Computing Science (CMPUT) 455

Search, Knowledge, and Simulations

James Wright

Department of Computing Science
University of Alberta
james.wright@ualberta.ca

Fall 2021

Today's Topics - Lecture 10

- More on Minimax and Alphabeta
- Python sample codes
- Solve TicTacToe again, now with alphabeta
- Compare alphabeta with naive negamax and boolean negamax
- Iterative deepening
- Alphabeta and proof trees; principal variation
- Search enhancements: transposition table

Coursework

- Work on Assignment 2
  - Deadline extended to Monday Oct 18
- Quiz 5: review minimax search parts 1 and 2. Double-length quiz
  - Deadline extended to Friday Oct 8 (tomorrow)
- Read Schaeffer et al, *Checkers is solved.* Science, 2007
- Quiz 6 (due Monday)
- Activities 10

Midterm

- The midterm is Oct 12 (this coming Tuesday)
- Topics: All material up to and including lecture 10 (today)
- Midterm study guide is available from main course page
- Exam on eclass, similar to quizzes
  - 90-minute time limit (modulo user-specific accomodations)
  - Opens 12:01am, closes 11:59pm Mountain time
  - You must start before 10:29pm if you want the full 90 minutes
- No lecture on Tuesday

Review - Minimax and Alpha-Beta

- Solve game tree for general case
- More than two (win-loss) outcomes
- Result in leaf nodes: numerical score
- Example: win-loss-draw, coded as e.g.
  win = +1, draw = 0, loss = -1
- Minimax: player maximizes their score,
  opponent minimizes
- Alphabeta: prune if move is outside alphabeta window
- Meaning of window: moves outside are too bad for one of the players, that player will make a different choice earlier on

Minimax and Alphabeta Sample Code

- New static evaluation function in `tic_tac_toe_integer_eval.py`
- The example is for negamax, from `toPlay`'s point of view
- Can also be used for depth-bounded search, if evaluation is also called for interior nodes: `alphabeta_depth_limited_tictactoe_test.py`
- Note: this uses *no* heuristic, so it is blind search
- Evaluation is exact at leaf nodes, 0 everywhere else

Solve TicTacToe with Negamax and Alphabeta

- Compare Three Search Algorithms
  - Solve TicTacToe in three different ways
- Naive negamax, alphabeta, boolean negamax
- boolean negamax test in `boolean_negamax_test_tictactoe.py`
- Naive negamax, alphabeta test in `alphabeta_tictactoe_test.py`
- All solve the game
- Performance of Alphabeta, boolean negamax is similar
- Naive negamax not competitive - no pruning at all
- None of these programs use a heuristic
- We can easily add a heuristic

Activity 10a: add a heuristic for TicTacToe

- Add a heuristic in the function `staticallyEvaluateForToPlay()`
- Idea: highest value for sure wins (3 in a row complete), lowest for losses
- If not won or lost: scan all eight lines on the board (3 horizontal, 3 vertical, 2 diagonal)
- Compute a score for each line depending on how good or bad it is for you
- Add up all those scores to get an evaluation function

Activity 10a Continued

- How to evaluate a line?
- Check different "features" and give a bonus or penalty when they are present
  - If the line is blocked for both, then value 0
    - Examples: `xox` `.ox` `xxo`
  - If a line is an "open two", then very valuable
    - Examples: `xx.` `o.o`
  - If a line is an "open one", then has some value for that player
    - Examples: `x..` `.o.`
  - Completely open line
    - Example: `...`

Activity 10a-c Questions to Explore

- For “open two”, should they be the same value for your own color and the opponent? Or is one more valuable than the other?
- How about a completely open line ... ? Should it be neutral, or a small advantage for the current player?
- What works better: adding up all features, or finding the most important one? Why?
- What are good “feature weights”? Experiment with different choices.
- Activities 10b and 10c: Test if the solver needs fewer nodes, and becomes faster.

