Paolo Sy-Quia CMPT 310

Reversi with Monte-Carlo Tree Search

Implementation

Specifications:

- The game was written in C++ and compiled with GCC
- OpenMP API was used to enable multithreading
    - bundled with GCC (compiled with -fopenmp)
- Code was built and tested with terminal on Ubuntu 20.04.1 LTS
    - recommended since Unicode and ANSI escape codes were used

Optimization:

Initially, my program averaged a suboptimal 1380 playouts per second (PPS) running on a single thread. After adding multithreading to the MCTS main loop, I saw a 245.55% average increase in PPS, going from 1380 to 4768.2 playouts.

Iterations	1	2	3	4	5	6	7	8	9	10	Avg
Single thread (PPS)	1387	1373	1408	1404	1285	1416	1383	1412	1331	1401	1380
Multithreading (PPS)	4957	4465	4800	4904	5003	4785	4444	4742	4829	4753	4768.2

Single Thread VS Multithreading (Playouts per second)

To increase performance further, I modified the most straining part of each playout— the "get all valid moves" function that was being called every turn of every playout.

I created two versions of that "get all valid moves" algorithm:
(1) one that would search starting from every empty tile
(2) and one that would search starting from occupied tiles.

The theory was that if most of the board was empty, it would be more efficient to search from occupied tiles and if the board was mostly full, it would be better to search from empty files.

Through testing, I found that swapping search algorithms from (2) to (1) when a team had 15 or more pieces on the board led to the most significant increase in performance. The average playouts per second increased from 4773.6 to 6075, a 27.3% increase.

Iter.	No swap (PPS)	Swap at 15 (PPS)
1	4957	6350
2	4465	6068
3	4800	5766
4	4904	6146
5	4742	6045
Avg.	4773.6	6075

With algorithm swapping VS no swapping

<u>Final playouts per second</u>: 6075

Pure Monte Carlo Tree Search

Quality of play:

- With just pure MCTS, the AI plays at a decent level.
- As a beginner, I had a hard time beating it.
  - Winning about ~30% of the time
- From what I observed, it generally prioritizes taking corners.

Modified Monte Carlo Tree Search

Heuristics used:

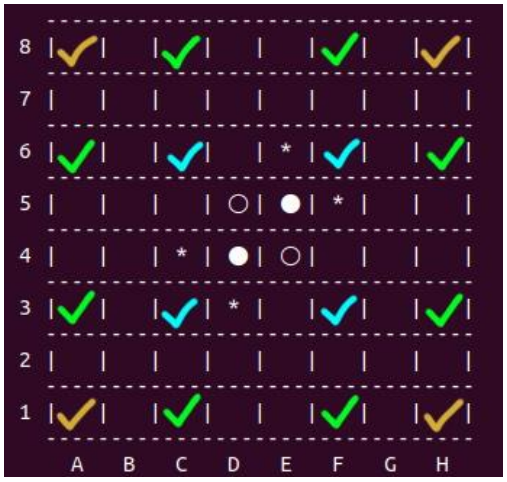
Two heuristics were applied during the random playouts of MCTS:

(1) Positional value heuristic – every position on the board is given a value

(2) Capture value heuristic – moves are scored based on the combined value of all captures (used in conjunction with (2))

Positional value heuristic:

Following the Reversi strategy that prioritizes the capture of corners and the positions that control those corners, I came up with values that are assigned to each position :



Good positions on the board

✓ Tier 1: Corners, the most valuable positions on the board

- Can't be captured, can be used to capture multiple directions at once
- Assigned a value of 5

✓ Tier 2: Positions controlling corners

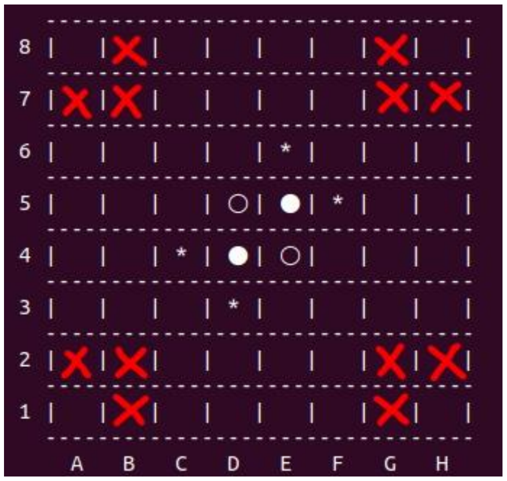
- Can be used to force opponent to take position adjacent to corner
- Assigned value of 3

✓ Tier 3: Positions controlling tier 2 positions

- Assigned value of 2

✗ Tier 0: Adjacent to corner (Bad)

- Allows opponent to take corner
- Assigned value of -5



Bad positions on the board

Capture value heuristic:

During MCTS playouts, the next move is decided by evaluating all the positions that would be captured by the move and summing all their positional values. The move that has the largest positional value sum is chosen as the next move. The heuristic attempts to maximize the number of captures while prioritizing positions that control the board.

Pure MCTS VS. Modified MCTS

Details:

- 500 games played
- AI with first turn is randomized
- Each algorithm is given five seconds to choose their next move

Set of 100 games	Set #1	Set #2	Set #3	Set #4	Set #5	Total	%
Pure MCTS	41	39	35	47	42	204	40.8%
Modified MCTS	59	61	65	53	58	296	59.2%

Pure MCTS VS. Modified MCTS (number of wins)

From a sample of 500 games, the modified MCTS algorithm won 59.2% of games against pure MCTS. This suggests that the heuristics allow the program to play better.

Notes:

- The sample size was limited by the length each game when using five second turns.
  - It took over three days to simulate 500 games.
- Possibly lost/Still reachable memory reported by Valgrind is caused by misunderstanding with OpenMP and Valgrind
  - No memory leaks when compiled without -fopenmp flag
  - Explanation according to Jakub Jelinek, Author at Red Hat Developer
    - https://gcc.gnu.org/bugzilla/show_bug.cgi?id=36298
    - Jakub Jelinek: https://developers.redhat.com/blog/author/rhjakub/