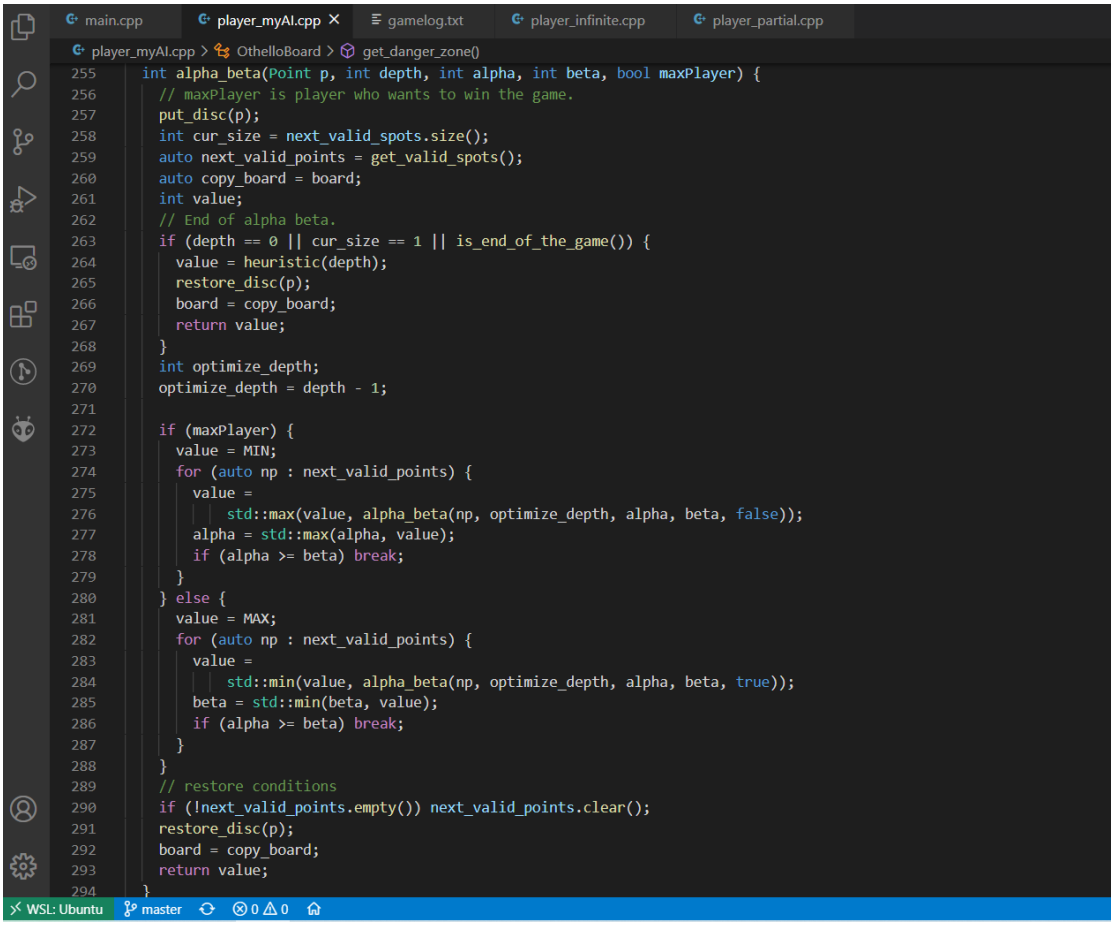
1. Heuristic

```cpp
int heuristic() {
  // size_t beta_option;
  if (cur_player == player) {
    cur_player = get_next_player(cur_player);
    next_valid_spots = get_valid_spots();
    // beta_option = next_valid_spots.size();
  } else {
    // beta_option = next_valid_spots.size();
    cur_player = get_next_player(cur_player);
    next_valid_spots = get_valid_spots();
  }
  size_t alpha_on_the_board = disc_count[player];
  size_t beta_on_the_board = disc_count[get_next_player(player)];
  // restore
  cur_player = get_next_player(cur_player);
  // minimize disc
  int corner = get_corner();
  if (corner == 0 && disc_count[EMPTY] > 40)  // first state : corner eater
    return alpha_on_the_board + corner * 3 - get_danger_zone() + get_stable_spots() * 3;
  else if (disc_count[EMPTY] > 21)            // second state : stable spread
    return alpha_on_the_board + corner * 3 - get_danger_zone() * 3 + get_stable_spots() * 5;
  else                                        // third state : greedy
    return alpha_on_the_board - beta_on_the_board + get_stable_spots();
}
```

2. Alpha-Beta Pruning




```cpp
int alpha_beta(Point p, int depth, int alpha, int beta, bool maxPlayer) {
  // maxPlayer is player who wants to win the game.
  put_disc(p);
  int cur_size = next_valid_spots.size();
  auto next_valid_points = get_valid_spots();
  auto copy_board = board;
  int value;
  // End of alpha beta.
  if (depth == 0 || cur_size == 1 || is_end_of_the_game()) {
    value = heuristic(depth);
    restore_disc(p);
    board = copy_board;
    return value;
  }
  int optimize_depth;
  optimize_depth = depth - 1;

  if (maxPlayer) {
    value = MIN;
    for (auto np : next_valid_points) {
      value =
          std::max(value, alpha_beta(np, optimize_depth, alpha, beta, false));
      alpha = std::max(alpha, value);
      if (alpha >= beta) break;
    }
  } else {
    value = MAX;
    for (auto np : next_valid_points) {
      value =
          std::min(value, alpha_beta(np, optimize_depth, alpha, beta, true));
      beta = std::min(beta, value);
      if (alpha >= beta) break;
    }
  }
  // restore conditions
  if (!next_valid_points.empty()) next_valid_points.clear();
  restore_disc(p);
  board = copy_board;
  return value;
}
```

3. Write Policy

```
296   void write_valid_spot(std::ofstream& fout) {
297     int value, child;
298     int alpha, beta;
299     Point p;
300     value = MIN, child = MIN, alpha = MIN, beta = MAX;
301     if (next_valid_spots.size() == 1)
302       p = next_valid_spots[0];
303     else {
304       for (auto np : next_valid_spots) {
305         // max_player's point of view
306         // race between depth(4) and depth(5), 4 wins always.
307         if (next_valid_spots.size() < 13)
308           child = alpha_beta(np, 5, alpha, beta, false);
309         else
310           child = alpha_beta(np, 4, alpha, beta, false);
311         alpha = std::max(alpha, child);
312         if (child > value) {
313           value = child;
314           p = np;
315           fout << np.x << " " << np.y << std::endl;
316           fout.flush();
317         }
318       }
319     }
320     // Keep updating the output until getting killed.
321     size_t count = 100;
322     while (count--) {
323       fout << p.x << " " << p.y << std::endl;
324       fout.flush();
325     }
326   }
327 };
```