# Bot for Hanabi Game

AJAYKIRAN PALAMTHODI, ap21903@essex.ac.uk

MSc, Advanced Computer Science

University of Essex, UK

ap21903@essex.ac.uk

We were not consistently getting a high score when playing the Hanabi bot. By introducing some new rules and taking cross over of some features from SHC, population-based GA and MCTS, I have improved the bot to score 15 almost consistently. The bot logs the data when it plays and uses it to improve itself further. So far, we have an **97.04 %** of score being higher than 13 in a lifetime of **5000** games.

## Introduction:

I have used here elements of 3 different algorithms to consistently get a score around 15 in the game of Hanabi. The 3 algorithms being Stochastic Hill Climber, Population based GA and Monte Carlo Tree Search. I have also introduced 5 more rule which I think had made the score higher. I have also logged the total number of games; total times score was higher than 13 and percentage of times score was higher than 13 in the bot's lifetime.

I have introduced a new bot **chromosome_logger** to log values initially when there is no data about the chromosome or fitness available. We use this to play the game and log the best and worst chromosomes. Then we play chromosome_evaluator to get optimum score while still recording the best and the worst chromosomes. Thus, the score gets increase as we play.

## Literature Review - Hanabi Game:

Hanabi is a Team Game. You must work towards a common goal of placing cards to the centre of the table, so they appear in sets of matching colours (1 white, 1red, 1 blue, 1 yellow, 1 green) each with ascending numeric values, (1, 2, 3, 4, 5). The coloured card sequence represents fireworks being launched. In Hanabi game concept Red -1-2-3-4-5 is read firework being launched.

Each player is dealt with 5 cards. Each player cannot see their own cards but can see all of their team-mates' cards, since all cards in a player's hand are displayed facing away from the owning player. You can't see the values or colours of the cards in your hand. Your team-mates can see them properly. Players take turns to play cards into the centre, in the hope that the card they play will be valid (e.g. a red 2 is only valid to be played if it can go on top of a red 1 already on the table). So, players are playing moves blindly, i.e., without seeing what cards they are playing! The team has 3 lives (indicated by 3 red tokens in the game). Hence if 3 invalid cards are played in total, by the whole team, then the game ends for all players immediately.

But the players take turns to communicate, specific details of the cards their teammates are holding. You are only allowed to give very limited amount of information. There are 8

information tokens overall at the beginning of the game. Giving information, costs these tokens. Information token can be gained by discarding cards in hand. Whenever a player discards a card, it is replaced with one from the deck.

When the deck is empty or when all the lives are lost or when all fireworks are correctly launched up, the game ends. Score is calculated by adding the number of cards correctly played at the centre. The maximum score you can get is 25.

The deck consists of 50 cards. 3 cards of value 1, 2 cars of value 2, 3, and 4and 1 card of value 5, for each of the colours, colours white, red, blue, yellow and green. 2 types of information can be given using the information tokens – The colour or The Value. Information must be complete and should point to the correct cards i.e., if the player has 2 green cards the informer cannot point to just 1 card and give the information. You cannot give negative information i.e.; you can only give information what a card is and not what is not.

## Experimental Study:

I have added some rules of my own to make the Hanabi bot score higher.

**Rule 7** – Play a card with both rank and colour and is playable. This is a safe move and can be played anytime.

**Rule 8** – Play a card when there is a hint about the Rank and the firework requires a card of that rank. This is kind of a gamble, but we go through with this rule only when the life tokens are still at maximum and when we run out of information tokens. So, it is still a safe rule.

**Rule 9** – Discard an unplayable card. This is done when the information token is 0 and you know that the card is totally unplayable. I have improved the function **filter_card_list_by_unplayable.** So, if all the 4's of a colour is discarded, then there is no point in keeping a 5 of that colour, because it can never be played. Similarly, for other ranks. So here we go through the 5 different colours for each card in the discard pile. We have a counter array for each colour. The array has 4 elements if we find a card of rank 0 of a particular colour in the discard pile the count [0] element gets incremented and if we find a card of rank = 1 of a certain colour in the discard pile count [1] gets incremented and so on. Then in the end we check if count [0] == 3. This means that all the rank 0 cards of a particular colour are in the discard pile so there is no point in keeping any higher rank cards as they are unplayable. If count [1], count [2], count [3] == 2 then there is no pint in keeping cards of rank 2,3,4 or 3,4 or 4 respectively. So, these cards are appended to the **unplayable_cards**. Then the count array is re-initialized for the next colour. Also, if the rank of card of a particular colour is lower than card of that colour in the firework, it goes to the **unplayable_cards** list.

