UNIVERSITY OF HERTFORDSHIRE
School of Computer Science

Modular BSc Honours in Computer Science

6COM0286 – Artificial Intelligence Project

Final Report
April 2015

IMPROVING PAC-MAN GHOST BEHAVIOUR USING GENETIC ALGORITHMS

K. C. Leddy

Supervised by: Rene Te Boekhorst

Abstract

The project investigates the feasibility of utilising a Genetic Algorithm to control the gameplay of the ghosts in the classic game of Pac-Man. Various simplifications were however made to the game in order to streamline it to purely focus on the ghosts' ability to catch Pac-Man. The code for the project was written in Python 2.7 and was based on an open source Pac-Man clone written by *David Reilly (2013)*.

The evolutionary algorithm used within the project was based on a chromosome representation of the action (up, down, left, right, random) to be taken within a given situation. These were encoded as an array of integers, each representing the action to be taken for that situation. Fitness scores were calculated by running a simulator and were based on the average 'Manhatten Distance' to Pac-Man per move. Bonuses were added to or removed from the fitness when the ghost caught Pac-Man or attempted to walk into a wall, at which point a random direction is chosen in order to keep the gameplay moving. The 'roulette-wheel' selection method was used along with 'two-point crossover' and 'uniform mutation' in order to create successive generations.

The simulator was implemented using a 'best-first' search algorithm for Pac-Man's path finding to the nearest pellet. The heuristic for the algorithm was calculated as the minimum 'Manhattan Distance' to each of the first 25 pellets on the map. This provided good results for the path finding, although it would freeze in certain specific situations; however it left much to be desired in terms of its comparison to the gameplay exhibited by human players. This also caused some minor issues with the fitness scores of the ghosts, as the simulated Pac-Man had no desire to avoid them; it simply wanted to collect pellets.

After 250 generations I found that the fitness scores of the ghosts were indeed going down, however the best fitness scores in each generation levelled out after roughly 100 generations and remained within the same range. Additionally, a couple of behaviours were discovered, such as the ghosts rushing certain map areas at the start of a level and blocking off corridors.

My evaluation of the project states that I am happy with the outcome of the project and believe that even though the results were not as prominent as I had hoped, I have made great progress towards what I hoped to achieve and feel that the project was a success.

Acknowledgements

I would like to thank my project supervisor, Rene Te Boekhorst for all of his help, support and advice throughout the project.

I would also like to thank the original creators of the 'Pac-Man' clone that I based my project on, David Reilly and Andy Sommerville for providing a stable implementation of the game that I was able to modify to fit the purposes of the report.

Contents

1 Introduction

1.1 Project Motivation

As an avid gamer and programmer, I've always been interested in making games and the various different applications of programming techniques that are used within them. As such, I felt that this project would be a good opportunity to investigate how a technique that is often overlooked in the area of gaming could be applied to a simple and well-known game, in order to present a new aspect of gameplay and put a twist on a classic game.

For this project, I chose to expand upon the classic game of 'Pac-Man' with a Genetic Algorithm to determine the behaviour of the enemies. I chose 'Pac-Man' as it is a classic, retro arcade game that contains fairly simple gameplay and is well-known by a large quantity of the population as it is commonly referred to as one of the most iconic games of all time. I decided to utilise a Genetic Algorithm for the behaviour of the enemies in the game, as Genetic Programming has been used several times before to evolve agents for the player, *(Gallagher & Ledwich, 2007)(Alhejali & Lucas, 2010)* and I felt it would be more interesting to attempt a different approach.

1.2 Brief Overview of Genetic Algorithms

Genetic Algorithms were invented in order to replicate some of the mechanisms observed in natural evolution. Essentially, they are designed to simulate 'survival of the fittest' over consecutive generations. Each generation is composed of a population of 'individuals' that each represents a possible solution. Individuals are then made to go through the process of evolution.

Evolution usually begins with a population of randomly generated individuals. Each individual is provided with a 'fitness' score that determines their level of capability to solve the given problem. The more fit individuals are then selected for use to combine their genes with one another in order to create the next generation. The new population is then usually subject to 'mutation', in which their genes are modified slightly in order to maintain genetic diversity within the population, and prevent it from getting stuck in a local optimum for the given problem. The entire evolution process is then repeated until some condition is satisfied (such as number of generations evolved or fitness level reached).

1.3 Brief Overview of Pac-Man

Pac-Man is an arcade game developed by Namco that was released in 1980. It is an incredibly popular game and is considered quite iconic, being recognized by "94 percent of U.S. residents" *("Pac-Man still going strong", 2010)*. Since its release, there have been several spin-offs of the game and it has become somewhat of a video game history landmark.

Gameplay is relatively simple; the player controls Pac-Man through a maze, eating 'pellets', once all pellets are eaten, the next stage begins. Four enemy ghosts traverse through the maze, trying to catch Pac-Man. Once Pac-Man is caught, a life is lost and once all lives have been lost, the game ends.

Near the corners of the maze there are four larger 'power-pellets' that once collected by Pac-Man, temporarily grant him the ability to eat the ghosts. Once eaten, the ghost's eyes remain and travel back to the box in the centre, where the ghost is regenerated.

There is no end to the game, it simply continues until the player has lost all of his/her lives and it is game over. The purpose of the game is to survive as many levels as possible and obtain the high score.

1.4 Gameplay Rules and Simplifications

For the purposes of the project, the gameplay of the Pac-Man clone has been simplified to remove various aspects that over-complicate the gameplay and are outside of the scope of what the project aims to achieve.

The first simplification that was made was the removal of levels, as the project only aims to investigate the capabilities of the ghosts and is not concerned with the level progression system. As such, only a single level has been implemented in the project, which conforms to the standard map layout as it appeared in the original game of Pac-Man (see Figure 1).

High scores and sounds were also removed from the project as they are not relevant to the investigation.

Additionally, the 'power-pellets' were removed from the game for the project as they over-complicate the gameplay by providing Pac-Man with the ability to eat the ghosts. This would interfere with the ghosts' chromosomes as they would need an entire separate set of genes for standard gameplay and gameplay after a 'power-pellet' has been activated. As such, for the purposes of the investigation within the report, the 'power-pellets' have been replaced with standard pellets.

The possibility to acquire additional lives has also been removed, as this would only extend the duration of the simulations and provide no extra benefits to the project.

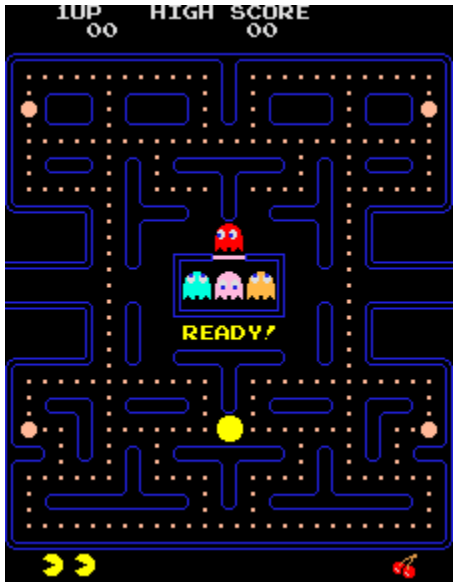

Figure 1: Screenshot of Original Pac-Man

1.5 Aims and Objectives

The project aims to produce a Pac-Man clone that utilizes a Genetic Algorithm to create enemy agents of increasing quality. Within this major aim of the project, there are several smaller objectives which needed to be achieved in order to successfully complete the project.

These objectives include:

- Deciding on and implementing an appropriate representation of the chromosomes and genes for the Pac-Man ghosts
- Rewriting the ghosts' behaviour to make them perform actions based on their genes
- Deciding on and implementing a relevant fitness function that will provide a good basis for scoring the ghosts' performance
- Implementing a simulator that will replicate the gameplay of the player in order to score the chromosomes for evolution without user intervention.
- Implementing a method for storing generations of chromosomes so that they may be used for gameplay against the user

1.6 Report Structure

Chapter 2 specifies the methods that were used in the investigation, including what technologies were used, how the simulator was architected and details about the genetic algorithm, such as; how the genes were encoded and the technique that was used for evolution of generations.

Chapter 3 encompasses the results that were obtained from the project. This section includes the results of the evolution of individuals and any peculiar or interesting behaviour that was exhibited by individuals after evolution through several generations.

Chapter 4 discussed the results of the project and the potential reasons why these results were discovered. This chapter also covers how these results compare to what the expected outcome was and how this reflects upon the investigation.

Chapter 5 evaluates the level of success of the investigation, any problems that were encountered along with their effect on the results, and how the project could have been improved upon.

Chapters 6 and 7 will contain the bibliography and appendices of the project to evidence and support any claims made within the report.

2 Method

2.1 Technologies Utilised

Only a small set of technologies were used within the project as it was heavily based upon the code which runs the game and Genetic Algorithm. Due to this, the majority of technologies that were used were in regards to the development and execution environment that was chosen for the project.

The chosen environment for development and execution was a computer with 8GB of RAM and an Intel Core i7 processor, running Arch Linux. This environment was chosen as it should be more than powerful enough for the project and Linux is the favourable Operating System for a lot of programmers as it is free, fast and provides a good environment for easily running programs via the command line and retrieving output whilst executing.

The programming language that was selected for the project was Python 2.7. This language was chosen as it is quite high level and most features can be implemented without requiring a large amount of code to be written. Additionally, it is the language that I have most experience with and therefore would be the best choice for me for implementation of complicated algorithms such as Evolutionary Algorithms. Although Python is a relatively slow language compared to others, due to the fact that it is usually interpreted rather than compiled, this should be offset by the use of a relatively powerful computer.

The other major technology that I took advantage of was the Python implementation of a Pac-Man clone by *David Reilly (2013)*. Due to this, the project also relies upon the 'PyGame' package for Python 2.7, as this package is used extensively throughout the core game code. As the project is mostly focused on the Genetic Algorithm that was employed, recreating the base game would have been an unnecessary amount of extra time and work. Considering the amount of Pac-Man clones that are available as open source projects, it was decided that building upon one of these would be the best approach to implementing the core game. As the chosen programming language was Python, the clone made by David Reilly was selected for use as it was the most comprehensive and simple to modify version of the game that I encountered available as an open source project.

A large quantity of the extra features was removed from David's implementation of the game as they were not needed. The maze editor is one example of this as the project only needed to focus on a single level that uses the standard map from the original Pac-Man. Other features that were removed were the ability to change the colours of the maze walls and pellets and the flashing of the level once completed. Because of these basic requirements, the first task that was undertaken was to strip out the unnecessary features of the game, and to separate the code into individual files and classes to make the code easier to work with and modify.

2.2 Genetic Encoding Scheme

As possibly the most important aspect of a genetic algorithm, a lot of thought went into creating a good representation of the chromosomes for individuals. Two attempts were made during the project to implement a decent representation for chromosomes. Both attempts involved the chromosome being represented by an array of genes, each containing a value for a given situation that signifies the direction to take (up, down, left, right).

2.2.1 First Attempt

The first attempt at an encoding scheme for the chromosomes involved implementing the situation based on the intersection that the individual has reached once a decision needs to be made. The genes were then arranged in the list such that each gene represents the direction to take at a single intersection.

After the initial implementation of this encoding scheme and some minor testing with the initial randomly generated population, it was decided that this representation was not appropriate for the problem. The reason for this was that it provided the individuals with too much knowledge of the map and did not accurately resemble the situation that the individuals encountered. From the start it was obvious that this encoding scheme was going to be a failure as almost all of the individuals would get stuck at particular areas on the map where they would simply move back and forth in one location. Although this would appear to be a good strategy, as the individuals were blocking off a particular area, such that Pac-Man would have to get caught by the ghost to collect the pellets within that area, it also seemed to be detrimental to the gameplay as it actually made it impossible for Pac-Man to complete a level. As a result of this, a new representation was needed.

2.2.2 Second Attempt

The second and final encoding scheme that was used in the project was based upon the one used for the 'Robby the Robot' simulation *(Chapter 9 in Mitchell, 2009)*. In this representation, the chromosome is based on actions and situations. As before, the actions are made up of the direction to take in a given situation, with the addition of a 'random' direction gene (up, down, left, right and random). The same as in the previous scheme, the chromosome is architected as an array of integers, representing the action to be taken, with each gene in the array representing a particular situation.

The major difference in the new representation is that the situation is made up of what can be seen in the following 3 grid squares in each direction and the 'majority direction' of Pac-Man. Each direction can be one of three states (wall, empty, pellets); if there is a wall in the direction, if there are pellets within the next 3 squares and if the next 3 squares are empty. Additionally, the ghosts are provided with information about the 'majority direction' of Pac-Man (up, down, left, right). For example, if Pac-Man is 6 squares to the right and 10 squares above, this is encoded as 'up', if Pac-Man is 5 squares to the left and 2 squares above, this is encoded as 'left' etc. In this representation, there is one gene in the array for each possible situation, totalling 324 genes (3^4 for each direction = 81 * 4 for each 'majority direction' of Pac-Man = 324). Some of these potential situations will never be encountered (such as a wall in every direction), but are left in for simplicity's sake.

