || cd != 0 )
                        matches += solveDirection( r, c, rd, cd );

    return matches;
}
```

8 directions

cd	rd
1	0
1	1
1	-1
0	1
0	-1
-1	0
-1	1
-1	-1

We give a direction by indicating a column direction and then a row direction. For instance, south is indicated by cd=0 and rd=l and northeast by cd=l and rd=-l; cd can range from -1 to 1 and rd from -1 to 1, except that both cannot be 0 simultaneously.

```java
/**
 * Search the grid from a starting point and direction.
 * @return number of matches
 */
private int solveDirection( int baseRow, int baseCol,
                            int rowDelta, int colDelta )
{
    String charSequence = "";
    int numMatches = 0;
    int searchResult;

    charSequence += theBoard[ baseRow ][ baseCol ];

    for( int i = baseRow + rowDelta, j = baseCol + colDelta;
             i >= 0 && j >= 0 && i < rows && j < columns;
             i += rowDelta, j += colDelta )
    {
        charSequence += theBoard[ i ][ j ];
        searchResult = prefixSearch( theWords, charSequence );

        if( searchResult == theWords.length )
            break;
        if( !theWords[ searchResult ].startsWith( charSequence ) )
            break;

        if( theWords[ searchResult ].equals( charSequence ) )
        {
            numMatches++;
            System.out.println( "Found " + charSequence + " at " +
                                baseRow + " " + baseCol + " to " +
                                i + " " + j );
        }
    }

    return numMatches;
}

/**
 * Performs the binary search for word search.
 * Returns the last position examined this position
 * either matches x, or x is a prefix of the mismatch, or there is
 * no word for which x is a prefix.
 */
private static int prefixSearch( String [ ] a, String x )
{
    int idx = Arrays.binarySearch( a, x );

    if( idx < 0 )
        return -idx - 1;
    else
        return idx;
}
```

figure 10.8

Implementation of a single search

This specifies direction!

No prefix found

A prefix found but not a match

Match found

Either not found, prefix, or match

figure 10.9

A simple `main` routine for the word search puzzle problem

```
  // Cheap main
public static void main( String [ ] args )
{
    WordSearch p = null;

    try
    {
        p = new WordSearch( );
    }
    catch( IOException e )
    {
        System.out.println( "IO Error: " );
        e.printStackTrace( );
        return;
    }

