Kandidatexamensarbete vid CSC, KTH

Reinforcement learning AI to Hive
Förstärkningslärande AI till Hive

Authors:
Blixt, Rikard
rblixt@kth.se
Professorsslingan 45 lgh 1404
114 17 Stockholm
076-894 37 88

Ye, Anders
aye@kth.se
Kungshamra 31 lgh 1118
17070 Solna
070-757 12 36

Course:
DD143X
Degree Project in Computer Science, First Level

Supervisor:
Austrin, Per

Reinforcement learning AI to Hive

Abstract

This report is about the game Hive, which is a very unique board game. Firstly we cover what Hive is, and then later details on our implementations of it, which issues we ran into during the implementation and how we solved those issues. Also we attempted to make an AI and by using reinforcement learning teaching it to become good at playing Hive. More precisely we used two AI that has no knowledge of Hive other than game rules. This however turned out to be impossible within reasonable timeframe, our estimations is that it would have to run on an upper-end home computer for at least 140 years to become decent at playing the game.

Förstärkningslärande AI till Hive

Sammanfattning

Denna rapport handlar om det unika brädspelet Hive. Rapporten kommer först berätta om vad Hive är och sedan gå in på detalj hur vi implementerar spelet, vad för problem vi stötte på och hur dessa problem löstes. Även så försökte vi göra en AI som lärde sig med hjälp av förstärkningslärning för att bli bra på spelet. Mer exakt så använde vi två AI som inte kunde något alls om Hive förutom spelreglerna. Detta visades vara omöjligt att genomföra inom rimlig tid, vår uppskattning är att det skulle ha tagit en bra stationär hemdator minst 140 år att lära en AI spel Hive på en godtagbar nivå.

Foreword

This report has been created in correlation with the course DD143X - Degree Project in Computer Science, First Level, 15 ECTS. The course was taken while studying at for a *Master of Science in Engineering, Major in Computer Science and Technology* at *the School of Computer Science and Communication* (CSC) in *the Royal Institute of Technology* in Stockholm, Sweden (*Kungliga Tekniska Högskolan*, KTH).
Both co-authors have done a fairly equal workload, both on the report writing side and the programming side. However Anders did slightly more work on the report while Rikard did slightly more on the programming side, especially related to testing and debugging.

Table of Content

1. Introduction 7
1.1 Background 7
1.1.1 Adding a new Piece 7
1.1.2 Moving a Piece 7
1.2 Artificial Intelligence 12
1.3 Purpose and Problem Statement 12
2. Implementation 14
2.1 Hive 14
2.2 AI 17
3. Results 18
3.1 AI Learning 18
3.2 Graphs 18
4. Discussion and Conclusion 21
4.1 Discussion 21
4.2 Conclusion 22
References 23
Appendix A 24
More Data from Learning Runs 24
Appendix B 25
Game and AI Code 25

1. Introduction

1.1 Background

Hive is relatively new strategic board game, compared to most board games, it was only created in 2001 by John Yianni. One key difference between Hive and other board games is that Hive does not have a board that is the playing field, instead of a board the playing field is made up by the hexagonal pieces known as *the hive*. Because you both move around pieces and add new pieces to the playing field while playing the game, this causes the playing field to change over time.
At the start of the match, each player has eleven pieces of five different types, all in the hand with the play field empty. These five types players get in different quantities, they consist of one queen bee, two beetles, two grasshoppers, three spiders and three soldier ants. The goal of the game is to surround the enemy's queen bee, on all six sides with pieces from either player. For example if the black queen bee has 5 friendly pieces (pieces that are black in this case) next to her, and a white spider moves in on the last empty hexagon next to the queen, causes white to win.
One of the most important rules in Hive is the "only one hive" rule. This rule refers to the playing field, it may not be broken up into multiple non-connected parts, and every piece must be connected to every other piece through other pieces. Each of the five different types of pieces has their own movement rules. Each turn the player has a choice between adding a new piece or moving an already existing piece based on a few rules. If the player cannot move or add a piece without breaking the rules, the turn will automatically go to the opposing player.
At the moment, there are two expansion packs available to the game, with a third one in development. Each expansion pack adds an additional piece to the game; the mosquito, the ladybug and the pillbug respectively. However they are not dealt with in this report. [1][2]

1.1.1 Adding a new Piece

On the first turn, each player may freely place a piece of his or her choice, as long as the one hive rule is not broken. However on future turns a player may only place a piece if the target location is next to a friendly piece and is not next to a hostile piece (piece of opposite color). In addition if it's the 4th turn the queen bee must be placed if it hasn't been already. An unofficial rule that tournaments often use is to disallowing the placing of the queen bee on the first turn. This is due to a tendency for draws if the queen bees from both players are placed on the first turn. [1] [2]

1.1.2 Moving a Piece

Each piece type has its own movement rule, but there are some general rules applying to all pieces. Firstly the one hive rule may not be broken during movement, even if the new

position of the piece would make it to only one hive again. Naturally, pieces may not move on top of each other, with one exception, the beetle. Additionally pieces may not be moved until the queen bee has been placed on the playing field. Finally, pieces may not move through spaces they do not naturally slide into, see the queen bee's image for an example. [1]

1.1.2.1 Image Explanation

In the pictures below we are showing the possible movements of the different pieces. For your convenience here's a short coverage the different colors mean:

* Black tiles represent any hive piece, from either player.
* Green tiles represent a valid move.
* Red tiles represent a specific invalid move we'd like to point out.
* White tiles are unoccupied and invalid to move to.
* Black tiles with green center means there's a hive piece here and the beetle can move on top of it.
* Blue tiles are used to explain how the spider can come to a green tile; it cannot stop on blue tiles however.

