Monte-Carlo Tree Search Algorithm in Pac-Man

Identification of commonalities in 2D video games for realisation in AI (Artificial Intelligence)

Name: Salman Tilkidagi
Registration Number: 1908852
Degree Course: Computer Science
Supervisor Name: Dr John Woods

Acknowledgements

I would like to take a second to acknowledge Dr. John Woods who was the assigned supervisor for taking time out of his busy day to arrange Skype/zoom meetings, One to one meeting, and his constant support/advice throughout the research. I would also like to give a special thanks to Dr. Michael Gardner for his guidance throughout the research.

Abstract

Pac-Man is a well-known game that has become popular since the 1980s. There were many attempts to develop a strategy that could earn as many points as possible and to compete against humans (achieving similar scores and, in some instances, higher scores than humans). The research is dedicated to the game strategy, which uses the Monte-Carlo Tree Search algorithm for the Pac-Man agent. Two main strategies were heavily researched for Pac-Man's behaviour (Next Level priority) and HS (Highest Score priority). The Pacman game best known as STPacman is a 2D maze game which will allow users to play the game using artificial intelligence and smart features such as, Panic buttons (where players can activate on or off when they want and when they do activate it Pacman will be controlled via Artificial intelligence). A Variety of experiments were provided to compare the results to determine the efficiency of every strategy**.** A lot of intensive research was also put into place to find a variety of 2D games (Chess, Checkers, Go, etc.) which have similar functionalities to the game of Pac-Man. The main idea behind the research was to see how effective 2D games will be if they were to be implemented in the program (Classes/Methods) and how well would the artificial intelligence used in the development of STPacman behave/perform in a variety of different 2D games. A lot of time was also dedicated to research an 'AI' engine which will be able to develop any 2D game based on the users submitted requirements with the use of a spreadsheet functionality (chapter 3, topic 3.3.1 shows an example of the spreadsheet feature) which will contain near enough everything to do with 2D games such as the parameters (The API/Classes/Methods/Text descriptions and more). The spreadsheet feature will act as a tool which will scan/examine all of the users submitted requirement and will have a give a rough estimation(time) on how long it will take for the chosen 2D game to be developed. It will have a lot of smart functionality and if the game is not unique like chess/checkers it will automatically recognize it and Program it. On the other hand, if the game is unique but have similar functionalities/commonalties to similar established 2D games it will also identify the commonalities the games got with each other to help minimise the time of development.

List of Figures

Table of Contents

Chapter 1 Introduction

The purpose of the thesis is focused on intensive research behind the theory of Identification of commonalities in 2D video games for realisation in Artificial intelligence. The research primarily focuses on the development of ‘STpacman’ to see how efficiently the classes/methods/objects in the game of Pacman will be against a variety of different 2D games such as Go, Chess, Checkers, five in a row, Othello, Pong, Cops and Robber, Mancala and connect four where surprisingly all of the games have somewhat a lot of commonalties with one another (in depth explanation in chapter 3 of the thesis). With the Successful results in any field give an excellent opportunity in applying them in other areas, especially if they relate to the specific algorithm, which can be utilized in different games. Not so long ago, the Monte-Carlo Tree Search algorithm was successfully applied to the game Go [1], [2], despite that this is a specific game, which has a large Number of variants after each position (much more than, for example, the game of Chess). The branching factor is a key factor to take into consideration as the higher the branching factor the quicker the expansion occurs. Besides, Go is difficult in the heuristic evaluation of the position; therefore, using different tree search algorithms (for example, Minimax) doesn’t offer a reliable result compared to algorithms such as Monte Carlo Tree search. Another suitable result was obtained in General game playing among the other algorithms [3]. It gave a reason to believe that the approach could be acceptable in other games, such as arcade game Pac-Man.

Further in-depth research would be outlined in different chapters clearly identifying the context in each chapter and will discuss/tackle different problems regarding the research. In chapter 2, you will find out about the literature review (background reading) were a lot of books, articles and publications were found and read to support the development of the project/research in Chapter 3, you will get an insight into the research behind the explanation of the AI concept, the identification of the parameters which is the API/Class/text description/Methods/objects etc. The spreadsheet functionality will contain all the parameters, classes, methods, objects used to develop games. Firstly, A Column for Maze - that will have different types like Triangular, square, rectangular, circular ext., secondly, Control - joystick - how the games played, up down, mouse, space bar, left right etc, thirdly, Algorithms - Minimax, MCTS, Alpha-beta pruning, Null move heuristic, Static board evaluation function and finally the AI can have a lot of functionalities: - check boxes, drop down menus and more for users to go through whilst submitting their requirements so the AI can take in all the information necessary. Chapter 3 will also outline the development of an ‘AI’ engine which will be able to develop any 2D game based on the user’s provided requirements. The AI will have a lot of smart functionalities like the use of a ‘**spreadsheet feature’** which will scan through the user’s requirements and build the game based on the requirements. In some cases, if the user wants games that are already established like chess, checkers and pong the AI will automatically track and trace the requirements and let the user know the game that’s going to be developed. This will help find the commonalties and identify them to the user which will help save time and resources in the future. In chapter 3, 3.3.1 you will be granted with a sketch of how the spreadsheet feature will look. Chapter 3 also contains examples of popular 2D games which the ‘AI’ will automatically recognise due to high demand and popularity allowing the development of these games to be done faster. Chapter 4 will show the projects overall design using UML class diagrams to show the structure on how the game was developed. In Chapter 5, you will get a breakdown on how the artificial intelligence was used within the game of Pacman and the chosen algorithm Monte-Carlo tree search. In chapter 6, you will be granted with the experiments chapter where you will see a comparison of results between NL (next level) and HS (high scores) with tables and graphs to show the outcome of each experiment. Chapter 7 draws a close to the ending of the research carried out (the conclusion) outlining the possible improvements in that could be implemented in the upcoming future as well as the positive/negative throughout the research.

Chapter 2 Literature Review

The literature review chapter is where a lot of intensive research was carried out using tools such as IEE Xplore digital library, Elsevier, Google scholar, Books and many publications to name a few. The chapter will give an outline on a variety of AI (algorithms) which will help determine the choice of AI in the development of Pacman as well as the AI engine which will develop any given 2D game. It will also where all the background reading was done in order to complete the thesis and was referenced using IEEE.

2.1 Monte Carlo Tree search (MCTS) – Choice of Algorithm for the game of Pac-Man

Further progress in the investigation of an efficient algorithm for Pac-Man was marked by the advent of Monte-Carlo Tree Search (MCTS) [24]. This algorithm was based on the tree search, but the main difference is that the tree is expanded not evenly but towards the most promising node. In this case, every agent (4 ghosts + 1 Pac-Man) played independently [24] [25]. Distinctive work in this area is to use the influence map approach [26]. According to this method, the game board can be divided into sections, which are (or are not) suitable for the specific agent. For example, the center of the board is preferable for the Pac-Man; the corners are more suitable for ghosts (of course, it depends on the position). The agent selects the next direction based on the section it is currently placed [27].

2.1.1 – Monte Carlo Tree Search approach

Tong, Ma & Sung (2011) [58] has adopted the Monte Carlo approach for developing the Pac-Man game. They have used the Monte-Carlo model to establish the endgame module for the Pac-Man in which the agent plays the game without the assistance of humans. Rather than completing the whole game, the model is only focused on ending the game when the remaining pills are less or equal to the specified target (Tong, Ma & Sung, 2011). Monte-Carlo path test is used to determine the safest path for the Pac-Man. The Probability limits have been set out for each ghost in the game module, which will determine their movement whenever they exceed or recede the limit of probability. The Endgame module has been placed on the algorithm, which says that when the pills limit known as m is less than the threshold η, then the module will be invoked. The algorithm sorts out the pills and their location and generates the shortest path for the agent by using the previously mentioned path. The algorithm created will maximize the distance of the agent of Pac-Man to minimize the distance to the ghosts.

2.1.2 MCTS Ghost Team

Nguyen & Thawonmas (2011) [22] have analyzed the Monte-Carlo Tree Search method to develop the ghost team in the Pac-Man game. The three ghosts are known as Pinky, Sue; Inky is controlled by using the MCTS algorithm, while the fourth ghost will act only on the rule-based approach. For predicting the path of the Ms. Pac-Man, the k nearest-neighbor (KNN) algorithm is used. A tree has been constructed for the ghost in the MCTS model from the ghost perspective. The branches in the tree are represented as Crossroad, the root node is the cross point for a ghost in which it moves to the next maze, and the child nodes are the adjacent cross-points in which the ghost can reach from the parent nodes. The prediction of Ms. Pac-Man in the path model will help the ghosts to take the shortest path and catch the character. The controlled prediction is made for the ghosts to take a shorter time and path in which the probability of attacking Ms. Pac-Man is high.

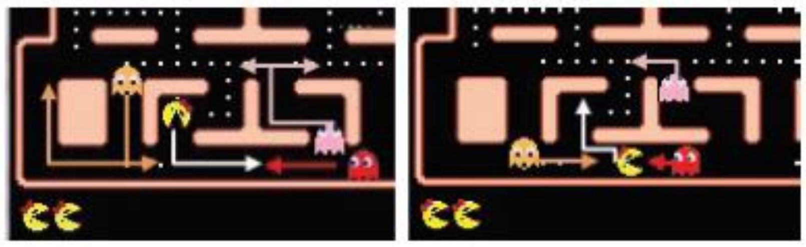


Figure 1 – Comparison of Predicted Path of Ghosts

2.1.3 MCTS More in-depth

While building a Chess game using the Minimax algorithm, which is one of many algorithms available in artificial intelligence, is where my interest in wanting to research and understand more techniques/theory behind Artificial intelligence began. A lot of time was spent researching the most efficient algorithm to use for the AI. Initially, the simple algorithms like 'Minimax' and Alpha-beta pruning were considered because of prior knowledge and comfortability.

While researching for the most appropriate algorithm to use for the agent (AI), I came across Monte Carlo Tree search and saw the growing interest in using MCTS inboard/Strategy games and seemed to be limited in real-time video games. The growth and the rise of MCTS stemmed from the game 'Computer Go' where players felt challenged by computer players. One of the main questions asked currently is if it's possible to 'Create a game that competes and beats 'expert human players' [51]. MCTS is used in a variety of games, starting from different student projects, and used in AlphaGO (where MCTS was used in row with machine learning). Also, MCTS can use Bayesian models to estimate the probability distribution of actions played by a strong player. It can also exist in NaiveMCTS, a MCTS algorithm designed for games with combinatorial branching factors. In MCTS, a tree search technique within the sector of Artificial intelligence is a probabilistic/heuristic search algorithm. MCTS focuses on Nodes that are created from the numerical simulations.

2.1.4 MCTS is split into four steps

1. Selection – Begins with the root node where each child node is chosen from the selection policy. If you reach a leaf node it does not terminate it gets expanded.
2. Expansion – where a new child node is added to the tree which is usually decided within the selection step.
3. Simulation – Simulation selects random moves satisfy and achieve predefined state
4. Backpropagation – Focuses and determines the expansion created (new added node) and updates the remaining tree with the additions. Once it updated the backpropagation is performed where it backpropagates new node → root node. It's also important to add that in the process the amount of 'simulation in each node is incremented' [52].

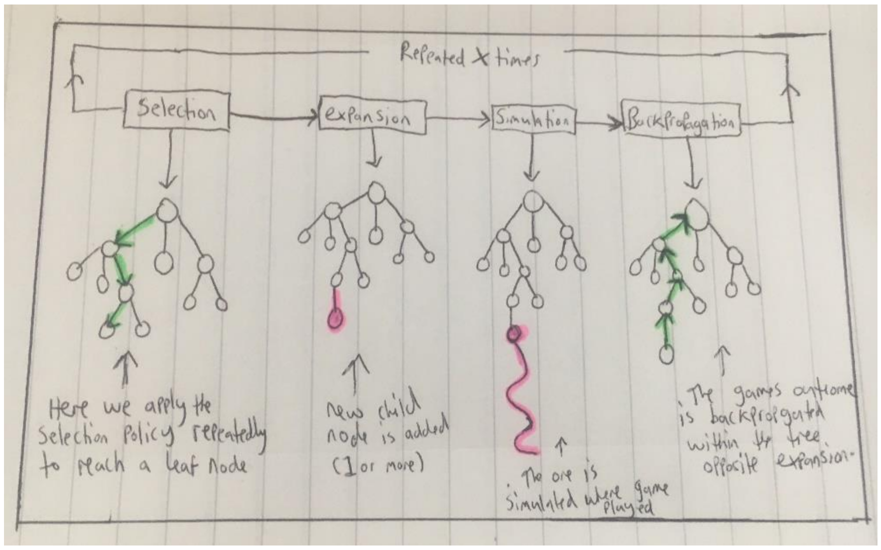

Repeated X times
Selection
expansion
Simulation
Backpropagation
Here we apply the Selection Policy repeatedly to reach a leaf node
New child node is added (1 or more)
The one is simulated where game played
The games outcome is backpropagated within the tree opposite expansion.

Figure 2 – Structure and strategy steps of MCTS

The figure above shows the 4 steps needed in order to use the Monte Carlo Tree search.

2.2 WCCI 2008 vs 2012 (hardcoded vs MCTS)

An interesting solution was proposed for Ms. Pac-Man in contest WCCI 2008, where the behavior of Pac-Man was entirely hardcoded in terms of the limited space 4X4 cells [28]. In this tactic, Pac-Man should firstly avoid ghosts, and only after that, it should collect dots [29]. The apparent advantage of this approach is that the Pac-Man is not distracted by ghosts while collecting dots. Another solution for the Pac-Man agent was developed to contest WCCI'12, where the MCTS method was used [18] [30]. To improve the algorithm, developers made some changes to the classic scheme, which led to better results in general.

2.3 Genetic Programming

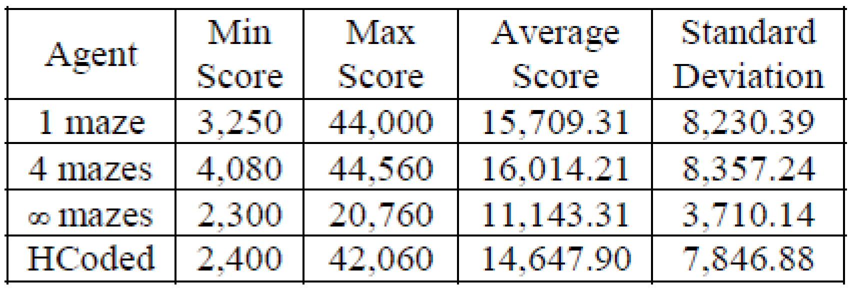
One of the first attempts to develop an AI algorithm for the game of Pac-man (specifically for genetic programming) was made by J.R.Koza [4] [5]. This research was based on the Genetic algorithm and priorities using the previously known list of rules. Such work allowed investigating how the Genetic algorithm is efficient and whether it is suitable for Pac-Man. Lucas and Alhejali [6] applied genetic programming to the different agents and different versions of

Pac-Man. As a result of the work, they demonstrated that the Genetic algorithm is suitable for different types of behavior (for example, it can be applied to Blinky as well as Pac-Man) [7]. Genetic programming was used by Cramer [39] [5] to develop a computational evolution method. This method opens good opportunities for Pac-Man AI. Generally, versus size in genetic algorithm, Distinctive investigation was provided by Rosca, which introduced the generality versus size in the genetic algorithm [44]. Family of genetic algorithms is characterized by similar behaviour – random small changes in basic data (behavior/set of steps/etc) and comparison with same level changes. “Chromosomes” in this case can be set to the steps of Pacman.

2.3.2 Genetic Programming Controlling the agent

Alhejali & Lucas (2010) [5] have adopted another technique for playing the Pac-Man in their article. They have used the genetic programming for controlling the agent that would intelligently play the game. Three types of game levels have been created to check the efficiency of the agent. In the first level game, there is only one level in which the character eats all pills or when eaten by ghosts will terminate the game. The second level is the four mazes, which consist of four levels, and the third level consists of endless mazes where the agent can make an unlimited score. There is a fourth agent made, which is known as a hand-coded agent, to compare the results of the first three agents and their performance in the score. The following table 1 shows the comparison between all agents used in this research and the model game where the hand coded agent operated 2 mazes and scored a min score of 2,400 and max score of 42.060.