2.3 Evolution Technique

The evolutionary algorithm is separated into a few smaller areas which will be explained in more detail in each of the following subsections. To begin with, the individuals in the population are given fitness scores based on how well they perform. Next, the individuals with higher scores are selected and put through the crossover in order to create new individuals to make up the population of the next generation. Finally, the new generation are mutated to prevent the occurrence of local optima.

2.3.1 Fitness Scores

During the project there fitness scores were modified a few times in order to provide sensible values to accurately distinguish the performance level of individuals. The major challenge that was encountered whilst determining how fitness would be scored was the method used to determine the performance of an individual. Without actually playing at least one level with each of the individuals, it would be almost impossible to estimate how they would perform during the game. Hence, before the fitness scores could be implemented, a gameplay simulator had to be coded in order to run a simulation level against the ghosts that would be used to determine their fitness. As the simulator ended up being a major chunk of the overall project, it will be detailed further in section '2.4 Simulator Architecture'.

The fitness of individuals is calculated during the simulation of the individual. The majority of an individual's fitness was the main thing that was modified throughout the project. Initially, it was implemented by awarding the individual 1 point every time it gets within its 'vision' range of 3 grid squares. However, the scores provided by this method were always very small integers and were quite close together for the entire population. In order to better distinguish the performance of each individual, a more granular fitness score was required.

The final variation that was used for simulation of fitness scores is based upon the sum of the 'Manhattan Distance' from the ghost to Pac-Man each time it moves to a new grid space. The concept of this fitness score was that each time a new grid square is reached, the fitness of the individual is increase by its 'Manhattan Distance' to Pac-Man (the sum of the distance vector components). For example, if Pac-Man was 4 squares to the right of the ghost and 7 squares above the ghost, its fitness would be increased by 11 (4 + 7). In order to keep these fitness scores fair, a counter of the total moves made by the ghost was maintained. At the end of the simulation, the fitness was calculated as the total of all the 'Manhattan Distances' calculated, divided by the total number of moves made during the simulation, plus any additional bonus fitness points gained/lost during the simulation.

Bonus fitness points were given to the individuals for performing certain actions. If the individual caught Pac-Man during the simulation, it was awarded a bonus of -25 points to its fitness score. However, if the ghost attempted to walk into a wall during the simulation, it was penalised with a +10 to its fitness score and a random direction was chosen in order to keep the gameplay moving.

These new fitness score calculations provided much better results that gave each individual a more unique fitness score. Additionally, due to the way the fitness scores are added up, within the project **smaller fitness scores are better**. As such, the individuals who obtain the best fitness scores and are more likely to have their genes used in the following generations, are those who spend more time close to Pac-Man or manage to catch him.

2.3.2 Selection

The selection method that was chosen for implementation in the project was a slightly modified 'roulette-wheel selection'. The major modification was that before using the scores to create the 'roulette-wheel', they were 'locally inversed', so that the smaller scores were converted into larger probabilities that are still proportionate to their fitness.

The basis for a 'roulette-wheel selection' is that each individual is assigned a probability based on its fitness score, which are then ordered, normalised and accumulated into a list that

acts as a 'roulette-wheel'. As the probabilities are all normalised between 0 and 1, due to the accumulation, the final 'section' of the wheel will end at 1, with all of the other sections spanning the area between 0 and 1. As such, a number can then be randomly generated between 0 and 1, and the individual whose section spans the area around the number is selected for crossover.

For the method used in the project, the individuals are first ordered by fitness, and only the best half of the individuals are used for creation of the wheel in order to create an aspect of elitism. After this, the scores are 'locally inversed' into what will be their relative probabilities by using the maximum score out of the group as the inverse point. To expand, the probability for each individual will be calculated as the maximum fitness of the group minus its fitness score. This inverses all of the scores so that lower scores present higher probabilities, with the highest score being converted into a probability of 0.

After the inversion, the rest of the algorithm for selection was implemented as standard for 'roulette-wheel' selection. The new probabilities are normalised between 0 and 1 and then accumulated in order to create the 'wheel', of which the section encompassing the randomly generated number is selected.

One final modification was implemented, which solves the two minor problems encountered by this approach. One issue is that the probability of the highest scoring individual is always 0 as this is the score used for the inversion point. The other issue is that due to floating point error, during accumulation of scores, the final score is often calculated as a value slightly below or above 1. In order to solve these problems, after accumulation, the final section of the wheel (representing the highest scoring individual) is forced to end at exactly 1. This resolves any issues that may be caused by randomly generating a value so close to 1 that it is outside the range of the wheel, whilst simultaneously providing the highest scoring individual with a small chance of being chosen for crossover.

2.3.3 Crossover

For the crossover component of the evolutionary algorithm, a simple 'two-point crossover' was implemented. This is very basic in that it simply selects two random positions within the chromosome and swaps the contents of the two parents between the two points in order to create the new children. The rest of the chromosomes outside of these 2 points remain the same.

2.3.4 Mutation

The mutation element of the algorithm is also very simple. A standard 'uniform' mutation was implemented, that selects a number of random genes based on the mutation rate (324 genes with a mutation rate of 5% means 16 genes should be mutated. 324 * 0.05 = 16), and then replaces these genes with randomly chosen genes from the selection of valid values.

2.4 Simulator Architecture

In order to generate fitness scores for the individuals in a population without having to endure the tedious process of manually playing against all of them, a simulator needed to be written for the Pac-Man game. The simulator's functionality from the user's perspective is relatively simple; it can be run just by executing the python program with command line arguments for the population size and number of generations to simulate. The simulator runs much faster than normal gameplay as its FPS (frames per second) limit has been removed, so it will run as fast as the computer will allow. Unfortunately however, the simulator only runs in a single thread and therefore gains no extra performance from processors with multiple cores. Additionally, several aspects of the game that are irrelevant for the simulator have been stripped out, such as the pause after being caught by a ghost, in order to increase the speed of simulation.

The evolutionary aspect of the simulator is not very complicated. It will retrieve the best generation that currently exists from the folder of serialised genomes, run the evolution algorithm on it to create the next generation, then simulate fitness scores for each individual (4 individuals at a time) before storing them in a new file. The simulator runs with a fixed mutation rate of 5%, which can be easily modified within the source code if necessary. For the genomes that were created during the investigation, a population of 100 individuals was used and a total of 250 generations were simulated.

The most challenging aspect of the simulator was how to simulate the movement of Pac-Man in a way that is similar to the behaviour a human player might exhibit. However, without first creating an intelligent agent for Pac-Man in order to collect pellets whilst simultaneously avoiding enemy ghosts, the approach used was to implement a search algorithm for Pac-Man to take the shortest route to the closest pellet.

At first, a depth-first search was implemented for the path finding of Pac-Man. However, this proved to be too slow and inefficient to serve the purposes of the simulator. Due to this a best-first search was used with a heuristic to estimate the cost from any potential node to a goal node. The costs used to determine the next path to expand when searching with the best-first algorithm was then made up of the existing actual cost of exploring each node plus the cost estimated to reach a goal node by the heuristic.

The heuristic used for the search algorithm was based on the minimum 'Manhattan Distance' to each of the remaining pellets in the maze. This heuristic provides an accurate estimate of the cost to reach a goal node as it will always be less than or equal to the actual cost to reach a goal node. Therefore, the heuristic is admissible and should provide a much more efficient method of finding a path for Pac-Man to follow. However, calculating the 'Manhattan Distance' to every pellet in the maze, from every grid space along each given path being explored proved to be too intensive to allow the simulator to run faster than standard gameplay. As such, the heuristic was cut down by only calculating the 'Manhattan Distances' to the first 25 or less pellets stored within the level.

Unfortunately, the path finding algorithm occasionally causes execution of the simulator to be paused when certain specific situations are encountered that cause several paths through the maze to have almost exactly the same number of moves to reach a pellet. This mostly happens when Pac-Man is at the top or bottom of the maze in the centre, when the last remaining pellets are also in the centre, but at the other end of the maze. Also, the simulator does not fully represent the behaviours that would be shown by a human player, as it makes no attempt to avoid ghosts, it simply focuses on collecting all of the pellets in the maze.

3 Results

3.1 Genetic Algorithm Results

The results obtained after simulating 250 generations with the genetic algorithm fell somewhat short of their expectations. However, the results still showed some improvement and began to display a couple of interesting behaviours. In order to analyse the data about the fitness scores of the genomes throughout generations, the files storing the genomes were stripped down to their fitness scores and converted into their best, worst and average fitness scores. This provides a decent amount of data to be able to graphically represent the progression of the genomes towards their desired goal. The best, worst and average fitness scores for each generation can be seen in section '3.1.1 Results Tables'.

3.1.1 Results Tables

Generation	Average	Best	Worst
0	176.33	5	473
1	150.4	-3	407
2	152.06	-11	417
3	150.85	-21	419
4	167.58	36	445
5	164.16	6	437
6	158.01	7	428
7	151.76	-22	359
8	152.86	1	361
9	164.32	9	475
10	173.07	24	438
11	128.39	2	301
12	154.59	-24	387
13	158.29	-14	381
14	170.05	-24	486
15	165.32	-15	402
16	167.3	29	462
17	160.64	13	499
18	153.51	-8	407
19	149.26	-11	328
20	159.18	-8	358
21	146.42	-34	300
22	159.75	5	545
23	160.21	-10	508
24	155.34	7	327
25	165.47	-12	373
26	150.69	-25	484
27	168.02	10	477
28	162.29	21	495
29	154.23	-14	338
30	162.08	1	502

Generation	Average	Best	Worst
31	161.23	-10	379
32	157.89	7	435
33	150.08	-15	418
34	165.22	26	396
35	149.77	16	397
36	152.8	-3	555
37	164.11	-10	402
38	148.03	13	398
39	161.18	14	366
40	144.92	-8	498
41	151.29	0	387
42	161.57	-15	382
43	165.43	15	414
44	152.91	1	487
45	155.9	-4	410
46	155.48	4	390
47	160.92	-15	419
48	152.9	-33	399
49	150.87	-23	338
50	145.46	11	315
51	155.27	3	421
52	153.77	11	491
53	172.99	4	425
54	174.55	29	414
55	156.77	14	397
56	165.21	22	477
57	154.23	10	386
58	160.92	3	360
59	166.99	28	490
60	162.53	0	404
61	157.38	-24	483

Generation	Average	Best	Worst
62	147.06	-23	430
63	163.69	12	390
64	164.11	11	412
65	132.83	2	399
66	113.61	-35	338
67	122.83	-33	318
68	125.16	-35	357
69	121.27	-29	358
70	122.63	-23	368
71	119.07	-34	350
72	118.57	-24	346
73	126.15	-51	462
74	121.35	-37	338
75	111.66	-35	378
76	113.97	-34	302
77	124.92	-14	376
78	125.48	-22	448
79	123.22	10	309
80	145.82	2	448
81	121.85	-7	413
82	133.63	-17	402
83	127.68	-8	371
84	134.45	1	412
85	115.1	-60	429
86	117.43	-13	352
87	113.15	-1	312
88	122.15	-25	325
89	124.93	-9	357
90	125.57	-8	343
91	119.44	-13	389
92	133.14	-9	298
93	105.34	-10	455
94	130.22	13	328
95	113.78	-35	337
96	121.14	-15	344
97	126.21	-12	368
98	104.6	-8	271
99	127.13	-5	366
100	118.12	-17	407
101	130.65	-10	407
102	109.72	-34	299

Generation	Average	Best	Worst
103	131.37	2	348
104	102.97	-33	450
105	98.03	-1	310
106	110.51	-9	421
107	90.94	-24	267
108	102.64	-11	360
109	106.8	-10	307
110	111.31	-13	330
111	90.04	-34	255
112	102.2	-34	327
113	109.82	-16	345
114	108.29	-14	328
115	101.97	-25	337
116	107.66	0	530
117	107.61	-34	291
118	114.82	-10	457
119	113.48	-16	357
120	115.47	-33	292
121	100.29	-8	317
122	111.63	-23	301
123	104.36	-36	347
124	95.3	0	286
125	92.9	-39	308
126	118.05	-14	367
127	95.96	-11	291
128	114.33	-60	349
129	92.36	-4	297
130	114.31	-13	324
131	90.43	-11	336
132	105.99	-23	366
133	97.53	-34	328
134	105.68	-4	401
135	106.58	-23	426
136	100.69	-22	300
137	93.4	-9	301
138	94.42	-24	384
139	106.71	-10	261
140	95.08	-11	386
141	104.17	-4	347
142	93.93	-7	266
143	100.53	-9	380