1.1.2.2 Queen Bee

The queen bee has the simplest movement rule of them all. It may move one step in any direction as long as that move doesn't break any other rules. [1]

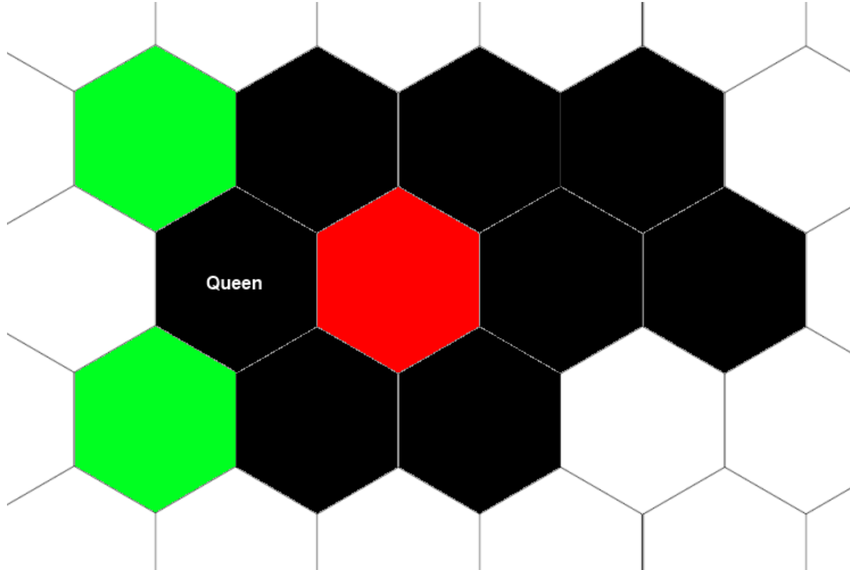

**Figure 1: This figure illustrates the movements of the Queen Bee.
In addition it shows the naturally slide into rule talked about in end of 1.1.2.
The queen does not naturally slide into the tight space on her right (red).**

1.1.2.3 Beetle

The beetle moves one step in any direction similar to the queen bee, however as mentioned before it may move on top of other pieces. Pieces below located below it may not move, as well as for the adding new piece rule, this entire stack of pieces is counted as the beetle at the top. You may move your beetle on top of another beetle, potentially creating a stack of up to five pieces. [1]

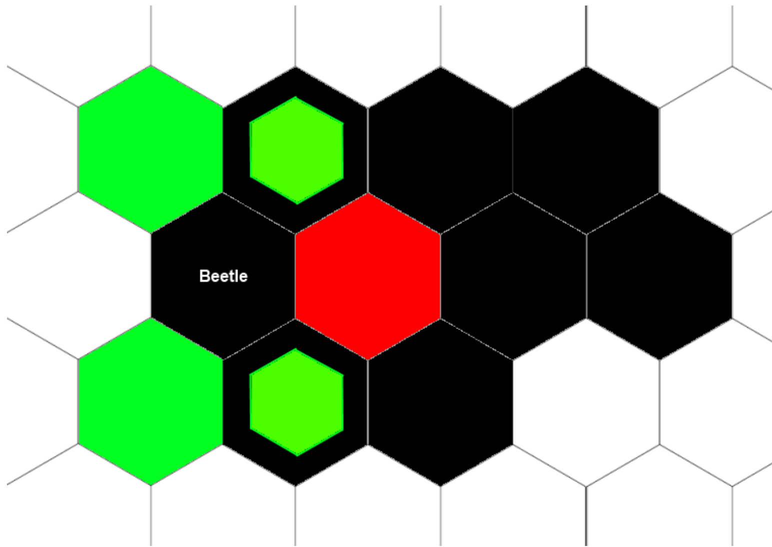

Figure 2: This figure illustrates the movements of the Beetle.
The beetle moves like the Queen, but can walk on other pieces.

1.1.2.4 Grasshopper

The grasshopper jumps over pieces, it may only jump in a straight line, until the first empty hexagon. It must jump over at least one piece. [1]

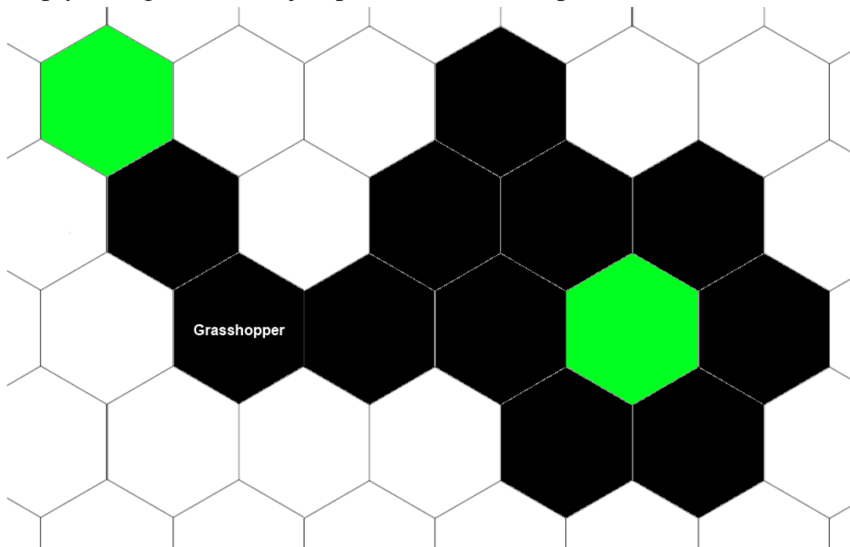

Figure 3: This figure illustrates the movements of the Grasshopper.