In this rule we go through the list of unplayable cards and if the hint of colour and rank both point to a card in the list, we discard it. This is pretty much a safe move by itself, and we don't need to check if the **information_tokens** is 0 or do it as a last resort.

**Rule 10** – Discard a card of unplayable colour. To do this I have added a new function **filter_unplayable_color**. In this function we create a full Hanabi deck and set **result** to it. Then we a have a dictionary **playable_colors.** The keys are the 5 colours and initially all the

values are False. Then we go through all the cards in discard pile and remove them from **result.** Then we go through all the cards in the firework and remove cards from **result** that are already in the firework. Now we go through the remaining cards and if the rank of the card is greater than or equal to card needed for the firework, we set the value of that colour in **playable_colors** to True. Finally, we go through the keys in **playable_colors** and if any of the color still has value False we add that colour to the list **unplayable_colors.**

Now if there is any card in hand with a hint of colour equal to a colour from the list **unplayable_colors**, we can discard it. This is again a safe move by itself, and we don't need to check if the **information_tokens** is 0 or do it as a last resort.

**Rule 11 –** Give a player without any clues a clue about a playable card. Here we go through each player and increment a counter variable called hints if there is a hint about colour or rank. Then we check if hints for any of the player is 0 if any player has 0 hints, we randomly give a hint about either colour or rank of a playable card in his hand.

I have tried to implement stochastic hill climber and MCTS to select the chromosome for improved score. We start with a **chromosome = [1, 11, 2, 7, 5, 8, 6]** which is one of the chromosomes that gives highest score **16.08**. I have created a new python file **chromosome_logger.py** and 3 new txt files to log the outputs. **logger.txt, best_chromosome.logger.txt and worst_chromosme_logger.txt**

**logger.txt –** Here the following are logged in the same order

- best Chromosome – the best chromosome
- up – an array upper limits of range within which a particular path (delete, insert, chose a new chromosome, take splices of 2 chromosome and add them together or swap the position of elements within a chromosomes) will be chosen.
- ignored – an array of number of times a path has been ignored.
- Best Fitness – This is the best score got so far. Not just in this iteration, in the lifetime.
- Total Game – Total games played in the lifetime.
- Times scored greater than 13 – Total times score was greater than 13 in the lifetime.
- Percentage of games score was higher than 13 – Total percentage of games where score was higher than 13.

**best_chromosome_logger.txt –** Here we log the chromosomes used when score was higher than 13

**worst_chromosome_logger.txt –** Here we log the chromosomes used when score was lower than 10

**How the code works**

We use **chromosome_logger.py** to log values into the 3 text files. This file must be run at-least once before running the **chromosome_evaluator.py.**