    System.out.println( "Solving..." );
    p.solvePuzzle( );
}
```

Tic-Tac-Toe

Tic-Tac-Toe

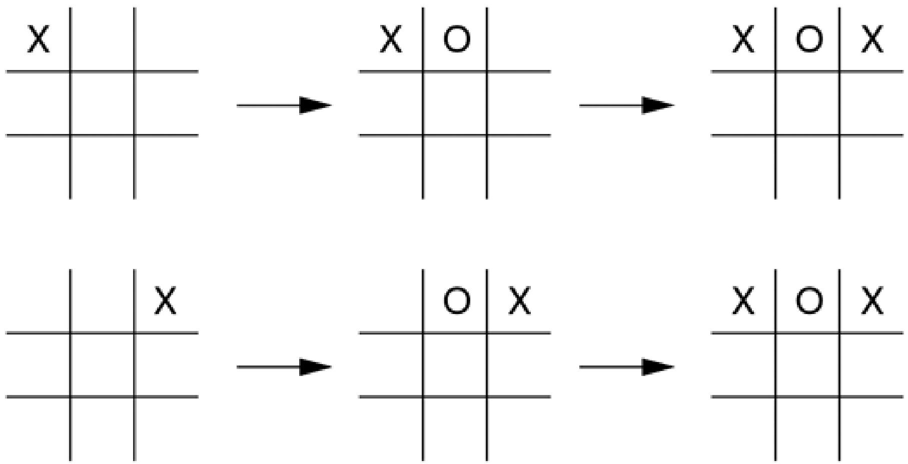


figure 10.12

Two searches that arrive at identical positions

Minimax

- Minimax (sometimes minmax) is a decision rule used in decision theory, game theory, statistics and philosophy for minimizing the possible loss for a worst case (maximum loss) scenario.
- Alternatively, it can be thought of as maximizing the minimum gain (maximin).
- Originally formulated for two-player zero-sum game theory, covering both the cases where players take alternate moves and those where they make simultaneous moves, it has also been extended to more complex games and to general decision making in the presence of uncertainty.

Minimax Theorem

- In the theory of simultaneous games, a minimax strategy is a mixed strategy which is part of the solution to a zero-sum game.
- In zero-sum games, the minimax solution is the same as the Nash equilibrium.
- This theorem was established by John von Neumann

Minimax Theorem

Minimax theorem

- For every two-person, zero-sum game with finitely many strategies, there exists a value V and a mixed strategy for each player, such that
  - (a) Given player 2's strategy, the best payoff possible for player 1 is V, and
  - (b) Given player 1's strategy, the best payoff possible for player 2 is −V.

Minimax Theorem

