Performance:

1. AI

1.1 All difficulties profiling tests

Tests AI performance across all difficulty levels (easy, medium, hard) by measuring execution time, CPU usage, and memory consumption for different game scenarios. Validates that each difficulty mode meets specific performance thresholds (easy: <10ms, medium: <100ms, hard: <500ms average).

```
TEST_F(AIPerformanceTest, AllDifficultiesProfile) {
    auto start = std::chrono::high_resolution_clock::now();
    std::vector<std::string> modes = {"easy", "medium", "hard"};

    for (const auto& mode : modes) {
        ai_->SetDifficulty(mode);
        std::vector<double> times;

        std::cout << "\n=== " << mode << " Mode ===\n";

        for (size_t i = 0; i < scenarios_.size(); ++i) {
            auto p = profile([&]() { ai_->GetBestMove(scenarios_[i], 'X'); });
            times.push_back(p.real_ms);
            printProfile(p, "Scenario " + std::to_string(i+1));
        }

        double avg = std::accumulate(times.begin(), times.end(), 0.0) / times.size();
        std::cout << "Average: " << std::setprecision(2) << avg << "ms\n";

        // Performance assertions (depth 6 for hard mode)
        double limit = (mode == "easy") ? 10.0 : (mode == "medium") ? 100.0 : 500.0;
        EXPECT_LT(avg, limit) << mode << " mode too slow: " << avg << "ms";
    }

    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(
        std::chrono::high_resolution_clock::now() - start);
    std::cout << "[TEST TIME] AllDifficultiesProfile: " << duration.count() << "ms\n";
}
```

Output :

```
=== easy Mode ===
[Scenario 1] Real: 0.010ms | CPU: 0.010ms | Memory: 9.098MB | CPU%: 103.8%
[Scenario 2] Real: 0.003ms | CPU: 0.004ms | Memory: 9.098MB | CPU%: 130.0%
[Scenario 3] Real: 0.003ms | CPU: 0.002ms | Memory: 9.098MB | CPU%: 72.7%
Average: 0.01ms

=== medium Mode ===
[Scenario 1] Real: 0.405ms | CPU: 0.407ms | Memory: 9.098MB | CPU%: 100.4%
[Scenario 2] Real: 0.041ms | CPU: 0.042ms | Memory: 9.098MB | CPU%: 102.4%
[Scenario 3] Real: 0.104ms | CPU: 0.104ms | Memory: 9.098MB | CPU%: 99.8%
Average: 0.18ms

=== hard Mode ===
[Scenario 1] Real: 21.265ms | CPU: 21.236ms | Memory: 9.098MB | CPU%: 99.9%
[Scenario 2] Real: 0.056ms | CPU: 0.057ms | Memory: 9.098MB | CPU%: 100.9%
[Scenario 3] Real: 0.236ms | CPU: 0.237ms | Memory: 9.098MB | CPU%: 100.2%
Average: 7.19ms
[TEST TIME] AllDifficultiesProfile: 22ms
```

1.2 Memory usage test

Monitors memory consumption during extended AI operations by running 50 sets of moves and measuring memory delta to detect potential memory leaks. Ensures memory growth stays under 1MB threshold.

```
TEST_F(AIPerformanceTest, MemoryUsageTest) {
    auto start = std::chrono::high_resolution_clock::now();
    std::cout << "\n=== Memory Usage Test ===\n";

    ai_->SetDifficulty("hard");
    long mem_before = profile([&]() {}).memory_kb;

    // Run 50 operations
    for (int i = 0; i < 50; ++i) {
        for (const auto& scenario : scenarios_) {
            ai_->GetBestMove(scenario, 'X');
        }
        if ((i + 1) % 10 == 0) {
            std::cout << "Completed " << (i + 1) << "/50 sets\n";
        }
    }

    long mem_after = profile([&]() {}).memory_kb;
    long mem_delta = mem_after - mem_before;

    std::cout << "Memory before: " << mem_before << " KB\n";
```

```
    std::cout << "Memory after:  " << mem_after << " KB\n";
    std::cout << "Memory delta:  " << mem_delta << " KB\n";

    EXPECT_LT(mem_delta, 1024) << "Memory leak detected: " << mem_delta << " KB";

    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(
        std::chrono::high_resolution_clock::now() - start);
    std::cout << "[TEST TIME] MemoryUsageTest: " << duration.count() << "ms\n";
}
```

