Introduction

This report details the agent I have built to try and win Pacman. The aim of the game is to get Pacman to eat all of the food whilst avoid being eaten by ghosts. The Pacman world is a deterministic and static environment, with well-defined locations of walls and food that do not change throughout the game. Because of this, I built a goal-based agent that uses searching algorithms to guide Pacman to his goal - the food on the map. Unfortunately, Pacman's environment is not accessible, and he has limited information about the world. I have implemented several strategies to help Pacman deal with this uncertainty. In particular, Pacman takes in information about the world and stores it for later use in a map. I chose to build my agent with these particular strategies because they use the information that Pacman can get from his state and the world to give him a high chance of winning the game.

Description

The approach my agent takes to play the Pacman game is based on three strategies, each strategy is called when a particular instance arises in the game. To determine which strategy is used I have created a strategy hierarchy. Both the corner and food strategy employ the same underlying algorithm based on an A* search to determine the best possible route. The priority ordering of the strategies is as follows:

1. Survival
2. Search for food
3. Head to the corners

From an implementation standpoint each strategy shares a common function work flow. The getAction() function sequentially calls each of the three strategies, following the priority ordering outlined above. Each strategy is called and if a None value is returned the next strategy is considered.

For each strategy a getStrategy() function is called which gathers information about the state and the world. Based on this information a decision is then made to either return None or call the initalizeStrategy() function. Finally a direction of travel is calculated and returned to the api.makeMove() function.

It is worth noting that the A* search algorithm is only called once per goal and not recalculated on every move. To achieve this, persistent storage has been used along with a generator function genMove() which returns an iterator and remembers its internal state between successive calls. A similar mechanism has been employed to remember the corners visited.

I have briefly outlined the A* search algorithm below and then go on to talk about each of the three strategies, I have also provided the work flow of the functions for each strategy if more details is wanted on the code implementation. In the testing section I have provided data on the performance of the strategies, as well as comparing the implemented strategies against others that were considered. I will refer to functions with their actual name followed by closed parentheses (without arguments). A fuller explanation of each function, it's arguments and return types can be found in the comments of the code.

The A* Search algorithm

Agents can solve the problem in Pacman by searching through many different solutions, this is known as performing a search. This can be viewed as an algorithm that searches through trees of nodes and their sub nodes until a goal has been reached. Practically this can become intractable very quickly, since enumerating all possible paths grows too large. The solution is to eliminate much of the search space by implementing some guiding heuristic. In this solution we have implemented an A* search algorithm and use the Manhattan distance as our heuristic. The idea is to use this heuristic to estimate the cost of potential nodes. The g, h and f score are three metrics that are calculated for each node to guide A* into picking the best route. In our implementation of A* the g score is the number of steps between the start node and the current node, the h score is the heuristic (manhattan distance) and the f score is the sum of the two.

I have implemented a Search() class, details of functions can be found in the comments in the code. In this class there is both an A* and BFS function, but the BFS functions is only used for comparison purposes.

The implementation of the A* is as follows: I instantiate two lists, openNodes for nodes that we are searching for and visitedNodes for nodes that we have expanded and searched. We start with the first node as the parent node and then get its child nodes, in this game that is the north, east, west and south neighbour. For each child node the f, g and h scores, along with the parent are stored in a dictionary. The child node is checked to see if it is already appended to openNodes and we compare the g scores and take the one with the lowest. The next parentNode is found based on the minimum f score node in openNodes. At each determination of the parentNode, the node is checked to see if it is the target and if it is the backtrace() function is called which builds the A* route from the parent information stored in the dictionary.

Survival Strategy

Throughout the game Pacman is under constant threat of being eaten by a ghost, which causes Pacman to lose the game. To minimize this outcome Pacman has implemented a survival strategy that takes precedence over all other strategies. Pacman uses his ability to see a radius of 2 steps around him at all times to find ghosts and make a move that reduces this danger. Initially Pacman considered a strategy that just looked directly north, south, east or west and if a ghost existed along one of these locations it would remove this from it's set of legal moves. However, this strategy is far from optimal, shown in the test. This is because the ghost could go undetected in a location diagonal Pacman. In this instance they could bump into each other when both are travelling along different directions, Figure 1 illustrates this below.