- Equivalently, Player 1's strategy guarantees him a payoff of V regardless of Player 2's strategy, and similarly Player 2 can guarantee himself a payoff of −V.

- The name minimax arises because each player minimizes the maximum payoff possible for the other — since the game is zero-sum, he also maximizes his own minimum payoff.

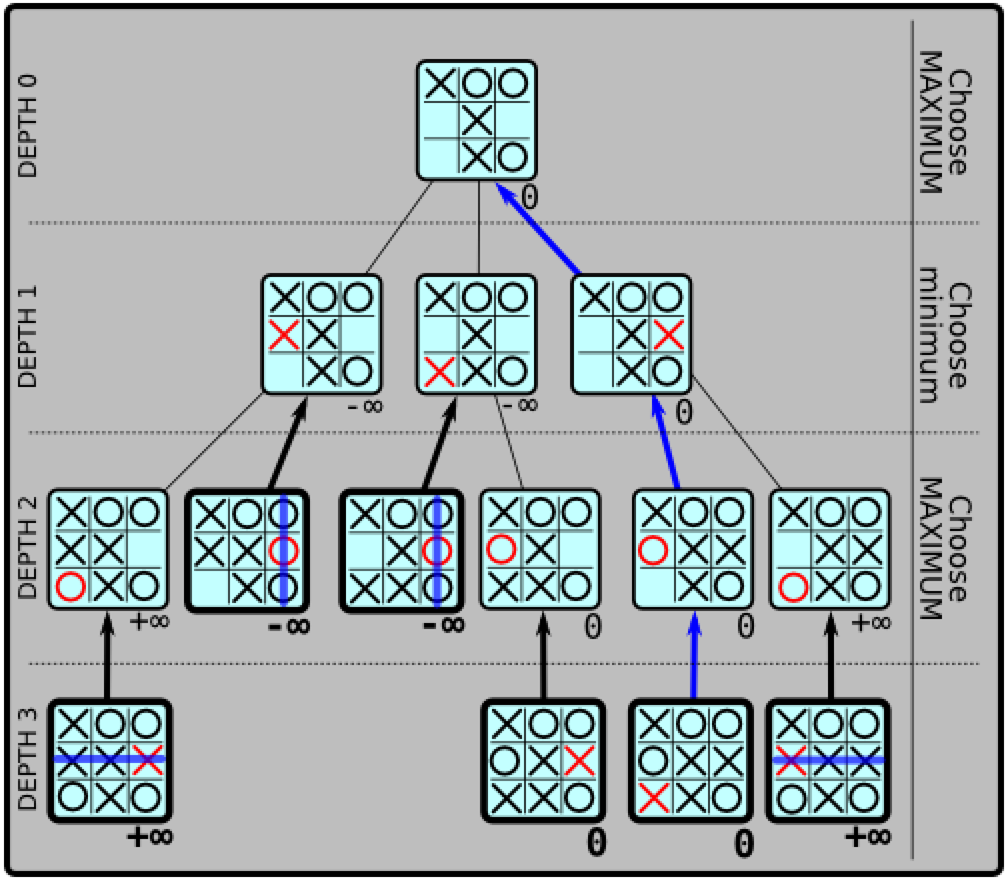
Minimax strategy

- The strategy used is the minimax strategy, which is based on the assumption of optimal play by both players.
- The value of a position is a COMPUTER_WIN if optimal play implies that the computer can force a win.
- If the computer can force a draw but not a win, the value is DRAW; if the human player can force a win, the value is HUMAN_WIN.
- We want the computer to win, so we have HUMAN_WIN < DRAW < COMPUTER_WIN.

Minimax

- For the computer, the value of the position is the maximum of all the values of the positions that can result from making a move.
- Scenario
  - Suppose that one move leads to a winning position, two moves lead to a drawing position, and six moves lead to a losing position.
  - Then the starting position is a winning position because the computer can force the win.
- Moreover, the move that leads to the winning position is the move to make.
- For the human player we use the minimum instead of the maximum.

Minimax for Tic-Tac-Toe



Minimax for Tic-Tac-Toe

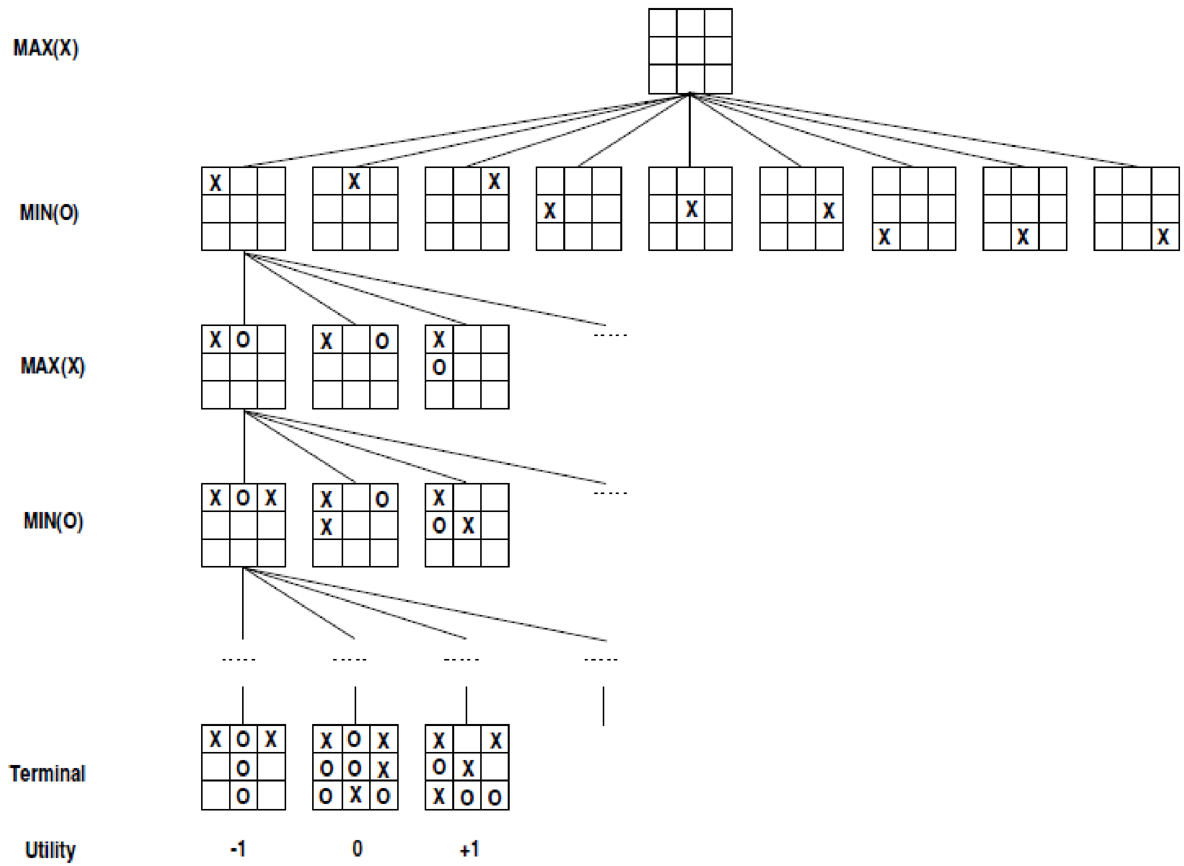


figure 7.26

Class to store an evaluated move

