Department of Computer Science
Ben-Gurion University of the Negev

Evolutionary Computing

Spring 2016

Prof. Moshe Sipper

Evolving Agents for "Bomberman" Project Report

Yuval Gelbard

Gal Bar-On

Table of Contents:

1. Introduction:

1.1. Genetic Programming:

"How can computers learn to solve problems without being explicitly programmed? In other words, how can computers be made to do what is needed to be done, without being told exactly how to do it?"

- Attributed to Arthur Samuel (1959)

Artificial Intelligence may provide a solution to the problem described in Arthur Samuel's question. When choosing an AI solution for a given problem, we are looking for the most general solution which can solve as many types of problems as possible, and is able to generalize whenever it encounters a yet-to-be-seen instance of a problem. We are looking for an adaptable solution- an algorithm or a program which is capable of changing itself until it reaches a suitable answer to the question we are investigating. Hence, designing a natural selection based, evolving AI, may give us a good base to such a solution. The field of Evolutionary Algorithms and Evolutionary Computing has been investigated since the late 1970's and is still being investigated today.

One area of Evolutionary Computing is Genetic Programming. That is, developing new algorithms and programs using an evolutionary process. A programs code is coded as a genome, usually using a variation of LISP trees, and is evaluated using problem instances as an input. The code's output and its relation with a pre-defined 'good' solution to the problem are then calculated to obtain fitness. LISP trees can also go through necessary genetic operators, such as Mutation and Crossover. A fitness based selection method is chosen and the evolution can begin.

Genetic Program has a very big part in game solving algorithms. Games such as Chess, Checkers, Tic-Tac-Toe and many others. Games intrigue the human mind and the effort to create a human-level AI still inspires a lot of research. Game research also addresses important cognitive functions of the human mind and contributes to research in many other fields.

Our project focuses on attempting to develop a GP based AI that plays a game and gains experience against a diverse set of players with different behaviors, thus making it capable of competing against any kind of player.

The project's final source code can be found in the following GitHub Repository:

https://github.com/Juvygelbard/BombermanAI

Further details about the implementation will be given later on.

1.2. Bomberman:

Bomberman is a strategic, maze-based video game franchise originally developed by 'Hudson Soft'. It was first published in 1983 and new games have been published at irregular intervals ever since. Bomberman has featured in over 70 different games on numerous platforms. The Bomberman franchise is one of the most commercially successful of all time.

The general goal throughout the series is to complete the levels by strategically placing bombs in order to kill enemies and destroy obstacles. Most Bomberman games also feature a multiplayer mode, where other Bomberman act as opponents and the last one standing is the winner. Although most games in the Bomberman series use the same type of maze-based levels established by the original game, some are Zelda-like adventure games, Mario-like platformers, Tetris-like puzzle games, and kart racers.

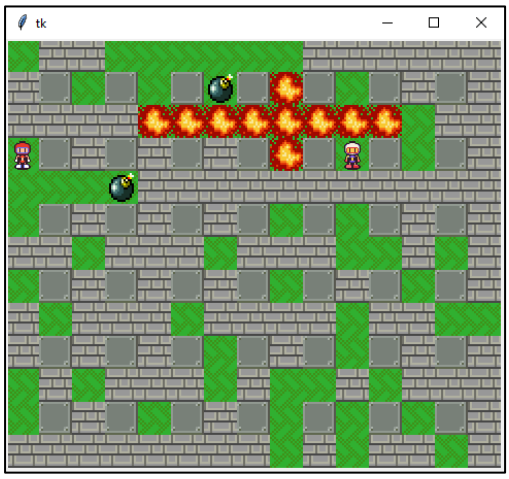
We, however, will be implementing a simple version of Bomberman. Since we didn't want to waste time developing a new version of our own to work with, we decided to search the web for an existing one. We found an announcement about a Summer Programming Competition in the University of Warwick, United Kingdom, to write an AI for Bomberman. The staff provided a server that allows AI players to register and play Bomberman against each other, communicating with it according to a given protocol.