- So first we declare a variable **best_fitness = 0 and best_chromosme = None.**

- We open logger.txt in read mode. In the 1st run this is an empty file. However, we read from this file and save to variable **logger.**
- **Chromomse –** This has been set to **[1, 11, 2, 7, 5, 8, 6],** one of the best chromosomes gotten so far.
- **up –** this is the upper limit of various paths path (delete, insert, chose a new chromosome, take splices of 2 chromosome and add them together or swap the position of elements within a chromosomes) that can be chosen and is set to [10, 20, 30, 40, 50] in the initial run.
- **ignored -** this is the times a path has been ignored. It has been set to [0,0,0,0,0] for the initial run.
- **total_game = 0, times_scored_higher_than_13 = 0, percentage_score_higher_than_13 = 0 –** these are 0 for the initial run.
- **chromosmomes_with_score_higher_than_13 = [], chromosomes_with_score_lower_than_10 = []-** these are empty lists for the initial run.
- Now we check if **logger** is an empty string and if it is not then we have values for the above variables. For the first run logger is empty and we move on without executing anything in the if block.
- We clear everything in the logger.txt and close the file. This is not important in the first run as there is nothing to delete.
- Now we open **best_chromosome_logger.txt** and read from it to a variable **logged_best_chromosm**e. Then we check if **logged_best_chromosome** is empty string and in the first run it is. So, we don not execute anything in the if block. We close **best_chromosme_logger.txt.**
- We open **worst_chromosome_logger.txt** and read from it to a variable **logged_worst_chromosme**. Then we check if **logged_worst_chromosome** is empty string and in the first run it is. So, we don not execute anything in the if block. We close **worst_chromosme_logger.txt**.
- We print all the values from the logger.txt. This is just for reference.
- We have 1 more variable **path** which is set to 0
- Now we start the iteration. From range of 1 to at-least 500 to collect data.
- **total_games** is incremented by 1 at the beginning of each iteration. Then we check if **best_chromosm** is None and if it is not None we set **chromosome** to **bext_chromome.copy().** But for the 1st run best_chromosme is none.
- **generate_random_number (lower_lmt, upper_lmt) -** this is a function which generates a random integer between 2 limits.
- We call this function with values (1, up [4]) to generate a random number between (1, 50) in the first run. up [4] is 50 for the first run. The random number is saved in a variable **random_number**
- We chose a path depending upon the **random_number**. If the **random_number** is between **1** and **up [0]** (**10** in first run) we chose to delete an element from the chromosome.
  If it is between **up [0] + 1** and **up [1]** (**11 to 20** in the first run) we chose to insert an element to the chromosome.

If it is between **up [1] +1** and **up [2]** (**21 to 30** in the first run) we chose, to clear the chromosome and chose a new chromosome.
If it is between **up [2] + 1** and **up [3]** (**31 to 40** in the first run) we chose to create a new chromosome and splice it with the older chromosome.
If it is between **up [3]** + **1** and **up [4]** (**41 to 50** in the first run) we chose to swap the position of 2 elements in the chromosome.

- **Delete –** We use the **generate_random_number (0, len(chromosme)-1)** to set a random position. The element in that position of the chromosome is popped out. We set the **path** variable to 1.
- **Insert –** We use **generate_random_number (0, len(chromosme)-1)** to select a position and select a chromosome to be inserted at that position.
  Here the chromosome we want to insert must be between **0 and 11** because that is the range of rules (0-11), and it must be a value not already in the **chromosome** list. We need to insert new element only after checking that the chromosome list is not already the full set of rules from 0-11. We set the **path** variable to 2.
- **New Chromosome –** Here we clear the chromosome. The we select a length for the length of new chromosome to be created. I have found that length between 4-8 to be ideal. Then we randomly append the chromosome in the range (0,11). As we do this, we need to make sure that we exclude the values already in the chromosome from being selected again. We set **path** to 3.

```
chromosome.append(random.choice([int(x) for x in range(0,12) if x not in chromosome]))
```

*Fig 1*

- **Splice –** Here we do the same step we used to create a new chromosome. Here however we need to check that the value we are entering is not in **new_chromosome** as well as **chromosome**. Also, we need to make sure that the range does not run out of values this could happen as the length of **new_chromosome** is still set to **4-8**. So, as we append values to **new_chromosome** excluding the values already in **new_chromosome** and **chromosome** we could make a full list of values from **0-11**. In this case we may run out of values to append to **new_chromosome.**
  So, what we do is append values to **new_chromosome** as normal. Then we subtract **chromosome** from **new_chromosome** and add the values that remain to a new variable **chrom**. Then we append **chromosome** with values in **chrom**. We set path to 4.
- **Swap –** We generate 2 random positions and swap the elements of those positions with ach other. We set path to 5.
- If 5 or 6 are not in the chromosome list, we append them.
- Now we call the run method pass the chromosome and get the **result**.
- If result is greater than 13, we increment **times_scored_higher_than_13** by 1 and add the chromosome to the list **chromosmomes_with_score_higher_than_13**
- If result is less than 10, we add the chromosome to the list **chromosmomes_with_score_lower_than_10**
- If **result** is greater than **best_fitness**, we set **best_fitness** to **result** and **best_chromosome** to **chromosome.**
- **Increasing Probability of the rewarding path:**
- Currently we have **Delete** working from 1-10, **Insert** from 11-20 and so on. When we find that **delete** gets a score greater than 13, we try to increase the range where