Output :

```
=== Memory Usage Test ===
Completed 10/50 sets
Completed 20/50 sets
Completed 30/50 sets
Completed 40/50 sets
Completed 50/50 sets
Memory before: 9316 KB
Memory after:  9316 KB
Memory delta:  0 KB
[TEST TIME] MemoryUsageTest: 1022ms
```

1.3 Stress test

Evaluates AI performance under heavy load by executing 25 iterations of all scenarios in hard mode, measuring throughput (operations per second) and ensuring average operation time remains below 50ms to detect performance degradation.

```
TEST_F(AIPerformanceTest, StressTest) {
    auto start = std::chrono::high_resolution_clock::now();
    std::cout << "\n=== Stress Test (Hard Mode) ===\n";

    ai_->SetDifficulty("hard");
    const int iterations = 25;

    auto p = profile([&]() {
        for (int i = 0; i < iterations; ++i) {
            for (const auto& scenario : scenarios_) {
                ai_->GetBestMove(scenario, 'X');
            }
            if ((i + 1) % 5 == 0) {
                std::cout << "Completed " << (i + 1) << "/" << iterations << " iterations\n";
            }
        }
    });
```

```
    printProfile(p, "Stress Test");

    int total_ops = iterations * scenarios_.size();
    double ops_per_sec = total_ops / (p.real_ms / 1000.0);
    double avg_per_op = p.real_ms / total_ops;

    std::cout << "Total ops: " << total_ops << " | Ops/sec: " << std::setprecision(1)
              << ops_per_sec << " | Avg/op: " << std::setprecision(3) << avg_per_op << "ms\n";

    EXPECT_LT(avg_per_op, 50.0) << "Stress test degraded: " << avg_per_op << "ms/op";

    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(
        std::chrono::high_resolution_clock::now() - start);
    std::cout << "[TEST TIME] StressTest: " << duration.count() << "ms\n";
}
```

Output :

```
=== Stress Test (Hard Mode) ===
Completed 5/25 iterations
Completed 10/25 iterations
Completed 15/25 iterations
Completed 20/25 iterations
Completed 25/25 iterations
[Stress Test] Real: 512.179ms | CPU: 511.313ms | Memory: 9.098MB | CPU%: 99.8%
Total ops: 75 | Ops/sec: 146.4 | Avg/op: 6.829ms
[TEST TIME] StressTest: 512ms
```

1.4 Decision quality

Tests both performance and correctness by presenting the AI with a critical game situation where it must block an opponent's winning move. Verifies the AI makes optimal decisions within 1 second while maintaining strategic accuracy.

```
TEST_F(AIPerformanceTest, DecisionQuality) {
    auto start = std::chrono::high_resolution_clock::now();
    std::cout << "\n=== Decision Quality Test ===\n";

    Board critical = {{'X', 'X', ' '}, {'O', 'O', ' '}, {' ', ' ', ' '}};
    ai_->SetDifficulty("hard");

    std::pair<int, int> move;
    auto p = profile([&]() { move = ai_->GetBestMove(critical, 'O'); });

    printProfile(p, "Critical Decision");

    bool optimal = (move.first == 0 && move.second == 2) || (move.first == 1 && move.second == 2);
```

```
    std::cout << "Move: (" << move.first << "," << move.second << ") - "
              << (optimal ? "OPTIMAL" : "SUBOPTIMAL") << "\n";

    EXPECT_TRUE(optimal) << "AI failed to block winning move";
    EXPECT_LT(p.real_ms, 1000.0) << "Decision too slow: " << p.real_ms << "ms";

    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(
        std::chrono::high_resolution_clock::now() - start);
    std::cout << "[TEST TIME] DecisionQuality: " << duration.count() << "ms\n";
}
```

Output :

```
=== Decision Quality Test ===
[Critical Decision] Real: 0.254ms | CPU: 0.254ms | Memory: 9.098MB | CPU%: 100.2%
Move: (1,2) - OPTIMAL
[TEST TIME] DecisionQuality: 0ms
```

[SUITE TOTAL TIME] 1558 ms (1.56 s)

