Roadmap

The roadmap is used to plan out (and log) features for each tagged version of the game. Git tags will be used as well so a simple “git checkout tags/v0.1.0” can be used to check what the code was like at a certain point.

Contents

[V0.1.0 2](#_Toc472603372)

[Final version for V0.0.X 2](#_Toc472603373)

[V0.0.4 2](#_Toc472603374)

[V0.0.3 3](#_Toc472603375)

[V0.0.2 3](#_Toc472603376)

[V0.0.1 3](#_Toc472603377)

[V0.2.0 4](#_Toc472603378)

[V0.1.3 6](#_Toc472603379)

[V0.1.2 7](#_Toc472603380)

[V0.1.1 7](#_Toc472603381)

# V0.1.0

This version should have the raw foundation of the game, including the GUI for the player, the class to represent a game board, a few pre-requisites for AI, and a way to represent a tree of moves.

Goals:

* Have a class to represent the game board **[V0.0.2]**
* Have a basic controller class setup for the Player (AI will come later) to interact with the board via a GUI **[V0.0.3]**
* The GUI should at the moment, only bother displaying the game board, as well as who’s turn it is **[V0.0.3]**
* The game board class should have the basic game rules implemented (can only place pieces on empty spaces. Game is won if 3 pieces are in a row, etc.) **[V0.0.2]**
* Have a way to hash the game board **[V0.0.1]**
* Have a way to represent a tree of moves (the Node class and the MoveTree class provide this) **[V0.0.1]**
* Have a way to calculate the statistically best path of moves to make (Average.statisticallyBest sorts this out, I know it’s a bit strange to have this before working on the AI, but I just needed it out the way) **[V0.0.1]**
* Allow any Node class to be serialised/unserialised. **[V0.0.1]**

## Final version for V0.0.X

The final version of every Minor version update should contain general clean-ups/refactoring of code. The final version is generally also the next minor version. So the version 0.0.4 is also v0.1

Goals:

* Remove the ‘MoveTree’ class. First, create a Node.root property to create an empty node (suitable as a root). Then, move the MoveTree.walk function into the Node class, since it makes more sense and provides more flexibility there. **[Done in V0.0.2]**
* Use the ‘Board’ class as a namespace for the ‘Hash’ class. So ‘Hash’ becomes ‘Board.Hash’. This better reflects its purpose, and becomes consistent with ‘Board.Piece’. **[Done in V0.0.4]**
* ~~Change ‘Node’ and ‘Hash’ to be created using~~ [~~Object Initialisers~~](https://msdn.microsoft.com/en-us/library/bb384062.aspx)~~, as it creates cleaner (and clearer) code.~~ **(After a change of thought, this could be more of a burden, as it would require me to make some read-only properties non-read-only, and would make lines look bloated)**

## V0.0.4

Achieved:

* Put the ‘Hash’ class under the ‘Board’ namespace, so it is used like ‘Board.Hash’.

My reason behind this decision is because, in my eyes, the code is more structured like this.

## V0.0.3

Achieved:

* Add support for running a thread for all the game logic, and providing a way for the two threads to speak to each other.
* Add a GUI, and a controller to let the player interact with the board.
* Add code to handle when two controllers tie.
* Fixed a bug in the win checking code. It was checking a wrong slot when checking “Top right to bottom right”.

When deciding how the game loop should work, there were 3 different ideas that came to mind.

The first idea was to run the game logic in the same thread as the GUI. However, with the way the game loop works and is structured, it would cause the GUI to freeze and prevent the user from actually interacting with the game. This obviously was not ideal.

The second idea, and the one I went with, was to use a separate thread for the game logic, and provide a way (in this case, a thread-safe queue) to allow the GUI thread to communicate with the game thread.

The final idea was using C#’s support for asynchronous tasks. However, in my past experience with asynchronous code (In C#, at least), I haven’t ever been able to structure things in a sane way, and usually end up with a buggy, unreadable mess. So, I decided to avoid it.

## V0.0.2

Achieved:

* Add basic serialisation support.
* Add the ‘Board’, and ‘Controller’ classes, with very basic tests. (This was a lot of code)
* Implement serialisation for the ‘Hash’ and ‘Node’ class.
* Remove the ‘MoveTree’ class, and move its functionality to ‘Node’.

