Tic-Tac-Toe Best First Search - Process Write-up

Algorithm Overview

This implementation uses **Best First Search** to find winning sequences in Tic-Tac-Toe games for both 3x3 and 4x4 grids. The algorithm explores the most promising game states first using a heuristic-guided priority queue approach.

Core Components

Input Processing (`read_inp()`)

- Reads board size n from first line
- Parses n×n board with symbols: X, 0, `.` (empty)
- Identifies current player to move
- Handles flexible input formatting (spaces or continuous characters)

Game State Representation

- **Board**: 2D list with symbols
- **Key generation** (`to_key()`): Converts board to tuple for efficient hashing
- **State tracking**: Uses `(board_key, next_player)` pairs to avoid revisiting identical positions

Winning Logic (`win()`, `lines_of()`)

- Extracts all possible winning lines: rows, columns, diagonals
- Checks if any line contains *n* identical non-empty symbols
- Returns `True` for terminal winning states

Heuristic Function (*heuristic()*)

The evaluation strategy compares line control:

hp = lines where opponent is absent (our potential)
ho = lines where we are absent (opponent potential)
score = hp - ho

- **Positive scores**: Favor current player
- **Higher values**: More winning opportunities
- **Simple but effective**: Guides search toward advantageous positions

Search Algorithm (*best_first()*)

Priority Queue Strategy

- Uses *heapq* with **negative heuristic values** (max-heap simulation)
- **Tie-breaking**: Counter ensures consistent ordering
- **State pruning**: Tracks visited *(board, player)* combinations

Search Process

1. Initialize with starting position
2. Pop highest-scoring state from priority queue
3. Generate all valid moves for current player
4. Check each resulting position for wins
5. Add non-winning states back to queue
6. Continue until win found or expansion limit reached

Move Generation (*gen_moves()*)

- Finds all empty cells (`.` positions)
- Creates new board states with player's symbol
- Returns *(new_board, move_coordinates)* pairs

File Operations

Input Format (*input.txt*)

3
X.O
.X.
O.X
X

- Line 1: Board size
- Lines 2-4: Board rows
- Last line: Current player

Output Files

- **steps.txt**: Step-by-step search log with board states and heuristic values
- **output.txt**: Final result showing winning sequence or search failure

Logging Strategy

- Records each expansion with depth, heuristic, and board state
- Traces complete solution path from root to winning position
- Provides detailed analysis for understanding search behavior

Key Features

Scalability

- **Variable grid sizes**: Works with any $n \times n$ board
- **Configurable limits**: `max_expansions` parameter prevents infinite search
- **Memory efficient**: State deduplication reduces storage requirements

Robustness

- **Input validation**: Handles malformed files gracefully
- **Terminal detection**: Identifies pre-solved boards
- **Error handling**: Clear messages for invalid inputs

Performance Optimizations

- **Heuristic guidance**: Prioritizes promising moves
- **Duplicate elimination**: Avoids redundant state exploration
- **Early termination**: Stops immediately when solution found

Algorithm Complexity

- **Time**: O(b^d) where b = branching factor, d = solution depth
- **Space**: O(states_visited) for duplicate detection
- **Practical performance**: Heuristic dramatically reduces search space

This implementation successfully demonstrates Best First Search applied to game tree exploration, providing both theoretical soundness and practical efficiency for Tic-Tac-Toe problem solving.