Generation	Average	Best	Worst
144	96.74	-25	345
145	118.95	-25	372
146	104.36	-12	360
147	108.2	-7	381
148	106.31	-57	360
149	104.13	-38	309
150	98.94	-9	297
151	111.53	-10	341
152	99.64	-37	269
153	101.98	-30	306
154	95.91	-27	318
155	98.18	-39	326
156	103.66	-24	338
157	103.53	-13	401
158	114.06	5	435
159	107.83	-9	404
160	100.77	-39	395
161	106.25	-35	395
162	98.28	-33	337
163	112.06	-28	345
164	100.06	-15	258
165	99.19	-4	429
166	97.12	-2	278
167	90.82	-9	401
168	101.61	-31	375
169	94.11	-24	382
170	86.14	-47	281
171	106.27	-39	444
172	104.04	-31	291
173	98.72	1	322
174	97.7	-14	294
175	104.83	-2	360
176	88.78	-7	260
177	85.48	-24	298
178	100.78	-33	345
179	98.67	-33	357
180	97.46	-34	339
181	103.82	-30	315
182	87.63	-34	297
183	89.05	-33	397

Generation	Average	Best	Worst
184	97.91	-9	349
185	99.1	-14	386
186	85.58	-34	258
187	102.54	-10	467
188	95.64	-24	324
189	100.57	-9	249
190	98.45	5	298
191	108.21	-34	328
192	91.09	-31	318
193	91.32	-5	320
194	99.48	0	267
195	75.4	-35	330
196	94.67	-8	281
197	82.41	-52	318
198	92.69	-61	265
199	83.51	-34	311
200	94.69	-9	387
201	82.01	-36	245
202	81.63	-11	265
203	90.15	-31	235
204	94.34	-8	330
205	79.23	-24	287
206	77.66	-33	282
207	99.25	-24	341
208	85.65	-6	221
209	93.87	-13	265
210	90.46	-59	295
211	96.9	-8	293
212	84.38	-25	280
213	83.68	-34	258
214	87.88	-16	238
215	83.69	-35	368
216	89.01	-13	267
217	87.58	-19	275
218	76.04	-32	235
219	84.05	-11	305
220	80.71	-33	327
221	96.87	-12	430
222	86.79	-33	270
223	77.8	-7	306

Generation	Average	Best	Worst
224	80.16	-9	215
225	74.14	-18	279
226	76.84	-22	231
227	83.65	-13	225
228	79.1	-60	214
229	75.6	-24	286
230	89.77	-23	292
231	89.99	-9	397
232	90.24	-22	242
233	73.13	-25	278
234	89.32	-58	392
235	92.19	-9	290
236	77.26	-34	313
237	89.51	-35	260

Generation	Average	Best	Worst
238	90.16	-11	257
239	82.7	-5	216
240	88.21	-14	308
241	99.25	-15	351
242	84.34	-23	297
243	87.96	2	337
244	88.1	-11	304
245	79.61	-18	376
246	83.63	-25	338
247	70.44	-14	311
248	76.35	-25	278
249	83.92	-23	341
250	72.96	-35	247

3.1.2 Results Graphs

As there is a large quantity of data in the results tables, several graphs have been made to illustrate how the fitness scores changed throughout generations.

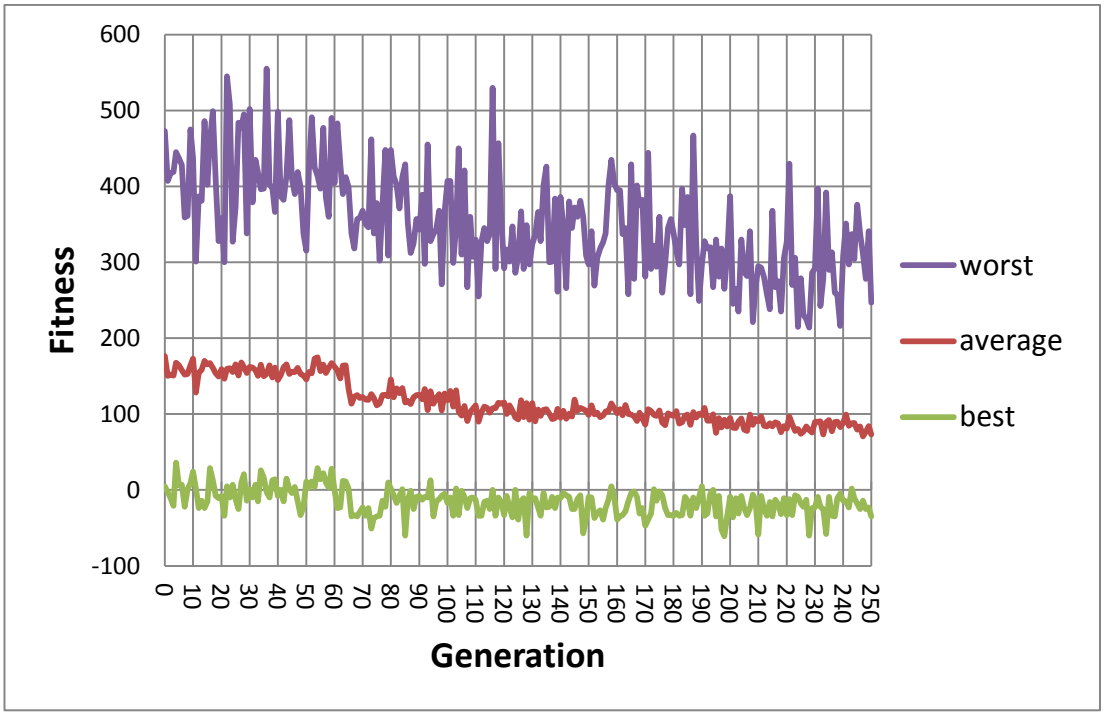

Figure 2: Average, best and worst fitness scores per generation

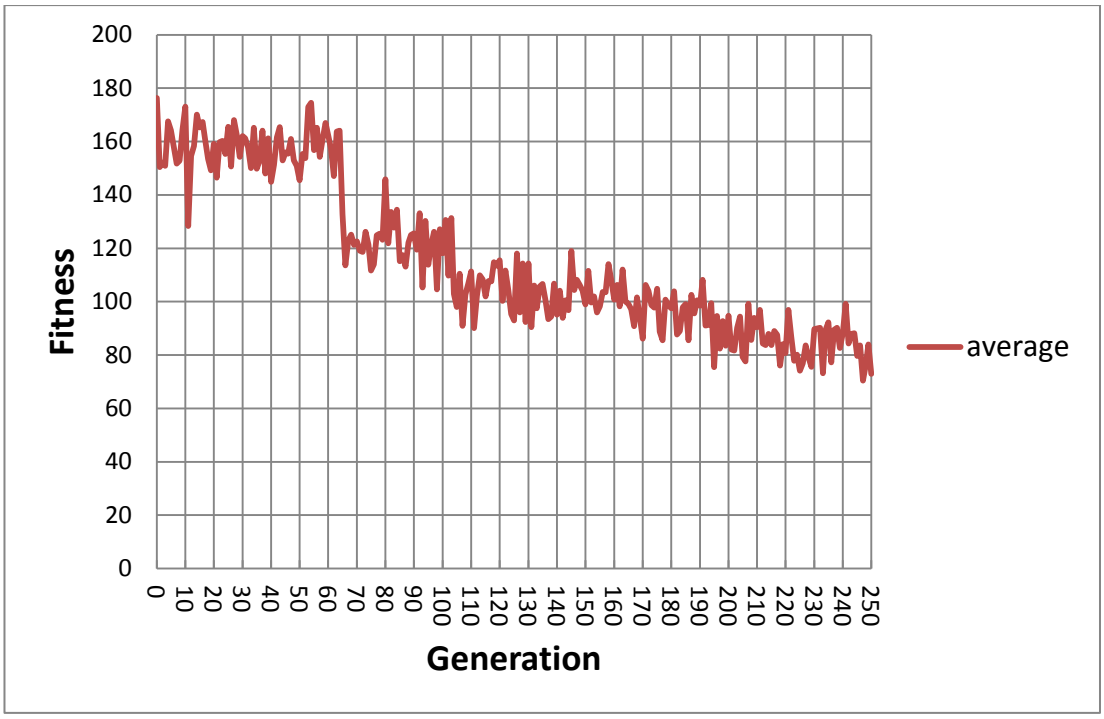

Figure 3: Average fitness score per generation

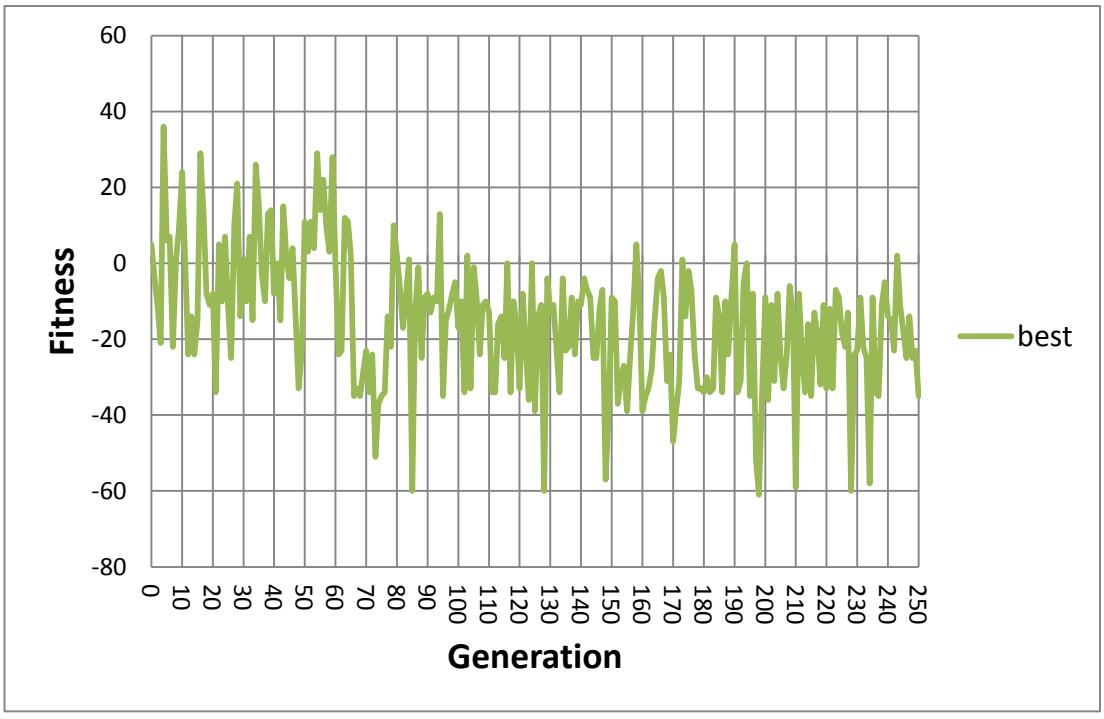

Figure 4: Best fitness score per generation

3.2 Behaviours Exhibited

There were only 2 noticeable behaviours that I observed beginning to emerge after the simulation of 250 generations of genomes. One of these behaviours was that at the start of a level, the ghosts appeared to rush towards certain areas of the maze. The other was that the ghosts would occasionally block off corridors of the maze by moving back and forth within them.

3.2.1 Level Start Rush

The first behaviour that I started to notice developing during the simulations was that the ghosts would often rush to particular places in the maze as soon as the level starts.

Figure 5 shows one example of this behaviour. We can see that the orange and purple ghosts have rushed to the corner to the top-left of their spawn box, the red ghost has rushed down the left-most vertical corridor and the blue ghost has rushed down the corridor to the right of the spawn box.

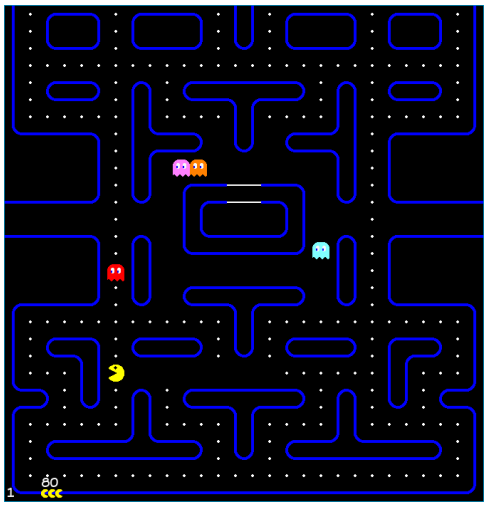

Figure 5: Rush example

All of the locations that were rushed in Figure 5 are areas that the simulated Pac-Man would frequent at the start of a level as he tries to make his way towards the top of the map. As a result of this, these are common positions for the simulated Pac-Man to get caught and as such, these behaviours have remained throughout the generations as they continue to provide good results in catching Pac-Man.