2. User system

2.1 Database operations profile

Measures and profiles the performance of core database operations by testing 10 registration and login cycles. Tracks real-time execution, CPU usage, and memory consumption for each operation, calculating average times and ensuring registration stays under 50ms and login under 30ms.

```
TEST_F(UserSystemPerformanceTest, DatabaseOperationsProfile) {
    auto start = std::chrono::high_resolution_clock::now();
    std::cout << "\n=== Database Operations ===\n";

    std::vector<double> reg_times, login_times;

    // Test registration and login
    for (int i = 0; i < 10; ++i) {
        std::string user = "user" + std::to_string(i);
        std::string pass = "pass" + std::to_string(i);

        auto p1 = profile([&]() { user_system_->registerUser(user, pass); });
        reg_times.push_back(p1.real_ms);
```

```
        printProfile(p1, "Register " + std::to_string(i+1));

        auto p2 = profile([&]() { user_system_->loginUser(user, pass); });
        login_times.push_back(p2.real_ms);
        printProfile(p2, "Login " + std::to_string(i+1));
    }

    double avg_reg = std::accumulate(reg_times.begin(), reg_times.end(), 0.0) / reg_times.size();
    double avg_login = std::accumulate(login_times.begin(), login_times.end(), 0.0) / login_times.size();

    std::cout << "Avg Register: " << std::setprecision(2) << avg_reg << "ms | Avg Login: " << avg_login <<
"ms\n";

    EXPECT_LT(avg_reg, 50.0) << "Registration too slow: " << avg_reg << "ms";
    EXPECT_LT(avg_login, 30.0) << "Login too slow: " << avg_login << "ms";

    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(
        std::chrono::high_resolution_clock::now() - start);
    std::cout << "[TEST TIME] DatabaseOperationsProfile: " << duration.count() << "ms\n";
}
```

Output :

```
=== Database Operations ===
[Register 1] Real: 1.939ms | CPU: 0.310ms | Memory: 9.098MB | CPU%: 16.0%
[Login 1] Real: 0.057ms | CPU: 0.058ms | Memory: 9.098MB | CPU%: 102.1%
[Register 2] Real: 3.125ms | CPU: 0.278ms | Memory: 9.098MB | CPU%: 8.9%
[Login 2] Real: 0.048ms | CPU: 0.048ms | Memory: 9.098MB | CPU%: 100.3%
[Register 3] Real: 1.951ms | CPU: 0.252ms | Memory: 9.098MB | CPU%: 12.9%
[Login 3] Real: 0.048ms | CPU: 0.048ms | Memory: 9.098MB | CPU%: 99.7%
[Register 4] Real: 2.964ms | CPU: 0.262ms | Memory: 9.098MB | CPU%: 8.8%
[Login 4] Real: 0.045ms | CPU: 0.046ms | Memory: 9.098MB | CPU%: 102.1%
[Register 5] Real: 3.797ms | CPU: 0.266ms | Memory: 9.098MB | CPU%: 7.0%
[Login 5] Real: 0.049ms | CPU: 0.048ms | Memory: 9.098MB | CPU%: 98.2%
[Register 6] Real: 1.722ms | CPU: 0.256ms | Memory: 9.098MB | CPU%: 14.9%
[Login 6] Real: 0.046ms | CPU: 0.048ms | Memory: 9.098MB | CPU%: 103.7%
[Register 7] Real: 1.476ms | CPU: 0.263ms | Memory: 9.098MB | CPU%: 17.8%
[Login 7] Real: 0.045ms | CPU: 0.045ms | Memory: 9.098MB | CPU%: 100.3%
[Register 8] Real: 1.664ms | CPU: 0.249ms | Memory: 9.098MB | CPU%: 15.0%
[Login 8] Real: 0.045ms | CPU: 0.045ms | Memory: 9.098MB | CPU%: 100.9%
[Register 9] Real: 3.373ms | CPU: 0.288ms | Memory: 9.098MB | CPU%: 8.5%
[Login 9] Real: 0.050ms | CPU: 0.051ms | Memory: 9.098MB | CPU%: 102.0%
[Register 10] Real: 1.871ms | CPU: 0.281ms | Memory: 9.098MB | CPU%: 15.0%
[Login 10] Real: 0.051ms | CPU: 0.052ms | Memory: 9.098MB | CPU%: 101.9%
Avg Register: 2.39ms | Avg Login: 0.05ms
[TEST TIME] DatabaseOperationsProfile: 25ms
```