Table 1 Scores of All Agents



Agent	Min Score	Max Score	Average Score	Standard Deviation
1 maze	3,250	44,000	15,709.31	8,230.39
4 mazes	4,080	44,560	16,014.21	8,357.24
∞ mazes	2,300	20,760	11,143.31	3,710.14
HCoded	2,400	42,060	14,647.90	7,846.88

The results show that four mazes agent has the best efficiency in scoring, which depicts the performance of genetic programming. The graphical variation has also been displayed in the research article, and it is as follows:

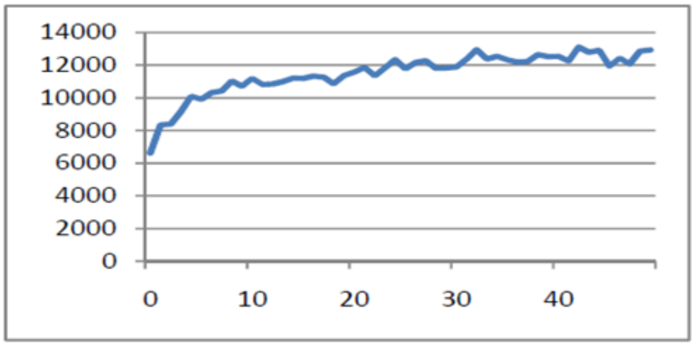


Figure 3 – Average Score of Agents in 4 Maze equipment

2.4 Temporal Difference Learning (TDL)

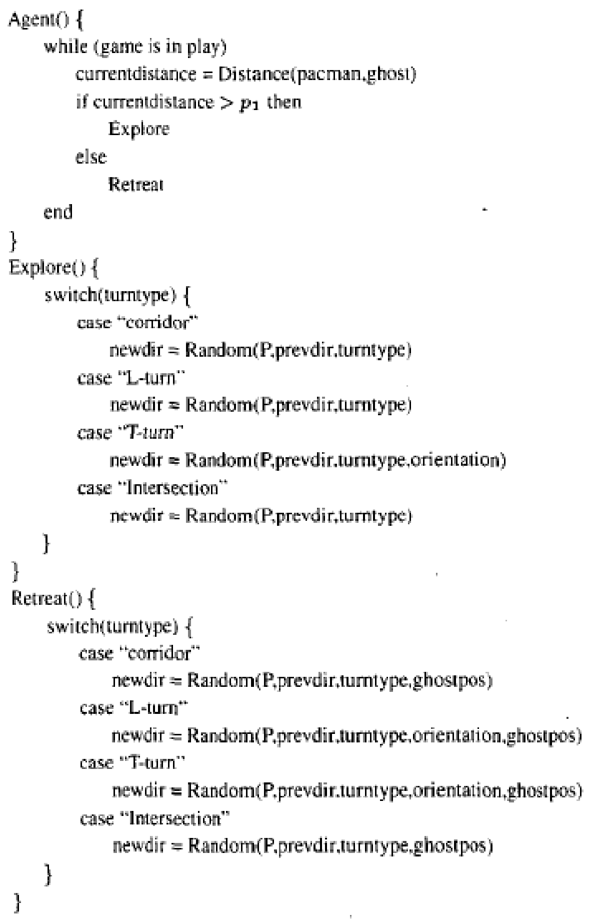
Another team [6] did some experiments with TDL (Temporal Difference Learning) and algorithms based on the evolutionary theory [8]. This analysis included approximation functions, where the arguments are multi-layer perception and interpolated table. The return value of such a function is an evaluation of the possibility of reaching the candidate node [9] [10]. The results of the experiments made it clear that TDL is less efficient that evolutionary algorithms.

2.5 Finite-State Automata

There was research dedicated to the finite-state automata [11]. Besides, Koza [4] used a specific list of predefined rules to manage the agent motion over the board. The current position of the agent defined every move [9]. Mentioned rules involved different changeable parameters, which are based on the incremental learning approach.

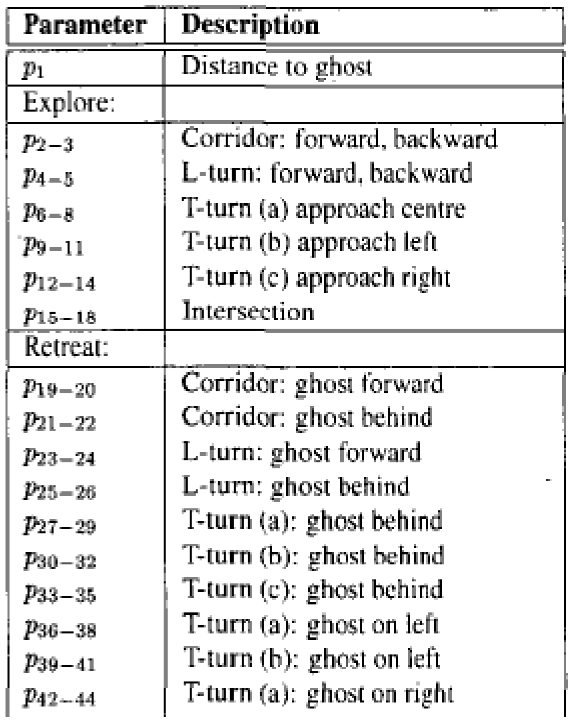
2.6 Rule -based method/Approach

The rule-based method was also applied in another work [12], where a list of rules is structured into groups of actions using the cross-entropy optimization algorithm. It allows getting the next possible direction for the agent using the previously assigned priorities to the groups. Gallagher & Ryan (2003) [5] have mentioned in their article about the system that can play the game of Pac-Man intelligently. In the article, authors have adopted the approach of artificial intelligence for the game. They have developed the artificial agent who will replace the human to play Pac-Man in a more simplified way. The artificial agent is acting like the state machine and the ruleset with some parameters, which will define the probability of the movement in the maze with some constraints. The algorithm created based on population-based increment learning (PBIL), by restructuring the central theme, in which only one ghost and Pac-Man have been mentioned. Parameters have been specified, which clearly defined the movement of the characters in the game and the function of each agent. This algorithm and some parameters are as follows:



```
Agent() {
    while (game is in play)
        currentdistance = Distance(pacman,ghost)
        if currentdistance > $p_1$ then
            Explore
        else
            Retreat
    end
}
Explore() {
    switch(turntype) {
        case "corridor"
            newdir = Random(P,prevdir,turntype)
        case "L-turn"
            newdir = Random(P,prevdir,turntype)
        case "T-turn"
            newdir = Random(P,prevdir,turntype,orientation)
        case "Intersection"
            newdir = Random(P,prevdir,turntype)
    }
}
Retreat() {
    switch(turntype) {
        case "corridor"
            newdir = Random(P,prevdir,turntype,ghostpos)
        case "L-turn"
            newdir = Random(P,prevdir,turntype,orientation,ghostpos)
        case "T-turn"
            newdir = Random(P,prevdir,turntype,orientation,ghostpos)
        case "Intersection"
            newdir = Random(P,prevdir,turntype,ghostpos)
    }
}
```

Figure 4 – Algorithm for Pac-Man Agent



Parameter	Description
p_1	Distance to ghost
Explore:	
p_{2-3}	Corridor: forward, backward
p_{4-5}	L-turn: forward, backward
p_{6-8}	T-turn (a) approach centre
p_{9-11}	T-turn (b) approach left
p_{12-14}	T-turn (c) approach right
p_{15-18}	Intersection
Retreat:	
p_{19-20}	Corridor: ghost forward
p_{21-22}	Corridor: ghost behind
p_{23-24}	L-turn: ghost forward
p_{25-26}	L-turn: ghost behind
p_{27-29}	T-turn (a): ghost behind
p_{30-32}	T-turn (b): ghost behind
p_{33-35}	T-turn (c): ghost behind
p_{36-38}	T-turn (a): ghost on left
p_{39-41}	T-turn (b): ghost on left
p_{42-44}	T-turn (a): ghost on right

Figure 5 – Parameters for Pac-Man movement

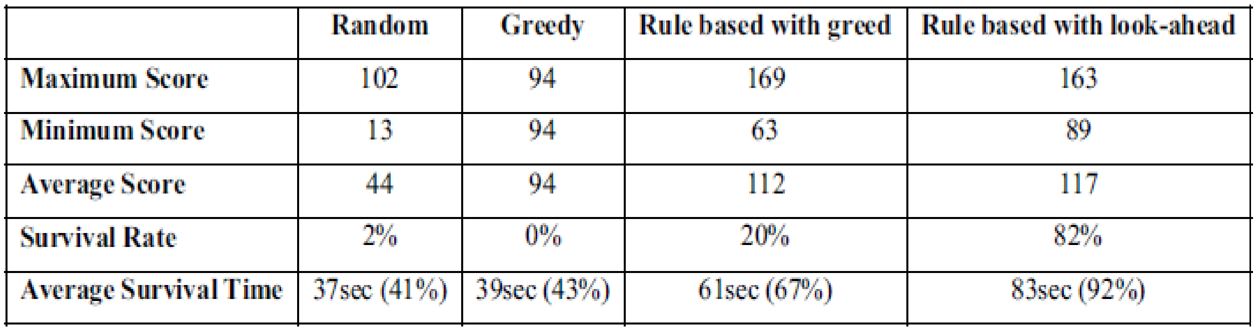
The figures above

Figure 4 and 5 show:

- Figure 5 shows the parameters, which depends on the behavior of algorithm for Pacman (movement of the Pacman agent)
- Figure 4 The global algorithm of Pacman agent's behaviour – All based on the specific movements the agent makes.

2.6.1 Rule- Based look ahead – fifth one critical

Ali, Munir & Farooqi (2011) [55] analyzed the theory on the Pac-Man game in their article. This technique is known as the rule-based look ahead strategy, which is used to check the performance based on the Pac-Man testbed. The authors have described the performance of the approach on the two parameters of the game, the first one is the survival rate, and the second is the number of dots eaten by the Pac-Man. The rule-based approach works in a way that captures information of the critical parts of the game and tells the strategy to the game agent for the next move. There are four methodologies which are adopted in the research to check the performance of each approach by comparing the results of the Pac-Man testbed. The following table shows the results of compared methodologies:



	Random	Greedy	Rule based with greed	Rule based with look-ahead
Maximum Score	102	94	169	163
Minimum Score	13	94	63	89
Average Score	44	94	112	117
Survival Rate	2%	0%	20%	82%
Average Survival Time	37sec (41%)	39sec (43%)	61sec (67%)	83sec (92%)

Table 2 Results of All Methodologies

The table above shows that the look-ahead strategy in the rule-based model has been successful as compared to other approaches. It is because, in this look ahead strategy, the Pac-Man will choose only that path where the number of pills is more and have no ghosts [55] Therefore, making it an excellent approach than the other three strategies which worked on only one or two path selection purpose which could be several ghosts or pallets.

2.7 First use of tree-search algorithm in Pac-Man

The first attempt to apply a tree search algorithm to the Pac-Man was made by Lucas [9]. The search tree was built on the all-possible moves of Pac-Man, but with a maximum 40 level of tree depth. In a bid to evaluate the position, the heuristics function was used [14]. Robles & Lucas (2009) [9] have adopted another perspective to play the Pac-Man in their article. The tree-based software agent is the technique used to represent the Pac-Man, and it is considered as the first attempt on the research of Pac-Man. This tree-based agent works in a way that it is screen capture technique to describe the best path for the agent. The system was developed by authors, only able to score 15000 on the original game while their developed game was 40,000 using a screen capture technique. Four main steps are taken in this research and are making the stimulator, creating an agent, developing the screen capture adapter, and reshaping the algorithm and parameters. The simulator is written in JavaScript duplicated the game; the agent will control the Pac-Man in a tree-based shape which is presented below, the screen capture adapter extracts the information of the original game and transfer to the model simulator. Path selection has been a critical part of this research because it will define the agent to adopt the safe path for gaining maximum score as shown in figure 4 – Safe and Non-safe path for Pac-man. These paths told us that if the agent adopted the thick red path, then it is non-safe, and if it adopts a thin greenway, then it is a safe path.

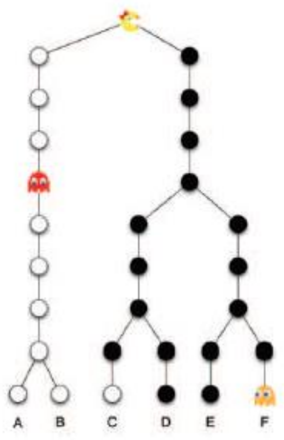


Figure 6 – Tree Model for Agent

(Image from [14] S. Samothrakis, D. Robles, and S. Lucas.

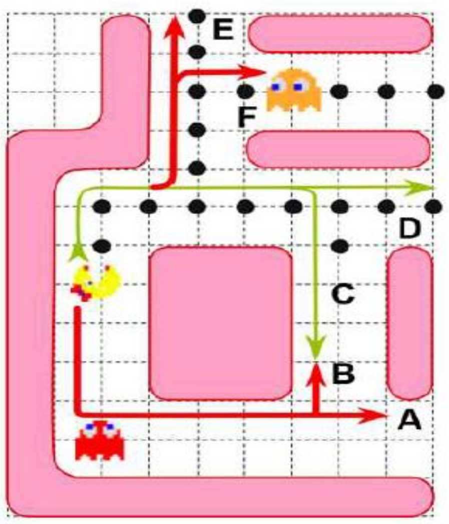


Figure 7 – Safe and Non-Safe Path of Pac-Man

(Image from [9] D. Robles and S. Lucas.

2.7.1 Mixing Rule-based approach with Tree search algorithm

An interesting method was offered by Bell et al. [10]. The main idea of this method is to mix a rule-based approach and a tree search algorithm. In a bid to find the next move for Pac-Man, Dijkstra's Shortest Path algorithm was applied [16]. Fruit, eatable Ghost, or Power Pellet could be served as the goals for this algorithm (depending on the situation on the board). Another investigation was made based on Dijkstra's algorithm and rule-based approach [17]. In this case, the neural network was responsible for the selection of the rules. Also, a situation on the board had a significant impact [18] [19].

2.7.2 Neural Network

Lucas [12] also used a neural network for the evaluation of all possible moves for Pac-Man. As it is known, the behavior of ghosts is not predictable, so a simple shortest path approach cannot be considered as an efficient way to select the next direction [20]. In this case, a neural network is a superior technique. The following research used neural network and evolutionary algorithm [21] to train agents (Pac-Man) how to play the game and how to do it efficiently. The main advantage of this method is the possibility to create agents that do not know anything about the rules [22] [23].

2.8 Ant Colony Optimization algorithm (ACO)

It is also possible to apply the Ant Colony Optimization algorithm to the Pac-Man agent [31]. Initially, ACO was designed for Ms. Pac-Man to improve performance [32]. There are two main goals when creating the fitness function: collecting dots and avoiding ghosts. This approach requires developing two kinds of ants: one is responsible for the dots, and another is responsible for avoiding ghosts.

2.9 Agent behavior (graphs with connected cells as vertices)

Another way to code the agent behavior is to use a graph with connected cells as vertices [33] [34]. This approach allows abstracting the maze and makes the development of algorithms easier. In the contest, CEC 2011 was presented the MCTS algorithm for the ghost team [35]. MCTS, in that case, controlled the Inky, Blinky, and Pinky (Clyde was hand-coded). A simplified version of the AI simulator was developed to evolve the Pac-Man agent [40]. There was a simple AI to control the agent, which moves through the limited area.

2.10 Neuro-Evolution Through Augmenting Topologies (NEAT)

There is also another approach called Neuro-Evolution Through Augmenting Topologies (NEAT) [36] [37]. It is one of the online learning techniques which allow controlling the ghosts. In this case, every ghost act as a separate agent, but there is a shared database (for example, all ghosts ‘know’ the current score) [38].

2.11 Swarm intelligence

It is also possible to use Swarm Intelligence techniques for controlling the ghosts [25]. The main problem in this approach is to develop an entirely suitable fitness function.

2.12 Greedy-look-ahead (Comparison between human and Greedy agent)

Some interesting comparisons were provided by Thompson et al. [41] with greedy-look-ahead and greedy-random agents [42] [43]. The essence of this work is to compare both agents with human players. The experiment started with the highest score for a human player and then a greedy-look-ahead agent [42] [43]. The investigation described the non-deterministic ghost motion and probability of luck.