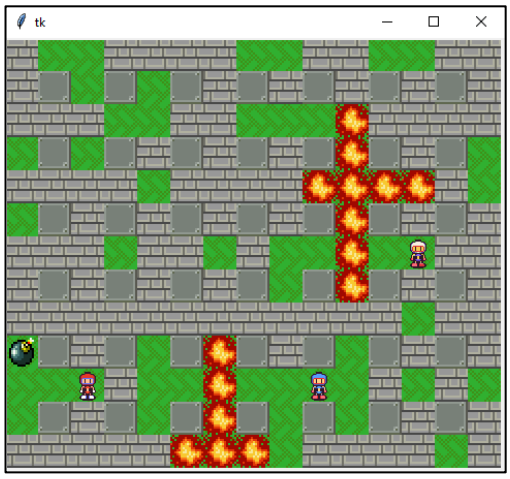
In this version of Bomberman there are no levels, just randomly generated maps and competition against other players in those maps. Here, enemies cannot be injured. If one is hit by a bomb, it is killed and eliminated from the game (even if one is hit by his own bomb). There are no power-ups, so the blast radius is constant and a player can place as many bombs as he pleases. Also, here bombs don't set off other nearby bombs.

As a real time game, Bomberman is based on ticks. In each tick, each player gets the current map state and can choose to move in one of four different direction or place a bomb. Walking on bombs isn't possible (though moving off a bomb after placing it is a valid move), same as walking through destructible or solid terrain. We chose the bomb delay to be 5 ticks and the blast radius to be 3 tiles long. At most 5 players can play the game simultaneously. When a player dies, the other active players gain one point (doesn't matter which one killed it). The winning player is the last one on the board and will have the highest score.

We should also mention that a single game can last at most 1000 ticks or until only one player remains in the game.

Since the package we found didn't have a Graphical User Interface (GUI), but a textual display of the game state at each tick, we decided to make a GUI of our own to make it easier for us to follow the evolutionary process. We matched to numerical values in the array representing the map to original images from the game, as simple as possible.





2. Bomberman – GP style:

This is the core of our project. An AI agent, specifically trained to play Bomberman.
In order to easily create and modify agents, we've created a wrapper for the agent's 'brain' – the *Agent* interface. This approach allows us to both develop sample agents which work according to constant, pre-defined heuristics, in order for our GP agent to fight diverse agents with different behaviors; and to develop the GP agent itself. The *Agent* interface implementations are then wrapped by the *Bot* class, which handles communication with the server. When the game server sends a tick message to the players, the *Bot* class sends its agent the current map state and asks it for its next move.

In order for our AI to train itself properly, we enabled a variety of players to play the game:

- Provided players (Such as the winner of the UWCS competition)
- A Greedy BFS player (further details later on)
- A Random player
- A Human player (mainly for testing)
- Our AI player

2.1. The greedy player:

Our 'Greedy Agent' works according to the following heuristics, given the current map, positions of all players and bombs status:

1. If I'm in range of the explosion of a currently ticking bomb - find closest safe place (BFS) and make a run for it.
2. If I'm in range of exactly one tile from an enemy player- place a bomb.
3. If there is an enemy player which I can walk to without removing any walls - walk towards it.
4. Walk towards closest enemy player. If ran into a wall, place a bomb.

Having only the agent from UWCS, which was too good in the beginning, we needed a simpler player that relies on constant heuristics for our AI to compete against. This way we could allow the evolution to start off without our agent getting 'crushed' by other players from the very beginning. The final product is an intermediate-level agent, which usually loses to the IBM contest winner, but still gives a good fight to our GP agent.

2.2. The Random Player:

This player makes one of the six possible moves each tick, with an equal probability. It was used mainly to compare its performance with our agent's performance after evolving it.