2.2 Game data operations

Evaluates the performance of game-related database operations including saving games with move data and retrieving game history. Tests game saving 10 times and history retrieval 5 times, measuring average execution times and ensuring game saves complete under 100ms and history retrieval under 50ms.

```
TEST_F(UserSystemPerformanceTest, GameDataOperations) {
    auto start = std::chrono::high_resolution_clock::now();
    std::cout << "\n=== Game Data Operations ===\n";

    // Setup users
    for (int i = 0; i < 5; ++i) {
        user_system_->registerUser("user" + std::to_string(i), "pass" + std::to_string(i));
    }

    std::vector<std::pair<int, std::string>> moves = {{0, "move1"}, {4, "move2"}, {8, "move3"}};
    std::vector<double> save_times, history_times;

    // Test game saving
    for (int i = 0; i < 10; ++i) {
        auto p = profile([&]() {
            user_system_->saveGameWithMoves("user0", "user1", "X", moves);
        });
        save_times.push_back(p.real_ms);
        printProfile(p, "Save Game " + std::to_string(i+1));
    }

    // Test history retrieval
    for (int i = 0; i < 5; ++i) {
        auto p = profile([&]() {
            auto history = user_system_->getGameHistory("user" + std::to_string(i));
        });
        history_times.push_back(p.real_ms);
        printProfile(p, "Get History " + std::to_string(i+1));
    }

    double avg_save = std::accumulate(save_times.begin(), save_times.end(), 0.0) / save_times.size();
    double avg_history = std::accumulate(history_times.begin(), history_times.end(), 0.0) /
history_times.size();

    std::cout << "Avg Save: " << std::setprecision(2) << avg_save << "ms | Avg History: " << avg_history
<< "ms\n";

    EXPECT_LT(avg_save, 100.0) << "Game saving too slow: " << avg_save << "ms";
    EXPECT_LT(avg_history, 50.0) << "History retrieval too slow: " << avg_history << "ms";

    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(
```

```
        std::chrono::high_resolution_clock::now() - start);
    std::cout << "[TEST TIME] GameDataOperations: " << duration.count() << "ms\n";
}
```

Output :

```
=== Game Data Operations ===
[Save Game 1] Real: 1.649ms | CPU: 0.268ms | Memory: 9.098MB | CPU%: 16.3%
[Save Game 2] Real: 1.475ms | CPU: 0.264ms | Memory: 9.098MB | CPU%: 17.9%
[Save Game 3] Real: 1.550ms | CPU: 0.263ms | Memory: 9.098MB | CPU%: 17.0%
[Save Game 4] Real: 1.385ms | CPU: 0.255ms | Memory: 9.098MB | CPU%: 18.4%
[Save Game 5] Real: 2.918ms | CPU: 0.308ms | Memory: 9.098MB | CPU%: 10.6%
[Save Game 6] Real: 1.640ms | CPU: 0.272ms | Memory: 9.098MB | CPU%: 16.6%
[Save Game 7] Real: 1.713ms | CPU: 0.299ms | Memory: 9.098MB | CPU%: 17.5%
[Save Game 8] Real: 1.753ms | CPU: 0.288ms | Memory: 9.098MB | CPU%: 16.4%
[Save Game 9] Real: 2.374ms | CPU: 0.276ms | Memory: 9.098MB | CPU%: 11.6%
[Save Game 10] Real: 2.662ms | CPU: 0.289ms | Memory: 9.098MB | CPU%: 10.9%
[Get History 1] Real: 0.077ms | CPU: 0.078ms | Memory: 9.098MB | CPU%: 101.1%
[Get History 2] Real: 0.050ms | CPU: 0.051ms | Memory: 9.098MB | CPU%: 102.5%
[Get History 3] Real: 0.030ms | CPU: 0.031ms | Memory: 9.098MB | CPU%: 101.8%
[Get History 4] Real: 0.027ms | CPU: 0.027ms | Memory: 9.098MB | CPU%: 101.7%
[Get History 5] Real: 0.025ms | CPU: 0.026ms | Memory: 9.098MB | CPU%: 103.6%
Avg Save: 1.91ms | Avg History: 0.04ms
[TEST TIME] GameDataOperations: 29ms
```