2.13 Cognitive model

Another aspect of AI to which attention can be paid is a cognitive model that estimates human efficiency in the computational operations during the bidirectional interactive tasks [45]. Such tasks can be described as the interactions between humans and computers with a changeable environment (visual, acoustic) [46].

2.14 Combined Fuzzy Logic

Some of the methods to solve various AI problems can be found in the combined fuzzy logic [47]. For example, there is a fuzzy approach q-learning, which is used to code the behavior of the agent, particularly Pac-Man. In a bid to use this method, the environment should be non-deterministic.

2.15 Simple Tree search Method to play Pac-Man

The paper written by Robbie & Simon M. Lucas was used to enter the IEEE CIG 2009, where he mentions the possible moves ‘Ms. Pac-Man agent can take to depth 40’ [9] with the use of hand-coded heuristics. while delving in further, the highest score obtained was 40,000 on the simulator and roughly 15,000 on the game (scores were gathered using screen-capture). [9]

The experiment focused on 4 steps – These steps were used to create an agent at a competitive level

1. Pac-Man simulator that copies the original game
2. Build an agent which will control Pac-Man
3. Use a screen capturer which can be used to transfer data
4. Add the parameters and algorithms

The simulator is written using java (object-oriented style). The 4 ghosts probabilities table showed the possible chances of Pac-Man getting captured.

The table outlined that Blinky (Red) has a chance of 0.9, Inky (Blue) has a chance of! (0.85), Pinky(pink) has a chance of! (0.80) and finally, Sue (Orange) has a chance of! (0.75). The paper helped give an understanding of the possible experiments that can be considered and potentially followed. It also shows the agents highest score showing the score to aim for when carrying out experiments.

2.16 Real-time AI Implementation: Challenges and Prospects

With the advent of computational intelligence, games have become the biggest testing platform for such techniques. The inherent quality of games to depict a real-life scenario with a simple and limited number of rules make them suitable for this kind of work. However, most of the games lack adaptability and employ already scripted AI techniques making the agents prone to exploitation. Adaptability can be added to the system by modeling the opponents with the help of evolutionary algorithms [53]. But the real-time implementation of this adds an extra layer of complexity to the problem. Another important aspect is the amount of time and computational resources available to train AI models. While offline training provides large time and CPU resources, real-time implementation has significant limitations. So, reducing the number of iterations along and speeding up the evolutionary algorithms become mandatory.

2.17 Training camp method in GP

The classic arcade game Ms. Pac-Man uses a setting in which to test the ability of the training camp method in GP to create a complete controller which can compete against the controller created with standard GP, using the same feature set and under similar conditions [54].The aim of the training camp is to break down the issue into a series of smaller representative problems that can be easily solved and studied before collecting all the solutions to create a complete strategy for such sub-problems.

The critical issue relates to the ability of the training camp to produce better agents than normal GP. The second question concerns the training camp's performance as contrasted with that of a conventional GP run. Specific issues to be addressed include how much work the developer needs to put into each approach to obtain the best possible solution and build training scenarios for the various phases of the training camps. It is observed that the following approach, among others, worked well:

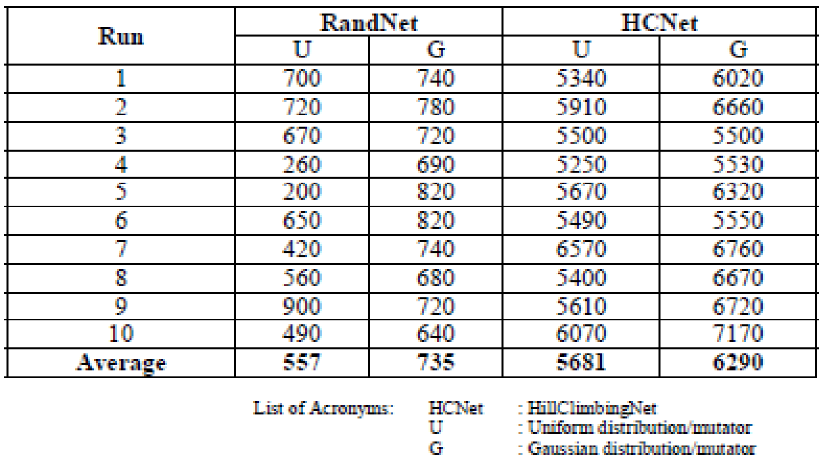
- Grouping the scenarios according to the behaviours that we are attempting to evolve (such as fleeing from unsafe situations)
- Creating agents explicitly for each form of behaviour
- Using each evolved agent as a feature node in the final GP run to learn how best to integrate individual behaviour.

All the regular GP run, and the final run of the training camp were checked under identical conditions, with certain differences, including the feature terminals, population size, and number of generations. The final run of the training camp (which incorporates the sub-behaviors) takes an average of 1 hour to complete, so it would take less than 7 hours to complete a single run of the training camp GP. For standard GP using the specified parameters, a single run on the same system and under the same conditions takes approximately 12 hours to complete for varying time ranges from 3 hours to 26 hours for one run. This enormous uncertainty makes it impossible for each solution to precisely equalize the computational effort employed. In each case, the GP program was run ten times to get a clear view of what the agent would do and to measure various statistics.

The training camp has proven capable of outperforming the regular GP from what's shown above. The training camp received an average score of over 11,000 in 60 percent of the tests with a best score of 11,650, while regular GP did not exceed 11,000 points and only exceeded 10,000 points 4 times, with a best average of 10,848 points. In addition, the training camp achieved a total of over 31,000 points in terms of overall achievement, about 12,000 points more than the regular GP, and in 7 cases out of 10 the average score was over 20,000 points.

2.18 Hill- Climbing

Tan, Teo & Anthony (2010) [57] have used a different approach to test the performance of agents on the Pac-Man game in their article. The authors have used the concept of hill climbing in the artificial networks to test the controllers that play the Pac-Man game. Hill-climbing is primarily used in artificial intelligence to optimize the problems in different games. The algorithm used in this game model is known as HillClimbingNet. This algorithm is based on two options, either with a uniform mutator or Gaussian mutator, and then compared it with another



Run	RandNet		HCNet	
	U	G	U	G
1	700	740	5340	6020
2	720	780	5910	6660
3	670	720	5500	5500
4	260	690	5250	5530
5	200	820	5670	6320
6	650	820	5490	5550
7	420	740	6570	6760
8	560	680	5400	6670
9	900	720	5610	6720
10	490	640	6070	7170
Average	**557**	**735**	**5681**	**6290**

List of Acronyms:		
	HCNet	: HillClimbingNet
	U	: Uniform distribution/mutator
	G	: Gaussian distribution/mutator

algorithm known as RandNet with two options uniform and Gaussian distribution. The HCNet has proved more efficient and better than the RandNet due to its scoring performance, which is shown in the table below:

Figure 8 – Score Results of Two Methods

The table above shows that the algorithm made within the Gaussian mutator of HCNet is more efficient as compared to the uniform mutator. The graph shows the comparison b/w U and G.

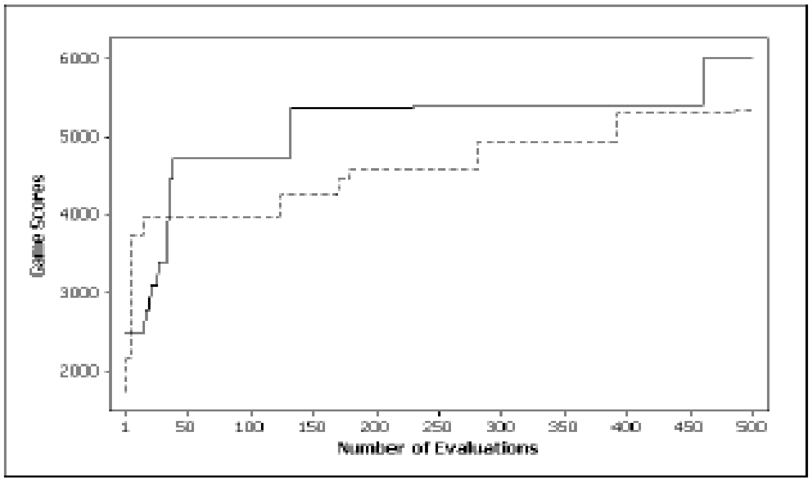


Figure 9 – Results of Hill Climbing Net

2.19 Map Based Controllers

Svensson & Johansson (2012) [56] have utilized the controller technique for Pac-Man and the Ghosts. They have implemented the influenced map-based controllers, which predicted the movements of the Pac-Man and the Ghosts in the game module. Potential fields and influenced maps have been used to compare the results of both techniques. The potential field method worked in a way that it emits the attracting fields in the surrounding to possible movement of the characters. While in the influence maps, the goal is to influence the adjacent tiles to adjacent tiles, thus propagating the influence far away in the game. The controller techniques defined the different aspects of the game in which the first one is used to define the direction of Ms. Pac-Man, in the second, the influence of the pills is designed in which the Ms. Pac-Man attract to the position of the pills. The next controller is the influence of the ghost, which takes care of the ghost by emitting the negative repelling to Ms. Pac-Man. Then the influence of the edible ghosts comes, which activates whenever Ms. Pac-Man eats the power pills.

Chapter 3 – Research Question and Methods

3.1 Can the development on Pac-Man Classes/Methods work efficiently in different 2D games

Chapter 3 will give an in-depth research on how the classes used to develop the game of Pac-Man will impact a variety of different 2D games. A lot of research dedicated to finding a range of similar 2D games (Pong, Chess, Checkers), to name a few. The tables below contain a list of all the classes/methods (in the columns) used within the implementation of the game (PAC-MAN). It also contains a variety of different 2D games (In the rows). The table will outline if different 2D games can follow the same/similar rules of the game Pac-Man for instance, if you get the game Chess you can see that it will have the following classes – AI, sound, Home etc. as Pac-Man and users who will be able to refactor the Pac-Man classes and develop it to follow the rules of chess with a few to no changes in the Code of Pac-Man. All 2D games with similar rules of Pac-Man will be Pac-Man variants only. Focusing on the possibility of finding an effective solution (win), using the same algorithms.

Most of the 2D games listed above have a similar pattern and structure to the one of Pac-Man. Pac-Man has been one of the oldest 2D games has inspired most of the newly developed 2D games with the use of enemy 'AI' (the four ghosts) and a refreshing concept of allowing players to pick up power-ups like cherries which help increase your speed and give you bonus points (cherry been 100 with the lowest amount of points) and collecting the infamous key rewards players with 5000 points. The game of Pac-Man offers players a range of 8 special 'Fruits'.

The game can use the class	✖
The game cannot use the class	—

Figure 10 – Key explaining what the symbols are

2D Games

	MazeBuilder	MazeBuilderTest	PowerPellet	Mobile	Mode	Node	Immobile	key
Chess	✖	✖	—	✖	✖	✖	✖	—
Checkers	✖	✖	—	✖	✖	✖	✖	—
Space Invaders	✖	—	✖	✖	✖	✖	✖	✖
Tetris	✖	✖	—	✖	✖	✖	✖	✖
Connect four	✖	✖	—	✖	✖	✖	✖	—
Tic-tac-toe	✖	✖	—	✖	✖	✖	✖	—
Go	✖	✖	—	✖	✖	✖	✖	—
Pong	✖	✖	✖	✖	✖	✖	✖	✖

Figure 11 – Maze/Game component classes

Classes used within the game of PACMAN

	AI	AI2	AITest	MazeBuilder	Sound	Pacman	Wall	Home	Direction	Scatter
Chess	✖	✖	✖	✖	✖	✖	—	✖	✖	—
Checkers	✖	✖	✖	✖	✖	✖	—	✖	✖	—
Space Invaders	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
Tetris	—	—	✖	✖	✖	✖	✖	✖	✖	✖
Connect four	✖	✖	✖	✖	✖	✖	✖	✖	✖	—
Tic-tac-toe	✖	✖	✖	✖	✖	✖	—	✖	✖	—
Go	✖	✖	✖	✖	✖	✖	—	✖	✖	—
Pong	—	—	✖	✖	✖	✖	✖	✖	✖	✖

Figure 12 – List of Algorithm classes/Main classes used within the game of Pacman

	Test	Strategy	Board	Chase	ChaseTest	Game	Ghost	Home	Dot	EmptySquare
Chess	✖	✖	✖	✖	✖	✖	✖	✖	—	✖
Checkers	✖	✖	✖	✖	✖	✖	✖	✖	—	✖
Space Invaders	—	✖	✖	✖	✖	✖	—	✖	—	✖
Tetris	✖	✖	✖	—	—	✖	—	✖	—	✖
Connect four	✖	✖	✖	—	—	✖	✖	✖	—	✖
Tic-tac-toe	✖	✖	✖	—	—	✖	✖	✖	—	✖
Go	✖	✖	✖		—	✖	—	✖	—	✖
Pong	✖	✖	✖	—	—	✖	—	✖	—	✖

Figure 13 – List of classes implemented in the game of Pacman

	Frightened	GalaxianBoss	Apple	Bell	Blinky	Cherry	Clyde	Inky	Pinky	Melon	Orange	Strawberry
Chess	✖	—	—	—	—	—	—	—	—	—	—	—
Checkers	✖	—	—		—					—		
Space Invaders	—	—	—	—	—	—	—	—	—	—	—	—
Tetris	—	—	—	—	—	—	—	—	—	—	—	—
Connect four	—	—	—	—	—	—	—	—	—	—	—	—
Tic-tac-toe	—	—	—	—	—	—	—	—	—	—	—	—
Go	—	—	—	—	—	—	—	—	—	—	—	—
Pong	—	—	—	—	—	—	—	—	—	—	—	—

Figure 14 – Structure/objects/methods in the game of Pacman

3.2 Pacman's comparison to Chess, Checkers, Go, Tic tac-toe and Connect four

STPacman is a 2D game, and it was developed using the object-oriented approach, so there is a good reason to introduce the AI code to another 2D game, such that chess, checkers, go, tic-tac-toe, connect four and so on. For example, game chess may have abstract class Piece with row and column as the properties (like the Entity class). Class Rook may extend Piece and add its property, such that the Boolean variable that defines whether this rook has already moved or not (it is required for castling). In the same way, other classes may extend the piece, such that Bishop, King, Queen, Knight, Pawn. In general, the structure of the model of chess game may be the same as in Pacman, so it is possible to apply this code in chess. Of course, the return value for the nextMove method in AI class should be changed because, for chess, the piece can move in any direction and different squares. For chess, all possible moves need to be collected. Method nextMove might return only the most promising move. In this case, the most suitable variant is to encapsulate move, in other words, to create a separate class to move with the Piece and the next square as the properties.

The same approach may be applied to checkers. The common abstract class in this game may also be Piece, but subclasses can be RegularChecker and King. Both of them have their move() method since the queen, and regular checker move differently. In checkers, a move also needs to be encapsulated because it can be quite complicated (for instance, captures). Method nextMove() in AI class may also return the most promising move.

It should be noted that the developing AI for game like chess and checkers (Turn-based games) may be a bit simpler than for Pacman. Pacman requires you to consider speed; also, four pieces can move simultaneously (ghosts). In chess and checkers, when one piece is moving, all the other pieces do not move (except castling in chess). MCTS builds the tree towards the most promising move using the random playout, so for chess and checkers, the playout does not depend on speed; only the depth of the tree is important.

All 2D games have common classes/methods with one another, and the most common class been the 'GUI.' While researching and studying games like Chess, Checkers, and Tic-tac-toe, there seemed to be a positive correlation with the pattern and structure of the 2D games. For instance, the class 'GUIMain.java' allows you to view the board, view the header, and view the footer. All the games mentioned above have the same instances.

The class 'GUIStart.java,' on the other hand, allows you to import java abstract window toolkit, which is an API used to show the graphical user interface. It also contains swing, which is also a GUI toolkit and is deemed as more prestigious than the user awt toolkit, in this class, you can add panels, border layouts and buttons (a lot more components) to the game. If you were to develop a Chess game, you can easily use the class to develop a GUI.