2.3. The Human Player:

The keyboard controlled human agent holds a dictionary of valid keys and a queue that stores moves to execute, one at each tick. The agent listens to the user's keyboard input, which can be one of the w,a,s,d keys to move or the spacebar to place a bomb.

2.4. Our Player:

To implement our Bomberman AI we took reference from the class material about GP and GP related games (Robocode, RARS etc.). We focused on real time, tick based games.

We decided to implement GP using decision trees made of 3 types of nodes (Which will be described later on): Terminals, Functions and Measures. Terminals can be either constants or random numbers, Functions can be arithmetic or logic functions and Measures are values which are re-calculated separately in the beginning of each tick and depend on the current game state. The Measures are calculated by simple non-evolutionary algorithms and some of them require a BFS scan of the map. When a tree is evaluated, it first replaces its measures with their corresponding values and then evaluates the Functions and their arguments recursively.

Our entire implementation is written in Python 3.5. We used python's 'tkinter' package to create the GUI for the game and also used the 'numpy' package for some calculations. Next, we will describe the different node types which were used to construct the decision trees. Specific nodes types will be further detailed.

2.5. Terminal Set:

{ConstantNum, RandomNum}

- ConstantNum: Returns a constant number in range $[0,9]$ when evaluated.
- RandomNum: Returns a random number in range $[0,9]$ when evaluated.

2.6. Function Set:

{Add(2), Sub(2), Mul(2), Div(2), Min(2), Max(2), Abs(1), Neg(1), If_A_ge_B(4), Compare(2)}

- Add, Sub, Mul, Div: Simple arithmetic operators with 2 operands.
- Min, Max: Return the minimum/maximum of the two operands, respectively.

- Abs: Returns the absolute value of its given operand.
- Neg: returns the negative value of its given operand.
- If_A_ge_B: checks if the first child is greater or equal to the second child. Returns the value of the third child if so and the value of the fourth one otherwise.
- Compare: compares between its two children. Returns 1 if the first child is greater than the other, -1 if the other is greater and 0 if both are equal.

2.7. Measure Set:

{CanMove_UP, CanMove_DN , CanMove_LT , CanMove_RT, EnemyDist_UP, EnemyDist,DN, EnemyDist_LT, EnemyDist_RT, InDanger_UP, InDanger_DN, InDanger_LT, InDanger_RT, EnemyInRange, NearTurn_UP, NearTurn_DN, NearTurn_LT, NearTurn_RT}

- CanMove: indicates if the individual can move in four different directions (UP, DOWN, LEFT or RIGHT).
- EnemyDist: represents the distance from the closest enemy in four different directions (UP, DOWN, LEFT or RIGHT). The closest enemy is the one with the shortest distance from the individual with minimum separating walls.
- InDanger: indicates if the individual is in danger (close to a bomb) in four different directions (UP, DOWN, LEFT or RIGHT).
- EnemyInRange: indicates if the enemy is in range for the individual to place a bomb.
- NearTurn: represents the distance from the closest turn if moving in four different directions (UP, DOWN, LEFT or RIGHT).

2.8. Main Actions:

The main actions are performed by sending a message to the server with the action the player wishes to make, in the form of "Action <action>", where 'action' is an element of:

$$PossibleMoves = \{"UP","DOWN","LEFT","RIGHT","BOMB","NONE"\}$$

(We made NONE a possible choice in case it is the better move to make at a given moment).

2.9. Fitness Function:

The fitness function is based on the game score of the player, as provided by the server at the end of the game. As mentioned above, each player gets a point when another player is killed. This way, a player can gain points based on how long it survived in comparison to other players (i.e. getting killed doesn't necessarily mean 'losing' the game; the score can still be higher than 0). To allow a different number of players in each game, we will have to normalize the score. We used the following formula:

$$PlayerGameScore = \frac{PlayerPoints}{NumOfPlayers - 1}$$

When running the game using $PlayerGameScore$ as the fitness measure, we've noticed that for some games, players gained a very high score just because they were too far away from the other plays for the entirety of the game, thus surviving most of it by doing nothing and still getting a high score. To avoid such a bias, we decided to run a constant number of games for each individual and get the average score. This leads us to the following formula:

$$MyFitness = \frac{1}{NumOfGames} \times \sum_{i=1}^{NumOfGames} PlayerGameScore(i)$$