2.3 Stress test

Performs comprehensive load testing by simulating high-volume database operations with 25 user registrations, 15 game saves, and multiple statistics queries. Measures total throughput (operations per second) and average time per operation, ensuring the system maintains performance under load with less than 20ms per operation.

```
TEST_F(UserSystemPerformanceTest, StressTest) {
    auto start = std::chrono::high_resolution_clock::now();
    std::cout << "\n=== Database Stress Test ===\n";

    const int users = 25, games = 15;
    std::vector<std::pair<int, std::string>> moves = {{0, "m1"}, {4, "m2"}, {8, "m3"}};

    auto p = profile([&]() {
        // Register users
        for (int i = 0; i < users; ++i) {
            user_system_->registerUser("stress_user" + std::to_string(i), "pass" + std::to_string(i));
        }

        // Save games
        for (int i = 0; i < games; ++i) {
```

```
            user_system_->saveGameWithMoves(
                "stress_user" + std::to_string(i % users),
                "stress_user" + std::to_string((i + 1) % users),
                (i % 2 == 0) ? "X" : "O",
                moves
            );
        }

        // Get statistics
        for (int i = 0; i < 10; ++i) {
            auto history = user_system_->getGameHistory("stress_user" + std::to_string(i));
            auto ai_stats = user_system_->getHumanVsAIStats("stress_user" + std::to_string(i));
        }
    });

    printProfile(p, "Stress Test");

    int total_ops = users + games + 20;
    double ops_per_sec = total_ops / (p.real_ms / 1000.0);
    double avg_per_op = p.real_ms / total_ops;

    std::cout << "Total ops: " << total_ops << " | Ops/sec: " << std::setprecision(1)
              << ops_per_sec << " | Avg/op: " << std::setprecision(3) << avg_per_op << "ms\n";

    EXPECT_LT(avg_per_op, 20.0) << "Stress test degraded: " << avg_per_op << "ms/op";

    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(
        std::chrono::high_resolution_clock::now() - start);
    std::cout << "[TEST TIME] StressTest: " << duration.count() << "ms\n";
}
```

Output :

```
=== Database Stress Test ===
[Stress Test] Real: 61.805ms | CPU: 10.457ms | Memory: 9.098MB | CPU%: 16.9%
Total ops: 60 | Ops/sec: 970.8 | Avg/op: 1.030ms
[TEST TIME] StressTest: 62ms
```

2.4 memory usage

Monitors memory consumption during extended database operations by performing 20 cycles of user registration, game saving, and history retrieval. Tracks memory usage before and after operations to detect potential memory leaks, ensuring memory growth stays under 1MB.

```
TEST_F(UserSystemPerformanceTest, MemoryUsage) {
    auto start = std::chrono::high_resolution_clock::now();
    std::cout << "\n=== Memory Usage Test ===\n";
```

```
    long mem_before = profile([&]() {}).memory_kb;

    std::vector<std::pair<int, std::string>> moves = {{0, "m1"}, {4, "m2"}};

    for (int i = 0; i < 20; ++i) {
        user_system_->registerUser("mem_user" + std::to_string(i), "pass" + std::to_string(i));
        user_system_->saveGameWithMoves("mem_user" + std::to_string(i), "AI", "X", moves);
        auto history = user_system_->getGameHistory("mem_user" + std::to_string(i));

        if ((i + 1) % 5 == 0) {
            std::cout << "Completed " << (i + 1) << "/20 cycles\n";
        }
    }

    long mem_after = profile([&]() {}).memory_kb;
    long mem_delta = mem_after - mem_before;

    std::cout << "Memory before: " << mem_before << " KB | after: " << mem_after
              << " KB | delta: " << mem_delta << " KB\n";

    EXPECT_LT(mem_delta, 1024) << "Memory leak: " << mem_delta << " KB";

    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(
        std::chrono::high_resolution_clock::now() - start);
    std::cout << "[TEST TIME] MemoryUsage: " << duration.count() << "ms\n";
}
```