In the game of "Cops and Robbers," it is possible to use abstract class Pieces (like Entity in Pacman) and two subclasses: Cop and Robber. This hierarchy is enough for this game because there are only five pieces on the board. AI class is also useful in performing MCTS for the game. In the case of "Cops and Robbers," the total speed of algorithm will be faster than in Pacman because, for example, for Robber, there are only two possible moves (maximum, there is a case when Robber has only one move).

Tetris has a bit of a different structure as it's a tile-based game, but it is still possible to utilize the abstraction like in Pacman. For instance, the whole playing field can be divided into the squares called "entities", so the common abstract class can also be Entity as it is in the Pacman project. Here Entity can have the following subclasses: Mobile (as a part of moving figures) and Immobile (simple static squares). In other words, the initial structure of the code can be the same as in Pacman.

Othello game has two types of pieces (black and white) as well, so this game could have one abstract class Piece and two subclasses: WhitePiece and BlackPiece. AI class with MCTS algorithm is suitable for Othello game, but the returned value in nextMove() method will be the next cell, not the Direction.ConnectFour, as well as Tic-Tac-Toe games, have the similar principles as Othello, so the code for Pacman can be easy adjusted to those games..

The game GO has a much-complicated structure and strategy, but MazeBuilder and AI classes are also appropriate choices that can be implemented into GO. Method nextMove(), in this case, may return a separate object Move because the move in GO is much more complex than in other games.

Games like Tetris, Minesweeper, and Pong have the unpredictable logic. For example, for Tetris, every next piece is generated randomly, so AI class is not the best choice for that game; on the other hand, it is possible using abstract class Piece and subclasses to encapsulate different figures. Pong game has a lot to do with Pacman (in both games a lot depends on the speed used by the user when he presses the keyboard). Minesweeper has hidden values (bombs, numbers), so the probability of correct behavior of the AI is less than 50%. There is a possibility to create AI for minesweeper as long as it takes the hidden information into account. Neither abstract class Piece nor its subclasses, like Mobile or Immobile, cannot be used in Minesweeper. The only suitable class will be MazeBuilder.

Most of 2D games in AI realization require the same algorithms, such that Depth-first search, Breadth-first search. Breadth First Search (BFS) is a crawl method used in graphs. It uses the queue to store visited peaks. In this method, the emphasis is on the vertices of the graph, first one vertex is selected, then it is visited and marked. Then the vertices adjacent to the visited vertex are visited and sequentially stored in the queue.

The Depth First Search (DFS) crawl method uses a stack to store visited vertices. DFS is an edge-based method and works recursively when vertices are explored along a path (edge). The study of a node is suspended as soon as another unexplored node is found, and the deepest unexplored nodes are passed. DFS passes / visits each vertex exactly once, and each edge is checked exactly twice.

The main difference between BFS and DFS is that BFS moves level by level, while DFS goes first from the start to the end node (top), then the other way from start to end, and so on until all nodes are visited. In addition, BFS uses the queue to store nodes, while DFS uses the stack to bypass nodes. For example, in Russian Checkers game the one of the most acceptable ways to find the next cell in case of capture is to use DFS. If to be more precisely, when a piece (king, for instance) searches for a next opposite piece to capture, it explores all neighbors (or all possible next cells) according to the DFS approach. At the same time, for the Pacman game in AI the DFS algorithm can be used to find the next cell for Pacman or ghost as well. Again, when a piece (ghost, for instance) searches for the next cell to move towards the Pacman, it explores all neighbors (in case of Pacman game, it can be maximum 4) following the DFS algorithm. In case of MCTS algorithm for Pacman, DFS, as well as BFS are used in finding the most promising move.

BFS is useful in Russian Checkers to find the shortest path to the promotion cell for simple checker. In evaluation function the piece with lower number of moves towards the promotion cell has highest priority. In case of Pacman game, BFS is used by eaten ghost ("eyes") to find the shortest path to the center of maze. In AI of "Cops and Robbers" game BFS is used for robber piece to find the shortest path to the last row.

To run with BFS, as well as DFS as part of AI need to have 2D maze with the specific number of rows and columns, pieces that occupy a specific cell, and possibility to move from one cell to another (even if the target cell is not adjacent to the starting cell). All of these criteria satisfy the most 2D games, so AI method could be build based on the DFS (or BFS), therefore one AI implementation could be used in more than one games.

As you can see, BFS, as well as DFS, as parts of AI could be used in various 2D games. Both of algorithm's act as example of commonality between different games.

3.3 The AI Concept, Parameters and description of the commonalties within 2D games

To create a 2D game using AI, one has to use many different input components. These components can be divided into three groups. The first group may contain classes (interfaces, enums, methods) that are needed to build the code. The second group may include a variety of graphical components, such as the size and form of a grid, background colour, and visual effects, among others. Lastly, the third group may comprise of the AI methods required to be utilized in the program (Minimax, Minimax with Alpha-Beta pruning, Monte-Carlo Tree search, and Negamax among others). All of these groups contain commonalities for 2D games, which could be developed based on the input parameters mentioned above.

When describing the internal structure of AI, it is possible to assume that AI relies on the possibility of creating a desired game using a specific data structure, for example, a spreadsheet. Such a spreadsheet contains data from every group mentioned above. Some of the 2D games require, for example, interface Movable with method move whereas other games may not require such interface but instead require abstract class move that has different subclasses, which encapsulate different Moves. The AI will scan through a spreadsheet to decide whether the given information is enough to create the desired game and if not enough, the user's request will be rejected, and the reasoning will be sent to the user in a .txt file (some examples may be missing data or unclear instructions).

One of the possible techniques that can be used by AI is a neural network. Such a network requires deep and vast knowledge to create specific connections between nodes (neurons). Such connections may allow the development of the 2D games using only the spreadsheet built based on their input components. On the other hand, this is just an idea and it will require many tests and experiments to develop it in the near future. However, it is possible to assume a sample sequence of actions. For example, a user enters the name of a class “Piece”, ticks’ “Abstract” checkbox, clicks on “+” to add subclasses, then he or she enters “Cop” as a name of the subclass, click on “+” again to add the name of the second subclass, and eventually, enters “Robber”. The user then opens the section responsible for the behavior of pieces. This section allows one to select the type of piece (for example, Cop) and prompts for the possible moves starting from the specific positions on the grid. Once the user sees the coordinates of the piece on the grid, he or she then fills the spinners (for instance, JSpinner widget in Java Swing library) with the possible next coordinates after the move. Next, the user selects “Robber” and enters coordinates for the next move for the “Robber” piece. For instance, Cop has the following coordinates: row = 3, column = 4, and the user selects Robber as a player. Since the Cop is the opposite piece and it moves only towards the “Robber.” The user should then enter the numbers 4,3 and 4,5 as the next possible coordinates (remember, all pieces in these games move only diagonally). If the “Robber” has the 3,4 initial coordinates, the user should enter 2,3, 2,5, 4,3, 4,5 as possible coordinates because Robber moves back and forth.

It should be noted that the quality of creating the code for the game highly depends on the input parameters entered by the user. If the user enters the wrong next coordinate, the AI will not be able to detect the mistake and notify the user. AI is only able to build the game based on the input parameters and it does not matter whether these parameters are valid or not. If they are not valid, AI will simply create another game, in other words, a game with different rules.

One possible example is the process of creating a chess game. Chess has several different pieces with every piece serving its function. Assuming the stated above, one of the parameters for the first group can be the abstract class piece and move. The user interface of the program may contain text fields that allow the user to enter the names of subclasses of the piece (Pawn, King, Knight, Rook, Bishop, Queen). AI makes the piece as an abstract class so that some of the methods related to Move class could be implemented by the subclasses. Subclass pawn, for example, has 3 types of moves, that is, 1 square, 2 squares (only once), and a specific move called “en-passant” when capturing. Also, a class move can have two abstract classes, that is, capture and simple move because every piece in chess can affect the pieces of the opponent.

For the second group, the program may contain colour options, which allows the user to select the colour of the background, as well as the colour of squares, grid, pieces, among others. Besides, the program may have a slider to allow the user to select the size of the grid. It is always possible to add some animation to the game, which shows the user a process of moving the piece from one square to another or gradual appearance of the piece in case of the “Connect four” game.

The third group allows the user to select the AI algorithm that can be used instead of humans, as well as the level of complexity. The structure of the AI algorithm, of course, depends on the type of the selected algorithm, but it is always possible to define the common methods for every algorithm in any game. One example of such a method can serve the next move. This method may return an instance of move class, or just a string representation of move ("4,3-4,4; 4,3:5,8 and so on). In the case of the last variant, the string is always passable.

One important thing concerned with the selected AI is a time spent when AI searches for the move using the predefined level of complexity. It is also a good practice to allow the user to set the maximum allowed time to avoid inconveniences with the computer moving after every 5 to 10 minutes or even more. Creating the timer is a good choice for this purpose.

The same approach can be used in developing the "Cops and Robbers" game. This game is much simpler than Chess and has only two types of pieces, that is, the Cops and Robbers. Following this, it is possible to create one abstract class Piece (common with Chess), but only 2 subclasses; Cop and Robber. Abstract class move here is not needed since the move for both Cop and Robber is pretty simple, and the only difference between them is that the Robber can move back, but for the Cop this action is forbidden.

Designing the user interface required for the second group is similar to the one required in Chess. The user can select the background colour, colour of pieces, and size of the grid, in other words, the same components as in Chess.

Selecting the AI algorithm (third group) and complexity requires the same input components as in Chess. Since "Cops and Robbers" are much simpler than Chess, it does not matter what algorithm the user selects since in all cases, the AI plays with the best strategy and if AI plays with Cops, then Robber is always the loser. Of course, it is always possible to select a low level of complexity, and in this case, the Robber has a chance to win the game.

"Connect four" the game does not have an initial set of pieces since they are being appeared while moving. However, it is always possible to create an abstract class piece and relate it to the square in the grid when the user moves. In this case, there will be only two subclasses, that is, MyPiece and OpponentPiece. The design of the user interface can be chosen by the user in the same way as in the other games, and this includes the size of the grid, background colour, and colour of pieces (only two colours; my pieces and opponent pieces) among others. The type of AI can also be selected from the list of possible algorithms for other 2D games.

Checkers (Russian Checkers) requires the same number of input components in the first group as Chess, "Connect Four", and "Cops and robbers". Again, here it is suitable to create an abstract class Piece with subclasses King and SimpleChecker, as well as the subclasses that can be divided into WhitePiece and BlackPiece. Based on this, the hierarchy of classes is the same as in the other 2D games described above. However, the moves in chess and checkers are more complicated than in other games. Checkers requires making an abstract class move with other subclasses to encapsulate capture, simple move, the capture of the simple checker, the capture of the king, as well as handling the case when a simple checker while the capturer becomes the king. Consequently, this needs to create multiple text fields for the user to engage in more classes as input components to the move section of the user interface.

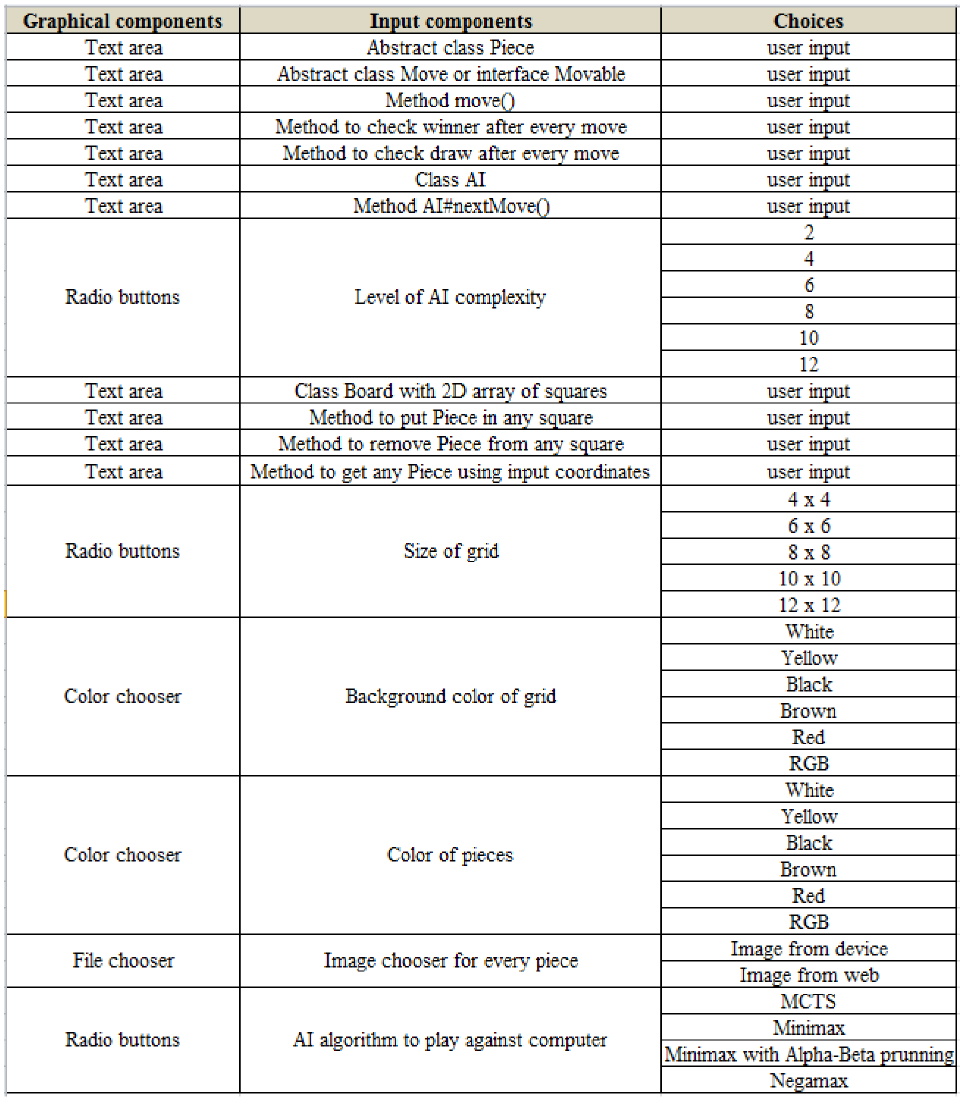
Thereby the commonalities between all 4 games described above may be generalized in the following components:

- Abstract class Piece
- Abstract class Move or interface Movable
- Method move()
- Method to check winner after every move
- Method to check draw after every move
- Class AI
- Method AI#nextMove()
- Level of AI complexity
- Class Board with a 2D array of squares
- Method to put Piece in any square
- Method to remove Piece from any square
- Method to get any Piece using input coordinates
- Size of grid
- The background colour of the grid
- Colour of pieces
- Image chooser for every piece
- AI algorithm to play against the computer

3.3.1 Spreadsheet Feature

1. **Graphical components –** Is the tools used which the user can enter input parameters.
2. **Input components –** Is the components required for AI to build a game

3. **Choices –** Is the variants of input if it is possible, otherwise just text input (user input – English text)



Graphical components	Input components	Choices
Text area	Abstract class Piece	user input
Text area	Abstract class Move or interface Movable	user input
Text area	Method move()	user input
Text area	Method to check winner after every move	user input
Text area	Method to check draw after every move	user input
Text area	Class AI	user input
Text area	Method AI#nextMove()	user input
Radio buttons	Level of AI complexity	2
		4
		6
		8
		10
		12
Text area	Class Board with 2D array of squares	user input
Text area	Method to put Piece in any square	user input
Text area	Method to remove Piece from any square	user input
Text area	Method to get any Piece using input coordinates	user input
Radio buttons	Size of grid	4 x 4
		6 x 6
		8 x 8
		10 x 10
		12 x 12
Color chooser	Background color of grid	White
		Yellow
		Black
		Brown
		Red
		RGB
Color chooser	Color of pieces	White
		Yellow
		Black
		Brown
		Red
		RGB
File chooser	Image chooser for every piece	Image from device
		Image from web
Radio buttons	AI algorithm to play against computer	MCTS
		Minimax
		Minimax with Alpha-Beta prunning
		Negamax

Figure 15 – Example of Spreadsheet feature

3.3.2 The usability of adoption in AI

AI methods are always useful in any game, especially when the user is able to select level of AI according to their needs. AI makes the game more attractive for potential users. For example, in Chess the user can setup a specific position and train against AI. It is very useful to improve the level of playing chess, since AI is able to find the best move in the position. In case of Pacman, the user can setup the AI for Pacman and see the strategy that could allow reaching the next level. In Russian Checkers there is a possibility to create problems (specific position where it needs to find the win or draw after defined number of moves). AI helps the developer to test the solution (for instance, AI can find win in less than initially defined moves).