1.1.2.5 Spider

The spider always move exactly three steps, and on none of these steps may it step on a title is has already visited during these three steps, including initial position. Each step the spider must move around another piece it is connected with, which means that the spider can't jump a gap (going from left 1 to the green next to it would be jumping the gap). [1]

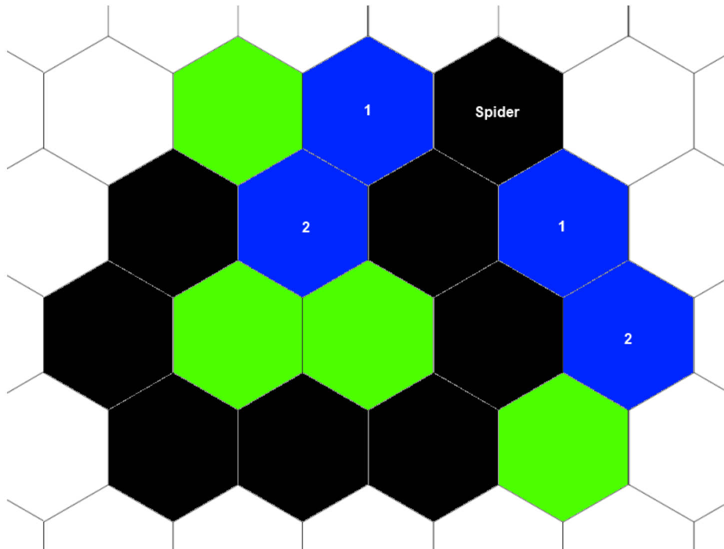

Figure 4: This figure illustrates the movements of the Spider.
The spider cannot move in a 1-2-1 pattern which would be backtracking from 2 to 1.

1.1.2.6 Ant

The ant moves very similar to the spider, but is not fixed at three steps, but instead as many as the player chooses. The ant is generally seen as the strongest tile in the Hive, since it can move to almost any position. [1]

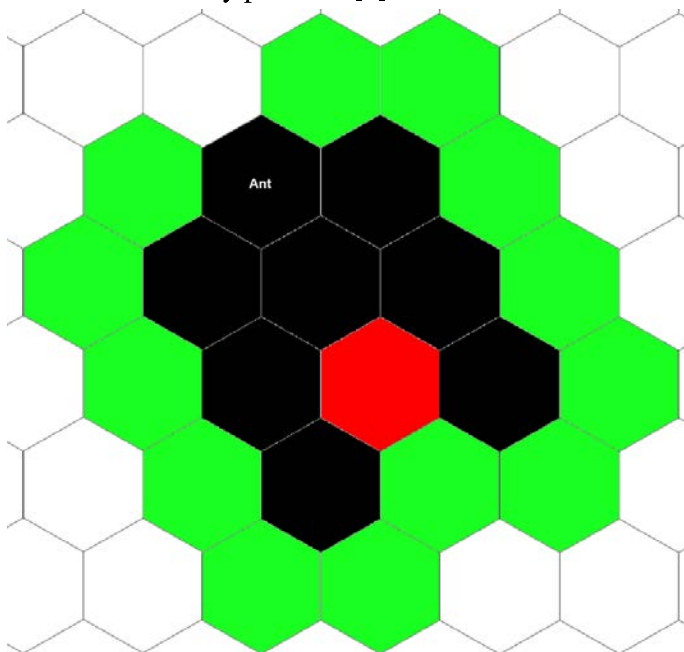

Figure 5: This figure illustrates the movements of the Ant.

1.2 Artificial Intelligence

Artificial Intelligence (AI) is a branch of computer science which aims to create machines that can take the best actions based on given data. [4] If we already know the best action according to given data and the problem's size is small, it is easy to create the perfect AI. But in most cases either the problem's size is too large, or we ourselves don't know the best action. A good example is chess, the game has been played and studied for several hundreds of years and we still don't know the best action in every state.

Machine learning is a major branch of AI which is about constructing machines which can study and learn from data and experience. One of the largest none machine learning branches of AI is search algorithms; here one can use large search trees and heuristic to pick the most optimal action. Search algorithms do not learn by experience. Back to machine learning, there are many way for a machine to learn by experience, all quite different. The one that's covered in this report is called reinforcement learning, it will be explained more in detail soon. Other common types of machine learning are decision tree learning, artificial neural network and clustering analysis. [5][7]

Reinforcement learning is a type of machine learning that aims to create a machine which tries different approaches and learns from these approaches based on a carrot and stick approach. The basics of reinforcement learning are to have a set of states and a set of actions where every action will lead to a new state. Every action also has a weight, which decides how good the AI sees this action. The machine will then give or take weight from the action depending on the result (carrot or stick). [6] This can be seen similar to the following real world scenario. In this scenario we have a student taking an exam without any knowledge in the subject, after each attempt the student will either receive a gold star ("carrot") if he passed the exam or a whip on the hands ("stick") if he failed the exam. After lots of attempts the student will ace the exam, scoring full points, still without any actual knowledge on the subject.

1.3 Purpose and Problem Statement

There are many multiplayer games where you can play competitively including a few board games such as chess and go. While playing these games, one almost always wants to play against someone on the same level as oneself, since the match will be a lot more interesting that way. Though there is an exception to this, it may still be fun to win against a friend or someone else even if they are obviously weaker, because it boosts ones confidence and pride. However this exception only applies while playing against a real person, when the opponent is an AI, the game will be boring if it is too easy. So when playing against an AI, the AI has to be strong in order for one to improve while playing and become stronger for the next match against a real person.

Hive is a relatively new board game and has not been studied as much as chess or other board games, there are therefore not many strategies and books about it. We will thus try to create a strong AI without knowing any strategies.

We also like to see if it’s possible to make a strong AI by playing two dumb AI against each other for a long time. An allusion to the real world could be two people that only know the very basics of mathematics trying to learn linear algebra by competing against each other … who can score the best on exams. Each time they only get to know who got the most correct answers, and not which questions were answered correctly.