Output :

```
=== Memory Usage Test ===
Completed 5/20 cycles
Completed 10/20 cycles
Completed 15/20 cycles
Completed 20/20 cycles
Memory before: 9316 KB | after: 9316 KB | delta: 0 KB
[TEST TIME] MemoryUsage: 121ms
```

[SUITE TOTAL TIME] 329 ms (0.33 s)

3. Game logic

3.1 Core operations profile

This test measures the performance of fundamental game operations by running 100 iterations and timing three core functions: game initialization (`Game_init()`), move validation (`isValidMove()`), and move execution (`makeMove()`). It calculates average execution times and enforces performance thresholds - initialization must complete under 1ms, moves under 0.5ms, and validation under 0.1ms. The test simulates realistic gameplay by testing moves 1-5 and stops early if the game ends.

```
TEST_F(GameLogicPerformanceTest, CoreOperationsProfile) {
    auto start = std::chrono::high_resolution_clock::now();
    std::cout << "\n=== Core Operations Performance ===\n";

    std::vector<double> init_times, move_times, validation_times;
    const int iterations = 100;

    for (int i = 0; i < iterations; ++i) {
        // Test initialization
        auto p1 = profile([&]() { game_engine_->Game_init(); });
        init_times.push_back(p1.real_ms);

        // Test move validation and making
        std::vector<int> moves = {1, 2, 3, 4, 5};
        for (int move : moves) {
            auto p2 = profile([&]() { game_engine_->isValidMove(move); });
            validation_times.push_back(p2.real_ms);

            if (game_engine_->isValidMove(move)) {
                auto p3 = profile([&]() { game_engine_->makeMove(move, "Test move"); });
                move_times.push_back(p3.real_ms);
            }

            if (game_engine_->isGameOver()) break;
        }

        if ((i + 1) % 20 == 0) {
            std::cout << "Completed " << (i + 1) << "/" << iterations << " cycles\n";
        }
    }

    double avg_init = std::accumulate(init_times.begin(), init_times.end(), 0.0) / init_times.size();
    double avg_move = std::accumulate(move_times.begin(), move_times.end(), 0.0) /
move_times.size();
    double avg_validation = std::accumulate(validation_times.begin(), validation_times.end(), 0.0) /
validation_times.size();

    std::cout << "Average Init: " << std::setprecision(4) << avg_init << "ms | "
              << "Move: " << avg_move << "ms | Validation: " << avg_validation << "ms\n";
```

```
    EXPECT_LT(avg_init, 1.0) << "Game initialization too slow: " << avg_init << "ms";
    EXPECT_LT(avg_move, 0.5) << "Move making too slow: " << avg_move << "ms";
    EXPECT_LT(avg_validation, 0.1) << "Move validation too slow: " << avg_validation << "ms";

    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(
        std::chrono::high_resolution_clock::now() - start);
    std::cout << "[TEST TIME] CoreOperationsProfile: " << duration.count() << "ms\n";
}
```

Output :

```
=== Core Operations Performance ===
Completed 20/100 cycles
Completed 40/100 cycles
Completed 60/100 cycles
Completed 80/100 cycles
Completed 100/100 cycles
Average Init: 0.001396ms | Move: 0.000298ms | Validation: 9.99e-05ms
[TEST TIME] CoreOperationsProfile: 2ms
```