The fact that most 2D games can be represented as 2D array of cells, allows introducing one AI approach to many games. It is very convenient since the developer can build a single AI interface with initially defined methods that accept 2D array as a parameter. This interface will work with any 2D game.

Apart from utilizing the AI in game logic, it could be possible to use it in design of the game interface and game options. For instance, depending on the last result of the game (win, or lose), the design of interface can be different (different color, brightness, blur and so on). Also, the AI can keep track the user mistakes (when clicking on the wrong place), and adjust the board position (buttons, text boxes and so on). Mistakes will occur If the user doesn't click on the relevant buttons (start, pause etc.). In this case AI can gather all the statistics (in which areas the user clicks most often instead of buttons) and can adjust the button position to reduce idle clicks.

3.3.3 The level of AI complexity and AI methods

Monte-Carlo tree search algorithm, unlike the Minimax, for example, tries to avoid iterating over all possible positions and selects only most promising subtree in the search tree. The important thing is how the parallelization is organized in the code. As mentioned before, MCTS considers only most suitable moves; therefore, number of positions increase after every such move affects the complexity as well. The last thing needs to be considered is the number of iterations in the simulation part, that is the number of moves when AI plays with itself until maximum number of iterations or terminal state of game is reached.

Assuming the mentioned above, the time complexity of MCTS could be computed as product of number of promising moves MCTS considers now, the number of parallel processes, and depth of algorithm divided by the number of available CPU cores. Of course, the more accurate value is not possible to compute for all cases because different users utilize different machines with different CPU and cores that allow parallelization. There was no use of parallelized MCTS. But I tested the algorithm in different computers (2, 4, 8 cores), and got the significant difference, so number of cores does matter. For example, for the computer with CPU Intel Core i7 with 8 cores it is possible to use >100 rounds/iterations of MCTS for depth 1 to get a suitable result (as you know, for Pacman game time is crucial).

As for the memory complexity, only the number of nodes being explored at the moment and number of parallel processes providing them matters. In other words, the memory complexity can be defined as the product of two mentioned values.

As for the methods of AI, then the method that performs simulation part is the most expansive in terms of complexity. Inside this method, AI will make moves up to roughly 1000 times (this is initially defined maximum number of iterations) in the worst case. In other words, the memory stores 1000 objects that represent a single position in one parallel process. And this is only for one level of the search tree. Number of levels in the tree depends on the depth of algorithm and defined by the user before the game starts. The Memory does not need to store all 1000 objects in the simulation phase, but in terms of how Java Garbage Collector works, 1000 copies (minimum) of positions are created (before each move the current position is copied). Of course, it is in the worst case.

3.4 API –APIS used in mapping

Application programming interface (APIs) is used for mapping multiple applications to navigate with one another. The commonalties will map to each API in a distinct order of operations. For instance, API's in the game of Pacman will identify and link classes with commonalties with one another e.g. Monte-Carlo Tree search – 2D joystick. In other words, the API behaves as a messenger that takes requests and tells the program (machine) what to do which then the machine sends the response back to you. Another example is the use of common rules - common function rule as they have similarities with one another. Like having a 2-player game that distinguishes similar pieces, different rules, each player turns, undo, Redo. The features of the games with common set rules and functions will all go under one API which combines all with one another. In some cases, like chess vs checkers they both obey one set of rules. Chess complies a different set of rules but similar APIs as mentioned above. The difference is within the rules of the games as they both have different rule sets for Chess its 1 of 9 rule sets whereas for Checkers its simply 1 of 2 rule sets. The two games however have very similar functionalities like the size of the board 8x8, options, menus, and algorithms to name a few.

3.5 Pacmans decision making in the maze

STpacman decides to make a movement at every tile. MCTS algorithm runs every time the Pacman is required to move. Since the behavior of the ghosts (including a fruit appearing) is unpredictable for the Pacman agent, Therefor MCTS will run before every next move.

3.6 depth of the tree parameter

“depth of tree” means the number of iterations/rounds of MCTS per one move. In other words, how many times selection – expansion – simulation - back propagation block is used per search. I experimented with different values starting from 1 iteration per 1 depth. In the game, the range of depth is 1 – 10, so, for example, for 1 I used 10 iterations of MCTS, for 2 – 30, for 3 – 50 and so on.

Chapter 4 Project Design

Chapter 4 shows steps taken using UML class diagrams to design the project. This project includes the development of the Pac-Man game with a graphical user interface on Java, so it requires selecting the appropriate pattern. The most suitable design is Model View Controller, which separates business logic from the view (GUI) [48].

MVC is widely known as a reliable and flexible pattern for the projects with GUI. In this project I put all java classes relating to the game logic in one package called ‘model’. All classes related to the view are inside the ‘view’ package. It is a convenient way for Pacman since this game has a lot of code required for the user interface and game logic. I believe the separation between these 2 structures is a good choice. For example, Pacman moves from one cell to another (view is updated). Controller catches this change and updates model (Pacman object changes its row and column coordinates). In response to the updates in the model, view is also updates to reflect the changes in the user interface (Pacman#draw() method).

Alternative is to use Model View Presenter pattern, but it is a lot more complex, and the idea of this project is to develop a reliable and well functioned java program, and not to experiment with variety of different patterns.

The general UML class diagram for the model is shown in Figure 16: there is one abstract common class called Entity, which defines common properties for all its subclasses. Two classes – Mobile and Immobile – extend Entity. Mobile class encapsulates entities, which can move and immobile encapsulates entities which are not able to move over the board.

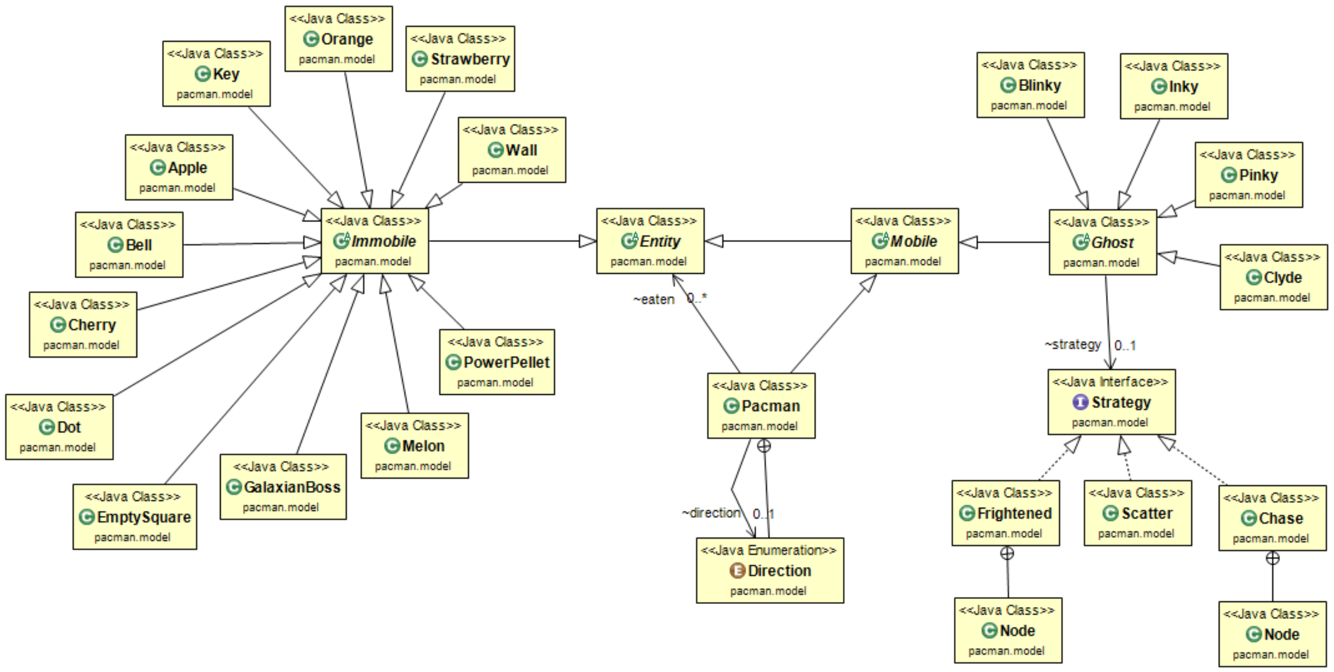
4.1 Moblie classes

There are two mobile classes: Ghost and Pacman, both of them are moving during the game. Ghost class is abstract because four different ghosts exist: Blinky, Inky, Pinky, and Clyde. All four classes above extend Ghost. Class Pacman is the concrete class that encapsulates the main game Agent for which the AI MCTS algorithm is developed [49]. It also contains inner enum Direction that determines the current direction where the Pacman is currently moving (right, left, up, or down). The inner enum is useful elsewhere as it can be placed as a separate class inside the model package. It seems like in the last version pf the game it was placed separately.

4.2 Abstract class (Immoblie)

The abstract class Immobile has multiple different subclasses, which can be divided into three groups: fruits (Strawberry, Orange, Key, Apple, Bell, Cherry, Melon, GalaxianBoss), pellets (Dot, PowerPellet), and obstacle (Wall). Special class EmptySquare encapsulates empty cells in the grid (after eaten Dot), and the GUI does not display it.

In a bid to define the behavior of ghosts, there is interface Strategy, which is implemented by three classes: Scatter, Frightened, and Chase. Each of these classes encapsulates different behavior for the ghosts, so every class implements the interface above. Such an approach comes from the Strategy design pattern.



<<Java Class>> Key pacman.model
<<Java Class>> Orange pacman.model
<<Java Class>> Strawberry pacman.model
<<Java Class>> Wall pacman.model
<<Java Class>> Apple pacman.model
<<Java Class>> Bell pacman.model
<<Java Class>> Cherry pacman.model
<<Java Class>> Dot pacman.model
<<Java Class>> EmptySquare pacman.model
<<Java Class>> GalaxianBoss pacman.model
<<Java Class>> Melon pacman.model
<<Java Class>> PowerPellet pacman.model
<<Java Class>> Immobile pacman.model
<<Java Class>> Entity pacman.model
~eaten 0..*
<<Java Class>> Mobile pacman.model
<<Java Class>> Pacman pacman.model
~direction 0..1
<<Java Enumeration>> Direction pacman.model
<<Java Class>> Blinky pacman.model
<<Java Class>> Inky pacman.model
<<Java Class>> Pinky pacman.model
<<Java Class>> Ghost pacman.model
<<Java Class>> Clyde pacman.model
~strategy 0..1
<<Java Interface>> Strategy pacman.model
<<Java Class>> Frightened pacman.model
<<Java Class>> Scatter pacman.model
<<Java Class>> Chase pacman.model
<<Java Class>> Node pacman.model
<<Java Class>> Node pacman.model

Figure 16 – UML Class diagram for STPac-Man

4.3 Entity - UML diagram for Mobile and Immoblie class

Figure 16 shows a Class diagram for Entity class, where there are all the instance variables and methods are. The two main properties of every entity are one being the row and two being the column; in other words, these are coordinates on the board. The board itself consists of the 15 rows and ten columns, so, for example, if Entity has row = 14 and col = 9, then this entity is placed in the right lower corner of the board.

There are two constructors of this class: constructor to initialize all instance variables and a constructor that accepts another Entity object as a parameter. The second constructor is intended to make a copy of this Entity (copying entities is the required operation in MCTS AI because it needs to get children from the current position without changing the current board) [50]. To make a copy of the Entity, this second constructor is invoked by a copy(Board) method.

There are some abstract methods in this class. The first method is to draw(Grahics2D). This method is used in View to draw the entities on the board. This way is very convenient since it does not require any additional checks or other methods in View; it assumes only iterating through all entities and call draw(..) method. However, every subclass of Entity must implement this method, so every class has a different implementation of this method. For example, Apple draws apple.png image; Cherry draws cherry.png image, and so on.

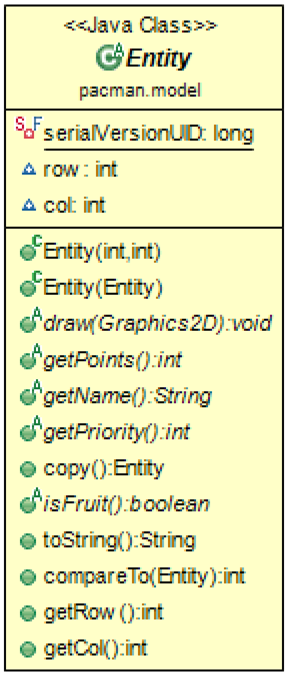


Figure 17 – UML Class diagram for Entity class

4.4 Entity - UML diagram for pacman.model

Next method getPoints() returns the number of points awarded if the Pacman "eats" this Entity. Here "eats" means moving to the square where this Entity is placed. For example, if an entity is Apple, then this method returns 700; if the entity is Melon, it returns 1000, but if entity is Ghost, then the returned value depends on if the ghosts are eatable or not, and the number of ghosts which have already been eaten. In other words, the whole Board consists of the grid of Entity objects, and after each move, Pacman "eats" another Entity (except Wall because it is an obstacle).

Other methods related to the different objects, for example, compareTo(Entity) accept Entity instance as an argument and compares the priorities of this Entity and given Entity. The idea of this method is to sort out the entities in the list that help GUI displaying them correctly (for example, it does not make sense to display Dot if there are Ghost and Dot on the same square).

Method copy() makes and returns a copy of this Entity using the serialization mechanism.

Also, the Entity class extends java built-in Observable class to notify GUI (Observer) every time the Entity is being changed. Here the Observer pattern is used.

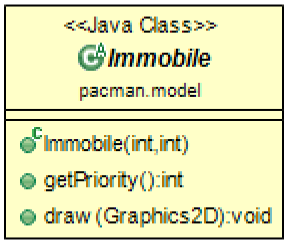


Figure 18 – UML Class diagram for Immobile class

4.5 UML – Immoblie Class

Figure 18 describes the Immobile class, which overrides the constructor and two methods described above. A constructor does not add new functionality, but the method getPriority() returns 0 because Immobile objects always have low priority (Mobile objects should have high priority). Priority relates to the order in which the objects are drawn through the Graphics2D. Immobile objects must have low priority to avoid situations when, for example, empty square is drawn over the pacman (pacman would be hidden in this case).

Method getPoints() is not overridden here because this abstract class does not have information about what instance will be created (Apple, Dot, etc.).

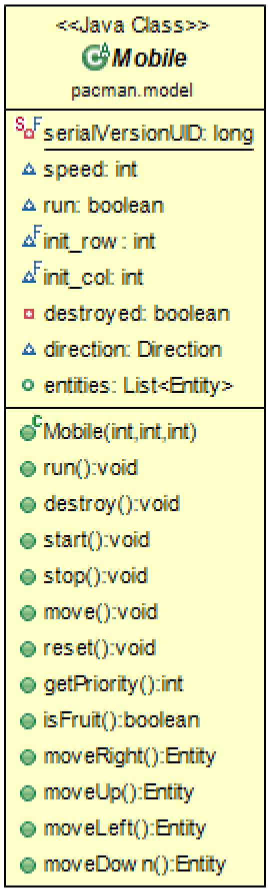


Figure 19 – UML Class diagram for Mobile class

4.6 UML – Moblie class

Abstract class Mobile is represented in Figure 19, where there is one overridden constructor of Entity superclass. This class adds four new methods to move Mobile up, down, left, or right. Each of these methods firstly validates the next cell to avoid out of bounds, then it retrieves the Entity of the next cell and invokes this method from the superclass Entity described above. The logic is the following: if the next cell, according to the current moving direction, is within the bounds, then any Mobile object (Pacman or Ghost) can "eat" this cell. It does not matter that the next cell can be Wall, Wall also overrides moveUp/Down/Right/Left(Mobile) methods, and in this case, it does not change the position of Mobile (method does nothing). Mobile class also contains four overridden methods (moveUp/Down/Right/Left)) to change the row or column of Mobile object and call this method from superclass to call "eat(Mobile)" method. Mobile class has the same logic as the logic is in the Immobile class, but in this case, that method is triggered when two Mobile objects collide on one cell (for example, Ghost and Pacman, or Blinky and Pinky).