**delete** so that it has more probability of being selected. So, we go through range of **numbers 1-5**. When the **path** is equal to the **number**, we increment the **up [number - 1]** by **2**. This is because **path** is from **1-5** and **up** being an array has index **0 to 4**. We also need to check that **2** elements of **up** does not collide or overlap. So, we do the increment only under these conditions i.e., **up [ number - 1] + 2 < up[number].** The last element of up however does not have this limitation.

- **Decreasing probability of punishing paths:**
- Quiet opposite to increasing probability of rewarding path, we try to narrow down the range of paths that give result less than 10. If the path gives score lower than 10 then we decrement the upper limit of that path by 2. Here again we need to make sure that the limits don't collide or overlap. So, we only do the decrement when **up[number-1]-2 > up[number]**. We don't need to implement this limitation on **up [0]** as there is nothing it would collide with.
- **Increasing probability of ignored paths:**
- First, we go through all the paths and increment the ignored array elements for all the paths except the path that was chosen. **Path=1** corresponds to **ignored [0]**, **path 2** to **ignored [1]** and so on.
- The we go through the elements of ignored and if any element has been ignored for 10 times in a row, we increase the upper limit of that path by 1. Now **up [0] corresponds to ignored [0] corresponds to path 1. up [ 1] corresponds to ignored [1] corresponds to path 2 etc.** Here again we need to make sure that the upper limits don't collide. So, we do the increment only when **up [number] < up [number+1] + 1.** Here too the last element of **up** doesn't have this limitation.
- Now we start logging the values.
- We open **logger.txt** in **r+** mode. First, we iterate through the elements of the **best_chromosome** and write them to the file each element separated by **',**'. When we reach the end of the list, we add a '|' instead of a **',**'.
- Then we go through elements of **up** and **ignored** and do the same.
- By now we would have written something like this to the **logger.txt** file **1, 11, 7, 2, 5, 8, 6| 690, 749, 752, 758, 776| 1375, 6613, 7060, 7035, 6637|**
- Then we also log **best_fitness, total_games, times_scored_higher_13, (times_scored_higher_13/total_games) * 100**, all separated by '**|**'.

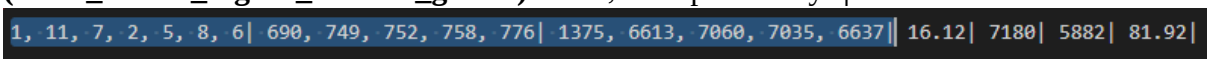


*Fig 2*

- We close the **logger.txt** file.
- We open **best_chromosome_logger.txt** file in **r+** mode.
- We go through the list of lists **chromosmomes_with_score_higher_than_13** and add each **chromosome** to the file. Each element of the **chromosome** is separated by **',**' and each chromosome is separated by '**|**'
- We close the file **best_chromosome_logger.txt**.
- Now we open file **worst_chromosome_logger.txt** file in **r+** mode do the same using the list of lists **chromosmomes_with_score_lower_than_10**
- **When we are not doing the 1st run:**
- Now logger.txt has some values and **logger** variable is not going o be an empty string. So, we first split the string by '**|**' and save it in **logged_values**. We can further split

the **logged_values [0], logged_values [1]** and **logged_values [2]** by **‘,’**. These are **chromose**, **up** and **ignored.**