3.2 Game simulation

This test runs 50 complete game simulations using three predefined move patterns: a quick win scenario, a draw scenario, and a diagonal win pattern. Each simulation includes full game initialization, move execution, win checking for both players, and draw detection. The test measures total game completion time and calculates games per second throughput, expecting each complete game simulation to finish within 5ms.

```
TEST_F(GameLogicPerformanceTest, GameSimulation) {
    auto start = std::chrono::high_resolution_clock::now();
    std::cout << "\n=== Game Simulation ===\n";

    const int num_games = 50;
    std::vector<double> game_times;

    std::vector<std::vector<int>> patterns = {
        {1, 2, 4, 5, 7},              // Quick win
        {1, 2, 3, 4, 6, 5, 8, 9, 7},  // Draw
        {5, 1, 9, 3, 2, 7}            // Diagonal win
    };

    for (int game = 0; game < num_games; ++game) {
        const auto& pattern = patterns[game % patterns.size()];

        auto p = profile([&]() {
            game_engine_->Game_init();
```

```
        for (int move : pattern) {
            if (game_engine_->isValidMove(move)) {
                game_engine_->makeMove(move, "Simulation move");
                game_engine_->checkWin("X");
                game_engine_->checkWin("O");
                game_engine_->checkDraw();
                if (game_engine_->isGameOver()) break;
            }
        }
    });

    game_times.push_back(p.real_ms);

    if ((game + 1) % 10 == 0) {
        std::cout << "Completed " << (game + 1) << "/" << num_games << " games\n";
    }
  }

  double avg_game = std::accumulate(game_times.begin(), game_times.end(), 0.0) /
game_times.size();
  double games_per_sec = num_games / (std::accumulate(game_times.begin(), game_times.end(),
0.0) / 1000.0);

  std::cout << "Average Game: " << std::setprecision(4) << avg_game << "ms | "
            << "Games/sec: " << std::setprecision(2) << games_per_sec << "\n";

  EXPECT_LT(avg_game, 5.0) << "Game simulation too slow: " << avg_game << "ms";

  auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(
      std::chrono::high_resolution_clock::now() - start);
  std::cout << "[TEST TIME] GameSimulation: " << duration.count() << "ms\n";
}
```

Output :

```
=== Game Simulation ===
Completed 10/50 games
Completed 20/50 games
Completed 30/50 games
Completed 40/50 games
Completed 50/50 games
Average Game: 0.0157ms | Games/sec: 63739.33
[TEST TIME] GameSimulation: 0ms
```

3.2 Stress test

This is an endurance test that performs 100 intensive cycles, where each cycle initializes a game, attempts all possible moves (positions 1-9), and then undoes all moves using the `undoMove()` function. It measures memory usage, CPU utilization, and timing performance under sustained load. The test expects each complete cycle to finish within 10ms and provides detailed profiling data including real time, CPU time, memory consumption, and CPU percentage utilization.

```
TEST_F(GameLogicPerformanceTest, StressTest) {
    auto start = std::chrono::high_resolution_clock::now();
    std::cout << "\n=== Stress Test ===\n";

    const int iterations = 100;

    auto p = profile([&]() {
        for (int i = 0; i < iterations; ++i) {
            game_engine_->Game_init();

            for (int pos = 1; pos <= 9; ++pos) {
                if (game_engine_->isValidMove(pos)) {
                    game_engine_->makeMove(pos, "Stress test");
                    if (game_engine_->isGameOver()) break;
                }
            }

            while (!game_engine_->get_moveHistory().empty()) {
                game_engine_->undoMove();
            }

            if ((i + 1) % 20 == 0) {
                std::cout << "Completed " << (i + 1) << "/" << iterations << " cycles\n";
            }
        }
    });

    printProfile(p, "Stress Test");

    double avg_per_cycle = p.real_ms / iterations;
    double cycles_per_sec = iterations / (p.real_ms / 1000.0);

    std::cout << "Cycles: " << iterations << " | Cycles/sec: " << std::setprecision(1)
              << cycles_per_sec << " | Avg/cycle: " << std::setprecision(3) << avg_per_cycle << "ms\n";

    EXPECT_LT(avg_per_cycle, 10.0) << "Stress test degraded: " << avg_per_cycle << "ms/cycle";

    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(
        std::chrono::high_resolution_clock::now() - start);
```

```
    std::cout << "[TEST TIME] StressTest: " << duration.count() << "ms\n";
}
```

Output :

```
=== Stress Test ===
Completed 20/100 cycles
Completed 40/100 cycles
Completed 60/100 cycles
Completed 80/100 cycles
Completed 100/100 cycles
[Stress Test] Real: 1.126ms | CPU: 1.096ms | Memory: 9.031MB | CPU%: 97.4%
Cycles: 100 | Cycles/sec: 88843.2 | Avg/cycle: 0.011ms
[TEST TIME] StressTest: 1ms
```

[SUITE TOTAL TIME] 5 ms (0.01 s)

Optimization:

1. Convolution to detect winning:

Apply a suitable kernel (vertical, horizontal, or diagonal) according to the required win pattern, to sweep on the board to detect when the convolution result equals a chosen sum.

This reduces the operation from a nested loop to 3 loops.

2. Alpha-Beta Pruning:

Detect when beta <= alpha which means this branch can’t occur. This reduces the number of paths.