Depth-limited Alphabeta

From `alphabeta_depth_limited.py`

```
def alphabetaDL(state, alpha, beta, depth):
    if state.endOfGame() or depth == 0:
        return state.staticallyEvaluateForToPlay()
    ...
        value = -alphabetaDL(state, -beta,
                             -alpha, depth - 1)
    ...

# initial call with full window
def callAlphabetaDL(rootState, depth):
    return alphabetaDL(rootState, -INFINITY,
                       INFINITY, depth)
```

Experiment: Explore Depth-limited Alphabeta

`alphabeta_depth_limited_tictactoe_test.py`

- TicTacToe with evaluation scores in `tic_tac_toe_integer_eval.py`
- Runs alphabeta with different depth limits: iterative deepening
- Example 1: empty board. Result always 0
- Example 2: win for X. Result changes to win score when win is proven at depth 5

Experiment: Explore Depth-limited Alphabeta

`alphabeta_depth_limited_tictactoe_test.py`

- TicTacToe with evaluation scores in `tic_tac_toe_integer_eval.py`
- Runs alphabeta with different depth limits: iterative deepening
- Example 1: empty board. Result always 0
- Example 2: win for X. Result changes to win score when win is proven at depth 5
- Interpretation: against best response:
  - Black can win in 5 moves
  - Black cannot win in 4 or fewer moves

Alphabeta and Proofs

- Assume the minimax value of a game is m
- alphabeta search (with no depth limit) computes the score m
- We can view alphabeta as finding two proofs at the same time
- Max player can guarantee at least m
- Min player can limit score to at most m

Alphabeta and Proofs - Continued

- If we store information about all nodes in the search, then we have these two strategies stored explicitly
- Also remember the relation to boolean minimax:
  - If we know a candidate value *m*:
  - We can do two boolean searches with tests $\geq m$ and $> m$
  - Together they can verify that *m* is the minimax result

Alphabeta and Playing Optimally - OR Node

- Assume both players follow best play based on the stored values
- Assume the root n is an OR node with minimax value $score(n) = m$
- Children $c_1, ..., c_k$:
- score(n) = m = max(score(c_1), score(c_2), ... score(c_k))
- OR player can find (at least) one child c_i with score(c_i) = m, and play that move
- This move leads to an AND node

Alphabeta and Playing Optimally - AND Node

- c_i is an AND node
- score(c_i) = m = min(score(c_{i1}), score(c_{i2}), ...
- AND player can find (at least) one child c_{ij} with score(c_{ij}) = m, and play that move
- This move leads to an OR node
- Repeat the same arguments until the end of the game

Principal Variation (PV)

- If both players play best moves:
  they follow a *principal variation* or PV of the search
- This is a move sequence with the property that
  each node in the sequence has a score of m
- Even with a depth-limited search and heuristic evaluation,
  a PV exists
- It will only go as deep as the search
- All these nodes have the same value as the
  heuristic evaluation of the last node in the sequence

Principal Variation and Proof Trees

- Consider the two parts of the proof that the minimax value is m
  - Proof that the max player can get at least m
  - Proof that the min player can get m or less
- If both players follow their proof, they will play out a PV
- The reverse is also true:
- Assume both players follow a move sequence S, such that for all nodes along that sequence the minimax score is m
- Then there exist two proof trees:
  - $P1$ for max
  - $P2$ for min
  - S is the intersection of $P1$ and $P2$ (set of nodes that are in both trees)

Summary of Basic Solving Algorithms

- Alphabeta and Negamax
- Alphabeta performance similar to boolean negamax here
- Naive negamax much worse, no pruning
- Discussed relation between alphabeta and proof trees
- Principal variation: a line with best play for both

Minimax Search Enhancements

Search Enhancements - Overview

- Hashing to cache search results
- Transposition table
- From searching a tree to searching a DAG
- Iterative deepening and move ordering
- History heuristic
- More alphabeta search improvements

How to Store Information About Search Nodes?

- In our minimax codes so far we did not store any information on search states
- The searches just returned a boolean, or an integer
- The searches used *depth-first* order
  - Go to first child, then first child of first child,...
- What if we want more detailed search results, and store them?
- Examples:
  - Store the best move
  - Store a proof tree
  - Store search statistics
  - Re-use search results for a later, deeper search

Get Best Move and PV

- It is easy to modify search to return both the score and a PV
- However, the overhead is quite large
- Very many move sequences are created during search
- Almost all of them are discarded later
- Much more efficient approach: use a **transposition table**
- Can get best move and PV information almost for free
- Several other important benefits

Big Idea: Caching Information

- A cache is an information store
- Example: on-chip cache for CPU
  - Accesses data much faster than loading from main memory
- Example: cache for rendered web pages
  - Data in cache is stored locally as opposed to loading from web, parsing html, loading images, building on-screen image, recomputing...