MoveTree was removed because it wasn’t offering any special functionality that I couldn’t just put inside ‘Node’. A special ‘Node.root’ function was made to replace usage of ‘MoveTree.root’.

## V0.0.1

Achieved:

* Have a way to hash the game board
* Have a way to represent a tree of moves
* Have a way to calculate the statistically best path to make
* Included this document

# V0.2.0

This version should be an unpolished version of what the final game will be like. This version should include at least 1 AI mode, a move tree which the AI uses to decide it’s move (and a way to load/save this data to a file), ~~and better management for when the game thread throws an exception.~~

Goals:

* Have at least 1 controller for the AI, which should use the ‘Average.statisticallyBest’ function to determine its next move. **[V0.1.2]**
* Create a static class which contains the code for interacting with the file system. The reason for a separate class is to allow a cleaner interface for the rest of the code, as well as separating file input/output code from things such as the AI. Example = ‘var move\_tree = GameFiles.loadMoveTree(MoveTrees.Global);’ to load the global move tree, the AI doesn’t need to know the specifics of it such as the file name, so the GameFiles class is used as an abstraction. **[V0.1.3]**
* ~~Provide a function inside of MainWindow for the game thread to call whenever an unhandled exception occurs (this function is called using the window’s dispatcher). The game~~ **~~should not crash~~** ~~due to its own mistakes, so any exception thrown must be reported and fixed. The game~~ **~~is only allowed to crash~~** ~~if, for example, the user decides to corrupt one of the files used by the game, and the game chokes when loading it. In short,~~ **~~crash due to the user, not due to buggy code.~~** ~~But even then, it’s ideal to just display an error message instead of crashing, since the program should still be in a valid state from a user error. [~~**Never mind, seems the program will crash regardless of what thread throws it. Which is the desired behaviour.] [V0.1.3 sort of implements this though]**
* Modify the ‘Node.walk’ function so it will instead, now allow a function to be called on every node that is walked to. The old functionality can still be replicated because the new functionality will allow much more flexible code, and will allow new opportunities. **[V0.1.1]**
* Provide an easy to use interface in the class described in the 2nd bullet point to save/load arbitrary trees of nodes.

Possible api = “GameFiles.saveTree(myRootNode, “super\_tree”);”,

“var root = GameFiles.loadTree(“super\_tree”)” **[V0.1.3]**

* Provide a way to merge the data from one move tree into another. For example, say I gave my friends the current version of my project to play with, and then I asked them for their AI move trees so I could merge them all into my personal AI move tree. This would allow for the AI to make use of more data. The algorithm the AI uses in onMatchEnd could be turned into a function for this. **[V0.1.3]**
* Provide a separate debug window that an AI can hook up into to allow a Graphical representation of their move tree to be seen, as well as tools to modify the tree. This window can also be used so the AI can report back any info I need. **[V0.1.2, no functionality to edit the node tree]**
* Using the algorithm used in “Average.statisticallyBest”, create a function such as ‘Node.walkEveryPath’ which will go over every possible path found in the tree, and perform an ‘Action<List<Node>>’ on every path found. **[V0.1.1]**

List of bugs that need fixing:

* When a match isn’t being performed, the player can queue up moves by pressing any of the slots. This will create a backlog of PlayerSelectMessages. Fixing it should be as easy as clearing the message queue anytime the player controller’s onDoTurn function is called.
* [Unsure] The AI, because it lacks enough data, will **always** start in the same position when the ‘doStatisticallyBest’ function is being used. I’m not too sure how to remedy this as of now…
* If an exception is thrown in the game thread, then the game thread will end early and the UI will effectively be doing nothing. This will just require changing where the try-catch is, as well as resetting some variables.
* The exception thrown in the Game thread, which is used to stop it when the window is closed, is passed into MainWindow.reportException. So, an error box appears every time the window is closed. This will simply require checking what exception is thrown, and preventing the one we don’t want from being passed.
* When the AI wins a round, the GUI doesn’t get updated to show where the AI put their winning piece. I’m pretty sure a single line of code will fix this.

## V0.1.4

Achieved:

* Fixed a bug where, if the AI placed the last piece of the match, then the GUI wouldn’t update to display where the AI put its piece.