2. Implementation

2.1 Hive

Since the game has not been released open source anywhere, barely been created for computer at all with only one browser version existing, we had to create the game ourselves. The first topic we discussed was problems we would run into during development. The first obvious problem was that the game board's tiles are hexagonal instead of square, how do you represent hexagonal tiles in the computer's memory? After staring on an image with several hexagonal tiles, we came up with an easy and elegant solution; using two simple arrays to form a matrix with a slight change of the definition of what a neighboring tiles are. When using squared tiles neighboring tiles are easy and intuitive; say [n][m] is the current tile { [n-1][m], [n+1][m], [n][m-1], [n][m+1]} are the tiles neighboring it. With our solution to the hexagonal problem, [n][m] have two additional neighbors { [n+1][m-1], [n-1][m+1] } for a total of six.

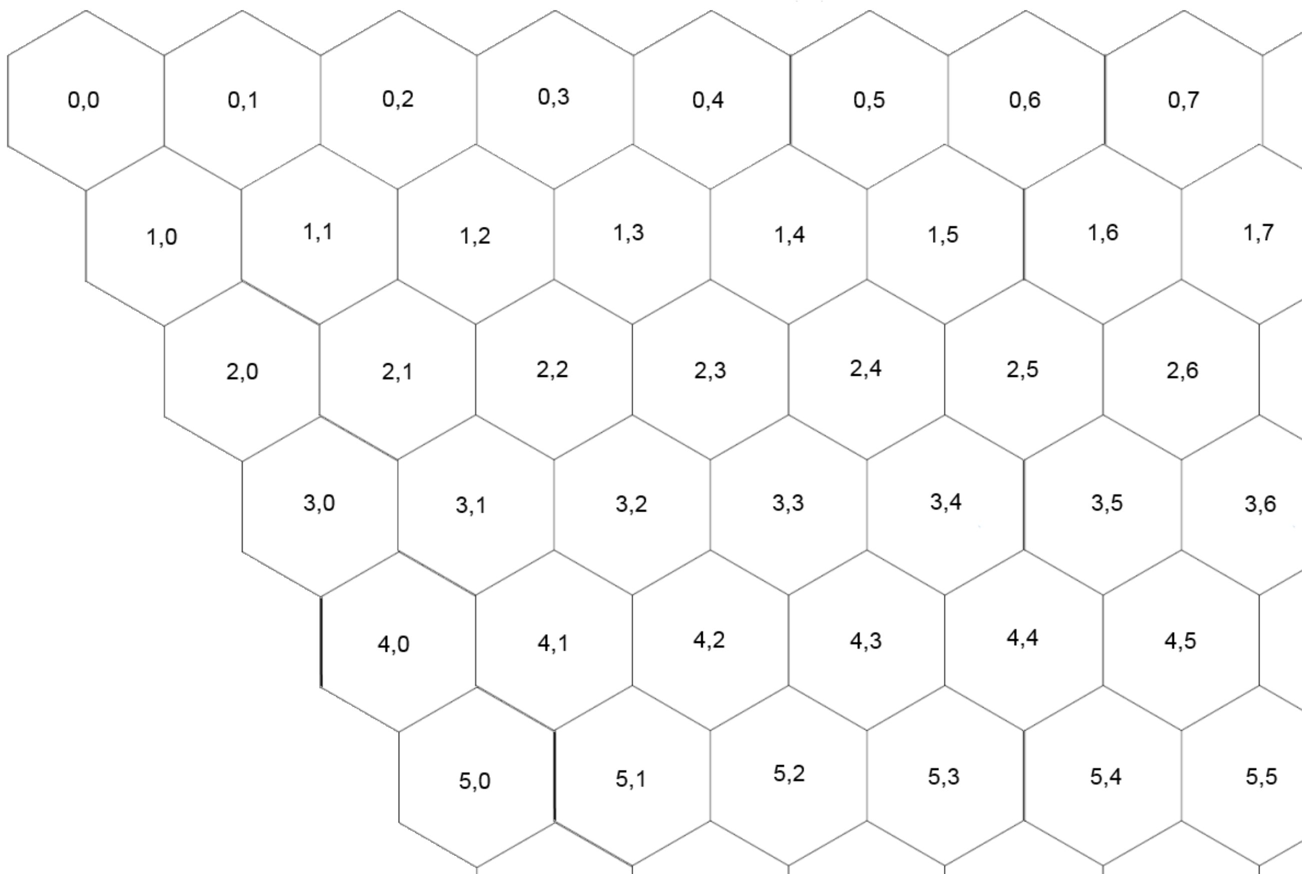

Figure 6: This figure illustrates how one can visualize our solution.

The other major problem we identified was that the game board can grow infinitely large by moving pieces in one direction forever, see picture below.