```
final class Best
{
    int row;
    int column;
    int val;

    public Best( int v )
      { this( v, 0, 0 ); }

    public Best( int v, int r, int c )
      { val = v; row = r; column = c; }
}
```

```
class TicTacToe
{
    public static final int HUMAN        = 0;
    public static final int COMPUTER     = 1;
    public static final int EMPTY        = 2;

    public static final int HUMAN_WIN    = 0;
    public static final int DRAW         = 1;
    public static final int UNCLEAR      = 2;
    public static final int COMPUTER_WIN = 3;

        // Constructor
    public TicTacToe( )
      { clearBoard( ); }

        // Find optimal move
    public Best chooseMove( int side )
      { /* Implementation in Figure 7.29 */  }

        // Compute static value of current position (win, draw, etc.)
    private int positionValue( )
      {  /* Implementation in Figure 7.28 */ }

        // Play move, including checking legality
    public boolean playMove( int side, int row, int column )
      { /* Implementation in online code */ }

        // Make board empty
    public void clearBoard( )
      { /* Implementation in online code */ }

        // Return true if board is full
    public boolean boardIsFull( )
      { /* Implementation in online code */ }

        // Return true if board shows a win
    public boolean isAWin( int side )
      { /* Implementation in online code */ }

        // Play a move, possibly clearing a square
    private void place( int row, int column, int piece )
      { board[ row ][ column ] = piece; }

        // Test if a square is empty
    private boolean squareIsEmpty( int row, int column )
      { return board[ row ][ column ] == EMPTY; }

    private int [ ] [ ] board = new int[ 3 ][ 3 ];
}
```

figure 7.27

Skeleton for class `TicTacToe`

figure 7.28

Supporting routine for evaluating positions

```
// Compute static value of current position (win, draw, etc.)
private int positionValue( )
{
    return isAWin( COMPUTER ) ? COMPUTER_WIN :
           isAWin( HUMAN )    ? HUMAN_WIN :
           boardIsFull( )     ? DRAW         : UNCLEAR;
}
```

```java
// Find optimal move
public Best chooseMove( int side )
{
    int opp;              // The other side
    Best reply;           // Opponent's best reply
    int dc;               // Placeholder
    int simpleEval;       // Result of an immediate evaluation
    int bestRow = 0;
    int bestColumn = 0;
    int value;

    if( ( simpleEval = positionValue( ) ) != UNCLEAR )
        return new Best( simpleEval );

    if( side == COMPUTER )
    {
        opp = HUMAN; value = HUMAN_WIN;
    }
    else
    {
        opp = COMPUTER; value = COMPUTER_WIN;
    }

    for( int row = 0; row < 3; row++ )
        for( int column = 0; column < 3; column++ )
            if( squareIsEmpty( row, column ) )
            {
                place( row, column, side );
                reply = chooseMove( opp );
                place( row, column, EMPTY );

                  // Update if side gets better position
                if( side == COMPUTER && reply.val > value
                    || side == HUMAN && reply.val < value )
                {
                    value = reply.val;
                    bestRow = row; bestColumn = column;
                }
            }