## V0.1.3

Achieved:

* Exceptions thrown in the game thread will trigger a message box to appear in the UI thread, detailing the exception. In debug mode, this message box contains a stack trace. Ideally, this should only ever be seen in cases of I/O issues (such as a game file being corrupt). Any other kind of exception should require a fix in the code to prevent it. Despite saying I wouldn’t do this, it’s more user-friendly to say something’s gone wrong, rather than being silent and having the program not act correctly.
* Add the GameFiles class, which currently allows code to save and load trees by using names. GameFiles.saveTree(“MyTree”, root) | var root = GameFiles.loadTree(“MyTree”)
* Modify the AI to save and load its global move tree using the GameFiles class.
* Add Node.merge, which is a static function used to merge a ‘source’ node tree into a ‘destination’ node tree. The AI was originally using this algorithm to merge its local move tree into its global move tree. The original algorithm had no support for if the ‘source’ node tree had more than 1 path, so that support was added.

After this version, all that’s left will be bug fixes and changes to the algorithm the AI uses. After those are done, then the UI/overall experience needs to be polished (V0.3.0 will have the details). Once that’s all done, then the game is in a finished state (V1.0.0).

I was surprised (as well as happy) to find that the AI worked perfectly fine, without any issues, when I added the code to save/load. It makes me feel it’s at the very least coded in a good way. It also confirms to me that GameFiles’ interface is simple enough, as only 5 lines of code was needed (2 lines were calls to the GameFIles class, 2 were for error checking, 1 was adding a variable. Comments are not included in this number).

The debug windows + the unit tests for GameFiles makes me certain that all the data is being written/read in correctly.

## V0.1.2

Achieved:

* Create Config.versionString, which stores a string representing the current version of the game. Every window will have this string appended to its title.
* Add a Debug window which can be used to visually see a tree of nodes.
* Add a ‘Start Match’ button which will start a new match between the AI and the player.
* Modify the Controller base class so that onAfterTurn and onMatchEnd are given the current state of the board, and the index of the last piece placed. This is so the controllers don’t have to track this information themselves, which creates bugs. It also means they can be certain the data is correct.
* Add an ‘AI’ controller, which currently has a ‘doRandom’ and ‘doStatisticallyBest’ mode.

This update is the first on to feature the AI’s main mode, ‘doStatisticallyBest’, in action. Currently the AI can’t load data from previous sessions, so it has to remake its global tree from scratch each time the program is run.

The AI, once it has played one winning match, will always try to place its first piece in the exact same position as the first winning match. This is simply because there’s not enough data for the AI to work with.

I will have to figure out a way to try and get the AI to randomise whether it uses ‘doStatisticallyBest’ one match, and ‘doRandom’ another match, so it has more data to work off of.

Another solution might be to pre-create a global tree, with one winning path for each starting position the AI could use. He should then have enough data to properly choose any of the slots to start off with.

## V0.1.1

Achieved:

* Display the current version of the project within the window’s title. E.g. “CS Project v0.1.1 prototype”. If “prototype” is displayed, then it is an incomplete version. So “v0.1.1 prototype” is an incomplete version of version 0.1.1, whereas “v0.1.1” would be the final, complete version of version v0.1.1.
* Create ‘Node.walkEveryPath’ which is a function based off of the algorithm used in ‘Average.statisticallyBest’. This function allows code to perform an action on every possible path in the tree. ‘Average.statisticallyBest’ was modified to use this new function.
* Modify ‘Node.walk’ to allow an action to be performed on every node walked to.

This version of the game will not have a single noticeable difference when playing the game, since this is exclusively backend stuff (that for the most part, isn’t even used yet). It focuses more on making it easier to write code that traverses a node tree, which will be very beneficial for the near future for the AI/any other algorithm that uses the node tree.

# Chronological Order of Development

This table is a compilation of every development point from every version of the game and is placed in chronological order. This is here as evidence of the game’s iterative development.

All dates are taken from when the changes were added to the git repository.