Hashing and Transposition Table

- Idea:
  Store game positions and its search information
- Examples: minimax score, win/loss flag, best move, search depth reached in iterative search, number of nodes searched, timestamp (when solved),...
- How to store?
- Typically, a fixed-size array is used for a search
- For our simple example,
  we just use a Python dictionary

Storing a TicTacToe Position

- How to store?
- Standard approach: compute a *code* or *hash code* for the position
- Store in the transposition table under this code
- For TicTacToe, less than $3^9 = 19{,}683$ states total
- Can easily store all states that we search

Code for a TicTacToe Position

- Remember our 1-d array representation
- Number code for each point
  - EMPTY = 0
  - BLACK = 1, used for ’X’
  - WHITE = 2, used for ’O’

```
# Board stored in array of size 9:
# 0 1 2
# 3 4 5
# 6 7 8
```

- View state as array of codes:
- `s = [1, 1, 2, 2, 2, 1, 1, 0, 0]`
- Example: `s[0] = 1` means top left corner is ’X’

Store Code and Data in Simple Transposition Table

- Array of codes: s = [1, 1, 2, 2, 2, 1, 1, 0, 0]
- Code of state: treat codes as a base 3 integer = 112221100 in base 3
- code(s) =
  $1 \times 3^8 + 1 \times 3^7 + 2 \times 3^6 + 2 \times 3^5 +$
  $+ 2 \times 3^4 + 1 \times 3^3 + 1 \times 3^2 + 0 \times 3^1 + 0 \times 3^0$
- Store pairs `(code(s), data(s))`
- To store in a Python dictionary
  - Use `code(s)` as the key
  - Store `data(s)` as the value under that key

Example: Simple Transposition Table

- `transposition_table_simple.py`
- Store boolean result score (True or False) as value
    - It is easy to store best move as well
- Use code as key
- Lookup failure: return `None`
- Lookup success - return score: `True`, or `False`

```
class TranspositionTable(object):
    ...
    def store(self, code, score):
        self.table[code] = score

    def lookup(self, code):
        return self.table.get(code)
```

Example: Boolean Negamax with Simple Transposition Table

- `boolean_negamax_tt.py`
- Always try lookup first
- If succeeds:
  - Done, no search needed
- Otherwise:
  - Do the regular search
  - Store result in table before return from function

```
def negamaxBoolean(state, tt):
  result = tt.lookup(state.code())
  if result != None:
      return result
  ...
```

`boolean_negamax_tt.py` continued

```
    if state.endOfGame():
        result = state.staticallyEvaluateForToPlay()
        return storeResult(tt, state, result)
    for m in state.legalMoves():
        state.play(m)
        success = not negamaxBoolean(state,tt)
        state.undoMove()
        if success:
            return storeResult(tt, state, True)
    return storeResult(tt, state, False)

def storeResult(tt, state, result):
    tt.store(state.code(), result)
    return result
```

Apply Transposition Table to Solving TicTacToe

- `tic_tac_toe_solve_with_tt.py`
- About 3x faster than without table
- For larger problems (Go, NoGo, Gomoku, ...) using the table can be several orders of magnitude faster

Full Transposition Table

- Problem with our simple approach so far:
- Does not scale to large searches
- Using dictionary to store all states will fill memory within seconds
  - For a fast program written in something like C++ anyway...
- We need a solution that works with fixed memory limit
- Only store most important states
- Need information-losing hash codes (see next slide)

Example: Code for 19×19 Go

- Why do we not always use the full code?
- Example: Full code for 19×19 Go
- 3 states per point, $19 \times 19 = 361$ points
- Total $3^{361} > 2^{572}$ different codes
- Not even considering history here, which is needed for Ko rule - Go has even more distinct states
- Storing everything in a table is not feasible
- Using full 573+ bit codes is not necessary
- Standard today: use 64 bit codes