This behaviour was not expected, however it is a good tactic, as Pac-Man will always spawn near the bottom of the maze at the start of every level and so moving towards the bottom from the start is a decent way to get close to Pac-Man early in the game.

3.2.2 Corridor Blocking

The other behaviour that I noticed was that ghosts would occasionally move to a location near to Pac-Man, but instead of chasing him as was expected, they simply block off a corridor nearby and wait for him to go towards them. As Pac-Man gets near to them, it appears quite common for them to suddenly move towards him, which is likely to catch human players out.

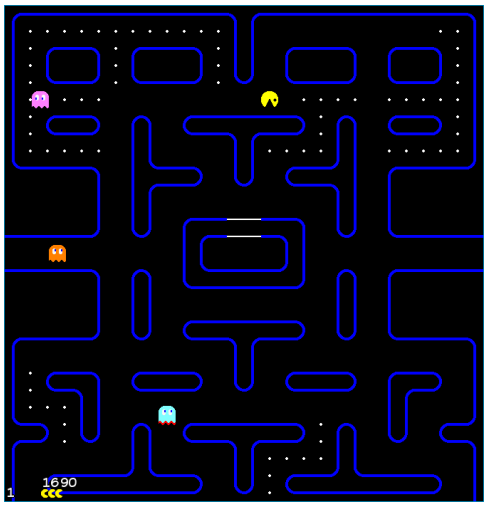

Figure 6: Blocking example 1

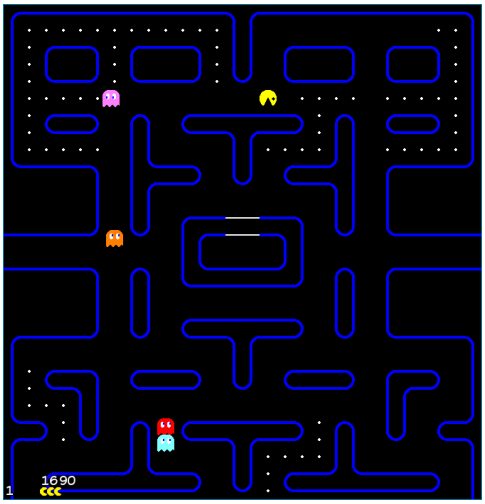

Figure 7: Blocking example 2

Between Figures 6 and 7 we can see that the purple ghost is continuously moving left and right within the corridor to the left of Pac-Man that still contains pellets. Additionally, these images show the red and blue ghosts at the bottom of the map exhibiting the same behaviour.

4 Discussion

The results within the tables in section '3.1.1 Results Tables' and the graph in Figure 2 both show that overall, the fitness scores of each generation appear to be decreasing. As a lower fitness score is better in this project, these results prove that the genetic algorithm functioned correctly and performed its purpose of increasing the quality of ghosts in increasing generations.

However, as we can establish by comparing the graph in Figure 3 and the graph in Figure 4, although the average fitness of the population was continually increasing, after about 100 generations, the best individuals' fitness was staying roughly the same for the remaining generations. This would imply that there has been a floor effect on the fitness scores of the individuals. I believe the reason for this floor effect to be that the ghosts are simulated in groups of 4. This means that when one ghost catches Pac-Man, the others have missed their chance to catch him and as all 4 ghosts cannot catch Pac-Man within one simulation, regardless of whether they are capable of doing so, when one ghost's fitness is decreased by catching Pac-Man, it will inhibit the others in the simulation from decreasing their fitness by catching him.

This essentially causes the ghosts to be in constant competition with one another to catch Pac-Man first. As a result of this, the ability to catch Pac-Man becomes less relevant to the fitness than **how fast** the individual is able to catch him. This theory is also supported by the behaviours detailed in section '3.2.1 Level Start Rush' and shown in Figure 5.

In addition to this, the quality of the individuals generated is also diminished by the poor level of difficulty that the best-first algorithm for Pac-Man presents. Quite often during the simulations I observed individuals getting better fitness scores by chance, because instead of actually catching Pac-Man themselves, Pac-Man would often run into them whilst on his way to the nearest pellet. This is an example of how the simulated Pac-Man is not very representative of human players' behaviour as it is focused entirely on collecting all of the pellets, whereas human players often prioritise avoiding the ghosts above collecting pellets.

Another issue with the simulator is that Pac-Man often seems to get caught by the ghosts in the areas surrounding their spawn box, which is an area that is usually avoided by most human players as it is fairly easy to get caught out in this area. The best way to improve the simulator and resolve these issues would be to add extra functionality to the Pac-Man agent to make it more intelligent. My two proposed changes would be to implement a new behaviour, in which once Pac-Man gets close to one of the ghosts, he ignores the pellets and runs away from them. Also, I would heavily increase the cost of the path finding algorithm using the grid spaces surrounding the ghosts' spawn box. In theory, these changes should imitate human players' behaviour better than utilising just a simple search algorithm.

In order to further test these claims, changes would need to be made to the behaviour of the Pac-Man simulator and a larger number of generations would need to be simulated to find out if these correlations and behaviours continue to occur. Unfortunately, due to the time that it takes to simulate an entire generation (estimated around 10 minutes for a population of 100 individuals), it was not possible for me to simulate more than 250 generations during the course of the project.

5 Evaluation

In conclusion, I feel that the project overall was a success, even though the results gained were not as prominent as I had hoped for. I believe that the methods I used were all necessary and well thought out and implemented, although the simulator could have been improved upon. However, the drawbacks of the current simulator implementation were not obvious until all of the data from it had been collected and analysed. The results that I obtained were relatively conclusive and showed a good correlation between fitness scores and the number of generations evolved through.

I believe the main reason that the project did not meet the expectations that I had for it was because of the shortcomings in the simulator. If I were to re-create the project again, I would definitely have implemented the changes to the simulator that I mentioned in section '4. Discussion'. If possible, another improvement that I would have made to the project would be to multi-thread the simulator so that one simulation of the game could run on each core, as this would drastically reduce the time required to simulate generations of individuals. This would have allowed me to collect a lot more data within the time that I had, which would have allowed me to closer examine the results and make more solid claims about the behaviours that the individuals exhibited.

I am happy with what I have achieved during the course of the project. Although it was not a strong success, I believe it was a good first attempt at something that to my knowledge has not been extensively researched before. As such, I am glad to have obtained the results that I did and was able to reflect upon.

I feel that I could have managed my time better during the course of the project, as there were a couple of areas which upon implementing, took a lot more time and required a lot more in-depth thought than I had initially anticipated. I should have accounted for extra time when managing my schedule to be able to deal with any unexpected nuances like these.

The project has been quite an experience to undertake and I have learnt a lot in regards to the depth of my knowledge in certain areas. I am also glad that I attempted a project as unusual and unique as I did, as it has allowed me to experiment a lot with the project until I was happy with the outcome of it, as the overall aim of the project was rather broad and did not present me with any incredibly specific requirements that needed to be met.

To summarise, even though the project did not meet my possibly excessive expectations, I am happy with the outcome and enjoyed the overall experience of the project. Choosing a subject area that was interesting to me definitely made me more motivated to complete the project and allowed me to investigate something that I would enjoy.

6 Bibliography

Mitchell, M. (2009). *Complexity: A guided tour*. Oxford University Press.

Reilly, D. (2013). *Pacman by David Reilly*. Retrieved from https://github.com/greyblue9/pacman-python

Pac-Man still going strong at 30. (2010, May 22). Retrieved from http://www.upi.com/Entertainment_News/2010/05/22/Pac-Man-still-going-strong-at-30/UPI-74821274544243/

Gallagher, M., & Ledwich, M. (2007, April). Evolving pac-man players: Can we learn from raw input?. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on* (pp. 282-287). IEEE.

Alhejali, A. M., & Lucas, S. M. (2010, September). Evolving diverse Ms. Pac-Man playing agents using genetic programming. In *IEEE Symposium on Computational Intelligence and Games* (pp. 53-60).

7 Appendix A - Final Code

7.1 pacman.pyw

```
#! /usr/bin/python

# pacman.pyw
# By Kai Leddy
# Original Pacman code by David Reilly
# With Modifications by Andy Sommerville

import sys, getopt

def print_help():
    print 'pacman.pyw [-h|--help] [-g|--generations <number>] [-p|--
population <number>]'
    print 'NOTE: population MUST be a multiple of 4!'
    sys.exit(2)

# if there are no command line arguments
if len(sys.argv) < 2:
    # run the game for the user to play
    import src.core
    src.core.run_game()
else:
    # define command line arguments
    short = 'hg:p:'
    full = ['help', 'generations=', 'population=']
    generations = 0
    population = 0
    # get arguments and print help if any problems occur
    try:
        opts, args = getopt.getopt(sys.argv[1:], short, full)
    except getopt.GetoptError:
        print_help()

    # set variables based on the arguments provided
    for opt, arg in opts:
        if opt in ['-h', '--help']:
            print_help()
        elif opt in ['-g', '--generations']:
            if arg.isdigit():
                generations = int(arg)
        elif opt in ['-p', '--population']:
            if arg.isdigit():
                population = int(arg)

    # if any parameters are 0 or population is not a multiple of 4
    if generations == 0 or population == 0 or population % 4 != 0:
        print_help()
    else:
        # run the simulator with the given arguments
        import src.simulator
        src.simulator.simulate(generations, population)
        sys.exit(0)
```

7.2 src/__init__.py

```
# List of modules in this package
__all__ = ["helper", "ghost", "pacman", "level", "game"]
```

7.3 src/core.py

```
1.  import pygame, sys, os, random
2.  from pygame.locals import *
3.  from src import *
4.
5.  # initialise pygame modules and setup screen
6.  clock = pygame.time.Clock()
7.  pygame.init()
8.  window = pygame.display.set_mode((1, 1))
9.  pygame.display.set_caption("Pacman")
10. screen = pygame.display.get_surface()
11. img_Background = helper.load_background()
12.
13. # constants for tile size, wait time and ghost colours
14. TILE_WIDTH = TILE_HEIGHT = 24
15. WAIT_TIME = 90
16. GHOST_COLORS = {}
17. GHOST_COLORS[0] = (255, 0, 0, 255)
18. GHOST_COLORS[1] = (255, 128, 255, 255)
19. GHOST_COLORS[2] = (128, 255, 255, 255)
20. GHOST_COLORS[3] = (255, 128, 0, 255)
21.
22. # constants for the score's position
23. SCORE_XOFFSET=50 # pixels from left edge
24. SCORE_YOFFSET=34 # pixels from bottom edge (to top of score)
25. SCORE_COLWIDTH=13 # width of each character
26.
27. # constants to easily reference tiles
28. ALL_TILES = {}
29. ALL_TILES[0] = None
30. ALL_TILES[1] = TILE_PELLET = (1, helper.load_tile("pellet.gif"))
31. ALL_TILES[2] = TILE_START = (2, None)
32. ALL_TILES[3] = TILE_DOOR = (3, None)
33. ALL_TILES[4] = TILE_GHOST_DOOR = (4, helper.load_tile("ghost_door.gif"))
34. ALL_TILES[5] = TILE_INTERSECTION = (5, None)
35. ALL_TILES[6] = TILE_INTERSECTION_EMPTY = (6, None)
36. # ghost tiles
37. ALL_TILES[10] = TILE_GHOST_INKY = (10, None)
38. ALL_TILES[11] = TILE_GHOST_BLINKY = (11, None)
39. ALL_TILES[12] = TILE_GHOST_PINKY = (12, None)
40. ALL_TILES[13] = TILE_GHOST_CLYDE = (13, None)
41. ALL_GHOST_TILES = [TILE_GHOST_INKY, TILE_GHOST_BLINKY,
42.                     TILE_GHOST_PINKY, TILE_GHOST_CLYDE]
43. # wall tiles
44. ALL_TILES[20] = TILE_WALL_H = (20, helper.load_tile("wall_horizontal.gif"))
45. ALL_TILES[21] = TILE_WALL_V = (21, helper.load_tile("wall_vertical.gif"))
46. ALL_TILES[22] = TILE_WALL_TL = (22, helper.load_tile("wall_top_left.gif"))
47. ALL_TILES[23] = TILE_WALL_TR = (23, helper.load_tile("wall_top_right.gif"))
48. ALL_TILES[24] = TILE_WALL_BL = (24, helper.load_tile("wall_bottom_left.gif"
    ))
49. ALL_TILES[25] = TILE_WALL_BR = (25, helper.load_tile("wall_bottom_right.gif
    "))
50. ALL_WALL_TILES = [TILE_WALL_H, TILE_WALL_V, TILE_WALL_TL, TILE_WALL_TR,
51.                   TILE_WALL_BL, TILE_WALL_BR, TILE_GHOST_DOOR]
52.
53. #      ____________________
54. # ___/  main code block  \_____________________________________________
    ____
55.
56. # create the pacman and ghosts
57. player = pacman.Pacman()
```

```
58. ghosts = {}
59. for i in range(4):
60.     ghosts[i] = ghost.Ghost(i)
61.
62. # create game and level objects and load first level
63. thisGame = game.Game()
64. thisLevel = level.Level()
65. thisLevel.loadLevel(thisGame.getLevelNum())
66.
67. # setup display options
68. window = pygame.display.set_mode(thisGame.screenSize,
69.         pygame.HWSURFACE | pygame.DOUBLEBUF)
70.
71. # method to continuously run the game
72. def run_game():
73.     while True:
74.         # process any user input and increase mode timer
75.         thisGame.checkInputs()
76.         thisGame.modeTimer += 1
77.
78.         # handle the current game mode
79.         if thisGame.mode == game.MODE_NORMAL:
80.             # move all characters
81.             player.move()
82.             for i in range(4):
83.                 ghosts[i].move()
84.         elif thisGame.mode == game.MODE_WAIT_HIT:
85.             # after waiting, restart the level
86.             if thisGame.modeTimer == WAIT_TIME:
87.                 thisLevel.restart()
88.                 # reduce player lives and change mode
89.                 thisGame.lives -= 1
90.                 if thisGame.lives == 0:
91.                     thisGame.setMode(game.MODE_GAME_OVER)
92.                 else:
93.                     thisGame.setMode(game.MODE_WAIT_START)
94.         elif thisGame.mode == game.MODE_WAIT_START:
95.             # after waiting, start the game again
96.             if thisGame.modeTimer == WAIT_TIME:
97.                 thisGame.setMode(game.MODE_NORMAL)
98.                 player.velX = player.speed
99.         elif thisGame.mode == game.MODE_WAIT_FINISH:
100.                # after waiting, show the game over screen
101.                if thisGame.modeTimer == WAIT_TIME:
102.                    thisGame.setMode(game.MODE_GAME_OVER)
103.
104.            # move the screen if necessary and redraw everything
105.            thisGame.smartMoveScreen()
106.            screen.blit(img_Background, (0, 0))
107.
108.            # only draw the map and characters if not game over
109.            if not thisGame.mode == game.MODE_GAME_OVER:
110.                thisLevel.drawMap()
111.                for i in range(4):
112.                    ghosts[i].draw()
113.                player.draw()
114.
115.            # draw the score, flip the buffer and wait for the next tick

116.            thisGame.drawScore()
117.            pygame.display.flip()
118.            clock.tick (60)
```

