An implementation of the game of Checkers with optimal AI

Georgi Mirazchiyski
40208393@napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures (SET09117)

Abstract

As part of Napier University's Algorithms and Data Structures class (SET09117), students were tasked to implement and report upon an implementation of the classic board game - Checkers. This report will cover one such implementation using creative solutions to the problem with the implementation of an optimal AI player that were learned out of the scope of the module's taught content.

Keywords – checkers, algorithms, data structures, C++, minmax, game loop

1 Introduction

This report's aim is to describe the implementation of a Checkers game written in a language of personal preference being C++ in my case. A good portion of a solid implementation of logic games such as this is properly picked data structures. That and well designed algorithms to make a good use these data structures. Both of these should almost always be coupled because one complements the efficiency of the other. Although, proper data structures and fast algorithms are not every that is needed for a fully functional and very performant system. A smart software design is also a key factor for the excellence of the result. The following parts of the report try to emphasize on these things and give an explanation followed by evaluations where neccessary. This assignment implementation provides human vs human, human vs computer and computer vs computer functionality allowing you to fully experience the game of checkers.

2 Design

The success to well written, efficient and extensible pieces of software lies in their architectural design. Thus, object oriented programming and design patterns[1] where necessary, good commenting practices and uniform naming convention for class members depending on their accessibility, were used strictly in the coding of this program.

2.1 System architecture

Design in software engineering is as much important as in art. Design patterns are created to be made use of where needed for achieving good code structure. In this assignment, I have found a good use of Singleton, Factory and Facade for the creation of Events Manager, who is responsible for handling the state of the game by invoking events that are created by the Events Factory.

2.2 Class overview

Abstract overview of the system's class structure:

- Checkers
  - Common
  - Game
  - Main
    - API
      - Board
      - Pawn
        - Events
          - EventFacility
          - EventFactory
          - EventManager
            - EventImpl
              - EntityPawnAction : EventFacility
              - LoadGame : EventFacility
              - SaveGame : EventFacility
              - WinGame : EventFacility
              - QutiGame : EventFacility
        - Utils
          - MovesGenerator
          - FileIO
          - Logger
          - Timer
    - Entity
      - Entity
        - AI
          - AI : Entity
          - EasyAI : AI
          - MediumAI : AI
          - HardAI : AI
        - Player

Player : Entity

Parallelism

Parallel

2.3 Game loop

Listing 1: The game loop in C++

```cpp

void gameLoop(Game& t_game)
{
    // create game board and assign players
    t_game.begin();

    // Game Loop!
    while (t_game.getIsRunning())
    {
        // update the frames (game movement)
        t_game.update();

        // switch players and re−draw board each turn
        if (t_game.getNextTurn())
        {
            // do the rendering (draw in console)
            t_game.draw();
        }

        // check for win condition & exit the loop
        t_game.end();
    }
}

```

3 Enhancements

1. Implement good object to file serialization for better saving and allowance of loading saved game with ease.
2. Add alpha-beta cuttoffs to avoid re-traversing unnecessary parts of the tree for Min-Max.
3. Add multi-threading to speed up some calculations in the AI algorithms.

4 Critical evaluation

Available game modes:

Human vs Human

Human vs Computer

Computer vs Computer

CPU: Easy / Medium / Hard

A player entity is capable of:

move: using MovesGenerator that returns a deque of a custom Movement data structure : pair(Position, Position) , where Position : pair(int, int) to describe coordinates

Listing 2: Human player move function in C++

```cpp
// search t_posTo in generatePossibleMoves
// if exists such move variant
// use board move func to update it
assert(std::is_sorted(t_moveGenerator−>getPossibleMoves().begin(), t_moveGenerator−>getPossibleMoves().end()));

if (std::binary_search(t_moveGenerator−>getPossibleMoves().begin(), t_moveGenerator−>getPossibleMoves().end(),
    std::make_pair(t_posFrom, t_posTo)))
{
    API::ActionState turnState = m_board−>move(t_posFrom, t_posTo);

    // if an action has happened −> a pawn with coords: t_posFrom and a pawn with coords: t_posTo will have their values swapped
    if (turnState == API::ActionState::JUMP)
        m_lastPlayedPawn = std::make_shared<API::Pawn>(m_board−>getBoardPawn(t_posTo));

    if (turnState == API::ActionState::MOVE)
        m_lastPlayedPawn = nullptr;
}
else
{
    throw std::logic_error(Action failed due to the selection of impossible move action.);
}