Zobrist Hash Codes

- How to compute a good 64 bit code for a state?
- Standard: Zobrist hashing, `https://en.wikipedia.org/wiki/Zobrist_hashing`
- Prepare one random number `code[point][color]` for each (point, color) combination
- Code of state is bitwise logical `xor` over all points on the board
- example:
  `board[0] = WHITE, board[1] = EMPTY, board[2] = BLACK,...`
- `hashcode = code[0][WHITE] xor code[1][EMPTY] xor code [2][BLACK] xor ...`

Bitwise xor in Python

- Option 1: use ˆ

```
0b1111011 ^ 0b111001000
```

- Option 2:

```
from operator import xor
xor(0b1111011, 0b111001000)
```

Transposition Table in Fixed Size Array

- Where in table to store state s?
- For a fixed size array, we need to compute an array index from the 64 bit code of s
- Typical solution:
  - Use array of some size 2^n
  - Take the first n bits of `code(s)` as the array index
- Avoid collisions: store full 64 bit code as part of data
- At each lookup, compare full 64 bit code
- Do not trust 64 bit codes for proofs!
  - Verify solution tree without using hashing

Transposition Table Entries

- What data to store?
- Depends on type of search
- For boolean negamax we only needed one bit
  - True/False minimax value of state
- For alphabeta, iterative deepening, need to store more
  - Best move from this state
  - Search score
  - Flags: exact value or upper or lower bound
  - Search depth
  - A flag whether it is exact result or heuristic score
- Details - `https://chessprogramming.org/Transposition_Table`

State Space of TicTacToe

Enumerating the State Space of TicTacToe

- TicTacToe has a small enough state space to create and count all states
- We will do it both for the tree and the DAG model
- How much do we save from using Transposition Table?

Estimating the Tree of TicTacToe

- `tic_tac_toe_estimate_tree.py`
- Estimate for the tree model
- Branching factor: 9 at root, then 8, 7, ...
- Model as in Lecture 4

Estimated Tic Tac Toe positions in the tree model:
$[1, 9, 72, 504, 3024, 15120, 60480, 181440, 362880, 362880]$

Enumerating the Tree of TicTacToe

- Now we can do the exact count for the tree model
- `tic_tac_toe_count_tree.py`

```
def countTicTacToeTree():
  t = TicTacToe()
  positionsAtDepth = [0] * 10
  countAtDepth(t, 0, positionsAtDepth)
  print("Tic Tac Toe positions in tree model: ",
         positionsAtDepth)
```

Enumerating the Tree of TicTacToe

```
def countAtDepth(t, depth, positionsAtDepth):
  positionsAtDepth[depth] += 1
  if t.endOfGame():
    return
  for i in range(9):
    if t.board[i] == EMPTY:
      t.play(i)
      countAtDepth(t, depth + 1,
                   positionsAtDepth)
      t.undoMove()
```

Enumerating the DAG of TicTacToe

- With transposition table, we can now count the size of the TicTacToe state space in the DAG model
- Main idea: skip states that we saw before
- `tic_tac_toe_count_dag.py`

```
def countTicTacToeDAG():
  tt = TranspositionTable()
  t = TicTacToe()
  positionsAtDepth = [0] * 10
  countAtDepth(t, 0, positionsAtDepth, tt)
  print("Tic Tac Toe positions in DAG model: ",
        positionsAtDepth)
```

Enumerating the DAG of TicTacToe

```
def countAtDepth(state, depth, positionsAtDepth, tt
  result = tt.lookup(state.code())
  if result != None:
    return
  tt.store(state.code(), True)
  positionsAtDepth[depth] += 1
  if state.endOfGame(): return
  for i in range(9):
    if state.board[i] == EMPTY:
      state.play(i)
      countAtDepth(state, depth + 1,
                   positionsAtDepth, tt)
      state.undoMove()
```

TicTacToe - DAG vs Tree Comparison

Run `tic_tac_toe_estimate_tree.py`, `tic_tac_toe_count_tree.py`, `tic_tac_toe_count_dag.py`

```
Estimated positions in the tree model:
[1, 9, 72, 504, 3024, 15120, 60480, 181440,
362880, 362880]
positions in tree model:
[1, 9, 72, 504, 3024, 15120, 54720, 148176,
200448, 127872]
positions in DAG model:
[1, 9, 72, 252, 756, 1260, 1520, 1140,
390, 78]
```