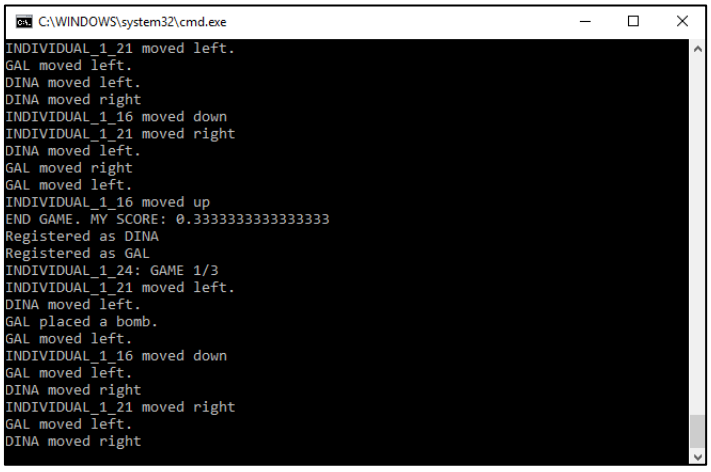
3. GP developmental process:

3.1. Building the GP Infrastructure:

Although we found python libraries that implement GP and could help us with our project, we decided to go with an implementation of our own that matches the specific requirements for our project.

We wrote a *Node* class that can be evaluated according to its content (the different kinds were described earlier). All terminals (including measures) and functions implement the *Node* class in their own unique way and have their own evaluation method. We wrapped the *Node* class with the *Tree* class, which has the *perform_crossover()* and *perform_mutation()* methods described later on. After testing the Tree class by creating trees with random roots and performing genetic operations on them, the *Tree* class was finally ready to be wrapped by the new *Genome* class.

The *Genome* object holds a set of 6 trees, one for each possible move. It has the *next_move()* method which evaluates all trees in the tree set (given the current measure values) and returns a list of the evaluated scores, where the first element belongs to the tree with the highest score.



Finally, we created the *gp_evolution* module, in which all the action happens. It creates the initial population, creates the genomes and assigns them to players and starts running the evolution process for the given number of generation. The data of each generation is saved in a file in case we want to continue or restart the evolution from whatever generation we stopped at.

3.2. Elitism:

To make the evolution process a bit faster, we chose a strategy of elitism. In other words, in the end of each generation, we kept the best n individuals and their genomes unchanged when moving to the next one. This allowed us to preserve good traits and create a better population in future generations. However, when choosing the remaining $popSize - n$ individuals for the next generation's population, offspring can be created by any pair of individuals from the previous population, even by those chosen by elitism.

3.3. The *perform_mutation()* Method:

(Defined in class *Tree* and extended by class *Genome*)
At the end of each generation, all the individuals that weren't selected by elitism go through mutation. For these individuals, each of their genome's move trees go through mutation with a given probability. For each tree, a new tree is created from the same set of terminals and functions and is considered the mutated gene. Then the algorithm iterates over the given tree and randomly chooses the node to be replaced by the new gene.

3.4. The *perform_crossover()* Method:

(Defined in class *Tree* and extended by class *Genome*)
Same as mutation, for every two individuals not selected by elitism, every move tree of the first individual goes through crossover with the other individual's corresponding move tree with a given probability. This way, each move tree evolves separately. The algorithm selects a random node from Tree X and replaces it with a randomly selected node from tree Y. However, the root of the tree cannot be replaced by another node. We allowed that to prevent trees from completely changing due to crossover.