```

Listing 3: EasyAI player move function in C++

```cpp
assert(std::is_sorted(t_moveGenerator−>getPossibleMoves().begin(),
t_moveGenerator−>getPossibleMoves().end()));

auto possibleMoves = t_moveGenerator−>getPossibleMoves();

// break out if there's no more possible moves
if (possibleMoves.empty())
    return;

// do a random move
randomMove(possibleMoves);

```

Listing 4: MediumAI player move function in C++

```cpp
assert(std::is_sorted(t_moveGenerator−>getPossibleMoves().begin(), t_moveGenerator−>getPossibleMoves().end()));

std::map<uint16, Movement> moveOrders;
auto possibleMoves = t_moveGenerator−>getPossibleMoves();

// break out if there's no more possible moves
if (possibleMoves.empty())
    return;

// the maximum element | the most swift move of all in the set of possible movements
    auto maxOrderValue = moveOrders.rbegin()−>first;

std::for_each(possibleMoves.begin(), possibleMoves.end(), [&](const Movement& move)
{
    ... Evaluate ...
    // order values for such: 3,4,5,6,7
    // 3 − move a king
    // 4 − take a pawn with a pawn
    // 5 − take a pawn with a king
    // 6 − take a king with a king
    // 7 − take a king with a pawn
    ... & insert in moveOrders ...
}

if (maxOrderValue > 2)
{
    // do a reasonable move
    auto maxOrderMove = moveOrders.crbegin()−>second;
    Position fromPos = maxOrderMove.first;
```

```
    Position toPos = maxOrderMove.second;
    // do movement
    move(fromPos, toPos);
}
else
{
    // do a random move (no other option left)
    // order value for such: 2
    randomMove(possibleMoves);
}

```

Listing 5: HardAI player move function in C++

```
assert(std::is_sorted(t_moveGenerator->getPossibleMoves().begin(), t_moveGenerator->getPossibleMoves().end()));

t_moveGenerator->generatePossibleMoves(m_board, m_pawnColor,    m_lastPlayedPawn);
auto possibleMoves = t_moveGenerator->getPossibleMoves();

// break out if there's no more possible moves
if (possibleMoves.empty())
    return;

// minimax (maximin) algorithm

auto maximinMove = maximin(*m_board, 0);
Position fromPos = maximinMove.first;
Position toPos = maximinMove.second;
// do movement
move(fromPos, toPos);

```

jump: calculates possible jump and checks if killing a pawn in the middle is possible

Listing 6: HardAI player move function in C++

```
// calculate jump possibilty
bool isJump = (abs(t_posFrom.first - t_posTo.first)) + (abs(t_posFrom.second - t_posTo.second)) == 4);

if(isJump)
{
    // find the middle pawn to kill
    killPawn(m_board[(t_lhs.getCoordX() + t_rhs.getCoordX()) / 2][(t_lhs.getCoordY() + t_rhs.getCoordY()) / 2]);
}
```

evolve: promotes man pawn to king pawn.

```
t_pawn.getMesh() = toupper(t_pawn.getMesh());
```

undo: uses stack, push back undo.top to redo, pop back from undo, display the board in the state of the top element in undo

Listing 7: Undo implementation in C++

```
// create temp board
Board::board<Pawn, Board::s_boardLen> tempBoard;

// save the current state of the board in the redo stack
m_redoStack.push(m_undoStack.top());
// remove it from the undo stack before display
m_undoStack.pop();
m_undoStack.pop();

// copy the board from the undo stack into the temp board
tempBoard = m_undoStack.top();
// copy the temp board into the game board
m_gameBoard->setBoard(tempBoard);

// discard the last moves from the gameHistory queue on undo
m_gameBoard->s_boardHistory.pop_back();
m_gameBoard->s_boardHistory.pop_back();
```

redo: uses stack, display the board in the state of the top element in redo, push back redo.top to undo, pop back from redo