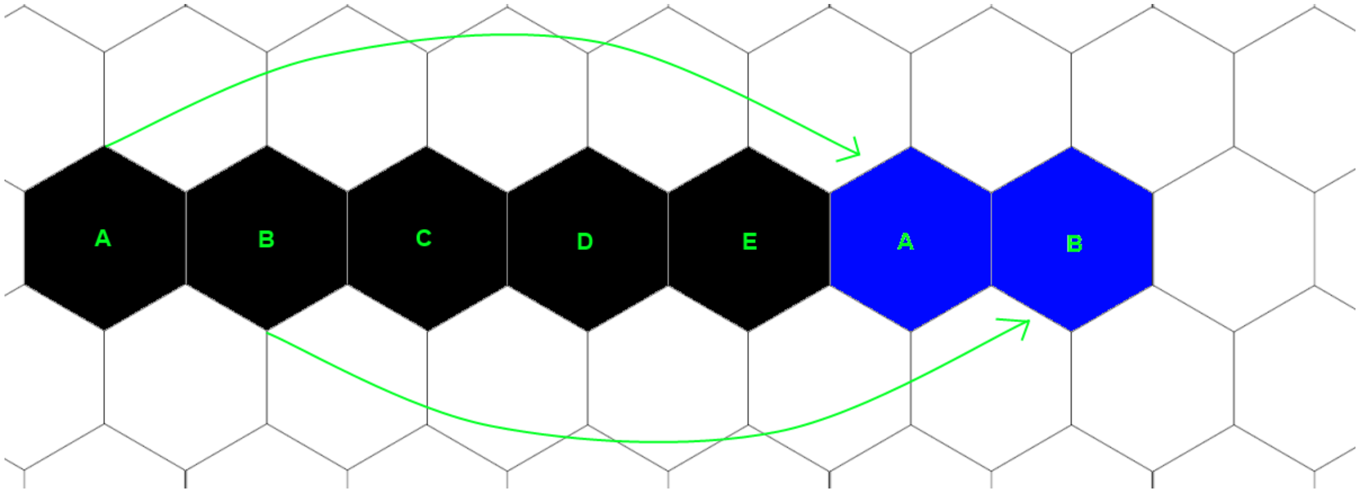

Figure 7: This figure illustrates how tile A, B, C, D and E can endlessly move to the right. If C move to the right side of the new position of B, D to the new position of C etc.

Quickly, it was realized that due to the amount of pieces for each player, eleven, the maximum needed size at any given time would be no larger than 24x24 (straight line in any direction with the possibility to move outside it by one step). After some discussion we came to the conclusion that we'd use a fix 24x24 matrix; centering the pieces after every move. This makes them never being able to grow beyond the bounds of the matrix.

Now with the solutions to upcoming issues figured out, we made a class diagram for the game. It's pretty simple, the game class keeps track of and handles general functionalities, such as turns and game board. Action and AI are interfaces for use with an AI with GameAction being an implementation of the Action interface; and Piece is the interface used by the pieces in the game (Ant, Beetle, Grass, Queen, and Spider). The individual pieces only have a custom toString and a custom move method.

During implementation it was noticed that handling coordinates on the map was an inconvenience, so an additional class was created. The class contains an i and a j value, representing the position in the game board matrix, in addition to lots of methods, such as "getNearbyCords()" and "isSurrounded()".

Here's a detailed class diagram of how the game looks like in the end.

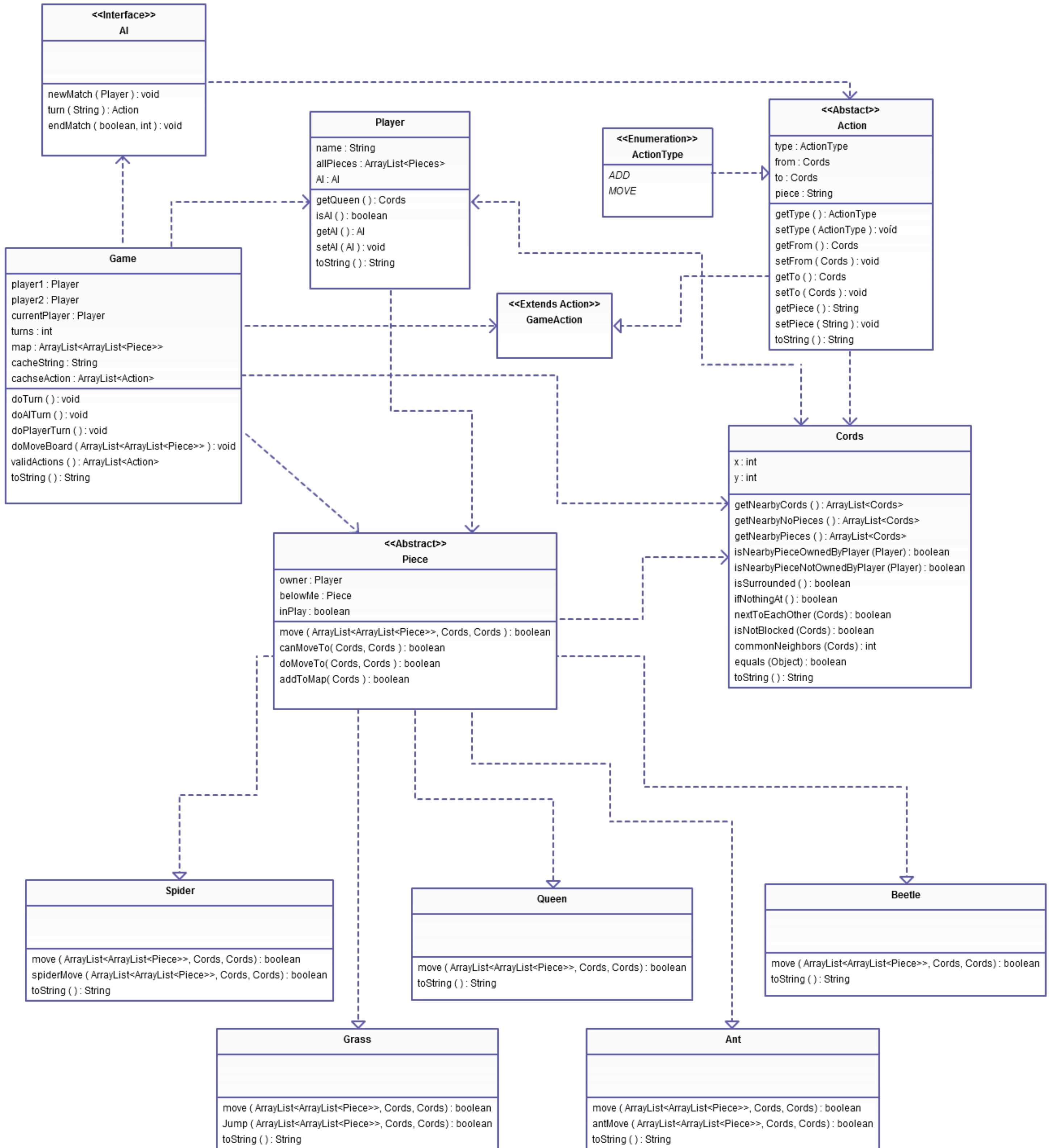

Figure 8: This figure illustrates our class diagram.