3.5. The *rank_selection()* Method:

(Defined in the *gp_evolution* module)
In order to prevent convergence to individuals with a relatively high score, i.e. prevent them from being chosen too frequently, we ranked all individuals from high to low according to their fitness function value. This way, there's a linear correlation between an individual's chance of being a parent and its position among the rest, where the latter is instead of its actual score.

3.6. Bloat Control:

When testing our *Tree* class, we realized that most of our randomly generated trees were either made of just one terminal node or just very small. This happened because we had a lot more terminals and measures than functions. Therefore, we needed to set a ratio between them so that functions get a higher probability of being chosen as root nodes, hence making us able to grow bigger trees.

However, bigger isn't always better. Since the trees change and grow all the time, and there was no limit, they could become enormous and take a lot of calculation time. We decided to set a maximum depth for our trees in order to control them and to prevent the "bloat" phenomenon; where trees grow without improving fitness. This is simply done by constructing a tree with an initial depth limit and passing this value to its children, along with a constant MAX_DEPTH.

We also made sure that whenever trees go through crossover or mutation, the product will not have a depth higher than MAX_DEPTH. When a node with a current depth of x is mutated, we generate a random tree part with a depth that, combined with the original tree depth, does not exceed MAX_DEPTH. We replace it with the previous node and update the depths accordingly. With crossover, however things are a bit more complex. We need to

make sure that when a child of one tree is replaced with a different child, the depth of their combination is valid. This is done by checking if the depth of the replaced node added to the height of the new child does not exceed MAX_DEPTH. If it does, a different random child is repeatedly chosen from the second tree until the combined depth is valid.

4. Experimental Results:

4.1. Improving Our Genetic Operators:

We tried running the evolutionary process with a population of 20 individuals for 30 generations with 3 games played by each individual per generation. We let each individual play more than one game because sometimes it wasn't really involved in the game and got a fake high score (as explained earlier). The tick length for this test was 300 milliseconds.

In the initial state, an individual could either go through mutation or crossover but not both. The mutation probability was 0.1 and the crossover probability was 1 - 0.1 = 0.9. Whenever two parents were chosen for crossover, only one randomly selected move tree would change (the random index is saved and the same move tree changes in both parents) out of all possible 6. Same goes for mutation, only this time individuals are chosen one by one. We noticed that each generation took a lot of time to calculate (around half an hour) with minor changes between generations, so we started thinking what could be done to accelerate the process.

Later, an individual would go through both mutation and crossover, each with a different probability (Mutation = 0.1, Crossover = 0.8). If chosen for crossover, all of its move trees would go crossover in pairs. Mutation changed in a way that each move tree of all individuals would mutate with a given probability. In this case the problem was that changes were too radical and good traits couldn't be kept for later generations.

Finally, we changed it to work as written in the *perform_crossover()* method description.

4.2. Adding and Modifying Game State Measures:

Our initial set of measures was:
{NearEnemy_CLR_UP, NearEnemy_CLR_DN , NearEnemy_CLR_LT , NearEnemy_CLR_RT, NearEnemy_WALL_UP, NearEnemy_WALL_DN, NearEnemy_WALL_LT, NearEnemy_WALL_RT, InDanger_UP, InDanger_DN, InDanger_LT, InDanger_RT, NearTurn_UP, NearTurn_DN, NearTurn_LT, NearTurn_RT}

In the beginning, we had two sets of measures which indicated an individual's distance from a near enemy. One was looking at a clear path without separating walls (NearEnemy_CLR) and the other took walls into account (NearEnemy_WALL). However, the latter did not consider the amount of walls standing between the individual and its enemy, thus giving a distance which isn't optimal. This led us to unite these sets into one set of measures indicating the distance from the nearest enemy with as little walls involved as possible (EnemyDist). The algorithm generates a map where at each stage, an inner layer of approachable walls is removed until an enemy is reachable. Then, the BFS result, which indicates the number of steps needed to reach that enemy, is returned. This measure required a long calculation time, which made us increase the tick length to 500 milliseconds.