- O we clear the 3 lists, **chromosme, up, and ignored** and then append these lists from values from **logged_values [0], logged_values [1]** and **logged_values [2]**
- We set **best_chromosme** to **chromosome.copy()**
- We set **best_fitness to logged_values [3], total_games to logged_values [4], total_scored_higher_13 to logged_values [5] and percent_scored_higher_than_13 to logged_values [6]**
- Then we clear the file and close it as before.
- We open **best_chromosome_logger.txt** and append the chromosomes as a list to the list **chromosmomes_with_score_higher_than_13.** We close **best_chromosome_logger.txt.**
- We open **worst_chromosome_logger.txt** and append the chromosomes as a list to the list **chromosmomes_with_score_lower_than_10.** We close **worst_chromosome_logger.txt.**
- Now during iteration if the number of chromosomes in the list **chromosmomes_with_score_higher_than_13** is greater than 100. We have a check if the current chromosome is a subset of one of the chromosomes in **chromosmomes_with_score_higher_than_13.** If not, we replace it with something in the list **chromosmomes_with_score_higher_than_13.**
- We also check if the chromosome is equal to one of the chromosomes in the list **chromosmomes_with_score_lower_than_10** and if so, we again replace it with one of the chromosomes in **chromosmomes_with_score_higher_than_13.**
- In the **chromosome_logger.py** this step is done before the path is chosen. So, we have more probability of finding new chromosomes.

In **chromosme_evaluator.py** we try to use the logged values from **chromosme_logger** and try to find a chromosome with high score. However, we log the values that we get here too. I think this would help improve the model.

- Difference between **chromosme_evaluator and chromosome_logger:**
- When we have path = 3 or when we need to choose a new chromosome. We do jot randomly select a new one. Instead, we can select a chromosome from the list **chromosmomes_with_score_higher_than_13.**
- When we have path = 4 or when we need to splice 2 chromosomes together. We do not splice the original chromosome with a new chromosome of randomly selected elements. Instead, we select 2 chromosomes from the list **chromosmomes_with_score_higher_than_13** and splice them together. I have set the maximum length of chromosome to be 8. So, I pop out elements from the final chromosome from random positions to assert length to be less than or equal to 8.
- Once we have gone through one of the 5 paths. If the chromosome is not a subset of one of the chromosomes in **chromosmomes_with_score_higher_than_13** or is in the list **chromosmomes_with_score_lower_than_10.** We replace it with one of the chromosomes in **chromosmomes_with_score_higher_than_13.** Here this step is done in the end because we are more focussed on getting a good score than logging values.

**Performance in Chromosome Tester –** For this I had to change the name of the class **RuleAgentChromosome** to **MyAgent and** I have set the chromosome to **[1, 11, 7, 2, 5, 8, 6].** The Score got was **15.7/25**

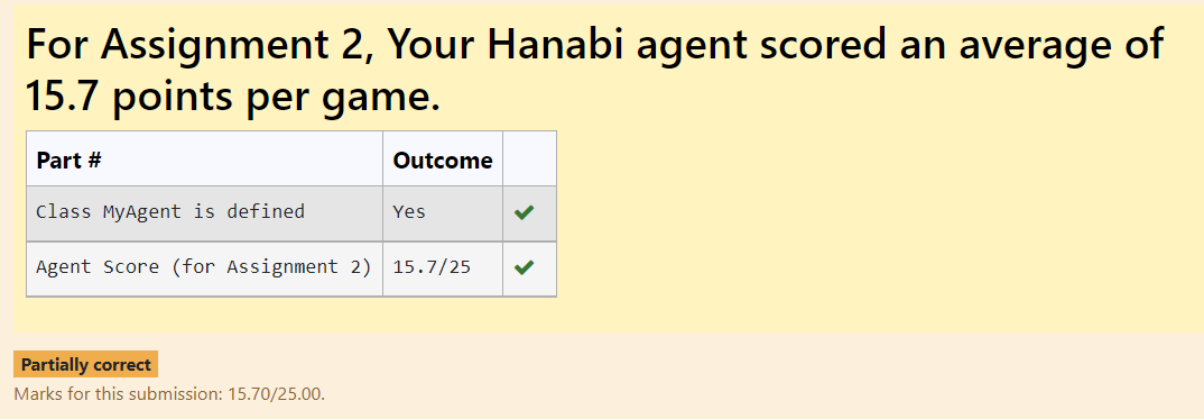


*Fig 3*

**Scatter Plot Showing Learning Rate –** I have added a new text file **fitness.txt.** This file is written by fitness which is a list of scores on each iteration. We use the data from this file to plot graph of score against the number of games played. The number of games in which the score is higher than 13 has increased over time as we trained the bot by playing more games.