2.2 AI

The basic needs of reinforcement learning are to have a set of states, a set of actions and rules to go from a state to another state. In our case, a state is how the game board looks like, an action is a move by a player which changes a state to another state. Except the already mentioned the most important thing in reinforcement learning is deciding how to reward or punish the AI. [3][6]

The first thing we had to figure out when doing the AI was how to represent a state and an action. The decision how to represent an action was taken during the implementing of the game, see figure 8. So when corned with the problem on how to represent a state, a simple solution was selected, during the development of the game a “print game’s board” had been created for use with the console interface. The print was taken to use as the game board’s current state but all unnecessary character was removed, such as space, new line and “-“ (in order to save memory). Also a way was needed to calculate all possible actions given the current state, so a method was implemented doing just that. After some careful considerations it was decided that this method was better located in the main game instead of in the AI. This is due to if the game was to be expanded in the future to include a GUI, a method just like this one would be required to highlight possible moves for each piece.

After the AI was implemented it had to play matches in order to learn, but before that commence, the way to reward and punish the AI had to be decided. After some consideration it was decided to let each action A of a state S have a weight which is an estimate on how good the action is based on previous experiences. Before any action was used in a certain state, all actions were given a total weight of 10000, so each individual action was given a weight of 10000/sumActionsInState. During a game every action used by the AI was record and then added or subtracted weight using the following algorithm; say an entire game is worth 1000 weight, so an used action will be rewarded or punished with e.g. [1000/amount of action used] (minimum weight 1 however), which means that short and quick games will be more rewarding, instead of prolonged games where there is a greater chance of doing a mistake. Which action will be chosen the AI is easy to pick, simply randomize an integer X between 0 and then sum of all weights for this state, then go through all actions one by one and subtract the action’s weight from X. The action that makes the [X <= 0] is the chosen action. This means that action with high weight is more favored, but still encourages exploring of new tactics. Furthermore if the last action resulted in a win, the AI will always use this action in the future, and the opposite if last action resulted in a loss, the AI will never use this action again. Because the wanted AI is supposed to be as dumb as possible (see purpose), no long term sight was given to the AI, it will not consider future states.

3. Results

3.1 AI Learning

For this phase we used a few variants on the AI when it comes to their configuration. They are called A, B and C.

Type	Start Weight	Change Weight
A	10000/actionCountInState	1000/usedActionsInMatch
B	10000/actionCountInState	10000/usedActionsInMatch
C	10000/actionCountInState	100000/usedActionsInMatch

Figure 9: actionCountInState refers to the amount of different actions in a specific state. usedActionsInMatch refers to total amount of actions used in the specific match.

Due to the extensive time it took to run just a few matches, unfortunately only two longer learning runs were completed.

Run	AI 1 Type	AI 1 Color	AI 2 Type	AI 2 Color	Matches Played	Running Time	Win Percentage (White)
1	B	White	A	Black	200	14h 34m	53%
2	C	White	C	Black	300	21h 45m	44%

Figure 10: This table shows our configurations for our 2 runs. White is the one who does the first action.

3.2 Graphs

Graphs are a nice and clean way to present data. The following graphs are based on the second run of matches. More data can be found in Appendix A.

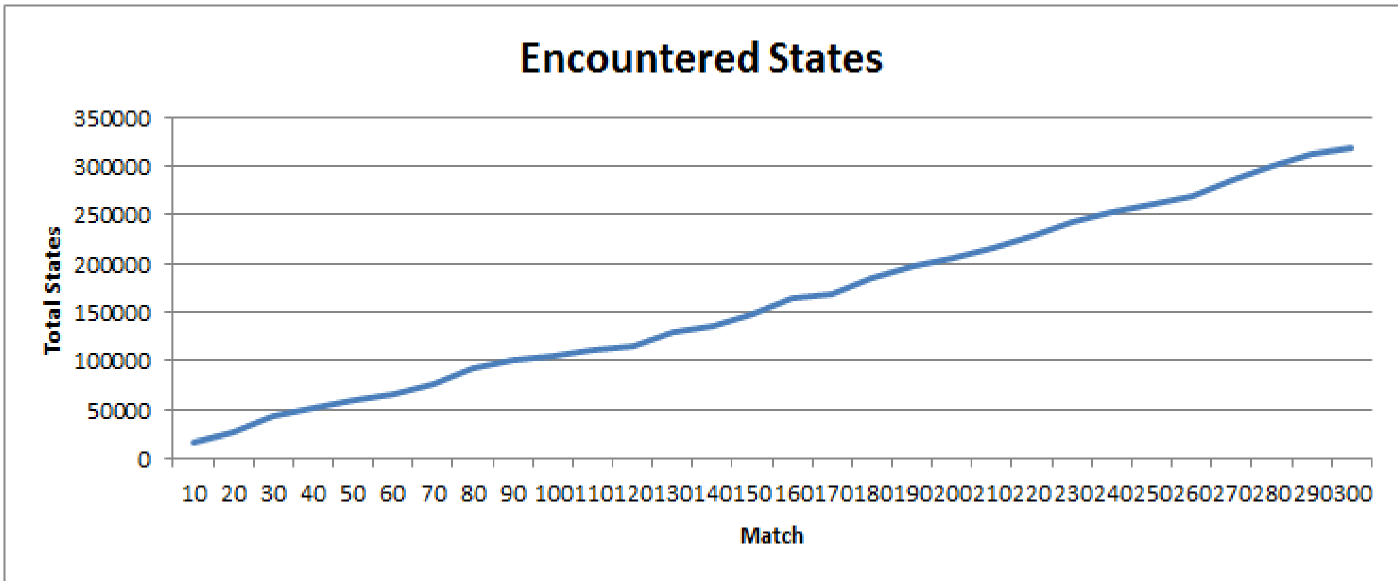

Figure 11: This figure illustrates how many states we have encountered after a number matches. As one can see, the growth is close to linear.