7.4 src/game.py

```
1.  import pygame, sys
2.  from pygame.locals import *
3.  import core, genetic, helper
4.
5.  # game mode constants
6.  MODE_NORMAL      = 1
7.  MODE_GAME_OVER   = 2
8.  MODE_WAIT_HIT    = 3
9.  MODE_WAIT_START  = 4
10. MODE_WAIT_FINISH = 5
11.
12. class Game():
13.     def __init__(self):
14.         # set initial variables
15.         self.levelNum = 0
16.         self.score = 0
17.         self.lives = 3
18.         self.mode = 0
19.         self.modeTimer = 0
20.         self.setMode(MODE_GAME_OVER)
21.         # variable to store single key buffer
22.         self.buffered_key = None
23.
24.         # camera variables
25.         self.screenPixelPos = (0, 0) # absolute x,y position of the screen
    from the upper-left corner of the level
26.         self.screenNearestTilePos = (0, 0) # nearest-
    tile position of the screen from the UL corner
27.         self.screenPixelOffset = (0, 0) # offset in pixels of the screen fr
    om its nearest-tile position
28.         self.screenTileSize = (28, 29)
29.         self.screenSize = (self.screenTileSize[0] * core.TILE_WIDTH,
30.                            self.screenTileSize[1] * core.TILE_HEIGHT)
31.
32.         # load the numerical display digits
33.         self.digit = {}
34.         for i in range(10):
35.             self.digit[i] = helper.load_text(str(i) + ".gif")
36.
37.         # load the images for lives, game over and ready
38.         self.imLife = helper.load_text("life.gif")
39.         self.imGameOver = helper.load_text("gameover.gif")
40.         self.imReady = helper.load_text("ready.gif")
41.
42.     # method to start a new game with the given individuals
43.     def startNewGame(self, individuals):
44.         # reset level variables
45.         self.levelNum = 1
46.         self.score = 0
47.         self.lives = 3
48.         # load the level and get ready to start the game
49.         self.setMode(MODE_WAIT_START)
50.         core.thisLevel.loadLevel(core.thisGame.getLevelNum())
51.
52.         # setup the genomes for the ghosts
53.         core.ghosts[0].set_individual(individuals[0])
54.         core.ghosts[1].set_individual(individuals[1])
55.         core.ghosts[2].set_individual(individuals[2])
56.         core.ghosts[3].set_individual(individuals[3])
57.
58.     # method to increase the score by a given amoount
59.     def addToScore(self, amount):
60.         self.score += amount
```

```
61.
62.     # method to draw the majority of the user interface
63.     def drawScore(self):
64.         self.drawNumber(self.score,
65.                 (core.SCORE_XOFFSET, self.screenSize[1] -
    core.SCORE_YOFFSET))
66.         # draw an icon for each life
67.         for i in range(self.lives):
68.             core.screen.blit(self.imLife,
69.                     (34 + i * 10 + 16, self.screenSize[1] - 18))
70.         # draw the game over or ready image if necessary
71.         if self.mode == MODE_GAME_OVER:
72.             core.screen.blit(self.imGameOver,
73.                     (self.screenSize[0] / 2 -
    (self.imGameOver.get_width()/2),
74.                      self.screenSize[1] / 2 -
    (self.imGameOver.get_height()/2)))
75.         elif self.mode == MODE_WAIT_START:
76.             core.screen.blit(self.imReady,
77.                     (self.screenSize[0] / 2 -
    20, self.screenSize[1] / 2 + 12))
78.         # draw the level number
79.         self.drawNumber (self.levelNum, (0, self.screenSize[1] - 20) )
80.
81.     # method to draw a number to the screen
82.     def drawNumber(self, number, (x, y)):
83.         strNumber = str(number)
84.
85.         for i in range(len(str(number))):
86.             iDigit = int(strNumber[i])
87.             core.screen.blit(self.digit[iDigit], (x + i * core.SCORE_COLWID
    TH, y))
88.
89.     # method to move the screen up or down as needed
90.     def smartMoveScreen(self):
91.
92.         possibleScreenX = core.player.x -
    self.screenTileSize[1] / 2 * core.TILE_WIDTH
93.         possibleScreenY = core.player.y -
    self.screenTileSize[0] / 2 * core.TILE_HEIGHT
94.
95.         if possibleScreenX < 0:
96.             possibleScreenX = 0
97.         elif possibleScreenX > core.thisLevel.lvlWidth * core.TILE_WIDTH -
    self.screenSize[0]:
98.             possibleScreenX = core.thisLevel.lvlWidth * core.TILE_HEIGHT -
    self.screenSize[0]
99.
100.                if possibleScreenY < 0:
101.                    possibleScreenY = 0
102.                elif possibleScreenY > core.thisLevel.lvlHeight * core.TILE_
    WIDTH - self.screenSize[1]:
103.                    possibleScreenY = core.thisLevel.lvlHeight * core.TILE_H
    EIGHT - self.screenSize[1]
104.
105.                core.thisGame.moveScreen((possibleScreenX, possibleScreenY))

106.
107.            # method to move the screen
108.            def moveScreen(self, (newX, newY)):
109.                self.screenPixelPos = (newX, newY)
110.            # nearest-tile position of the screen from the UL corner
111.                self.screenNearestTilePos = (int(newY / core.TILE_HEIGHT),
112.                        int(newX / core.TILE_WIDTH))
113.                self.screenPixelOffset = \
114.                    (newX - self.screenNearestTilePos[1]*core.TILE_WIDTH,
```

```
115.                     newY - self.screenNearestTilePos[0]*core.TILE_HEIGHT)
116.
117.             def getScreenPos(self):
118.                 return self.screenPixelPos
119.
120.             def getLevelNum(self):
121.                 return self.levelNum
122.
123.             # method to setup the next level
124.             def setNextLevel(self):
125.                 # load the next level
126.                 self.levelNum += 1
127.                 self.setMode(MODE_WAIT_START)
128.                 core.thisLevel.loadLevel( core.thisGame.getLevelNum() )
129.                 # reset player variables
130.                 core.player.velX = 0
131.                 core.player.velY = 0
132.                 core.player.anim_pacmanCurrent = core.player.anim_pacmanS
133.
134.
135.             # method to change the mode and reset the timer
136.             def setMode(self, newMode):
137.                 self.mode = newMode
138.                 self.modeTimer = 0
139.
140.             def get_individuals(self):
141.                 # return the 4 best individuals so far
142.                 return genetic.get_best_generation(4)[:4]
143.
144.             def checkInputs(self):
145.                 # loop through all game events
146.                 for event in pygame.event.get():
147.                     # handle the quit event
148.                     if event.type == QUIT:
149.                         sys.exit(0)
150.                     # handle key press events
151.                     elif event.type == KEYDOWN:
152.                         # if game over and key pressed was return
153.                         if self.mode == MODE_GAME_OVER and event.key == K_RE
    TURN:
154.                             self.startNewGame(self.get_individuals())
155.                         # if in normal game state, add any keys to buffer
156.                         elif self.mode == MODE_NORMAL and event.key in [K_RI
    GHT, K_LEFT, K_UP, K_DOWN]:
157.                             self.buffered_key = event.key
158.
159.                 # handle any key in the buffer
160.                 if self.buffered_key == pygame.K_RIGHT:
161.                     if not (core.player.velX == core.player.speed and core.p
    layer.velY == 0) and not core.thisLevel.checkIfHitWall((core.player.x + cor
    e.player.speed, core.player.y), (core.player.nearestRow, core.player.neares
    tCol)):
162.                         core.player.velX = core.player.speed
163.                         core.player.velY = 0
164.                         self.buffered_key = None
165.
166.                 elif self.buffered_key == pygame.K_LEFT:
167.                     if not (core.player.velX == -
    core.player.speed and core.player.velY == 0) and not core.thisLevel.checkIf
    HitWall((core.player.x -
     core.player.speed, core.player.y), (core.player.nearestRow, core.player.ne
    arestCol)):
168.                         core.player.velX = -core.player.speed
169.                         core.player.velY = 0
170.                         self.buffered_key = None
171.
```

```
172.                elif self.buffered_key == pygame.K_DOWN:
173.                    if not (core.player.velX == 0 and core.player.velY == co
    re.player.speed) and not core.thisLevel.checkIfHitWall((core.player.x, core
    .player.y + core.player.speed), (core.player.nearestRow, core.player.neares
    tCol)):
174.                        core.player.velX = 0
175.                        core.player.velY = core.player.speed
176.                        self.buffered_key = None
177.
178.                elif self.buffered_key == pygame.K_UP:
179.                    if not (core.player.velX == 0 and core.player.velY == -
    core.player.speed) and not core.thisLevel.checkIfHitWall((core.player.x, co
    re.player.y -
     core.player.speed), (core.player.nearestRow, core.player.nearestCol)):
180.                        core.player.velX = 0
181.                        core.player.velY = -core.player.speed
182.                        self.buffered_key = None
```