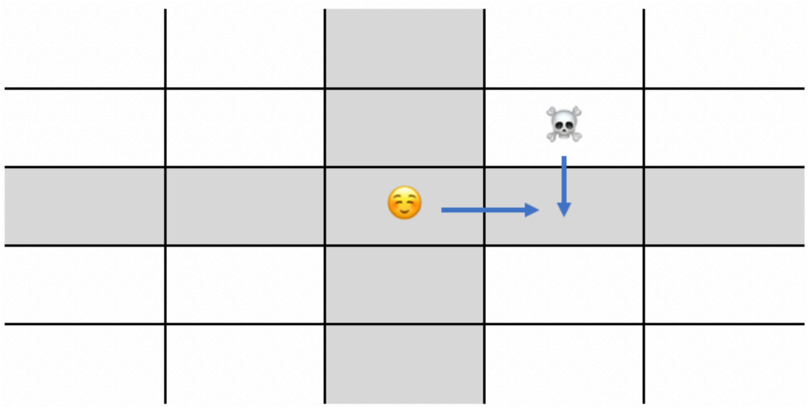


Figure 1 shows Pacman (smilley face) moving east, whilst the ghost (skull and cross bones) is moving south. The highlighted grey area is where Pacman is scanning for Ghosts. In this instance Pacman will not detect the ghost and they will bump into each other.

Pacman has implemented a better solution by building a safe area around him of radius two and checking for ghosts within this vicinity. If ghosts are found, Pacman has to take action to evade danger. Pacman considered two possible solutions

1. Pacman passes the ghost location to the A* search class and determines the shortest route to the ghost, this is shown in Figure 2. From this, Pacman knows the shortest number of steps it takes to get to the ghost and the route that needs to be taken. Therefore, Pacman uses the first location in the route to work out the direction the ghost will come from if this route is taken. Pacman can then remove this direction accordingly from the set of legal moves. Whilst the ghost may not take this route, if it does not it will be taking a longer pass which gives Pacman plenty of time to continue the escape on a next move. Given the presence of walls it may be the case that the ghost is in the safe area but has to take a longer route to get to Pacman if they are separated by a wall. In this instance, Pacman ignores routes that are greater than 4 steps.

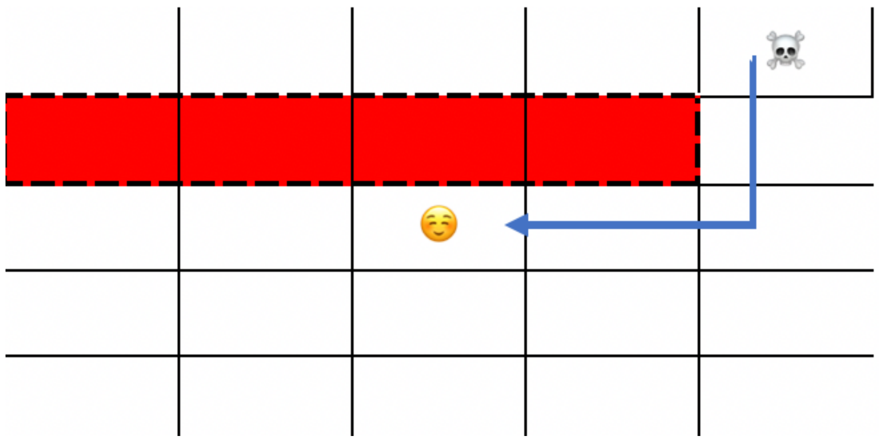


Figure 2 shows the shortest route from a ghost to Pacman. Pacman could uses this route to work out a way to escape. The red area is a wall.

2. Pacman splits the safe area into four quadrants and two axes, each quadrant maps to two directions and the axes map to one, the areas are illustrated in figure 3. If the ghosts are located in one or more of these areas the corresponding directions are removed accordingly. This strategy is conservative as it removes at least the two most likely directions that a ghost could come from if they are locate in the quadrant area.

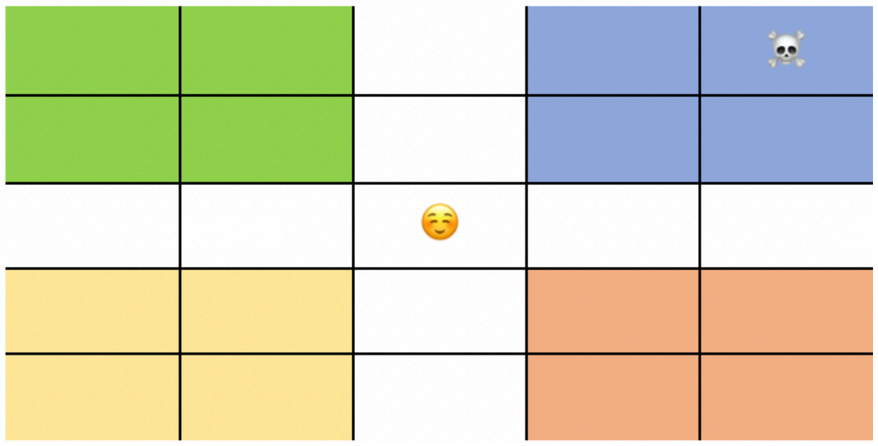


Figure 3 shows how Pacman divides his safe area. The four coloured squares each map to two directions, whereas the areas directly north, east, west and south map to one.