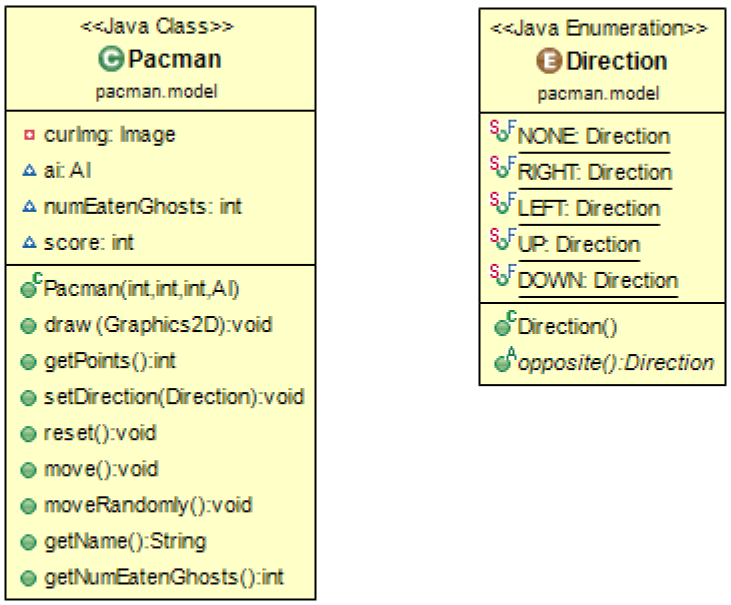


Figure 20 – UML Class diagram for Pacman class

4.7 UML – Pacman class

Figure 20 describes the class for the main agent of this project – Pacman. First, it is a subclass of Mobile class, so it overrides the constructor of Mobile. Next, Pacman adds new instance variables: curImg (a current image which has to be drawn by View – left, right, up or down), numEatenGhosts (counter for number of eaten ghosts). Then, Pacman overrides getPoints() method that returns 0 because Pacman cannot eat another Pacman. Next, Pacman overrides the move method. Inside this method, curImg variable is changed according to the current direction. If AI mode is selected, the current direction depends on the returned value of the AI#nextMove() method.

The last method in Pacman class is move(), where Pacman calls moveUp/Down/Right/Left() method according to the current direction. Inside this method, it first checks whether it is AI mode or not, and if it is AI mode, then it updates the current direction by calling nextMove() method from AI.

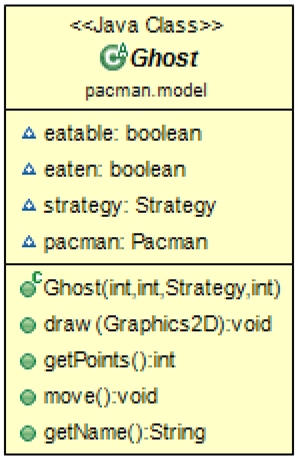


Figure 21 – UML Class diagram for Ghost class

4.8 UML – Class diagram for Ghost class

Figure 21 describes Ghost class, where there is a constructor of the Mobile superclass that is overridden, but the constructor takes an additional parameter called Strategy. Strategy is the interface that defines behavior for the Ghost: it can be either frightening, scatter, or chase. Also, Ghost adds a new instance variable called eatable, which is true if this Ghost is eatable, otherwise false. According to the Pac-Man rules, ghosts become eatable during the specified amount of time after Pacman eats the power pellet.

The ghost class overrides move() method. Inside this method, ghost updates its current direction using the strategy. If successful, it calls this method from the superclass (Mobile) to change if it is possible. In other words, method move() invokes move() method of the Strategy interface. The next direction for the Ghost is defined by the Frightened, Scatter, or Chase classes, which implement the Strategy interface.

Method getPoints() returns 0 if this Ghost is not eatable; otherwise, it returns 200 if Pacman eats a first ghost, 400 if Pacman eats the second ghost, 800 if Pacman eats third ghost, and 1600 if Pacman eats the fourth ghost. After that, it increments the number of eaten ghosts in Pacman object and removes itself from the Board.

4.9 Explanation on NL vs HS

The program introduces a Novel feature called “Assistance.” It is a button, and it’s available only in Manual mode. When the user clicks on it, then Manual mode changes to AI mode (NL strategy), and Pacman moves automatically according to the NL strategy using the MCTS algorithm. This option is useful when the user is close to losing and needs help. At any moment of the game, assistance can be turned off.

It should be noted that there are situations when the assistance button cannot help. For example, Figure 22 shows such a position.

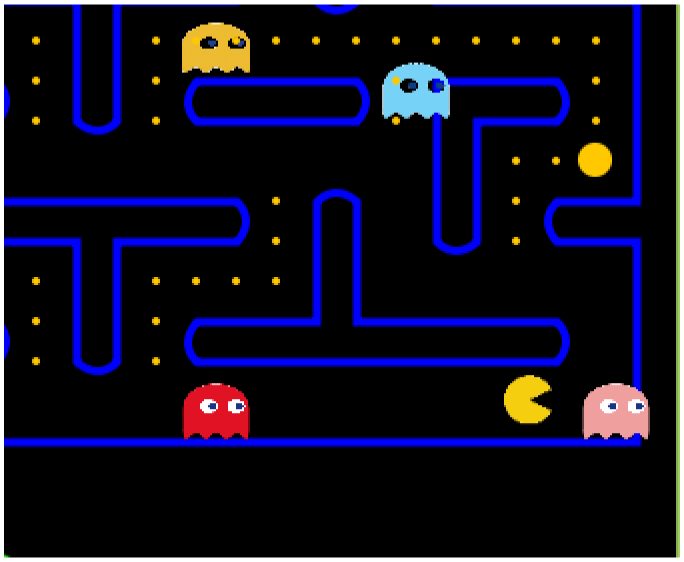


Figure 22 – Position where Pacman must die

4.9.1 Select depth of Tree search

This program also allows the user to select the depth of tree search, as well as the chosen depth randomly. For the first case, it is useful when doing the experiments to know the results for every level of the tree. The second option selects the depth of tree search randomly, so the user does not know in advance what level of tree the AI is currently using.

Chapter 5 Artificial Intelligence

5.1 Chase strategy for Ghost

Algorithm 1 Next direction function for Chase strategy

The code goes through all possible points of route construction, starting from the point of presence of the character. If the next point is not the target, then its unvisited neighbours are added to the next search. This is a classic implementation of the width search algorithm.

```
function nextDirection(Ghost)
        vertices ← all entities except Wall and Ghost excluding input Ghost
        src ← source vertex which encapsulates Ghost
        goal ← goal vertex which encapsulates Pacman
        preds ← map of predecessors to store path from src to goal
        visited ← map of visited vertices
        q ← queue of vertices
        q.add(src)
        for vertex in vertices do
                visited put vertex, false except src
        while q is not empty, do
                v ← q.poll
                if v = goal then
                        break
             end if
             for w in adjacentNodes(v, vertices) do
                     if visited.get(w) = false then
                             visited.put(w, true)
                             preds put(w, v)
                             q.add(w)
                     end if
             end for
     end while
     pred ← null
     for next in preds do
             if next = src or next = null then break
             end if
     end for
     if pred = null then return
     end if
     diffCol ← src.col – pred.col
     diffRow ← src.row – pred.row
     if diffCol ≠ 0 then
             if diffCol > 0 then
                     return Direction.LEFT
             else
                     return Direction.RIGHT
        end if
    else
        if diffRow > 0 then
            return Direction.UP
        else
            return Direction.DOWN
        end if
    end if
```

5.2 Scatter strategy for Ghost

Algorithm 2 Next direction function for Scatter strategy

```
function nextDirection(Ghost)
    vertices ← all entities except Wall and Ghost excluding input Ghost
    src ← source vertex which encapsulates Ghost
    goal ← ghost == Blinky ? (1;26) : Inky ? (1;1) : Pinky ? (29;1) : (29;26)
    preds ← map of predecessors to store path from src to goal
    visited ← map of visited vertices
    q ← queue of vertices
    q.add(src)
    for vertex in vertices do
        visited put vertex, false except src
    while q is not empty, do
        v ← q.poll
        if v = goal then
            break
        end if
        for w in adjacentNodes(v, vertices) do
            if visited.get(w) = false then
                visited.put(w, true)
                preds put(w, v)
                q.add(w)
            end if
        end for
    end while
    pred ← null
    for next in preds do
        if next = src or next = null then break
        end if
    end for
    if pred = null then return
    end if
    diffCol ← src.col – pred.col
    diffRow ← src.row – pred.row
    if diffCol ≠ 0 then
```

```
        if diffCol > 0 then
            return Direction.LEFT
        else
            return Direction.RIGHT
        end if
    else
        if diffRow > 0 then
            return Direction.UP
        else
            return Direction.DOWN
        end if
    end if
```

```
public Node getGoal(Node goal, Node src, Set<Node> vertices) {
        // Predecessors to store path.
        Map<Node, Node> preds = new HashMap<>();

        // Make all nodes unvisited except ghost.
        Map<Node, Boolean> visited = new HashMap<>();
        vertices.forEach(v -> visited.put(v, v == src));

        // Create queue to push nodes.
        Queue<Node> q = new LinkedList<>();
        q.add(src);

        // While queue contains at least 1 node.
        while (!q.isEmpty()) {
            Node v = q.poll(); // Retrieve this node.
            if (v == goal) { // If this node is goal, out of loop.
                break;
            }

            // Otherwise, loop through the adjacent nodes.
            for (Node w : adjacentNodes(v, vertices)) {
                if (!visited.get(w)) { // If this node is not visited yet.
                    visited.put(w, true); // Put this node into the visited map.
                    preds.put(w, v); // Put node v as a predecessor of adjacent
node.
                    q.add(w); // Add it to the queue.
                }
            }
        }

        // Here we need to iterate over predecessors map to reconstruct the whole path
        // from
        // src to the goal.
```

```
                Node pred = null;
                for (Node next = goal; next != src && next != null; pred = next, next =
preds.get(next)) {
                        assert (next != null);
                }
                return pred;
```

5.3 Frightened strategy for Ghost

Algorithm 3 Next direction function for Frightened strategy

```
function nextDirection(Ghost)
        return Chase.oppositeDirection
```

5.4 Home strategy for Ghost

Algorithm 4 Next direction function for Home strategy

```
function nextDirection(Ghost)
         vertices ← all entities except Wall and Ghost excluding input Ghost
        src ← source vertex which encapsulates Ghost
        goal ← (13;13)
        preds ← map of predecessors to store path from src to goal
        visited ← map of visited vertices
        q ← queue of vertices
        q.add(src)
        for vertex in vertices do
                visited put vertex, false except src
        while q is not empty, do
                v ← q.poll
                if v = goal then
                        break
                end if
                for w in adjacentNodes(v, vertices) do
                        if visited.get(w) = false then
                                visited.put(w, true)
                                preds put(w, v)
                                q.add(w)
                        end if
                end for
        end while
        pred ← null
        for next in preds do
                if next = src or next = null then break
                end if
        end for
```

```
if pred = null then return
end if
diffCol ← src.col – pred.col
diffRow ← src.row – pred.row
if diffCol $\neq$ 0 then
    if diffCol > 0 then
        return Direction.LEFT
    else
        return Direction.RIGHT
    end if
else
    if diffRow > 0 then
        return Direction.UP
    else
        return Direction.DOWN
    end if
end if
```

5.5 MCTS algorithm for Pacman

One of the strategies used in this version of the MCTS algorithm is NL, or Next Level priority. According to this strategy, the main priority for Pacman is to eat dots to go to the next level, and eating eatable ghosts, power pellets, or fruits are the secondary goals.

For the behavior of the Pacman agent, the Monte-Carlo tree search algorithm was used. This algorithm allows finding the optimal next direction for the Pacman in every specific position to achieve the goal (eating dots or killing ghosts). Every time the Pacman has four possible directions: up, down, left, or right (in the best case), but in the worst case, there are only two directions due to the obstacles (walls). In the first case MCTS selects among the four variants, the last case allows selecting only 1 of 2 variants. Selecting the direction only among two variants significantly reduces the time spent for search. Like other similar tree search algorithms, MCTS is very dependent on time [14]. in some cases, MCTS select not 4 variants, but only 2, due to position on the map. Number of iterations depend only on the games map size.

The implementation of the MCTS algorithm used in this program includes four main parts: selection, expansion, simulation, and backpropagation. In part selection, the search starts from the root node and goes down the tree towards the leaf (node with no children) selecting the node with the highest estimated number every time. One of the first citation of that algorithm was "Abramson, Bruce (1987). The Expected-Outcome Model of Two-Player Games. Technical report, Department of Computer Science, Columbia University".It can be possible to use the specific formula called UCT to define whether the node is successive or not [16]. Figure 8 shows this formula, where w_i means the number of wins after the i_{th} move, n_i – number of simulations after the i_{th} move, c – exploration parameter (in theory, √2), t – total number of simulations of the parent node.

5.5.1 Selection formula MCTS

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}}$$

Figure 23 – UCT formula for selection part of MCTS

In other words, this formula helps to select the node with the maximum win rate. Algorithm 5 shows the function which returns such a node using the UCT formula. It should be noted that for a situation where the number of playouts and the number of wins is zero, then the heuristics function is used (UCT, in this case, does not make any sense). A simple heuristics function counts the steps to the nearest ghost, and in this case, the node with the largest distance from Pacman to the nearest ghost is the preferable node.

5.6 MCTS – 4 step Implementation

5.6.1 Step 1 - Selection

Algorithm 5 Selection method of Monte-Carlo tree search algorithm for Pacman agent

```
function selection(Root)
    childs ← Root.children
    for child in childs do
        child.parent ← Root
    if Root.children.isEmpty then return Root
    end if
    return childs.MAX (for child in childs do
        if child.playouts == 0 and child.wins == 0 then child.heuristics
        otherwise child.uct
        end if)
```

The next step is to implement the expansion part. The essence of this part is to create child nodes from the current node and select the node from where the random playout will be simulated. The child node is any valid position after the move of an agent. During this stage, the tree grows in depth. Algorithm 6 shows the pseudocode for this operation. Promising.children is the method that contains only actual successors of this node. Promising.childs is a property of Node class that stores all childs returned by children method. It is used to avoid calling children() method every time it needs to iterate over childs of a particular node.

5.6.2 Step 2 – Expansion

Algorithm 6 Expansion method of Monte-Carlo tree search algorithm for Pacman agent

```
function selection(Promising)
    for child in Promising.children do
        child.parent ← Promising
        Promising.childs.add(child)
```

The next step is the simulation. At this stage, the algorithm performs the random playout until the terminate state of the game is reached (Pacman eats all dots or dies). The algorithm selects a node from where the playout starts randomly or using the simple heuristics function described above. Algorithm 7 describes the process in detail.

5.6.3 Step 3 – Simulation

Algorithm 7 Simulation method of Monte-Carlo tree search algorithm for Pacman agent

```
function simulation(Node)
    next ← MAX(Node.heuristics)
    next ← next.copy
    pacman ← entities.pacman
    numlives ← game.lives
    pacmanmove ← Node.pacmanmove
    maxiterations ← 1000
    curiteration ← 0
    while game.lives == nummlives and entities.dot.count > 0 and curiteration <
maxiterations do
        if pacmanmove then pacman.moveRandmoly
        else next.entities.ghost.move
        end if
        pacmanmove ← !pacmanmove
        curiteration ++
    return game.lives < numlives ? LOSE: WIN
```

The last step is backpropagation. This part updates the number of wins and the total number of playouts in every node starting from the bottom of the tree towards the root. After this stage, it can be possible to define the most promising nodes comparing the number of wins and total playouts. Algorithm 8 describes a function that allows for updating every node.