7.5 src/genetic.py

```
###
# genomes are made up of one of the following actions
# 0 = RANDOM
# 1 = UP
# 2 = DOWN
# 3 = LEFT
# 4 = RIGHT
###
import random, cPickle
import helper, simulator

MUTATION_RATE = 0.05

# method to serialize a generation
def pickle_generation(generation):
    with helper.next_gen_file() as gen_file:
        cPickle.dump(sorted(generation), gen_file)

# method to return the best generation so far
# or generate a new population if none currently exist
def get_best_generation(fallback_population):
    # if no generations have been simulated yet
    if not helper.any_gen_files():
        # create a new population as a fallback
        return population(fallback_population)

    # unpickle the best generation from the file
    with helper.best_gen_file() as gen_file:
        return cPickle.load(gen_file)

# method to convert the numerical direction into a usable tuple
def gene_to_direction(gene):
    if gene == 0:
        return helper.random_direction()
    elif gene == 1:
        return (0,-1)
    elif gene == 2:
        return (0,1)
    elif gene == 3:
        return (-1,0)
    elif gene == 4:
        return (1,0)

# method to pick a random gene to be used
def random_gene():
    return random.randint(0,4)

# method to generate a valid individual, without a fitness score
def individual():
    # randomly choose actions for all 324 possible situations
    return (None, [random_gene() for x in xrange(324)])

# method to create a population of individuals
def population(count):
    return [individual() for x in xrange(count)]

# method to normalise a list of individuals
def normalise(population):
    # calculate the total fitness of the parents
    total = sum([x[0] for x in population])
    # normalise each fitness score
    normalised = [(float(x[0]) / total, x[1]) for x in population]
    return normalised
```

```
64.
65. # method to generate a roulette wheel for selection
66. def roulette_wheel(all_parents):
67.     # split the list to only use the best half as parents
68.     parents = all_parents[:len(all_parents)/2]
69.     # get the maximum of all fitness scores to use for inverse
70.     maximum = max([x[0] for x in parents])
71.     # calculate the inverse of each fitness relative to the maximum
72.     local_inverse = [(maximum - x[0], x[1]) for x in parents]
73.     # normalise each inversed fitness score
74.     normalised = normalise(local_inverse)
75.     # sort the parents in descending order and accumulate scores
76.     wheel = sorted(normalised, reverse=True)
77.     acc = 0
78.     for i in xrange(len(wheel)):
79.         acc += wheel[i][0]
80.         wheel[i] = (acc, wheel[i][1])
81.     # force the last score to be exactly 1
82.     wheel[-1] = (1.0, wheel[-1][1])
83.     return wheel
84.
85. # method to select a parent from the roulette
86. def roulette_select(roulette):
87.     # generate a random float
88.     rand = random.random()
89.     # loop through each item and select the first with a larger value
90.     for each in roulette:
91.         if each[0] > rand:
92.             return each[1]
93.
94. # method to select parents to be used for evolution
95. def select(parents):
96.     roulette = roulette_wheel(parents)
97.     for i in xrange(len(parents) / 2):
98.         # make sure 2 parents that are different are selected
99.         male = None
100.                female = None
101.                while male == female:
102.                    # select a male and female
103.                    male = roulette_select(roulette)
104.                    female = roulette_select(roulette)
105.                # yield the 2 parents to be used
106.                yield male, female
107.
108.        # method to crossover genomes from parents into a child
109.        def crossover(male, female):
110.            # create 2 copies to use as children
111.            child_1 = male[:]
112.            child_2 = female[:]
113.            # select 2 points to use for crossover
114.            p1 = random.randint(0, len(male) - 1)
115.            p2 = random.randint(0, len(male) - 1)
116.            # crossover the 2 points selected
117.            child_1[p1:p2], child_2[p1:p2]  = child_2[p1:p2], child_1[p1:p2]
118.            # return the children with no fitness score attached
119.            return [(None, child_1), (None, child_2)]
120.
121.        # method to mutate the genomes of some of the children
122.        def mutate(children):
123.            # get the number of genes to mutate
124.            num_genes = len(children[0][1])
125.            num_mutate = int(num_genes * MUTATION_RATE)
126.            # loop through all children to be mutated
127.            for child in children:
128.                # select num gene locations to mutate
```

```
129.                locations = random.sample(xrange(num_genes), num_mutate)
130.                # for each chosen location, replace with a random gene
131.                for each in locations:
132.                    child[1][each] = random_gene()
133.
134.        # method to evolve a full generation
135.        def evolve(parents):
136.            children = []
137.            # continue selecting parents for a full generation
138.            for male, female in select(parents):
139.                # crossover the parents and add the child to the list
140.                children.extend(crossover(male, female))
141.            # mutate some of the children and return the new generation
142.            mutate(children)
143.            return children
```

7.6 src/ghost.py

```
1.  import pygame, os, random
2.  import helper, core, game, genetic
3.
4.  class Ghost():
5.      def __init__(self, ghostID):
6.          # gameplay variables for ghosts
7.          self.x = 0
8.          self.y = 0
9.          self.velX = 0
10.         self.velY = 0
11.         self.speed = 1
12.         self.nearestY = 0
13.         self.nearestX = 0
14.         # setup ghost specific variables
15.         self.id = ghostID
16.         self.genome = []
17.         self.fitness = 0
18.         self.fitness_bonus = 0
19.         self.fitness_count = 0
20.         self.leftSpawn = False
21.         # set the animation and home variables
22.         self.homeX = 0
23.         self.homeY = 0
24.         self.animFrame = 0
25.         self.animDelay = 0
26.
27.         # load the frames in the animation
28.         self.anim = {}
29.         for i in range(4):
30.             self.anim[i] = helper.load_sprite("ghost_" + str(i + 1) + ".gif
    ")
31.             # change the ghost color in each frame
32.             for y in xrange(core.TILE_HEIGHT):
33.                 for x in xrange(core.TILE_WIDTH):
34.                     # if the pixel is red
35.                     if self.anim[i].get_at((x, y)) == (255, 0, 0, 255):
36.                         # change the pixel to match the ghosts color
37.                         self.anim[i].set_at((x, y), core.GHOST_COLORS[self.
    id])
38.
39.     # method to return the final fitness score
40.     def get_fitness(self):
41.         return (self.fitness / self.fitness_count) + self.fitness_bonus
42.
43.     # method to apply the bonus for catching pacman
44.     def hit_pacman(self):
45.         self.fitness_bonus -= 25
46.
47.     # method to setup the ghost to mimic and individual
48.     def set_individual(self, individual):
49.         self.genome = individual[1]
50.         self.fitness = 0
51.         self.fitness_bonus = 0
52.         self.fitness_count = 0
53.
54.     # method to reset the ghost
55.     def reset(self):
56.         self.x = self.homeX
57.         self.y = self.homeY
58.         self.velX = 0
59.         self.velY = 0
60.         self.state = 1
61.         self.speed = 1
```

```
62.         self.leftSpawn = False
63.
64.     # method to draw the ghost
65.     def draw(self):
66.         # dont draw anything if the game is over
67.         if core.thisGame.mode == game.MODE_GAME_OVER:
68.             return False
69.
70.         # ghost eyes --
71.         for y in range(6,12,1):
72.             for x in [5,6,8,9]:
73.                 self.anim[self.animFrame].set_at((x, y), (0xf8,0xf8,0xf8,25
    5))
74.                 self.anim[self.animFrame].set_at((x+9, y), (0xf8,0xf8,0xf8,
    255))
75.
76.         if core.player.x > self.x and core.player.y > self.y:
77.             #player is to lower-right
78.             pupilSet = (8,9)
79.         elif core.player.x < self.x and core.player.y > self.y:
80.             #player is to lower-left
81.             pupilSet = (5,9)
82.         elif core.player.x > self.x and core.player.y < self.y:
83.             #player is to upper-right
84.             pupilSet = (8,6)
85.         elif core.player.x < self.x and core.player.y < self.y:
86.             #player is to upper-left
87.             pupilSet = (5,6)
88.         else:
89.             pupilSet = (5,9)
90.
91.         for y in range(pupilSet[1], pupilSet[1] + 3, 1):
92.             for x in range(pupilSet[0], pupilSet[0] + 2, 1):
93.                 self.anim[self.animFrame].set_at((x, y), (0, 0, 255, 255))

94.                 self.anim[self.animFrame].set_at((x+9, y), (0, 0, 255, 255)
    )
95.         # -- end ghost eyes
96.
97.         # draw regular ghost (this one)
98.         core.screen.blit(self.anim[self.animFrame],
99.                 (self.x - core.thisGame.screenPixelPos[0],
100.                         self.y - core.thisGame.screenPixelPos[1]))
101.
102.                # don't animate ghost if the level is complete
103.                if core.thisGame.mode == game.MODE_WAIT_FINISH:
104.                    return False
105.
106.                # increase delay and change frame if necessary
107.                self.animDelay += 1
108.                if self.animDelay == 4:
109.                    self.animFrame += 1
110.                    # wrap to beginning
111.                    if self.animFrame == 4:
112.                        self.animFrame = 0
113.                    # restart the delay
114.                    self.animDelay = 0
115.
116.            # method to force ghosts to leave their spawn
117.            def leave_spawn(self):
118.                # if the ghost is lined up with the grid again
119.                if (self.x % core.TILE_WIDTH) == 0 and (self.y % core.TILE_H
    EIGHT) == 0:
120.                    # if the ghost is out of the spawn
121.                    if self.y / core.TILE_HEIGHT == 11:
122.                        self.leftSpawn = True
```

```
123.                         self.velY = 0
124.                         self.velX = self.speed * random.choice([1, -1])
125.                     # if this is ghost one or two, simply move up until out
    of spawn
126.                     elif self.id == 1 or self.id == 2:
127.                         self.velY = -self.speed
128.                     # for other ghosts, move left/right, then move up
129.                     elif core.ghosts[1].leftSpawn:
130.                         if self.velX == 0 and self.velY == 0:
131.                             self.velX = self.speed if self.id == 0 else -
    self.speed
132.                         else:
133.                             self.velX = 0
134.                             self.velY = -self.speed
135.
136.             def move(self):
137.                 # make the ghost leave spawn if it hasnt already
138.                 if not self.leftSpawn:
139.                     self.leave_spawn()
140.
141.                 # move the ghost based on its current velocity
142.                 self.x += self.velX
143.                 self.y += self.velY
144.
145.                 # calculate the nearest grid square to the ghost
146.                 self.nearestY = int(((self.y + (core.TILE_HEIGHT/2)) / core.
    TILE_HEIGHT))
147.                 self.nearestX = int(((self.x + (core.TILE_WIDTH/2)) / core.T
    ILE_WIDTH))
148.
149.                 # if the ghost is lined up with the grid again
150.                 if (self.x % core.TILE_WIDTH) == 0 and (self.y % core.TILE_H
    EIGHT) == 0:
151.                     x = self.nearestX
152.                     y = self.nearestY
153.                     # if the ghost has hit a door
154.                     doors = core.thisLevel.getPathwayPairPos()
155.                     if (x,y) in doors:
156.                         if (x,y) == doors[0]:
157.                             self.x = doors[1][0] * core.TILE_WIDTH
158.                             self.y = doors[1][1] * core.TILE_HEIGHT
159.                         else:
160.                             self.x = doors[0][0] * core.TILE_WIDTH
161.                             self.y = doors[0][1] * core.TILE_HEIGHT
162.
163.                     # if the ghost has reached an intersection
164.                     if (x,y) in core.thisLevel.intersections:
165.                         # check for vision of pacman and increase fitness if
     necessary
166.                         self.checkForPacman()
167.                         # get the action from the genome
168.                         direction = self.getGenomeAction()
169.                         # whilst the chosen direction contains a wall
170.                         while core.thisLevel.isWall((direction[1] + y, direc
    tion[0] + x)):
171.                             # deduct points and move in random direction
172.                             self.fitness_bonus += 10
173.                             direction = helper.random_direction()
174.                         # start moving in the direction
175.                         self.velX = self.speed * cmp(direction[0], 0)
176.                         self.velY = self.speed * cmp(direction[1], 0)
177.
178.             def getGenomeAction(self):
179.                 # calculate the majority direction to pacman
180.                 distX = core.player.x - self.x
181.                 distY = core.player.y - self.y
```

```
182.                 if abs(distY) > abs(distX):
183.                     pacmanDir = 1 if distY < 0 else 2
184.                 else:
185.                     pacmanDir = 3 if distX < 0 else 4
186.                 # get a string representing the encoded state of each direct
    ion
187.                 stringNSEW = self.getVision()
188.                 # calculate the gene number for this situation
189.                 gene_number = pacmanDir * (int(stringNSEW, 3) + 1) - 1
190.                 # return the direction to move given by that gene
191.                 return genetic.gene_to_direction(self.genome[gene_number])
192.
193.             def getVision(self):
194.                 # create a string to represent the situation in each directi
    on
195.                 # encoded in base 3, as there are 3 possible states
196.                 north = '0' if core.thisLevel.isWall((self.nearestY -
     1, self.nearestX)) \
197.                     else '1' if core.thisLevel.getMapTile((self.nearestY -
     1, self.nearestX)) == core.TILE_PELLET \
198.                         or core.thisLevel.getMapTile((self.nearestY -
     2, self.nearestX)) == core.TILE_PELLET \
199.                         or core.thisLevel.getMapTile((self.nearestY -
     3, self.nearestX)) == core.TILE_PELLET \
200.                     else '2'
201.                 south = '0' if core.thisLevel.isWall((self.nearestY + 1, sel
    f.nearestX)) \
202.                     else '1' if core.thisLevel.getMapTile((self.nearestY + 1,
     self.nearestX)) == core.TILE_PELLET \
203.                         or core.thisLevel.getMapTile((self.nearestY + 2, s
    elf.nearestX)) == core.TILE_PELLET \
204.                         or core.thisLevel.getMapTile((self.nearestY + 3, s
    elf.nearestX)) == core.TILE_PELLET \
205.                     else '2'
206.                 east =  '0' if core.thisLevel.isWall((self.nearestY, self.ne
    arestX + 1)) \
207.                     else '1' if core.thisLevel.getMapTile((self.nearestY, sel
    f.nearestX + 1)) == core.TILE_PELLET \
208.                         or core.thisLevel.getMapTile((self.nearestY, self.
    nearestX + 2)) == core.TILE_PELLET \
209.                         or core.thisLevel.getMapTile((self.nearestY, self.
    nearestX + 3)) == core.TILE_PELLET \
210.                     else '2'
211.                 west =  '0' if core.thisLevel.isWall((self.nearestY, self.ne
    arestX - 1)) \
212.                     else '1' if core.thisLevel.getMapTile((self.nearestY, sel
    f.nearestX - 1)) == core.TILE_PELLET \
213.                         or core.thisLevel.getMapTile((self.nearestY, self.
    nearestX - 2)) == core.TILE_PELLET \
214.                         or core.thisLevel.getMapTile((self.nearestY, self.
    nearestX - 3)) == core.TILE_PELLET \
215.                     else '2'
216.                 return north + south + east + west
217.
218.             # manhatten distance fitness
219.             def checkForPacman(self):
220.                 # calculate the difference vector to pacman
221.                 xDiff = abs(self.nearestX - core.player.nearestCol)
222.                 yDiff = abs(self.nearestY - core.player.nearestRow)
223.                 # add the manhatten distance to pacman to fitness
224.                 self.fitness += xDiff + yDiff
225.                 self.fitness_count += 1
```