TicTacToe - DAG vs Tree Discussion

- Tree: estimate is exact at depth $0 \ldots 5$ (why?)
- DAG: No savings at lower levels (why?)
- Massive savings deeper in DAG

Another Application of Transposition Table: Solve All TicTacToe States

- `tic_tac_toe_solve_all.py`:
  Traverse whole state space
- Solve each state
- Store all solved nodes in transposition table
- Optional activity: modify the code to print each DAG state only once (use tt)

```
Solve all TicTacToe states black win/draw/white win
Depth 0: 0 black, 1 draws, 0 white, 1 total positions
Depth 1: 0 black, 9 draws, 0 white, 9 total positions
Depth 2: 48 black, 24 draws, 0 white, 72 total positions
Depth 3: 128 black, 276 draws, 100 white, 504 total positions
Depth 4: 2336 black, 544 draws, 144 white, 3024 total positions
Depth 5: 5472 black, 3168 draws, 6480 white, 15120 total positions
Depth 6: 38016 black, 7200 draws, 9504 white, 54720 total positions
Depth 7: 59472 black, 28800 draws, 59904 white, 148176 total positions
Depth 8: 81792 black, 46080 draws, 72576 white, 200448 total positions
Depth 9: 81792 black, 46080 draws, 0 white, 127872 total positions
```

More Alphabeta Improvements

Alphabeta Improvement: Iterative deepening and Move Ordering

- We have seen iterative deepening before
- Search with depth limit of 1, 2, 3, ...
- Scenario now: heuristic alphabeta search with a (good) evaluation function
- Even shallow searches will often find a good move
- Remember - alphabeta is most effective if strongest move is tried first
- Alphabeta window reduced most, can cut more moves
- Idea: first try the strongest move from previous search
- This is a very strong heuristic and used in most alphabeta implementations

Alphabeta Improvement: History Heuristic

- Invented by Jonathan Schaeffer (past Dean of Science) in 1983
- Game-independent improvement
- Idea: keep track of which moves are effective in causing beta cuts in the search
- Give a bonus for those moves,
  try them earlier among all children
- Similar idea: countermove heuristic (Uiterwijk)
  - store good reply to a move

Many More Alphabeta Enhancements

- Huge number of ideas have been tried in last 70 years
- Examples:
  - Minimal window search, Scout, PVS
  - Quiescence search
  - Parallel search
  - Late move reductions
- Very good website:
  `https://chessprogramming.org/`
- Do we still need to learn all these enhancements?

Alpha Zero vs Alphabeta Enhancements

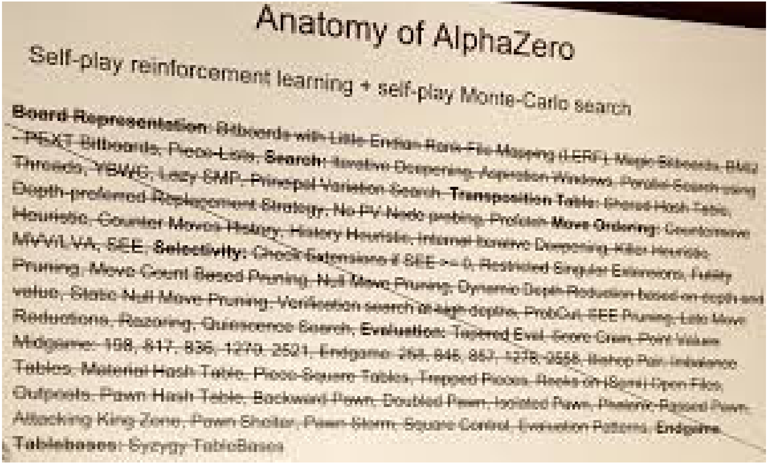

Image source: `lifein19x19.com`

- If the Alpha Zero approach works,
- and if we have enough computing power: no!

Summary and Preview

- This concludes our discussion of standard minimax algorithms
- Next topic: closer look at using knowledge in search
- After that: Monte Carlo Tree Search (MCTS) - a quite different way to approach minimax search problems
- However, it has the same goals as alphabeta
  - Heuristic search: play as well as possible when time limit is given
  - Solve: with unlimited time, eventually find the (perfect play) minimax solution
  - Most work on MCTS is on heuristic search, play well
  - Still, also interesting for solving