Listing 8: Redo implementation in C++

```
// create temp board
Board::board<Pawn, Board::s_boardLen> tempBoard;

// copy the board from the redo stack into the temp board
tempBoard = m_redoStack.top();
// copy the temp board into the game board
m_gameBoard->setBoard(tempBoard);

// save the current state of the board in the undo stack
m_undoStack.push(m_redoStack.top());
m_undoStack.push(m_redoStack.top());
// remove it from the redo stack after display
m_redoStack.pop();

// save the moves in the gameHistory deque's back since these are the last current ones
m_gameBoard->s_boardHistory.push_back(m_undoStack.top());
m_gameBoard->s_boardHistory.push_back(m_undoStack.top());
```

save: uses FileIO which internally uses file streams to write the game data buffer of characters into a file

Global to the entire system:

load: will be supported after dealt with Serialization...!

replay: using deque, push back on redo, pop back on undo, display the front and pop front after that

Listing 9: Display replay game in C++

```
while (t_board->s_boardHistory.size() > 0)
{
    // create temp board
    Board::board<Pawn, Board::s_boardLen> tempBoard;

    // copy the board from the redo stack into the temp board
    tempBoard = t_board->s_boardHistory.front();
    t_board->s_boardHistory.pop_front();

    ... clear screen ...

    // copy the temp board into the game board
    t_board->setBoard(tempBoard);
    t_board->display();
    Utils::Timer::getInstance().applyTimeDelayInSeconds(1.0);
}
```

restart: simply resets the game state and the board data

4.1 Optimal AI using Min-Max

The Min-Max[2] algorithm is a traditional one that is usually applied in two-player games such as tic-tac-toe, chess, go and in our case - checkers.

4.1.1 Theory and visualization

The one thing most if not all logic games have in common so they can incorporate the same algorithm is that they can be described by a set of rules and premisses. Having this knowledge, it is possible to know from a given point in the game, what next moves could be available. Simply said, checkers is 'full information game' and each player knows everything about the possible moves of the adversary.

Search trees are used to represent the searches. A search tree is generated, and traversed depth-first, starting with current game position up to a specified level of depth - end game position.

The two players involved are described as **MAX** and **MIN**. The final game position is evaluated from the MAX's point of view, making the algorithm act more like a Max-Min rather than Min-Max, though Min-Max is how it should be called formally. The inner node values of the tree are filled bottom-up with the evaluated values. The nodes of the MAX player receive the maximum value of its children. Those of the MIN player will have the minimum value of its children. These values represent how good a game move is. What is going on is that the MAX player will try to select the move with the highest value in the end while the MIN player will try to select those best to him, thus minimizing the MAX's outcome. To summarize in short, Min-Max is an algorithm designed to maximise gain and minimise loss in the worst case scenario of a game play. It's perfect play for deterministic fully observables games.

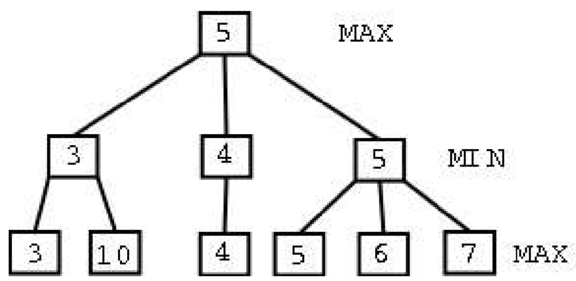

Figure 1: **Min-Max search tree** - depth = 3

4.1.2 Hack it in C++ code

Listing 10: Min-Max algorithm in C++

```
Movement HardAI::maximin(API::Board t_board, uint32 t_depth)
{
    API::Utils::MovesGenerator moveGenerator;

    // get all valid/possible moves in from the generator
    auto p_board = std::make_shared<API::Board>(); *p_board = t_board;
    moveGenerator.generatePossibleMoves(p_board, m_pawnColor, m_lastPlayedPawn);
    auto possibleMoves = moveGenerator.getPossibleMoves();

    // ensure you have any possible moves
    if (possibleMoves.empty())
        return Movement{ {0, 0}, {0, 0} };

    // populate with scores for every possible move
    std::array<int32, 32> scores;
    for (int i = 0; i < scores.size(); i++) scores[i] = 0;
    size_t count = 0;
    for (auto posMove : possibleMoves)
    {
        scores[count] = MIN(t_board, t_depth + 1, posMove);
        count++;
    }

    int32 maxVal = scores[0];
    size_t maxPos = 0;
    // go through scores and find POS of maxScore
    for (size_t i = 0; i < possibleMoves.size(); i++)
    {
        if (maxVal < scores[i])
        {
            maxVal = scores[i];
            maxPos = i;
        }
    }

    // get the best move based on the max score positon
    return possibleMoves[maxPos];
}
```