7.7 src/helper.py

```
1.  # This module contains all helper functions
2.  from os import listdir
3.  from os.path import abspath, join, dirname
4.  import pygame, random
5.
6.  # Constant reference to resources
7.  RES_PATH = abspath(join(dirname(__file__), "..", "res"))
8.
9.  # method to return a random direction
10. def random_direction():
11.     return random.choice([(0,-1), (0,1), (-1,0), (1,0)])
12.
13. # Helper to load the background image
14. def load_background():
15.     return pygame.image.load(join(RES_PATH, "background.gif")).convert()
16.
17. # Helper to load any image from the resources
18. def load_image(folder, image_file):
19.     return pygame.image.load(join(RES_PATH, folder, image_file)).convert()
20.
21. # Helper to load sprites
22. def load_sprite(sprite_file):
23.     return load_image("sprites", sprite_file)
24.
25. # Helper to load text
26. def load_text(text_file):
27.     return load_image("text", text_file)
28.
29. # Helper to load tiles
30. def load_tile(tile_file):
31.     return load_image("tiles", tile_file)
32.
33. # Helper to open a level file
34. def open_level(level):
35.     return open(join(RES_PATH,"levels",str(level) + ".txt"), 'r')
36.
37. # Helper to check if any generations exist
38. def any_gen_files():
39.     return bool(listdir("genomes"))
40.
41. # Helper to open a file for a generation
42. def next_gen_file():
43.     num = 0
44.     # if generation(s) already exist, get the next generation number
45.     if any_gen_files():
46.         # get the next generation number
47.         num = max([int(each[-3:]) for each in listdir("genomes")]) + 1
48.     # get the filename for the next generation
49.     filename = "generation_{:03}".format(num)
50.     # return a file to store the next generation in
51.     return open(join("genomes", filename), "wb")
52.
53. # Helper to open a file containing the best generation
54. def best_gen_file():
55.     # if no generations have been simulated yet, return nothing
56.     if not any_gen_files(): return
57.     # get the largest existing generation number
58.     num = max([int(each[-3:]) for each in listdir("genomes")])
59.     # get the filename for the best generation
60.     filename =  "generation_{:03}".format(num)
61.     # return a file containing the best generation so far
62.     return open(join("genomes", filename), "rb")
```

7.8 src/level.py

```
1.  import core, game, helper, genetic
2.
3.  class Level():
4.      def __init__(self):
5.          # level variables
6.          self.lvlWidth = 0
7.          self.lvlHeight = 0
8.          self.map = {}
9.          self.intersections = []
10.         self.pellets = 0
11.
12.     # method to set a specific map tile
13.     def setMapTile(self, (row, col), tile):
14.         self.map[(row, col)] = tile
15.
16.     # method to return the contents of a map tile
17.     def getMapTile(self, (row, col)):
18.         # only return tile contents if within map boundaries
19.         if row >= 0 and row < self.lvlHeight and col >= 0 and col < self.lv
    lWidth:
20.             return self.map[(row, col)]
21.         else:
22.             return 0
23.
24.     # method to determine whether a map tile contains a wall
25.     def isWall(self, (row, col)):
26.         # return true if outside of the map boundaries
27.         if row > core.thisLevel.lvlHeight - 1 or row < 0:
28.             return True
29.         if col > core.thisLevel.lvlWidth - 1 or col < 0:
30.             return True
31.
32.         # check the offending tile ID
33.         result = core.thisLevel.getMapTile((row, col))
34.         # if the tile was a wall
35.         if result in core.ALL_WALL_TILES:
36.             return True
37.         else:
38.             return False
39.
40.
41.     # method to check if the given position would hit a wall
42.     def checkIfHitWall(self, (possiblePlayerX, possiblePlayerY), (row, col)
    ):
43.         numCollisions = 0
44.         # check each of the 9 surrounding tiles for a collision
45.         for iRow in range(row - 1, row + 2, 1):
46.             for iCol in range(col - 1, col + 2, 1):
47.                 # only check if the position is within the map boundaries
48.                 if  (possiblePlayerX -
     (iCol * core.TILE_WIDTH) < core.TILE_WIDTH) \
49.                 and (possiblePlayerX - (iCol * core.TILE_WIDTH) > -
    core.TILE_WIDTH) \
50.                 and (possiblePlayerY -
     (iRow * core.TILE_HEIGHT) < core.TILE_HEIGHT) \
51.                 and (possiblePlayerY - (iRow * core.TILE_HEIGHT) > -
    core.TILE_HEIGHT):
52.                     # add to collision count if it is a wall
53.                     if self.isWall((iRow, iCol)):
54.                         numCollisions += 1
55.
56.         # return whether any collisions occured
57.         return True if numCollisions > 0 else False
```

```
58.
59.     # method to check if the player has hit a cushion
60.     def checkIfHit(self, (playerX, playerY), (x, y), cushion):
61.         if (playerX - x < cushion) and (playerX - x > -cushion) \
62.         and (playerY - y < cushion) and (playerY - y > -cushion):
63.             return True
64.         else:
65.             return False
66.
67.
68.     # method to check if the player has hit something
69.     def checkIfHitSomething(self, (playerX, playerY), (row, col)):
70.         # check each of the 9 surrounding tiles for a collision
71.         for iRow in range(row - 1, row + 2, 1):
72.             for iCol in range(col - 1, col + 2, 1):
73.                 # only check if the position is within the map boundaries
74.                 if  (playerX -
    (iCol * core.TILE_WIDTH) < core.TILE_WIDTH) \
75.                 and (playerX - (iCol * core.TILE_WIDTH) > -
   core.TILE_WIDTH) \
76.                 and (playerY -
    (iRow * core.TILE_HEIGHT) < core.TILE_HEIGHT) \
77.                 and (playerY - (iRow * core.TILE_HEIGHT) > -
   core.TILE_HEIGHT):
78.                     # check the offending tile ID
79.                     result = core.thisLevel.getMapTile((iRow, iCol))
80.                     if result == core.TILE_PELLET:
81.                         # remove pellet, decrease count and increase score

82.                         core.thisLevel.setMapTile((iRow, iCol), None)
83.                         core.thisLevel.pellets -= 1
84.                         core.thisGame.addToScore(10)
85.                         # if there are no more pellets left
86.                         if core.thisLevel.pellets <= 0:
87.                             core.thisGame.setMode(game.MODE_WAIT_FINISH)
88.
89.                     # if a door was hit, move the player to the other door

90.                     elif result == core.TILE_DOOR:
91.                         for i in xrange(core.thisLevel.lvlWidth):
92.                             if not i == iCol:
93.                                 if core.thisLevel.getMapTile((iRow, i)) ==
   core.TILE_DOOR:
94.                                     core.player.x = i * core.TILE_WIDTH
95.                                     if core.player.velX > 0:
96.                                         core.player.x += core.TILE_WIDTH
97.                                     else:
98.                                         core.player.x -= core.TILE_WIDTH
99.
100.            # method to return the position of the ghost box
101.            def getGhostBoxPos(self):
102.                for row in xrange(self.lvlHeight):
103.                    for col in xrange(self.lvlWidth):
104.                        if self.getMapTile((row, col)) == core.TILE_GHOST_DO
    OR:
105.                            return (row, col)
106.                return False
107.
108.            # method to get a pair of doors
109.            def getPathwayPairPos(self):
110.                doorArray = []
111.                for y in xrange(self.lvlHeight):
112.                    for x in xrange(self.lvlWidth):
113.                        if self.getMapTile((y, x)) == core.TILE_DOOR:
114.                            # found a horizontal door
115.                            doorArray.append((x, y))
```

```
116.                 return doorArray
117.
118.             # method to draw the map
119.             def drawMap(self):
120.                 for row in xrange(-
    1, core.thisGame.screenTileSize[0] +1, 1):
121.                     outputLine = ""
122.                     for col in xrange(-
    1, core.thisGame.screenTileSize[1] +1, 1):
123.                         # row containing tile that actually goes here
124.                         actualRow = core.thisGame.screenNearestTilePos[0] +
    row
125.                         actualCol = core.thisGame.screenNearestTilePos[1] +
    col
126.
127.                         useTile = self.getMapTile((actualRow, actualCol))
128.                         if useTile and useTile[1]:
129.                             # if this isn't a blank tile
130.                             core.screen.blit(useTile[1],
131.                                 (col * core.TILE_WIDTH -
     core.thisGame.screenPixelOffset[0],
132.                                  row * core.TILE_HEIGHT -
     core.thisGame.screenPixelOffset[1]))
133.
134.             # method to load a given level
135.             def loadLevel(self, levelNum):
136.                 self.map = {}
137.                 self.pellets = 0
138.
139.                 # setup variables for reading the file
140.                 f = helper.open_level(levelNum)
141.                 lineNum=-1
142.                 rowNum = 0
143.                 useLine = False
144.                 isReadingLevelData = False
145.
146.                 for line in f:
147.                     lineNum += 1
148.
149.                     while len(line)>0 and (line[-1]=="\n" or line[-
    1]=="\r"): line=line[:-1]
150.                     while len(line)>0 and (line[0]=="\n" or line[0]=="\r"):
    line=line[1:]
151.                     str_splitBySpace = line.split(' ')
152.
153.                     j = str_splitBySpace[0]
154.
155.                     if (j == "'" or j == ""):
156.                         # comment / whitespace line
157.                         useLine = False
158.                     elif j == "#":
159.                         # special divider / attribute line
160.                         useLine = False
161.                         firstWord = str_splitBySpace[1]
162.
163.                         if firstWord == "lvlwidth":
164.                             self.lvlWidth = int( str_splitBySpace[2] )
165.                         elif firstWord == "lvlheight":
166.                             self.lvlHeight = int( str_splitBySpace[2] )
167.                         elif firstWord == "edgecolor":
168.                             # edge color keyword for backwards compatibility
     (single edge color) mazes
169.                             red = int( str_splitBySpace[2] )
170.                             green = int( str_splitBySpace[3] )
171.                             blue = int( str_splitBySpace[4] )
172.                             self.edgeLightColor = (red, green, blue, 255)
```

```
173.                             self.edgeShadowColor = (red, green, blue, 255)
174.                         elif firstWord == "edgelightcolor":
175.                             red = int( str_splitBySpace[2] )
176.                             green = int( str_splitBySpace[3] )
177.                             blue = int( str_splitBySpace[4] )
178.                             self.edgeLightColor = (red, green, blue, 255)
179.                         elif firstWord == "edgeshadowcolor":
180.                             red = int( str_splitBySpace[2] )
181.                             green = int( str_splitBySpace[3] )
182.                             blue = int( str_splitBySpace[4] )
183.                             self.edgeShadowColor = (red, green, blue, 255)
184.                         elif firstWord == "fillcolor":
185.                             red = int( str_splitBySpace[2] )
186.                             green = int( str_splitBySpace[3] )
187.                             blue = int( str_splitBySpace[4] )
188.                             self.fillColor = (red, green, blue, 255)
189.                         elif firstWord == "pelletcolor":
190.                             red = int( str_splitBySpace[2] )
191.                             green = int( str_splitBySpace[3] )
192.                             blue = int( str_splitBySpace[4] )
193.                             self.pelletColor = (red, green, blue, 255)
194.                         elif firstWord == "startleveldata":
195.                             isReadingLevelData = True
196.                             rowNum = 0
197.                         elif firstWord == "endleveldata":
198.                             isReadingLevelData = False
199.                     else:
200.                         useLine = True
201.
202.                     # this is a map data line
203.                     if useLine == True:
204.                         if isReadingLevelData == True:
205.                             for k in xrange(self.lvlWidth):
206.                                 tile = core.ALL_TILES[int(str_splitBySpace[k
    ])]
207.                                 self.setMapTile((rowNum, k), tile)
208.
209.                                 if tile == core.TILE_PELLET:
210.                                     self.pellets += 1
211.                                 elif tile == core.TILE_INTERSECTION:
212.                                     self.intersections.append((k, rowNum))
213.                                     self.setMapTile((rowNum, k), core.TILE_P
    ELLET)
214.                                     self.pellets += 1
215.                                 elif tile == core.TILE_INTERSECTION_EMPTY:
216.                                     self.intersections.append((k, rowNum))
217.                                 elif tile == core.TILE_START:
218.                                     # starting position for pac-man
219.                                     core.player.homeX = k * core.TILE_WIDTH

220.                                     core.player.homeY = rowNum * core.TILE_H
    EIGHT
221.                                     self.setMapTile((rowNum, k), None )
222.
223.                                 elif tile in core.ALL_GHOST_TILES:
224.                                     # one of the ghosts
225.                                     core.ghosts[tile[0] -
     10].homeX = k * core.TILE_WIDTH
226.                                     core.ghosts[tile[0] -
     10].homeY = rowNum * core.TILE_HEIGHT
227.                                     self.setMapTile((rowNum, k), None )
228.
229.                             rowNum += 1
230.
231.                 # do all the level-starting stuff
232.                 self.restart()
```

```
233.
234.            # method to restart the level
235.            def restart(self):
236.                # reset all ghosts
237.                for i in range(4):
238.                    core.ghosts[i].reset()
239.
240.                # reset the player and it's animation
241.                core.player.x = core.player.homeX
242.                core.player.y = core.player.homeY
243.                core.player.velX = 0
244.                core.player.velY = 0
245.                core.player.anim_pacmanCurrent = core.player.anim_pacmanS
246.                core.player.animFrame = 3
```