It should also be noted that a few points not mentioned earlier in the document will be present in this table.

|  |  |  |
| --- | --- | --- |
| Description | Version | Date |
| Have a way to hash the game board | 0.0.1 | 5/12/2016 11:56 PM |
| Have a way to represent a tree of moves | 0.0.1 | 15/12/2016 8:41 AM |
| Have a way to calculate the statistically best move to make | 0.0.1 | 15/12/2016 10:10 AM |
| Include this document | 0.0.1 | 15/12/2016 10:36 AM |
| Add basic serialisation support | 0.0.2 | 26/12/2016 11:50 PM |
| Implement serialisation for the Hash class | 0.0.2 | 26/12/2016 11:50 PM |
| Remove the MoveTree class, and move its functionality into the Node class | 0.0.2 | 27/12/2016 12:03 AM |
| Implement serialisation for the Node class | 0.0.2 | 27/12/2016 12:39 AM |
| Add the Board and Controller classes | 0.0.2 | 28/12/2016 12:11 AM |
| Add the game logic thread, as well as a way for the UI and game threads to talk to each other | 0.0.3 | 28/12/2016 1:31 AM |
| Add the GUI, as well as the PlayerController class | 0.0.3 | 28/12/2016 4:21 AM |
| Fixed a bug in the win checking code, where a wrong slot was being checked | 0.0.3 | 28/12/2016 5:43 PM |
| Prevent the player from placing a piece in a non-empty slot | 0.0.3 | 28/12/2016 5:58 PM |
| Modify the Board class to support when two controllers tie | 0.0.3 | 28/12/2016 7:01 PM |
| When the user hovers their mouse over a slot in the GUI’s grid, change its background colour. Aka, make it more interactive. | 0.0.3 | 29/12/2016 3:48 PM |
| Add a label to the GUI, informing the user what piece they’re playing as | 0.0.3 | 29/12/2016 3:49 PM |
| Put the Hash class under the Board namespace | 0.0.4  0.1.0 | 4/1/2017 11:49 PM |
| Display the current version of the project in the window’s title | 0.1.1 | 7/1/2017 12:52 AM |
| Modify Node.walk to allow an Action<Node> to be passed to it | 0.1.1 | 7/1/2017 1:58 AM |
| Create the function Node.walkEveryPath | 0.1.1 | 7/1/2017 3:28 AM |
| Add a Debug window to easily view a tree of Nodes | 0.1.2 | 10/1/2017 4:58 PM |
| Create static variable Config.versionString | 0.1.2 | 10/1/2017 5:04 PM |
| Modify the Debug window so a piece of text on screen can be changed, so things like the AI can give some extra feedback | 0.1.2 | 11/1/2017 6:24 AM |
| Add the first version of the AI. It only uses its ‘doRandom’ algorithm, and has an untested, and unused ‘doStatisticallyBest’ algorithm.  (While the global move tree existed, there currently wasn’t any data going into it. This mean that the AI always had to fall back onto doRandom) | 0.1.2 | 11/1/2017 7:36 AM |
| Add a button to start a new match (also remove the requirement of having to restart the game to play more than one match) | 0.1.2 | 11/1/2017 7:46 AM |
| Add code to the AI so it merges its local move tree into the global one.  (After this was done, I then manually tested the AI’s doStatisticallyBest algorithm… and it worked without issue) | 0.1.2 | 11/1/2017 8:51 AM |
| Modify the Controller base class, so certain functions are given more information | 0.1.2 | 12/1/2017 9:16 AM |
| Add the MainWindow.reportException function, which will display an exception inside of a message box so the user knows something’s gone wrong. | 0.1.3 | 13/1/2017 6:55 AM |
| Add the GameFiles class | 0.1.3 | 13/1/2017 7:52 AM |
| Modify the AI so it saves/loads its global tree to a file.  (This surprisingly worked without any issue as well) | 0.1.3 | 13/1/2017 8:05 AM |
| Add the Node.merge function | 0.1.3 | 17/1/2017 12:13 PM |
| Improve Node.merge so the source tree (which goes into a destination tree) can have more than one path.  (If there were more than one path, it simply wouldn’t be merged over. The debug windows allowed me to catch this) | 0.1.3 | 17/1/2017 1:21 PM |
| Begin the writeup of the project’s report  (I did do this severely out of order, I admit) | 0.1.4 | 24/1/2017 3:11 PM |
| Add this table of chronological development | 0.1.4 | 24/1/2017 11:08 PM |