    return new Best( value, bestRow, bestColumn );
}
```

figure 7.29

A recursive routine for finding an optimal Tic-Tac-Toe move

Set value

Recursive call with opponents' turn

Alpha-Beta Pruning

Alpha-Beta Pruning

- Although the minimax strategy gives an optimal Tic-Tac-Toe move, it performs a lot of searching.
- Specifically, to choose the first move, it makes roughly a half-million recursive calls.
- One reason for this large number of calls is that the algorithm does more searching than necessary.

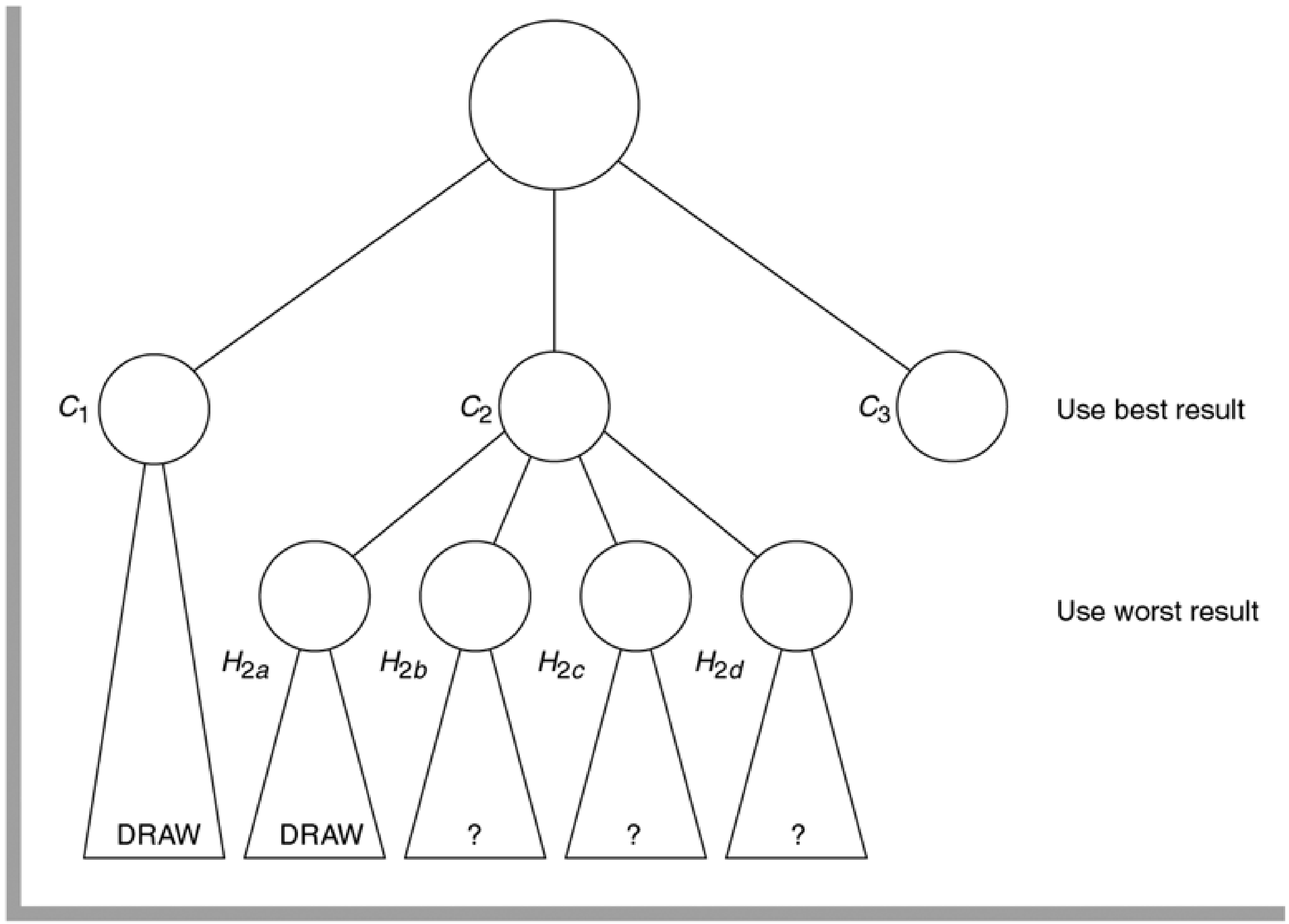
TTT Scenario

- Suppose that the computer is considering five moves: C1, C2, C3, C4, and C5. Suppose also that the recursive evaluation of C1 reveals that C1 forces a draw.
- Now C2 is evaluated.
- At this stage, we have a position from which it would be the human player's turn to move.
- Suppose that in response to C2, the human player can consider H2a, H2b, H2c, and H2d, Further, suppose that an evaluation of H2a shows a forced draw.
- Automatically, C2 is at best a draw and possibly even a loss for the computer (because the human player is assumed to play optimally). Because we need to improve on C1, we do not have to evaluate any of H2b, H2c, and H2d.
- We say that H2a is a refutation, meaning that it proves that C2 is not a better move than what has already been seen.

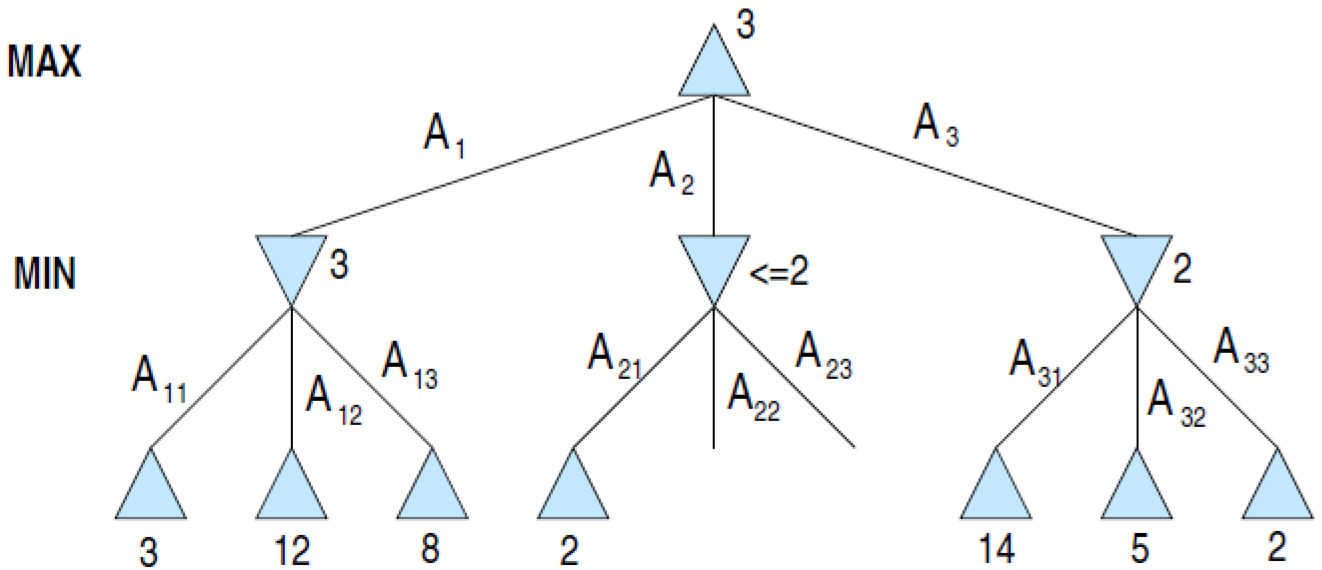
Alpha-Beta Pruning

figure 10.10

Alpha–beta pruning: After H_{2a} is evaluated, C_2, which is the minimum of the H_2's, is at best a draw. Consequently, it cannot be an improvement over C_2. We therefore do not need to evaluate H_{2b}, H_{2c}, and H_{2d} and can proceed directly to C_3.



Alpha-Beta Pruning



- A_{21} gives worse choice than A_1, so prune at A_2 since we know that A_2 will always be worse than A_1
- All A_3's children expanded, since it is the *last child* that gives value of 2.