7.9 src/pacman.py

```
1.  import helper, core, game
2.
3.  class Pacman():
4.      def __init__(self):
5.          # pacman variables
6.          self.x = 0
7.          self.y = 0
8.          self.velX = 0
9.          self.velY = 0
10.         self.speed = 3
11.         self.nearestRow = 0
12.         self.nearestCol = 0
13.         self.homeX = 0
14.         self.homeY = 0
15.         # animation variables
16.         self.anim_pacmanL = {}
17.         self.anim_pacmanR = {}
18.         self.anim_pacmanU = {}
19.         self.anim_pacmanD = {}
20.         self.anim_pacmanS = {}
21.         self.anim_pacmanCurrent = {}
22.
23.         # load all of the frames of all animations
24.         for i in range(1, 9, 1):
25.             self.anim_pacmanL[i] = helper.load_sprite("pacman_l_" + str(i)
    + ".gif")
26.             self.anim_pacmanR[i] = helper.load_sprite("pacman_r_" + str(i)
    + ".gif")
27.             self.anim_pacmanU[i] = helper.load_sprite("pacman_u_" + str(i)
    + ".gif")
28.             self.anim_pacmanD[i] = helper.load_sprite("pacman_d_" + str(i)
    + ".gif")
29.             self.anim_pacmanS[i] = helper.load_sprite("pacman.gif")
30.
31.     # method to move the player
32.     def move(self):
33.         # get the nearest row and column
34.         self.nearestRow = int(((self.y + (core.TILE_WIDTH/2)) / core.TILE_W
    IDTH))
35.         self.nearestCol = int(((self.x + (core.TILE_HEIGHT/2)) / core.TILE_
    HEIGHT))
36.
37.         # make sure the current velocity will not cause a collision before
    moving
38.         if not core.thisLevel.checkIfHitWall(
39.                 (self.x + self.velX, self.y + self.velY),
40.                 (self.nearestRow, self.nearestCol)):
41.             # it's ok to Move
42.             self.x += self.velX
43.             self.y += self.velY
44.
45.             # check for collisions with other tiles (pellets, etc)
46.             core.thisLevel.checkIfHitSomething(
47.                     (self.x, self.y), (self.nearestRow, self.nearestCol))
48.
49.             # check for collisions with the ghosts
50.             for i in range(4):
51.                 if core.thisLevel.checkIfHit((self.x, self.y),
52.                         (core.ghosts[i].x, core.ghosts[i].y), core.TILE_WID
    TH/2):
53.                     core.ghosts[i].hit_pacman()
54.                     core.thisGame.setMode(game.MODE_WAIT_HIT)
55.
```

```
56.         # we're going to hit a wall -- stop moving
57.         else:
58.             self.velX = 0
59.             self.velY = 0
60.
61.     # method to draw the player
62.     def draw(self):
63.         # dont draw if the game is over
64.         if core.thisGame.mode == game.MODE_GAME_OVER:
65.             return False
66.
67.         # set the current frame array to match the direction pacman is faci
    ng
68.         if self.velX > 0:
69.             self.anim_pacmanCurrent = self.anim_pacmanR
70.         elif self.velX < 0:
71.             self.anim_pacmanCurrent = self.anim_pacmanL
72.         elif self.velY > 0:
73.             self.anim_pacmanCurrent = self.anim_pacmanD
74.         elif self.velY < 0:
75.             self.anim_pacmanCurrent = self.anim_pacmanU
76.
77.         #draw the player to the screen
78.         core.screen.blit(self.anim_pacmanCurrent[self.animFrame],
79.                 (self.x - core.thisGame.screenPixelPos[0],
80.                  self.y - core.thisGame.screenPixelPos[1]))
81.
82.         if core.thisGame.mode == game.MODE_NORMAL:
83.             # only Move mouth when pacman is moving
84.             if not self.velX == 0 or not self.velY == 0:
85.                 self.animFrame += 1
86.             # wrap to beginning
87.             if self.animFrame == 9:
88.                 self.animFrame = 1
```

7.10 src/pathfinder.py

```
1.  import heapq, random
2.  import core
3.
4.  directions = [(0,-1), (0,1), (-1,0), (1,0)]
5.
6.  class Path:
7.      #constructor for a path, optionally taking a parent
8.      def __init__(self, parent = None, path = [], length = 1, heuristic = 0)
    :
9.          if parent:
10.             self.path = parent.path + path
11.             self.length = parent.length + length
12.         else:
13.             self.path = path
14.             self.length = length
15.         self.heuristic = heuristic
16.
17.     #method to return the current node in the path
18.     def current(self):
19.         if not self.path:
20.             raise Exception, "Path is empty"
21.         return self.path[-1]
22.
23.     #method overrides
24.     def __le__(self, path2):
25.         return len(self) <= len(path2)
26.
27.     def __len__(self):
28.         return self.length + self.heuristic
29.
30.     def __iter__(self):
31.         for node in self.path:
32.             yield node
33.
34.
35.
36. #method to return whether the node is the goal or not
37. def goal(node):
38.     return core.thisLevel.getMapTile((node[1], node[0])) == core.TILE_PELLE
    T
39.
40. #method to return the list of successors
41. def successors(node):
42.     global directions
43.     # get all surrounding squares
44.     squares = [(x + node[0], y + node[1]) for (x,y) in directions]
45.     # filter to only those that are not walls
46.     non_wall = [(x,y) for (x,y) in squares if not core.thisLevel.isWall((y,
    x))]
47.     # return the possible moves in a random order
48.     random.shuffle(non_wall)
49.     return non_wall
50.
51. #method to get the heuristic for following a node
52. def heuristic(node):
53.     # get the list of all pellet positions left in the map
54.     all_pellets = [pos for pos in core.thisLevel.map
55.                    if core.thisLevel.getMapTile(pos) == core.TILE_PELLET]
56.     # get the city-block distances to the first 25 pellets
57.     distances   = [(abs(x - node[0]) + abs(y - node[1]))
58.             for (y,x) in all_pellets[:25]]
59.     return min(distances)
60.
```

```
61. def find_path():
62.     #initialise the candidates heap
63.     candidates = []
64.     #add single path containing only the start node
65.     start = (core.player.nearestCol, core.player.nearestRow)
66.     start_path = Path(path = [start], length = 1)
67.     heapq.heappush(candidates, start_path)
68.
69.     #loop until a solution is found
70.     while True:
71.         #get first candidate and its current node
72.         cand = heapq.heappop(candidates)
73.         node = cand.current()
74.
75.         #base case
76.         if goal(node): return cand
77.
78.         #expand the node's successors
79.         for succ in successors(node):
80.             #candidates.append(cand + [succ])
81.             new_path = Path(parent = cand, path = [succ],
82.                     heuristic = heuristic(node))
83.             heapq.heappush(candidates, new_path)
```

7.11 src/simulator.py

```
1.  import sys, pygame, cPickle
2.  import core, game, genetic, simulator_player
3.
4.  def simulate_play(individuals):
5.      # start a new game playing immediately
6.      core.thisGame.startNewGame(individuals)
7.      core.thisGame.setMode(game.MODE_NORMAL)
8.      core.player.velX = core.player.speed
9.
10.     # continuously run the game
11.     while True:
12.         # handle the current game mode
13.         if core.thisGame.mode == game.MODE_NORMAL:
14.             # move all characters
15.             simulator_player.move()
16.             for i in range(4):
17.                 core.ghosts[i].move()
18.         elif core.thisGame.mode == game.MODE_WAIT_HIT:
19.             # immediately restart the level
20.             core.thisLevel.restart()
21.
22.             # reduce player lives
23.             core.thisGame.lives -= 1
24.             # if no lives are left, stop the current simulation
25.             if core.thisGame.lives == 0:
26.                 break
27.             else:
28.                 # start the level again
29.                 core.thisGame.setMode(game.MODE_NORMAL)
30.                 core.player.velX = core.player.speed
31.         # if the game has ended, stop the current simulation
32.         elif core.thisGame.mode == game.MODE_GAME_OVER \
33.             or core.thisGame.mode == game.MODE_WAIT_FINISH:
34.             break
35.
36.         # move the screen if necessary and redraw everything
37.         core.thisGame.smartMoveScreen()
38.         core.screen.blit(core.img_Background, (0, 0))
39.         core.thisLevel.drawMap()
```

```
40.         for i in range(4):
41.             core.ghosts[i].draw()
42.         core.player.draw()
43.         core.thisGame.drawScore()
44.
45.         # flip the buffer and wait for the next tick
46.         pygame.display.flip()
47.         core.clock.tick()
48.
49.     # return a list of fitness and genome pairs
50.     return [(core.ghosts[i].get_fitness(), core.ghosts[i].genome) for i in
    range(4)]
51.
52. def simulate(generations, population):
53.     # get the best generation so far to start with
54.     pop = genetic.get_best_generation(population)
55.
56.     # run for the given number of generations
57.     for i in xrange(generations):
58.         fit = []
59.         #if the population has already been fitness scored
60.         if pop[0][0] != None:
61.             # evolve the generation to generate the next
62.             pop = genetic.evolve(pop)
63.         # loop through the population in chunks of 4 individuals
64.         for i in xrange(0, len(pop), 4):
65.             # get the current selection of 4 individuals
66.             sel = pop[i:i+4]
67.             # simulate gameplay for the given selection
68.             fit.extend(simulate_play(sel))
69.
70.         # print fitness scores and average
71.         print sorted([each[0] for each in fit])
72.         print "Average Fitness = " + str(sum([each[0] for each in fit]) / l
    en(fit))
73.
74.         # pickle the generation into a file
75.         genetic.pickle_generation(fit)
```

7.12 src/simulator_player.py

```
1.  import random
2.  import core, pathfinder
3.
4.  path = []
5.
6.  def move():
7.      global path
8.      core.player.move()
9.      # if the player is aligned to the grid
10.     if (core.player.x % core.TILE_WIDTH) == 0 and (core.player.y % core.TIL
    E_HEIGHT) == 0:
11.         # if there is no current path to follow
12.         if not path:
13.             # find the path to the closest pill
14.             path = pathfinder.find_path().path
15.
16.         # find the direction for the first node in the path
17.         square = path.pop(0)
18.         direction = (core.player.nearestCol -
     square[0], core.player.nearestRow - square[1])
19.
20.         # start moving in the direction
21.         core.player.velX = -core.player.speed * cmp(direction[0], 0)
22.         core.player.velY = -core.player.speed * cmp(direction[1], 0)
```