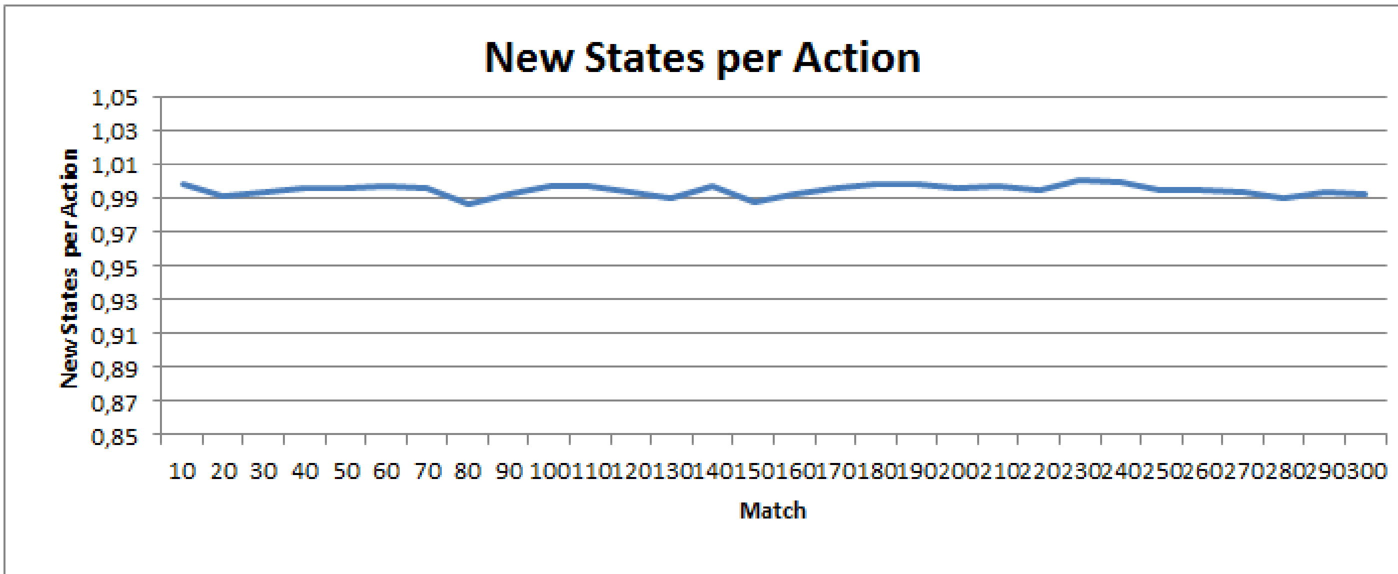

Figure 12: This figure illustrates how many new states we encounter per action made. 1.00 means that we encounter a new state each action. As we can see, the encounter ratio is over 99%.

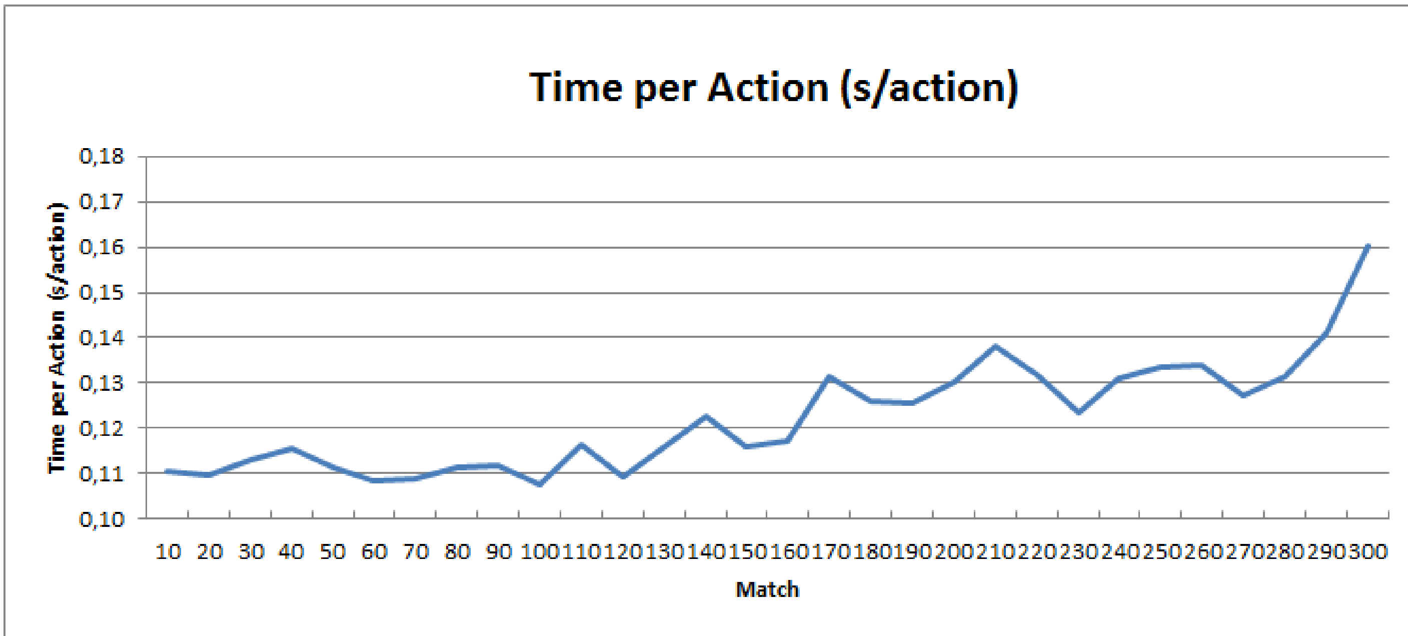

Figure 13: This figure illustrates how long time the AI needs to make a decision which action it should take next.