5.6.4 Step 6 – Backpropagation

Algorithm 8 Backpropagation method of Monte-Carlo tree search algorithm for Pacman agent

```
function backPropagation(Promising, Result)
    n ← Promising
    while n ≠ 0 do
        n.playouts ++
        if n.result == Result then n.wins ++
        end if
        n = n.parent
```

5.6.4.1 - Backpropagation values mid game

It depends on the depth (number of MCTS iterations per move). In the middle of the game the values can be 800/1500, 1100/1200 and so on.

Chapter 6 Experiments

6.1.NL (next level priority)

Table I shows the results of experiments with the NL strategy. For every depth of search tree, ten simulations were performed, and the average score along with the average reached stage was calculated and stored.

TABLE I

Table of the results of experiments with NL (Next Level priority) strategy

Number of simulations	Tree depth	Average score	Average stage
10	1	6160	2
10	2	7120	3
10	3	7280	3
10	4	7890	3
10	5	8420	3
10	6	8940	4
10	7	10610	4
10	8	12550	4
10	9	14480	5
10	10	15160	5
Total:		**9861**	**3.6**

As per the results above, the algorithm reaches at least 2nd stage when the minimal depth of the tree (1) is used, and it earns at least 6160 points. The maximum score 15, 160 is available when using tree depth = 10; the maximum stage which could be reached in this case is 5.

6.2. HS (highest score priority)

Table II shows the results of experiments with HS strategy. For every depth of search tree, ten simulations were performed, and the average score along with the average reached stage was calculated and stored.

6.3 – Further Settings of AI

AI requires only 1 setting – depth. There are 10 (1 through 10) possible values for depth. Depth 1 may include from 1 to 100 rounds depending on the computer where this application is launched. In the experiments I used average values because results were very different even for the same number of rounds. In the last version I used 10-20 rounds for depth 1, 30-40 rounds for depth 2, 50-60 rounds for depth 3, 70-100 rounds for depth 4, 80-150 rounds for depth 5, 120-200 rounds for depth 6, 180-300 for depth 7, 250-400 for depth 8, 350-500 for depth 9, and 400 – 800 for depth 10. I also tried to use values > 1000, but the results were dramatically different due to the lack of time.

TABLE II

Table of the results of experiments with HS (Highest Score Priority) strategy

Number of simulations	Tree depth	Average score	Average stage
10	1	3200	1
10	2	3600	1
10	3	3890	1
10	4	4620	1
10	5	4810	1
10	6	5200	1
10	7	5800	2
10	8	6100	2
10	9	6550	2
10	10	7120	2
Total:		**5089**	**1.5**

According to the results above, the algorithm reaches at least 1st stage when the minimal depth of the tree (1) is used, and it earns at least 3, 200 points. The maximum score 7, 120

6.4 Average scores for NL vs Hs

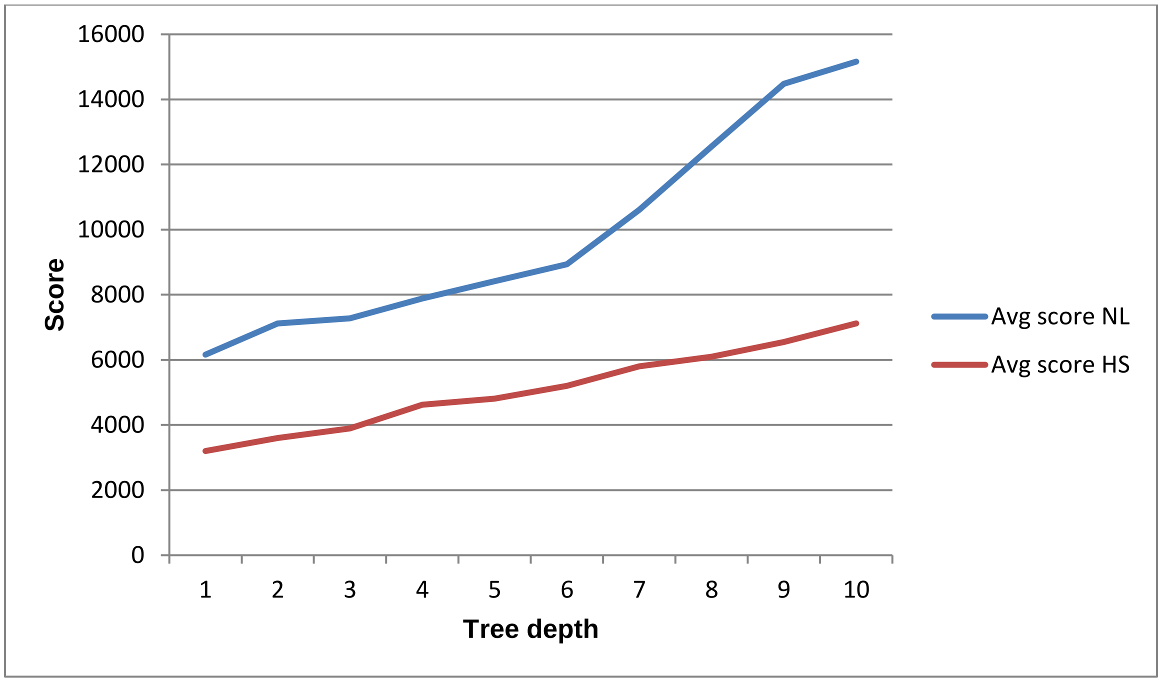


Figure 24 – Graph representation of results of average scores NL vs. HS strategies

Figure 24 clearly shows that the Next Level strategy is more efficient than HS. The highest Score strategy increases performance very slowly during the growing depth of the tree. Next Level strategy highly increases performance after the 6th level of tree depth. Differences between scores are outlined via the graphs: NL strategy is intended to go up to the next level, while HS strategy is not intended to do so.

The HS algorithm has a significant drawback. For example, if there is a fruit (say, Melon, which brings 1000 points) on the board, and last dot (brings only 10 points) to go up to the next level, then Pacman will select next cell towards the Melon because 1000 > 10. This strategy often leads to the unwanted results since the path to the Melon can be fraught with risk to be eaten. The point is that, appearing the fruits that accumulate quite a large number of points cannot be predictable by the algorithm (especially location). Also eating ghosts (this action brings plenty of points as well) always associated with the risk to be eaten. In other words, selecting NL strategy is preferable over HS.

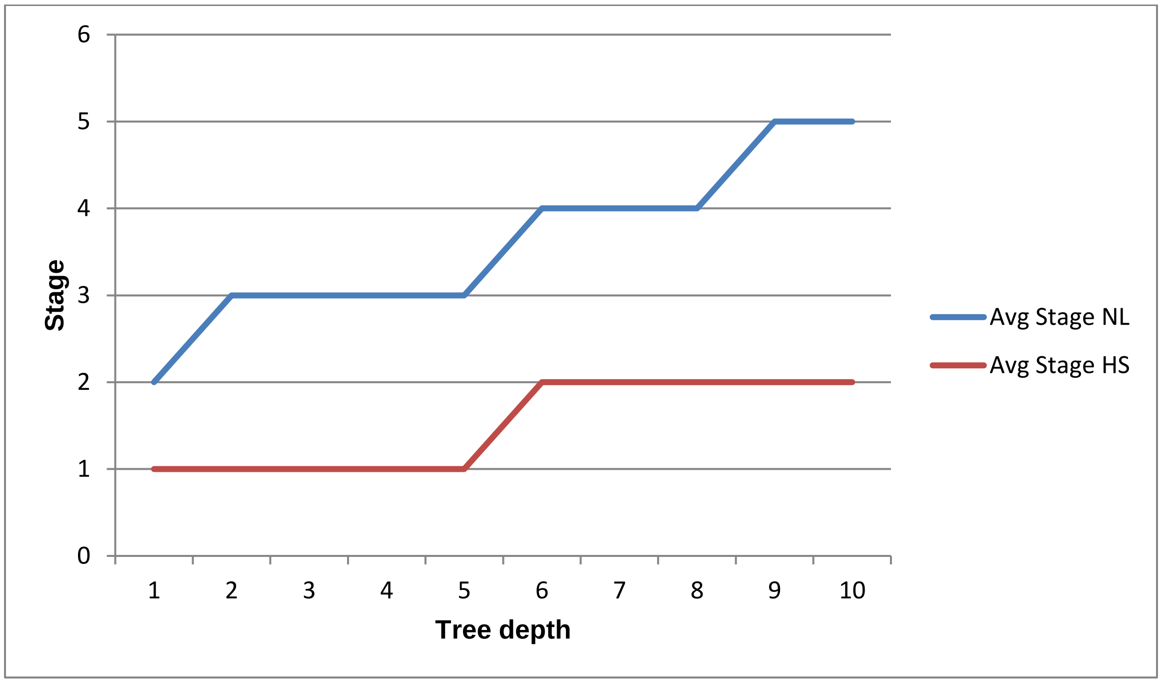


Figure 25 – Graph representation of results of average stages NL vs. HS strategies

As shown in Figure 24, the maximum stage obtained by HS strategy is 2, but the NL strategy reaches level 5. As you can see, the comparison of reached stages clearly shows that the NL strategy is more efficient than HS.

6.5 HS algorithm differs from the NL algorithm - MCTS implementations differ

In case of HS (Highest Score), MCTS expands the tree towards the ability to earn maximum number of points. For example, Pacman decides between 2 targets: Cherry (100 points), and Dots (10 points). HS strategy selects Cherry (100 > 10), so in the simulation phase of MCTS the terminal state is when Pacman eats Cherry.

In case of NL (Next Level), MCTS expands three towards the next level, i.e. between Cherry and Dot the Pacman selects Dot because in the simulation phase of MCTS the terminal state is when all dots are eaten. In NL strategy number of points does not matter.

6.6 Game tree expansion – The way Child nodes for parent nodes are found

The Game tree is expended at each iteration by selecting one of the possible moves randomly. Parent node in this case must be a leaf node, i.e. a node from where no rollout has been performed yet.

There is a maximum of 4 possible child nodes for Pacman: up, down, left, and right. Every one of these nodes can be found by shifting a row or column of a Pacman node. For example, to find the "up" child node, row -1, to find the "right" child node, col + 1. The target cell is checked for possible wall or out of bounds, and if so, it is not added to the list of child nodes.

The same approach can also be applied to the opponent (ghosts) as well. Since there are 4 ghosts and they are moving simultaneously, each enemy move can be considered as a common move of all 4 enemies. For example, Inky moves up, Blinky moves down, Clyde moves right, and Pinky moves right as well, and it can be determined as 1 complex move. There are 4 enemies, and each of them can have 4 possible moves (up, down, left, and right), so the maximum possible child nodes for the opponent is 4 x 4 x 4 x 4 = 256

Minimum number of child nodes for Pacman is 2 (for example, row = 3, col = 6; here means starting from 0), maximum number of child nodes is 4 (for example, row = 5, col = 6, starting from 0). Considering the stated above, the average branching factor for Pacman is 3 ((2 + 4) / 2).

Minimum number of child nodes for enemies is 16 (2 x 2 x 2 x 2) if considering the case when each enemy has only 2 possible moves. Maximum number of child nodes is 256, so the average branching factor for opponent move is (256 + 16) / 2 = 136.

6.6.1 Rollout simulation phase

Simulation phase in MCTS algorithm assumes moving until the end of the game (when Pacman dies, next level reached, maximum number of iterations reached, or Pacman eats entity to earn maximum points in case of HS strategy), starting from the node after expansion phase. For example, after expansion the next move belongs to Pacman, and Pacman moves from cell (3, 3) to cell (3, 4). Next turn: Blinky moves from cell (2, 1) to cell (3, 1), Inky moves from cell (12, 8) to cell (12, 7), Pinky moves from cell (5, 5) to cell (6, 5), Clyde moves from cell (4, 3) to cell (4, 4). Next turn: Pacman move from cell (3, 4) to cell (3, 5). Next turn… and so on until the terminal state of the game (in case of NL strategy), or Pacman earns maximum possible points the current position allows. Of course, when Pacman dies, or maximum number of iterations (1000 by default) is reached, the simulation ends.

6.5.1 Algorhim that controls Pacman during the Rollout – (Win and end of the rollout)
Algorithm for rollout phase uses a while loop to iterate until the terminal state is reached (as

mentioned above). Inside the while loop, every next move is selected randomly. Decision is made when terminal state is reached: WIN if Pacman reaches the next level (in case of NL strategy), or if Pacman eats entity to earn the maximum points; LOSE if Pacman dies.

In case of a WIN, number of playouts are incremented by 1 (denominator), number of wins incremented by 1 (numerator). For example, if the node from where the rollout initiated contains 3/7 score, then after the simulation this score will be 4/8. Note, score 4/8 after 3/7 can be only if the rollout initiated from the node where Pacman moves. If rollout starts from the node where opponent moves, the score will be 3/8 (i.e., only number of playouts is incremented, but number of wins remains the same).

6.7 Novelty

6.7.1 Novel contribution – (AI Panic button)

Novelty is important whilst developing any game as players want to be intrigued and drawn into a feature that has not been implemented before as it helps enhance the player's experience. Whilst researching and testing multiple versions of Pacman, it was very difficult to come up with a completely new concept as the game of Pacman has been covered/revised and refactored multiple times. The challenge was very time consuming but one that could most definitely be figured. After a few days/weeks of playing and testing Pacman games, something clicked where no one thought of adding an 'AI Panic Button', which will be used to control the game of Pacman and give you a break from playing (autopilot/cruise control feature). The assistance will have a 'Turn on panic' button, and a 'Turn off panic' button where users can come back to the game where the AI left off and can turn on and off the panic button as many times as possible (until it loses all its lives and the game ends). The idea came alight when I was playing the game and had to go downstairs for dinner (not wanting to pause the game and still obtain as many points as possible). This is where the panic button idea was developed and was tested multiple times to allow people to take short breaks etc. and return to the game at any instance without having to worry about pausing the game and not maximizing points. It's also a good tool for new users who have little to no experience with the concept of the game. They can watch how the AI maneuver around the maze and can get an idea of how the game mechanic's work.

6.7.2 Use of Random parameters tool

Another novel feature was to possibly implement the use of parameters for the depth of the tree search where players can select from a random list parameter to control the AI (ranging from 1-10) there is also a 'R' parameter option where a random parameter would be selected out of the 10. The parameter in the GUI main screen will look like the image below with a drop-down menu allowing users to pick a parameter of their choice.

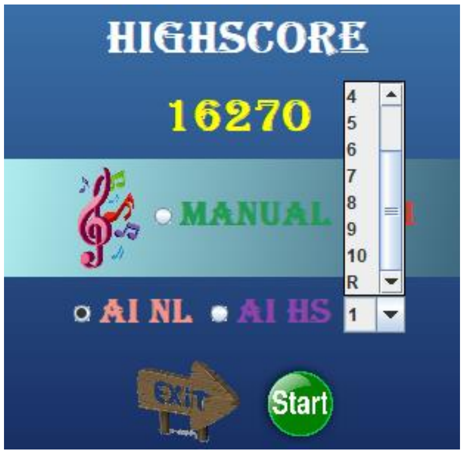


Figure 26 – Drop down menu of the parameter option

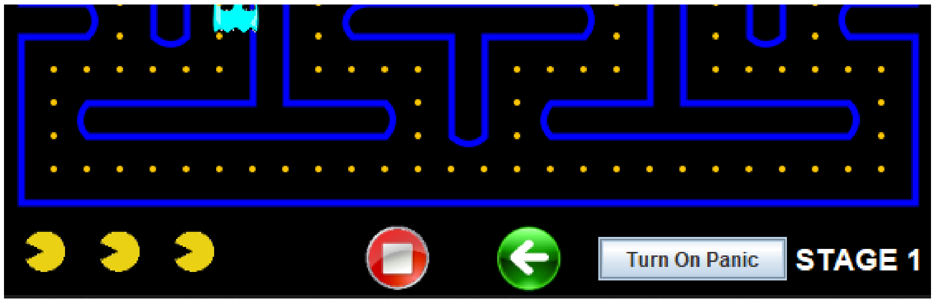


Figure 27 – Screenshot of the 'Turn on Panic' button.