```java
    // Find optimal move
private Best chooseMove( int side, int alpha, int beta, int depth )
{
    int opp;              // The other side
    Best reply;           // Opponent's best reply
    int dc;               // Placeholder
    int simpleEval;       // Result of an immediate evaluation
    int bestRow = 0;
    int bestColumn = 0;
    int value;

    if( ( simpleEval = positionValue( ) ) != UNCLEAR )
        return new Best( simpleEval );

    if( side == COMPUTER )
    {
        opp = HUMAN; value = alpha;
    }
    else
    {
        opp = COMPUTER; value = beta;
    }

Outer:
    for( int row = 0; row < 3; row++ )
        for( int column = 0; column < 3; column++ )
            if( squareIsEmpty( row, column ) )
            {
                place( row, column, side );
                reply = chooseMove( opp, alpha, beta, depth + 1 );
                place( row, column, EMPTY );

                if( side == COMPUTER && reply.val > value ||
                    side == HUMAN && reply.val < value )
                {
                    if( side == COMPUTER )
                        alpha = value = reply.val;
                    else
                        beta = value = reply.val;

                    bestRow = row; bestColumn = column;
                    if( alpha >= beta )
                        break Outer;  // Refutation
                }
            }

    return new Best( value, bestRow, bestColumn );
}
```

Cut-point

figure 10.11

The chooseMove routine for computing an optimal Tic-Tac-Toe move, using alpha–beta pruning

Transposition tables

- Another commonly employed practice is to use a table to keep track of all positions that have been evaluated.
- For instance, in the course of searching for the first move, the program will examine the positions shown in Figure 10.12. (next slide)
- If the values of the positions are saved, the second occurrence of a position need not be recomputed; it essentially becomes a terminal position.
- The data structure that records and stores previously evaluated positions is called a transposition table;
  - It is implemented as a map of positions to values.