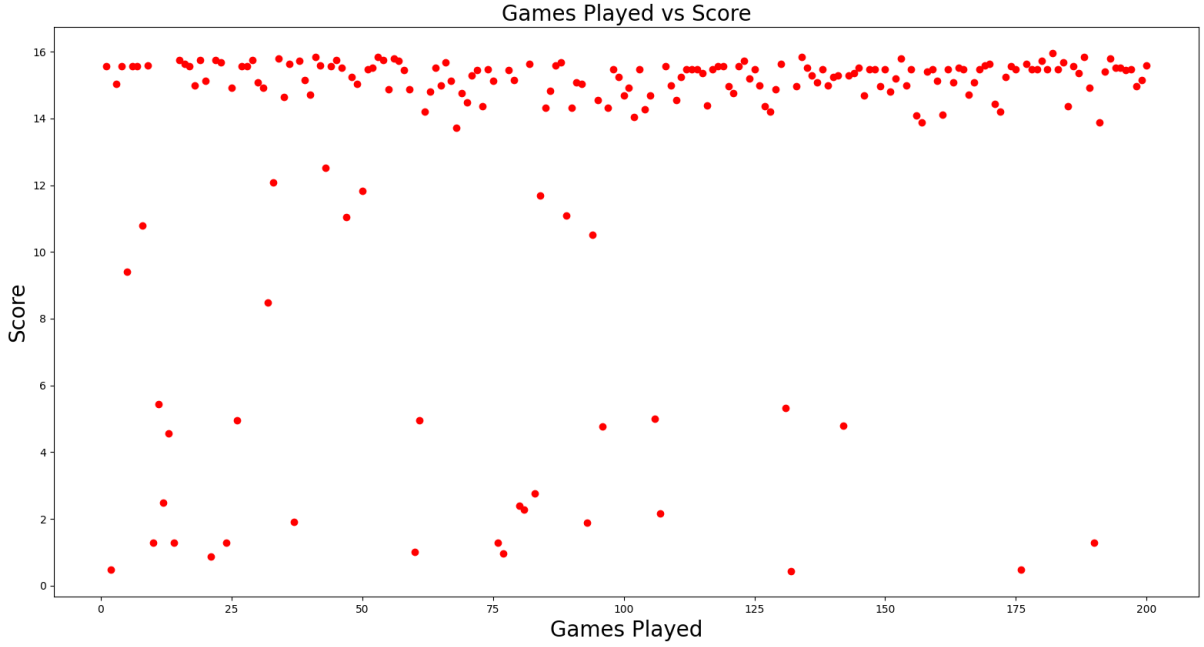


*Fig 4 – 1st 200 run (Chromosome_logger)*

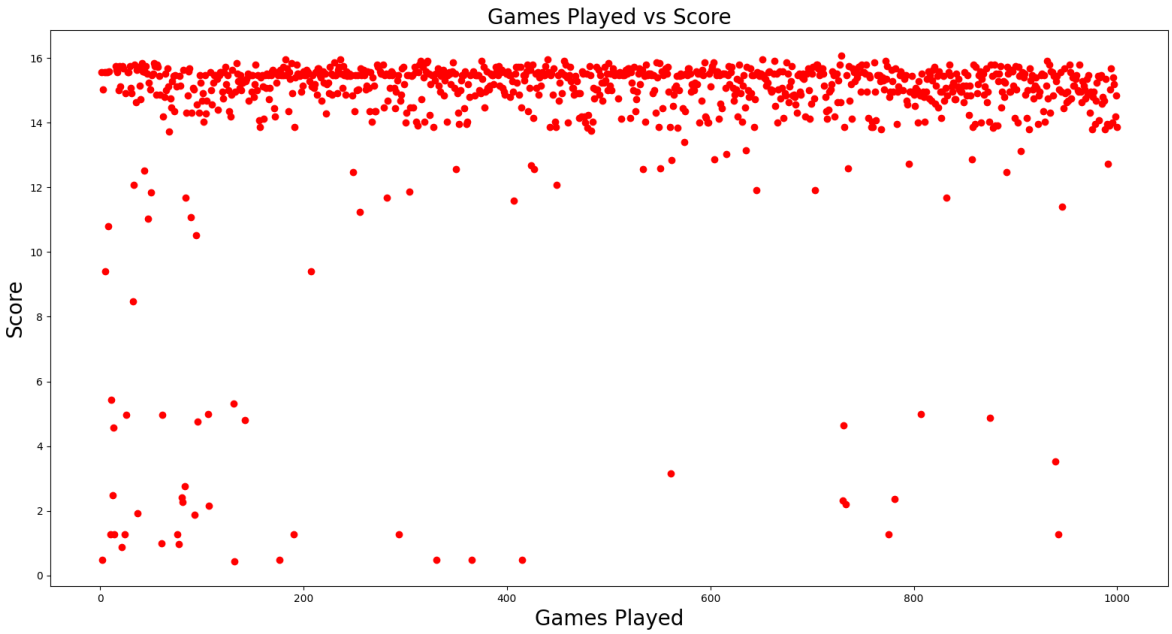


*Fig 5 – 1st 1000 run (200-1000 Chromosome_evaluator)*

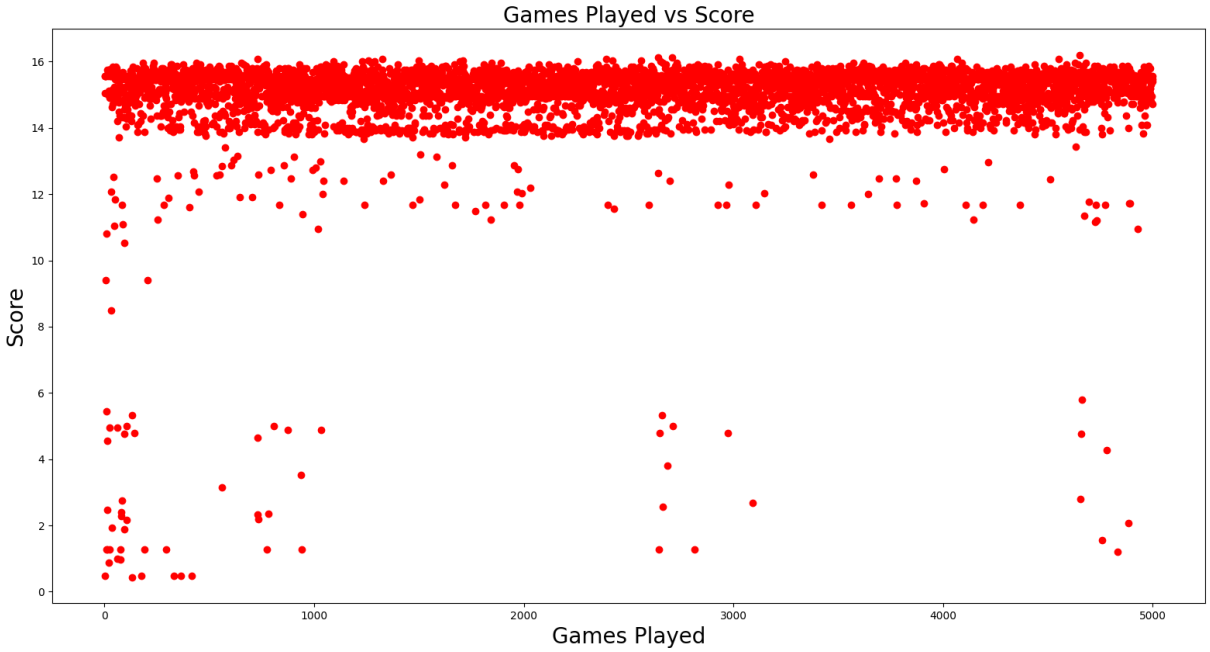


*Fig 6 – 1st 5000 run (200-5000 Chromosome_evaluator)*

# Techniques Implemented:

- **Stochastic Hill-climbers:** Stochastic Hill Climber is often called the Random Mutation Hill Climber because it randomly selects a way to mutate the chromosome. Delete, Insert or Swap.