In addition, we needed a measure that indicated if a move was possible. Since this was only checked when an attempt was made to perform that move, it couldn't be done in advance. Therefore a set of measures was added (CanMove) to check all directions to see if moving there is an option or not.

We also realized that even when an enemy was close, our agent didn't necessarily place a bomb to kill him. For instance, an enemy could be two steps away but hiding around a corner. A bomb wouldn't reach it in that case. We decided to add a measure that indicates whether an enemy is in range for the agent to place a bomb (EnemyInRange). This measure could also tell if one is in range for an enemy to place a bomb.

4.3. Improving runtime efficiency:

Since the average game length was 60 ticks, a tick length of 500 milliseconds would give us a 30 seconds long game. This sums up to 90 seconds of game playing for each individual per generation. For a population size of 30, a generation would last 45 minutes. Therefore we looked for a way to significantly improve our runtime. We achieved that by modifying our program to run 6 separate servers, which made the evolution process run 6 times faster. Each server had a different set of players playing on it to increase the variety of players. With the new settings, a generation lasts approximately 7.5 minutes. Eventually we decided to use 5 servers. Otherwise, calculating the values for more than 5 individuals at once would be too much for the computer to calculate in 500 milliseconds.

4.4. Running the evolution process:

4.4.1. Generations 1-100 (With the initial mutation and crossover settings):

At first the bot didn't do much since it got randomly generated move trees. It either stayed in the same place for an entire game, moved back and forth between two positions or killed itself. The initial configuration included 5 servers described as follows:

- Two servers running 2 greedy bots and one UWCS competition bot each.
- Two servers running 1 greedy bot and two UWCS competition bots.
- The last server was running only one UWCS competition bot.

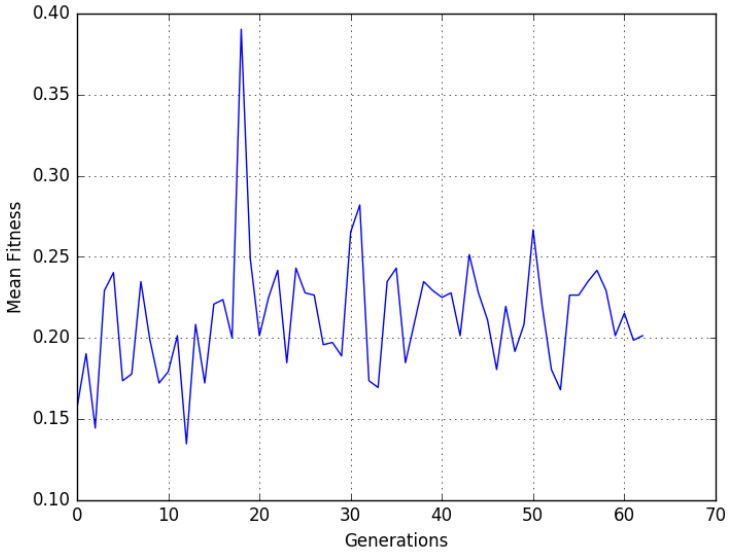
(The players described above are in addition to our agent in each server).

After about 30 generations we started seeing some changes in our agent's behavior. One agent followed an enemy once it was close enough, but then suddenly started running away. Another agent managed to avoid bombs but always ended up killing itself. In generations 31-100 there were only minor changes.

After analyzing the result of those generations we realized we should both change the measures and the genetic operator settings, both done as described earlier.