```
final class Position
{
    private int [ ][ ] board;

    public Position( int [ ][ ] theBoard )
    {
        board = new int[ 3 ][ 3 ];
        for( int i = 0; i < 3; i++ )
            for( int j = 0; j < 3; j++ )
                board[ i ][ j ] = theBoard[ i ][ j ];
    }

    public boolean equals( Object rhs )
    {
        if( ! (rhs instanceof Position ) )
            return false;

        Position other = (Position) rhs;

        for( int i = 0; i < 3; i++ )
            for( int j = 0; j < 3; j++ )
                if( board[ i ][ j ] != ( (Position) rhs ).board[ i ][ j ] )
                    return false;
        return true;
    }

    public int hashCode( )
    {
        int hashVal = 0;

        for( int i = 0; i < 3; i++ )
            for( int j = 0; j < 3; j++ )
                hashVal = hashVal * 4 + board[ i ][ j ];

        return hashVal;
    }
}
```

figure 10.13

The `Position` class

```
// Original import directives plus:
import java.util.Map;
import java.util.HashMap;

class TicTacToe
{
    private Map<Position,Integer> transpositions
                                    = new HashMap<Position,Integer>( );

    public Best chooseMove( int side )
      { return chooseMove( side, HUMAN_WIN, COMPUTER_WIN, 0 ); }

        // Find optimal move
    private Best chooseMove( int side, int alpha, int beta, int depth )
      { /* Figures 10.15 and 10.16 */ }

      ...
}
```

figure 10.14

Changes to the `TicTacToe` class to incorporate transposition table and alpha–beta pruning

```
    // Find optimal move
private Best chooseMove( int side, int alpha, int beta, int depth )
{
    int opp;              // The other side
    Best reply;           // Opponent's best reply
    int dc;               // Placeholder
    int simpleEval;       // Result of an immediate evaluation
    Position thisPosition = new Position( board );
    int tableDepth = 5;   // Max depth placed in Trans. table
    int bestRow = 0;
    int bestColumn = 0;
    int value;

    if( ( simpleEval = positionValue( ) ) != UNCLEAR )
        return new Best( simpleEval );

    if( depth == 0 )
        transpositions.clear( );
    else if( depth >= 3 && depth <= tableDepth )
    {
        Integer lookupVal = transpositions.get( thisPosition );
        if( lookupVal != null )
            return new Best( lookupVal );
    }

    if( side == COMPUTER )
    {
        opp = HUMAN; value = alpha;
    }
    else
    {
        opp = COMPUTER; value = beta;
    }
```

Check the table and get the value of the board

figure 10.15

The Tic-Tac-Toe algorithm with alpha–beta pruning and transposition table (part 1)

```
    Outer:
        for( int row = 0; row < 3; row++ )
            for( int column = 0; column < 3; column++ )
                if( squareIsEmpty( row, column ) )
                {
                    place( row, column, side );
                    reply = chooseMove( opp, alpha, beta, depth + 1 );
                    place( row, column, EMPTY );

                    if( side == COMPUTER && reply.val > value ||
                        side == HUMAN && reply.val < value )
                    {
                        if( side == COMPUTER )
                            alpha = value = reply.val;
                        else
                            beta = value = reply.val;

                        bestRow = row; bestColumn = column;
                        if( alpha >= beta )
                            break Outer;  // Refutation
                    }
                }

        if( depth <= tableDepth )
            transpositions.put( thisPosition, value );

        return new Best( value, bestRow, bestColumn );
    }
```

figure 10.16

The Tic-Tac-Toe algorithm with alpha–beta pruning and transposition table (part 2)

Speedup with data structures

- The use of the transposition table in this Tic-Tac-Toe algorithm removes about half the positions from consideration, with only a slight cost for the transposition table operations.

- The program's speed is almost doubled.

End of class

- Readings
  - Minimax: chapter 7
  - Today’s class: chapter 10