4. Discussion and Conclusion

4.1 Discussion

After only 300 games each AI had already seen 318 thousand different states with a total of 17.4 million possible actions to commit, since each AI only saves data from its own turn, there were around 636 thousand states the game had been in. This was a large toll on the primary memory causing more and more usage of the page file. This caused a drastic increase in time per move that can be seen in figure 13, in the first 100 matches it takes around 0.11 seconds per move but increase to 0.16 seconds per move for the last few matches of the test. The change in speed is first noticed after 120 matches, and the delay is increasing fast with an average move after 300 matches takes 50% longer than a match at 100 matches and we suspect it will continue to increase in a similar fashion.

As we can see in figure 12 the amount of new states per move is over 99% and not decreasing, this indicates that we almost never go to an old state, more exactly we only revisit an old state approximately 0.49% of the turns. On average, an AI does 1000 turns per match, which means that each match the AI only visits 5 old states. Given that the first state is guaranteed to be an old state after the first match, it is actually only 4 old states per match, implying 0.39% is the probability of visiting an old state after each action.

Since we have 0.39% chance to visit an old state each time we make a move it means that at best we have visited 0.39% of all states the game can be in. There is a chance for backtracking which will visit an old state and increases the chance to visit another old state. Also, as one can see in figure 12, the chances of visiting an old state has yet to decrease which indicates that there are still more than 99.61% states to visit.

Let us assume that we actually have visited 0.39% of all states in our 300 matches, and let us also make the assumption that we will visit the same amount of new states in the upcoming 300 matches, and so on. With these assumptions, we will need to play: *100/0.39 * 300 = 76923* matches to encounter every possible state. Playing 300 matches for the two AIs took 21 hours and 45 minutes on the computer we used, 76923 matches with the same speed would take us *21.75 hours/300 * 76923 = 5576.92 hours* = 93 days to only visit each and every state. For our AI to learn something, we would want it to at least try each and every action at least once, since the average amount of actions in a state is 54.55 it would mean that we would need to use at least 55 actions from the average state, and hence visit every state at least 55 times so for a minimum to use every action once, it would be *93 days * 55 = 5115 days* = 14 years. The AI would not learn much by using an action once, 10-100 times would be needed to actually learn something useful which would take 140-1400 years. And this is calculated with a best case scenario and even before considering the additional time each action takes due to paging when reaching larger amount of known states and actions.

A possible solution for the memory problem would be to create a file for each state which has information about every action and its weight. Every turn the AI would find the correct file, load the data from it and then decide which move to make. This is somewhat manual paging but would be required when the program exceed the page file limit. Although this is likely to be substantially slower while still using the RAM for most the work in the normal case, it may actually have improved speed when reaching very large page files. Additionally, compared to the page file, small separate file would only be limited by secondary memory (e.g. HDD) size; unlike the page file which generally has a maximum size.

4.2 Conclusion

Ultimately, we could not use reinforcement learning to create a strong AI for hive by using two as dumb as possible AIs to teach each other. This has mostly to do with the amount of possible actions and the time it takes to try them out. The time it takes for the AI to play enough matches to encounter each and every state a numerous times to try each and every action at least once would be too long. Without calculating the increment of time it takes when changing to use page file instead of primary memory and ignoring the probability that it may use an action it already has used before. So the calculated 14 years needed to use each and every action at least once is calculated in the best case scenario, hence ignoring the two increasing factors just mentioned.

References

1. John Yianni. Hive: a game buzzing with possibilities [homepage on the Internet]. [updated 2010; cited 2013 Mar 7]. Available from: http://www.gen42.com/downloads/rules/Hive_Rules.pdf

2. Hive (Game) [homepage on the Internet]. [updated 2013 Mar 1; cited 2013 Mar 7]. Available from: http://en.wikipedia.org/wiki/Hive_%28game%29

3. Sutton R, Barto A. Reinforcment Learning: An Introduction [serial on the Internet]. [updated 2005 Jan 04; cited 2013 Mar 21]. Available from: http://www.incompleteideas.net/sutton/book/ebook/the-book.html

4. Artificial Intelligence [homepage on the Internet]. [updated 2013 Mar 8; cited 2013 Mar 11]. Available from: http://en.wikipedia.org/wiki/Artificial_intelligence

5. Machine Learning [homepage on the Internet]. [updated 2013 Mar 4; cited 2013 Mar 11]. Available from: http://en.wikipedia.org/wiki/Machine_learning

6. Reinforcement Learning [homepage on the Internet]. [updated 2013 Mar 5; cited 2013 Mar 11]. Available from: http://en.wikipedia.org/wiki/Reinforcement_learning

7. Cory Janssen. What is Artificial Intelligence [homepage on the Internet]. No Date [cited 2013 Apr 05]. Available from: http://www.techopedia.com/definition/190/artificial-intelligence-ai

Appendix A

More Data from Learning Runs

Can be downloaded at http://csc.kth.se/~rblixt/kex/stats.xlsx

Appendix B

Game and AI Code

Can be downloaded at http://csc.kth.se/~rblixt/kex/code.zip