Documentation for semestral project
Programming II
“Gomoku”

1. Introduction

Gomoku is a very simple game. The aim is to build an unbroken chain of 5 stones of your color in one line (diagonal is also possible). The main difference is that here you play against the computer.

2. User Manual

When the user runs the program, a 15x15 grid appears. As stated earlier, the whole aim is to build a chain of 5 stones horizontally, vertically or diagonally. The user plays against the computer and the first move goes to the user. The user is red while the computer is blue. The user can choose any cell to put in his/her red stone. After the user’s move, the computer plays back and chooses where to put its blue stone. The user plays again trying to achieve the goal of the game and then the computer plays back trying to win as well. This process is repeated until either the user or the computer manages to form a 5x5 row of cells to win the game.

- When the user or the computer wins, the game finishes and the 5 cells are highlighted by a line crossing through them.

After that, the user can click once on any cell to restart the game.

3. Algorithms/optimizations

For this project, I used the GtkSharp library (for the graphical interface). Everything else was written using clear C# without additional libraries.

I decided to use this game to study the Minimax algorithm more deeply (this is the main aim of my project). It is smart and able to control the game situation a few steps further.

This algorithm allows you to make a tree of the current game situation and consider all possible moves starting from the current situation to as deep as you need.

The Minimax algorithm alone by itself is not efficient enough. For instance, in my game, there are 225 squares. If you consider all these

steps as possible moves, you will have more than 11 million gaming situations at depth 3 and therefore the following optimizations were implemented:

- Alpha-beta pruning
- Limiting the number of the possible moves - the program considers possible positions by only looking at those that are near the previous moves done.
- Checking efficiently the existence of the winner - The program does not look at the whole grid after every move, it checks if this move created a line of 5 stones in one of the 4 possible directions. This method improved the complexity from quadratic in terms of the board to linear in terms of the number of stones in a line you need to win.
- Making `UnMove()` instead of copying the current game situation. - It optimizes the algorithm because we just change some properties of the game and not clone the object millions of times.
- Evaluation function for limiting the recursion depth. - Not to explore the whole gaming tree, the program goes to some constant depth and then evaluates the current game state to optimize the algorithm speed. If any player occupies a cell, it counts the number of stones in each possible win line and depending on the number of them in one of the directions points are added to the player who created this line. - Additionally, the computer checks the outside cells of that line, if a cell contains the stone of the opponent or a cell is on the border of the gaming grid), the counter of one player is subtracted to get the approximation of the minimax value in this game state.

4. Code decomposition

My code is splitted into 3 parts (classes):

- Program.cs - here the Main() function lies and the application starts. Also I decided to place here classes that are responsible for drawing the game on the screen (Area that implements the DrawingArea and MyWindow that implements Gtk.Window)
- Game.cs - here we can find the single class of the game. It holds everything connected to the current game situation (grid, player, winner, etc.). Also there are lots of the game methods and methods for the Minimax algorithm (for example CheckWinner(), Move(), PossibleMoves(), UnMove(), etc.)
- Minimax.cs - here is the Minimax class with the most important Predict() method that predicts the winner of the current game situation and returns the best possible move for the computer. and the Evaluate() method to get the approximate minimax value of the current game situation.

5. **The most challenging parts**
   During the development I met lots of difficulties such as:
   - Big number of possible moves which affected badly the speed
   - Coding efficient check-win algorithm depending on the last stone placed on the grid
   - Evaluate() method was quite difficult to implement. Several options were tried but sometimes it would not cover all the chains correctly.

6. **Future work**
   In the near future I would like to add the following parts:
   - Some more graphical interface. Such that the user will be able to choose the color he/she wants to play. Menu with two gaming modes (against the computer and with a friend)
   - Continue developing the minimax algorithm and compare it to the algorithms of other developers
7. **Summary**

   To summarize everything above mentioned, I would like to say that this project was very useful for me as I studied one very important algorithm very deeply. I was trying to optimize the algorithm as best as it was possible to get the best performance. In the end I achieved the expected result and I am fully satisfied with my work.