4.4.2. Re-running the process after the measure and genetic operator changes:

We used the same configuration as last time for the servers, only this time we used an improved set of measures and managed to fix a few bugs with their calculation methods. The population size was raised to 40 (which changed the proportion of the population chosen by elitism to 5%). After 62 generations, we created a graph showing the mean fitness of each generation's population (shown in the next page):

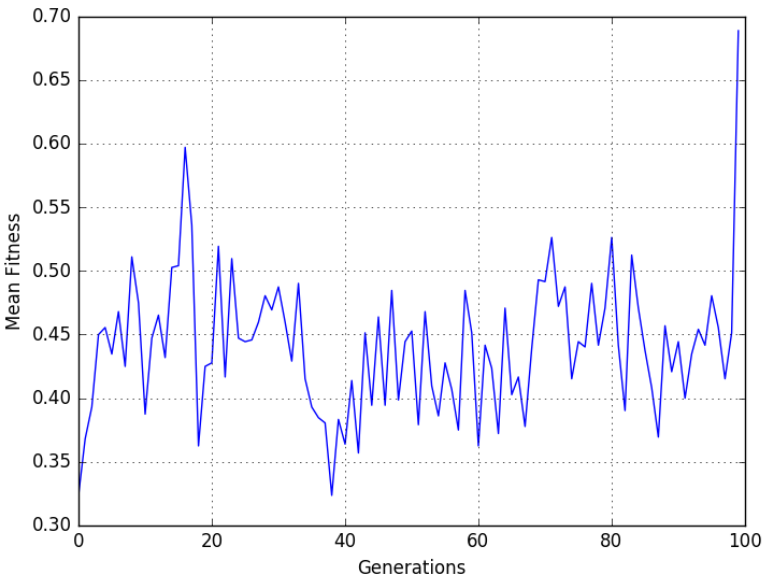


It is easy to see that there isn't a significant change between each generation, let alone an improvement. The only major improvement was a slight peak around generation 18. There was something that prevented the individuals from gradually increasing fitness through time, which later turned out to be the fact that some individuals were playing in servers including the UWCS competitor alone. That caused such individuals to lose their games and get 0 fitness immediately, even if their traits were better compared to other individuals. They were just not ready yet to face such a skilled opponent. These losses prevented them from carrying their possibly good traits to the next generation.

We tried making the following changes in the player settings:

- One server running two greedy bots and one UWCS bot.
- One server running one greedy bot and one UWCS bot.
- Three servers running one greedy bot each.

We got the following result (using the configuration described above):



Throughout the first 100 generations there were no major improvements to the mean generation fitness, besides a visible peak, again, around generation 18. There is another clear peak at exactly generation 100, which makes another future improvement possible. We tried running the three best individuals from generation 100 in a series of games and wrote down the mean scores of those individuals (though their performance wasn't as good as expected). Each row represents 50 games, played with a different set of players. We also ran a Random Agent to be used as a "control group" to our experiment.

Server Composition	Random Agent Mean score	Individual 1 mean score	Individual 2 mean score	Individual 3 mean score
1 Greedy Bot	0.08	0.52	0.4	0.6
1 UWCS Bot	0.02	0	0	0
1 Greedy Bot 1 UWCS Bot	0.03	0.37	0.27	0.27

Though its score was not the highest, individual number 3 showed interesting behavior and managed to avoid a few bombs before it was brutally killed by its enemies.

As an example of our evolution's result, the tree expression for individual 3's "DOWN" move tree is presented below:

*abs(compare((((min((((((min(CanMove_LT, InDanger_DN)-neg(CanMove_RT))-max(compare(EnemyDist_UP, InDanger_UP), min((EnemyDist_RT-EnemyDist_UP), NearTurn_DN))))/min(compare((EnemyDist_UP*7), (EnemyDist_UP-3)), (if (EnemyDist_DN>=abs(EnemyDist_DN)) then NearTurn_RT else NearTurn_RT)))-EnemyDist_LT), ((EnemyInRange-max((InDanger_UP/(EnemyDist_LT+InDanger_UP)), (InDanger_RT*(if (EnemyDist_LT>=NearTurn_RT) then EnemyDist_DN else abs(CanMove_DN)))))-CanMove_RT))-InDanger_RT)/InDanger_DN)+min(min((((EnemyDist_RT/NearTurn_RT)/EnemyDist_UP)-((InDanger_LT*NearTurn_DN)/NearTurn_DN)), ((((random/CanMove_RT)*InDanger_DN)-max((CanMove_UP/InDanger_DN), (InDanger_RT*(if ((CanMove_LT-EnemyDist_DN)>=NearTurn_UP) then EnemyDist_DN else abs(InDanger_RT)))))-CanMove_RT)), InDanger_LT)), ((InDanger_LT+NearTurn_UP)+min(min((((InDanger_LT*(InDanger_LT+(if (InDanger_RT>=NearTurn_UP) then CanMove_RT else CanMove_RT)))/EnemyDist_UP)-(NearTurn_DN/NearTurn_DN)), ((EnemyInRange-max((CanMove_UP/(2-InDanger_RT)), (InDanger_RT*(if ((max(0, InDanger_DN)+compare(InDanger_DN, NearTurn_DN))>=NearTurn_UP) then EnemyDist_DN else abs(abs(InDanger_LT))))))-CanMove_RT)), InDanger_LT))))*