  **Initialization** – Randomly initialize a value within the search space. We are not doing this step now because have already found a chromosome which give the best fitness so far. But initially random chromosomes are created within search space.

  **Mutation** – We have 5 different paths to choose 3 of which are mutations of the chromosome. Insert, delete and swap.

**Evaluation** – We evaluate the fitness by comparing best score to current score.

**Replacement** – We replace the chromosome with the chromosome that gives best score and replace current best score with score if score is greater than best score.

- **Population Based GA –** We have implemented a crossover of SHC with population-based GA by adding 2 new paths to mutate towards. Selecting a chromosome from best chromosomes and splicing up 2 best chromosomes together.

  **Elitism –** We find chromosomes which give score greater than 13 and add them to a list these are elite.

  **Splice –** We take chunks from 2 elite chromosomes to create a new better chromosome.

- **Monte Carlo Tree Search –**

  **Tree Selection -** The tree is expanded in regions where ucb1 algorithm looks more promising. The selection process keeps selecting the most promising node in main tree. We have implemented this by increasing the probability of selecting a path that was promising earlier.

  **Back propagation –** We are updating the best fitness each time we get a score greater than the current best score. We are updating the best chromosome as the chromosome corresponding to the best score.

  **Expansion –** We are selecting the best chromosome and exploring new values by undergoing one of the 5 mutation paths available.

  Once a path is ignored for too long the probability of choosing that path is increased.

- We have also implemented new rules to try improved the score. We have improved the list of unplayable cards and added a new strategy to discard unplayable coloured cards. We have also implemented a rule to give clue about a playable card to a player who doesn't have any clues.

# Analysis:

**Mutation and Replacement** is a good feature of SHC which helps in find new possible chromosomes which might give a better score. We also replace the best score with score if the current score is higher than best score.

**Elitism and Splicing** are 2 good features of Population based GA, which allows to retain chromosomes which has previously given good scores and create a new chromosome by taking elements of 2 elite chromosomes.

**Selection and Expansion** are 2 good features of MCTS. It is good to explore more in a node which has previously given good results. We do this by increasing the probability of a node being chosen. We also reduce the probability of a node being chosen when it has previously given bad scores. We increase the probability when a node has been ignored for too long.

I have also added some rules which made the bot better.

We can discard an unplayable card when information token is 0. Instead of just checking into cards in the firework. I have also improved the method by making a higher rank card unplayable when cards of a lower rank of the same colour are all in the discard pile.

We can discard cards when it is of an unplayable colour, and we have hint about colour of the card. I have added a method to know what unplayable colours are.

We can give clues to a player who has a playable card and is out of clues.

We can obviously play a card which has both rank and colour and is playable.

We can play a card when we have the rank and one of the colours in the firework needs that rank.

## Overall Conclusions and Future Scope:

Taking some elements of SHC, Population based GA and MCTS has helped improve the score of the Hanabi bot. Also, adding some obvious rules has helped in improving the score.

Logging the values and using the data to influence new generation of chromosomes has helped in consistently getting a score around 15 in the Hanabi game.

I think playing the game more improves the score as we are logging the best chromosomes. The bot can be improved by addition of more rules like providing hints about an unplayable card. We can also include splicing of 3 or even more best chromosomes if there are enough rules.

## References:

https://moodle.essex.ac.uk/pluginfile.php/1393836/mod_resource/content/4/02b_genetic_algorithms.pdf

https://moodle.essex.ac.uk/pluginfile.php/1407583/mod_resource/content/9/05_game_theory_and_tree_search.pdf

https://moodle.essex.ac.uk/mod/book/view.php?id=86957