Listing 11: Min-Max: calculate MIN in C++

```
int32 HardAI::MIN(API::Board t_board, uint32 t_depth, Movement t_move)
{
    // apply test move on the temp board
    t_board.move(t_move.first, t_move.second);

    API::Utils::MovesGenerator moveGenerator;

    auto color = (m_pawnColor == Red) ? Black : Red;
    // get all valid/possible moves in from the generator
    auto p_board = std::make_shared<API::Board>(); *p_board = t_board;
    moveGenerator.generatePossibleMoves(p_board, color, m_lastPlayedPawn);

    auto possibleMoves = moveGenerator.getPossibleMoves();

    // if no possible moves return an enormously big score
    if (possibleMoves.empty())
        return hasFoundEnemy(t_board, color) ? 10000 : -10000;

    std::array<int32, 32> scores;
    for (int32 i = 0; i < scores.size(); i++) scores[i] = 0;
    size_t count = 0;
    for (auto posMove : possibleMoves)
    {
        scores[count] = MAX(t_board, t_depth + 1, posMove);
        count++;
    }

    std::sort(scores.begin(), scores.end(), std::less<int32>());
    return scores[0];
}
```

Listing 12: Min-Max: calculate MAX in C++

```
int32 HardAI::MAX(API::Board t_board, uint32 t_depth, Movement t_move)
{
    // apply test move on the temp board
    t_board.move(t_move.first, t_move.second);

    API::Utils::MovesGenerator moveGenerator;

    // exit recursion
    if (t_depth >= MAX_LEVEL) ///> 4
        return calculateBoard(t_board);

    // get all valid/possible moves in from the generator
    auto p_board = std::make_shared<API::Board>(); *p_board = t_board;
    moveGenerator.generatePossibleMoves(p_board, m_pawnColor, m_lastPlayedPawn);
    auto possibleMoves = moveGenerator.getPossibleMoves();

    // if no possible moves return an enormously big score
    if (possibleMoves.empty())
        return hasFoundEnemy(t_board, m_pawnColor) ? 10000 : -10000;

    std::array<int32, 32> scores;
    for (int32 i = 0; i < scores.size(); i++) scores[i] = 0;
    size_t count = 0;
    for (auto posMove : possibleMoves)
    {
        scores[count] = MIN(t_board, t_depth + 1, posMove);
        count++;
    }

    std::sort(scores.begin(), scores.end(), std::greater<int32>());
```

```
        return scores[0];
    }
```

4.1.3 Performance

The level of depth is a sort of optimization because for most logic games generating full tree can take ages and I really mean it since I have tested depth of 12 on release. I could cook a dinner and watch quarter a season of some TV series and the program will most-likely still process the generation. There's a branching factor to be taken in account here. So let's assume if a game tree has a branching factor of 3, meaning each node has 3 children, the total number of nodes for a tree with depth n will have $\sum_{n=0}^{n} 3^n$.

General case:

$$\sum_{n=0}^{n} m^n$$

Time complexity: O(m^n)
Space complexity: O(m)

Memory note: Since Min-Max is implemented as a recursive algorithm and it creates a massive number of boards to test moves on depending on the level of depth, its recommended usage for most optimal result is from 4th to 8th level since it gets too slow above that and also very important, depending on the data structures and types used for the implementation, the stack memory that is storing all local variables can be filled quickly. Such worries are more valid for lower level languages such as C/C++ and must be taken in account, but either way it is important consideration.

5 Personal Evaluation

To sum it all up, checkers was a great exercise to improve skills and help gain more knowledge in both algorithms data structures and designing good software systems. The outcome is a well-looking piece of software with a working Min-Max algorithm, which gave me more confidence and better understanding in algorithmic problem solving approaches. This also helped me a great deal to become a lot better in debugging which was inevitable for the bug-fixing in the implementation of Min-Max. Last but not least, the evaluation of such interesting recursive algorithm made me think even more about performance and how to optimise code blocks when needed and only if needed and not to try and do premature optimisations that I got rid of some at later stage of the development.

References

[1] A. Shvets and G. Frey, "Design patterns," Aug. 2016.

[2] P. H. Winston, "Lecture 6: Search: Games, minimax, and alpha-beta," *MIT 6.034 Artificial Intelligence, Fall 2010*, Oct. 2010.