5. Conclusion:

Our AI managed to evolve a bit and may have evolved even further if we had extra time to try different strategies and add more measures if needed. As the evolutionary process was relatively slow, we only managed to run about 100 generations in 24 hours which consumed a significant amount of our time. Technical issues such as calculation mistakes and concurrency issues would often get in our way and require much time to detect and fix.

We realized towards the end that our bot mainly spent time hiding in corners and watching other bots destroy each other, which made it develop a rather evasive strategy. It even managed to dodge a few bombs, but would never attempt to attack another player. If we had more time, we could have tried one of these strategies to force our bot to develop an offensive, less passive behavior:

- Developing a passive opponent, which would make our bot leave its comfort zone and attack its enemies.
- Evolving our bot using a co-evolution strategy, at least for the first few generations. This would allow our bot to compete against passive, yet not previously evolved bots which are already trained to avoid bombs and enemies.
- Adding more measures which focus on the position of possible enemies and on the possibility of attacking them.

Some other changes we could apply to accelerate the evolutionary process:

- Changing our evolutionary process in a way that each bot would compete against a few different sets of bots for each generation, as opposed to our current configuration, in which a bot competes 3 times against a fixed set of bots per generation.
- Our current configuration runs all the evolving bots on a single machine, while the servers and opponents can run on different machines. Since the bots of a single generation are independent of one another, we could run the bots on different machines with a single computer managing the process. By that we can significantly increase the amount of parallel games being played, and run more generations in a shorter period of time.

Finally, we've managed to get some interesting results and a bot which is much more successful than a random bot, in addition to the added value of improving our personal knowledge in the field of Evolutionary Computing. We strongly believe that with some added time, effort and resources we can eventually build a true Bomberman winner.

6. Bibliography:

- 'IBM Summer Programming Competition", Ed. Ruth King. Jul 29, 2012. University of Warwick Computing Society. Aug 9, 2012. <https://www.uwcs.co.uk/details/1583/>

- "UWCS/bomberman-progcomp", Aug 22, 2012. Github repository. <https://github.com/UWCS/bomberman-progcomp>

- "Bomberman" Wikipedia: The Free Encyclopedia. Wikimedia Foundation, Inc. 27 Dec 2003. Web. 26 Jun. 2016. <https://en.wikipedia.org/wiki/Bomberman>

- "Introduction to Genetic Programming – GECCO 2003", Ed. John R. Koza. Jul 13, 2003. Chicago IL, USA. Mar 29, 2016.

- "Burke 2003 Tutorial - Chapter 8 - A Genetic Programming Tutorial", Ed. John R. Koza and Riccardo Poli. 2003. Stanford University, Stanford CA. Mar 29, 2016. <http://www.genetic-programming.com/jkpdf /burke2003tutorial.pdf>

- "Attaining Human-Competitive Game Playing with Genetic Programming", Ed. Prof. Moshe Sipper. Ben-Gurion University, Israel. Mar 29, 2016. <http://www.cs.bgu.ac.il/~sipper/gpgames.ppt>