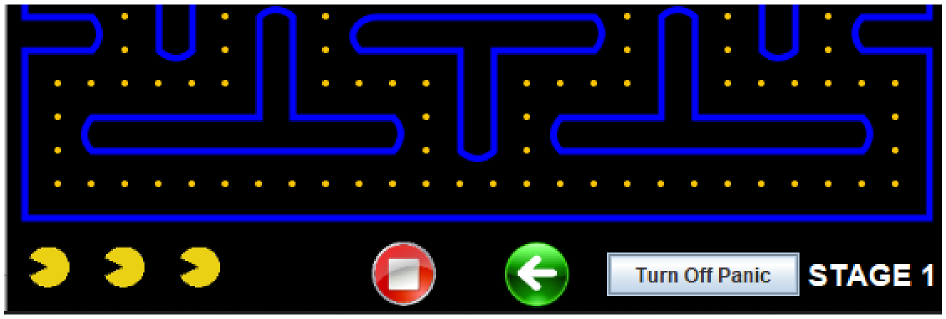


Figure 28 – Screenshot of the 'Turn off Panic' button.

Chapter 7 Conclusion

In this research, one of the objectives was the try to develop a suitable MCTS (Monte-Carlo Tree Search) algorithm for the Pacman agent with two possible strategies: Highest Score priority (HS) and Next Level priority (NL). To compare the two strategies, a variety of experiments were ran to measure the different depth of tree search (1 through 10). For every depth, ten simulations were made to get an average value. According to the results of experiments, the NL strategy was more efficient with a higher performance (9, 861 points, 3.6 level) than HS (5, 089 points, 1.5 level).

NL strategy obtained the maximum score 15, 160, and reached the 5th level with the maximum tree depth (10). HS strategy obtained the maximum score 7, 120, and reached 2nd level with the maximum tree depth (10). In comparison with the results obtained by Robbie & S. Lucas [9], the results seem to be more than modest. For example, the average score obtained in the program developed by S. Lucas was 21, 063 points (against 9, 861 in STPac-man program), where the lowest score as 10, 590 (against 6, 160 in STPac-man program), and highest score 34, 760 (against 15, 160 in STPac-man). As you can see, the tree search method created by S. Lucas seemed to work more than two times better than the method used in the development of the program.

The result indicates a good reason to revise the algorithm in the future to match S.Lucas results and even beyond. I tried to use different ways to setup MCTS algorithm for 2 strategies (NL and HS). For example, I used different numbers of iterations in simulation phase (now it used 1000 iterations). Also, I tried to use different ranges of MCTS rounds for the specific depth. Every time I tested one or another setting, I got different results. In other words, the results were not stable to end up with the specific settings. The significant drawback is with HS strategy. I have already explained earlier that between two available paths the Pacman agent always select path where it is possible to earn maximum points. But this strategy often fails due to the risk to be eaten until the target entity is reached. To improve that, it needs to evaluate some risks on the path. For example, if Pacman has a choice between Melon and a final dot, the right choice would be the dot rather than Melon, even if the dot brings much less points. Also, during the testing phase I noticed that the Pacman doesn't always select the right path, for both the HS/NL strategy and this mostly relates to avoiding the ghosts. Concerning this case, MCTS needs to be optimized and refactored with one of the possible ways been to increase the number of rounds per one search.

On the hand, a lot of time was spent researching the identification of commonalties in 2D games with the realisation of AI. The idea behind the research was to find a suitable AI which would be able to develop a variety of different 2D games using the development of STpacman as a base starting point (similarity in classes, methods, objects and more). The research outlined that a lot of 2D games have similar commonalties/functionalities and methods with one another which allowed the idea of developing a '**2D AI Engine game developer'** which can develop any 2D game of your choice using a lot of smart functionalities as mentioned in the chapter 3.

A lot of student from ‘Bridge Academy’ kindly volunteered to play and tested the Pacman game ‘STpacman’ a fellow student kindly opted to give their initial thoughts behind how usable the system was to them. ‘Janis Lokma’ stated that the game has a lot of unique functionalities such as the AI panic button which I used a fair few time when heading out to collect my parcels the game carried on playing and once I headed back to my machine I turned it off and I was able to carry on from where the AI left off from. Secondly, a comment from ‘Maran’ who mentioned that the User interface was very simple, clear and easy to navigate through the settings. He also mentioned that the research behind the development of the 2D AI engine game developer was very cool, novel and he cannot wait to see someone in the develop it in the future for him to use.

Multiple attempts were made to get into contact with S.Lucas as there is no access to the code of S.Lucas’s work and whilst researching online and reading journals and publications written by S.Lucas I did not find any src code (on git too). Therefore, it’s quite hard to determine whether the results obtained could be comparable with the results of S.Lucas. Experimental conditions cannot be the same because I have never seen the program made by S.Lucas other than reading the publications he has published online. The Algorithm made by S.Lucas can have additional settings or improvements. As of now STPacmans implementation is a lot simpler, and in the future the algorithm can be refactored and improved. For example, I can improve the HS strategy, or combine 2 strategies into 1. Also I believe the program of S.Lucas does not have HS and NL strategy settings, so our results cannot be comparable in the long run.

References

[1] R. Coulom, “Efficient selectivity and backup operators in Monte-CarloTree Search,” in Proceedings of the 5th International Conference on Computers and Games (CG2006), 2006, pp. 72–83.

[2] S. Gelly, Y. Wang, R. Munos, and O. Teytaud, “Modifications of UCT with Patterns in Monte-Carlo Go,” INRIA, Tech. Rep. 6062, 2006.

[3] Y. Bjornsson and H. Finnsson, “CadiaPlayer: A simulation-based general game player,” IEEE Transactions on Computational Intelligence and AIin Games, vol. 1, pp. 4–15, 2009.

[4] J. R. Koza.Genetic Programming: on the Programming of Computers by Means of Natural Selection. MIT Press, 1992.

[5] A. M. Alhejali, and S. Lucas. Evolving diverse Ms. Pac-Man playing agents using genetic programming. In IEEE Symposium on Computational Intelligence and Games, pages 53–60. IEEE Press, 2009.

[6] P. Burrow and S. Lucas. Evolution versus temporal difference learning for learning to play Ms. Pac-Man. In IEEE Symposium on Computational Intelligence and Games, pages 53–60. IEEE Press, 2009.

[7] M. Gallagher and A. Ryan. Learning to play Pac-Man: An evolutionary, rule-based approach. In IEEE Symposium on Computational Intelligence and Games, pages 2462–2469. IEEE Press, 2003.

[8] I. Szita and A. Lorincz. Learning to play using low-complexity rule-based policies. Journal of Artificial Intelligence Research, 30(1):659–684, 2007.

[9] D. Robles and S. Lucas. A simple tree search method for playing Ms. Pac-Man. In IEEE Symposium on Computational Intelligence and Games, pages 249–255. IEEE Press, 2009.

[10] N. Bell, X. Fang, R. Hughes, G. Kendall, E. O’Reilly, and S. Qiu. Ghost direction detection and other innovations for Ms. Pac-Man. In IEEE Symposium on Computational Intelligence and Games, pages 465–472.IEEE Press, 2010.

[11] K. Oh and S. Cho. A hybrid method of Dijkstra algorithm and evolutionary neural network for optimal Ms. Pac-Man agent. In Second World Congress on Nature and Biologically Inspired Computing, pages239–243, 2010.

[12] S. M. Lucas. Evolving a neural network location evaluator to play Ms. Pac-Man. In IEEE Symposium on Computational Intelligence and Games, pages 203–210. IEEE Press, 2005.

[13] M. Gallagher and M. Ledwich. Evolving Pac-Man Players: Can we learn from raw input? In IEEE Symposium on Computational Intelligence and Games. IEEE Press, 2007.

[14] S. Samothrakis, D. Robles, and S. Lucas. Fast approximate max-n Monte-Carlo Tree Search for Ms. Pac-Man. IEEE Transactions on Computational Intelligence and AI in Games, 2011.

[15] N. Wirth and M. Gallagher. An influence map model for playing Ms. Pac-Man. In IEEE Symposium on Computational Intelligence and Games, pages 228 – 233. IEEE Press, 2008.

[16] A.Previti, R.Ramanujan, M.Shaerf, B.Selman. Monte-Carlo Style UCT Search for Boolean Satisfiability, 2011.

[17] J. Flensbak and G. N. Yannakakis. Ms. Pacman AI Controller.2008.

[18] T. Pepels, M. HM Winands, and M. Lanctot. Real-time montecarlo tree search in ms pac-man. Computational Intelligence and AI in Games, IEEE Transactions on, 6(3):245–257, 2014.

[19] M. Emilio, M. Moises, R. Gustavo, and S. Yago. Pac-man: Optimization based on ant colonies applied to developing an agent for Ms. Pac-Man. In Computational Intelligence and Games (CIG), 2010IEEE Symposium on, pages 458–464. IEEE, 2010.

[20] G. Foderaro, A. Swingler, and S. Ferrari. A model-based approach to optimizing ms. pac-man game strategies in real time.

[21] G. Foderaro, A. Swingler, and S. Ferrari. A model-based cell decomposition approach to on-line pursuit-evasion path planning and the video game ms. pac-man. In Computational Intelligence and Games (CIG), 2012 IEEE Conference on, pages 281–287. IEEE, 2012.

[22] K. Quang Nguyen and Ruck Thawonmas. Applying Monte-CarloTree Search to Collaboratively Controlling of a Ghost Team in Ms. Pac-Man. In Games Innovation Conference (IGIC), 2011 IEEE International, pages 8–11. IEEE, 2011.

[23] M. Wittkamp, L. Barone, and P. Hingston. Using NEAT for continuous adaptation and teamwork formation in Pacman. 2008.

[24] F. Liberatore, A. M. Mora, P. A. Castillo, and Juan Juli ́an Merelo Guerv ́os. Evolving evil: optimizing flocking strategies through genetic algorithms for the ghost team in the game of Ms. Pac-Man. In Applications of Evolutionary Computation, pages 313–324.Springer, 2014.

[25] Cramer, N. A Representation for the adaptive generation of simple sequential programs, Proc. of an Intl. Conf. on Genetic Algorithms and their Applications, Carnegie-Mellon University, July 24-26, 1985.

[26] Bonet, J. and C. Stauffer. Learning to play Pac-Man using incremental reinforcement learning. In the Congress on Evolutionary Computation. 1999.

[27] Thompson, J., L. MacMillan, and A. Andrew. An evaluation of the benefits of look-ahead in Ms. Pac-Man.   In IEEE Symposium on Computational Intelligence and Games. 2008. Perth, Australia.

[28] Luke, S. and L. Spector. Evolving teamwork and coordination with genetic programming. In the First Annual Conference on Genetic Programming. 1996: MIT Press.

[29] Rosca, J. Generality versus size in genetic programming. In the First Annual Conference on Genetic Programming. 1996: MIT Press.

[30] Ohno, T. and H. Ogasawara. Information acquisition model of highly interactive tasks. In ICCS/JCSS. 1999. Charlotte, North Carolina, USA: Association for Information Systems.

[31] DeLooze, L. and W. Viner, Fuzzy q-learning in a nondeterministic environment: developing an intelligent Ms. Pac-Man agent.

[32] S. M. Lucas and G. Kendall, "Evolutionary computation and games," IEEE Computational Intelligence Magazine, February 2006.

[33] M. L. Marcus Gallagher, "Evolving pac-man players: Can we learn from raw input?" in IEEE Symposium on Computational Intelligence and Games,2007.

[34] Fitzgerald, P. Kemeraitis, and C. B. Congdon, "Ramp: A rule-based agent for ms. pac-man," in World Congression Computational Intelligence,2008.

[35] R. T. Hiroshi Matsumoto, Chota Tokuyama, "Ice ambush 2," inhttp://cswww.essex.ac.uk/staff/sml/pacman/cec2009/ICEPambush2.pdf,2008.

[36] S. Rabin, AI Game Programming Wisdom.CharlesRiverMedia, Inc.,2002.

[37] C.-S. Lee, M.-H. Wang, G. Chaslot, J.-B. Hoock, A. Rimmel, O. Tey-taud, S.-R. Tsai, S.-C. Hsu, and T.-P. Hong, "The computational intelligence of MoGo revealed in Taiwan's Computer Go tournaments," IEEE Transactions on Computational Intelligence and AI in Games, vol. 1, pp. 73 – 89, 2009.

[38] J. H. Holland, Adaptation in natural and artificial systems. Ann Arbor: University of Michigan Press, 1975.

[39] E. Alba, J. F. A. Montes, and J. M. Troya, "Fully automatic ANN design: A genetic approach," in IWANN '93: Proceedings of the International Workshop on Artificial Neural Networks. London, UK: Springer-Verlag, 1993, pp. 399–404.

[40] Bajurnow, A. and V. Ciesielski. Layered learning for evolving goal-scoring behavior in soccer players. In Proceedings of the 2004 Congress on Evolutionary Computation (CEC2004). 2004: IEEE.

[41] G.Chaslot,M.Winands, H.Herik, J.Uiterwijk, and.Bouzy, "Progressive strategies for Monte-Carlo tree search, "New Math. Natural Comput., vol. 4, no. 3, p. 343, 2008.

[42] P. Hingston and M. Masek, "Experiments with Monte Carlo Othello," inProc. IEEE Congr. Evol. Comput., 2007, pp. 4059–4064.

[43] R. Balla and A. Fern, "UCT for tactical assault planning in real-time strategy games," in Proc. 21st Int. Joint Conf. Artif. Intell., 2009, pp.40–45.

[44] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in Proc. 15th Eur. Conf. Mach. Learn., 2006, pp. 282–293.

[45] M. Enzenberger, M. Muller, B. Arneson, and R. Segal, "FUEGO: An open-source framework for board games and go engine based on Monte-Carlo tree search," IEEE Trans. Comput. Intell. AI Games, vol.2, no. 4, pp. 259–270, Dec. 2010.

[46] C. Luckhardt and K. Irani, "An algorithmic solution of n-person games," in Proc. 5th Nat. Conf. Artif. Intell. (AAAI), 1986, pp.158–162.

[47] N. Ikehata and T. Ito, "Monte Carlo tree search in Ms. Pac-Man," in Proc. 15th Game Program. Workshop, IPSJ Symposium Series Vol.2010/12, 2010.

[48] T. Galaxies, Twin Galaxies—Ms Pac-Man High Scores @ON-LINE 2006 [Online]. Available: http://www.twingalaxies.com/index.aspx?c=22&pi=2&gi=3162&vi=1386, [Online].

[49] Tong BKB, Sung CW. A Monte-Carlo approach for ghost avoidance in the Ms. Pac-Man game. International IEEE Consumer Electronics Society's Games Innovations Conference (ICE-GIC).2010.

[50] C.B. Browne, E. Powley, D. Whitehouse, S.M. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. Computational Intelligence and AI in Games, IEEE Transactions on, 4(1):1 –43, March 2012.

[51] Simon M Lucas. Computational intelligence and AI in games. IEEE Transactions on 02 May 2009.

[52] Tom Pepels, Mark H. M. Winands, Marc Lanctot, Real-Time Monte Carlo Tree search in Ms Pac-man. IEEE Transactions on 04 February 2014.

[53] M. Quinn, L. Smith, G. Mayley, and P. Husband, "Evolving teamwork and role allocation with real robots," in In Proceedings of the 8th International Conference on The Simulation and Synthesis of Living Systems (Artificial Life VIII), 2002.

[54] S. M. L. Atif M. Alhejali, "Using a Training Camp with Genetic Programming to Evolve Ms Pac-Man Agents," in *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, Seoul, South Korea, 2011

[55] Ali, R. B., Munir, A., & Farooqi, A. H. (2011, June). Analysis of rule-based look-ahead strategy using Pacman Testbed. In *2011 IEEE International Conference on Computer Science and Automation Engineering* (Vol. 3, pp. 480-483). IEEE.

[56] Svensson, J., & Johansson, S. J. (2012, September). Influence Map-based controllers for Ms. PacMan and the ghosts. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)* (pp. 257-264). IEEE.

[57] Tan, T. G., Teo, J., & Anthony, P. (2010, March). Uniform versus Gaussian mutators in automatic generation of game AI in Ms. Pac-man using hill-climbing. In *2010 International Conference on Information Retrieval & Knowledge Management (CAMP)* (pp. 282-286). IEEE

[58] Tong, B. K. B., Ma, C. M., & Sung, C. W. (2011, September). A Monte-Carlo approach for the endgame of Ms. Pac-Man. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)* (pp. 9-15). IEEE.