Pacman has implemented option two. As shown in the testing section this leads to a better survival rate. Despite this survival strategy Pacman is still eaten on rare occasions, usually this is because he is cornered by both ghosts down a corridor with walls either side or at the apex of the corners of the grid.

Survival Function Work Flow

- Pacman calls getSurvival() to begin checking if the strategy needs to be implemented. The ghost locations are found and checkGhosts() is called find the ghosts considered dangerous based on their location

- To determine the safe area Pacman uses the function getSafeArea(). This is based on Pacmans current location and uses the variable self.ghstrad, set to two, to build a list of tuples which contains the locations in (x, y) cartesian form of the surrounding grid.

- Once the safe area is known Pacman cross checks the ghosts locations against the safe area and returns a list of ghost locations that match. If no ghosts appear in these locations an empty list is returned.

- If an empty list is returned the strategy returns None and subsequent strategies are considered. If the ghosts appear in the safe area we call planEscape(), this use getQuadMapping() to determine which directions are removed from the set of legal actions based on the location within the safe area. The set of legal actions is then returned, and a move is randomly chosen from these.

Food strategy

To win the game Pacman needs to eat all the food on the map. Our agents' approach is to find the nearest food using an A* search algorithm to find the shortest path.

Pacmans first step is to find the nearest food using the Manhattan distance. Pacman does not have a full view of the world and can only see food in a limited area, nonetheless, Pacman builds a view of the world as he moves, storing both visited and food locations in a map. When Pacman cannot see food in its visible area, it can look in the map, to find food locations and determine which of those is closest. If Pacman still can't find any food in its current state once the map has been checked it will return None for this particular strategy and the corner strategy will be called. Note Pacaman adds capsules to his list of food locations.

If a food location has been found this is passed to the A* method in the Search class and an A* route is found and stored in self.foodAstarRoute. Pacman follows this route until either danger is afoot and a survival strategy is implemented, or the food is reached. Below I outline the functional work flow of the code for the food strategy.

Food Function Work Flow

- getFood() is called and obtains available food locations from the api and mapWorld() is called to add these positions to self.pacWorld. If no food is found pacman checks the map, self.pacWorld. If we still can't find any food from this list we return None to getAction() otherwise Pacman calls getFoodAction().

- In getFoodAction one of two things will happen depending whether Pacman is pursuing an existing target or finding new food:

  1. If the previous strategy was food and Pacman is already pursuing a food location (self.newFood == False and self.strategy == 'Food) Pacman calls chasefood() which returns the next step in the A* star route that is stored in self.foodAstarRoute. A next(genMove) call will return this location.

  2. If self.newFood is True we use the manhatten distance function to return the closest food from our food locations. This is passed along with the strategy 'Food' to the initializeRoute() function. This function instantiates a new A* route and stores it in the self.foodAstaRoute class variable and a new self.genMove function and stores it in self.foodMove so we don't have to calculate the route again.

Pacman then returns to getFoodAction() and calls chaseFood() which calls the next(genMove) and generates the direction for pacman to make. If the target food location is reached, we set self.newFood = True so a new target and A* search route are set.

Corner Strategy

Pacman is constantly scanning for both food and ghosts. However, if no ghosts are present in the safe area and no food can be found, Pacman employs a corner strategy. Pacman will visit each corner sequentially searching for food along the way. If either the food or survival strategy is employed in-between the corner strategy, Pacman will continue to the same corner when the corner strategy is called again. The Pacman agent employs this strategy to maximise its chances of covering the whole map and finding all the food along the way. By the end our agent will hopefully have covered sufficient ground to have eaten all the food or mapped where it is. Each corner is visited only once until all have been visited. Pacman then uses resetCorners() to start again.

This strategy aims to guide Pacman to finding food when the food strategy can't be used on a particular move, by implementing 'way-points'. An alternative strategy that Pacman tried used a random function to randomly guide Pacman around. As expected, the survival rate was materially worse. The results are discussed in testing.

Food Function Work Flow

- getCorners() function retrieves the corners from the api. If all corners have been visited we call the resetCorners() function. Current location is passed to mapWorld() and then getCornerAction is called.

- getCornerAction() has to make a decision out of three options dependant on what self.newCorner and self.strategy equal. I will outline these three decisions below

    1. if self.newCorner == 'False and self.Strategy == 'Corners' we have not reached the current corner and the strategy from the previous move was corner we can continue along the route to the current corner using the route stored in self.survivalAstarRoute. We call the chaseCorner() function which returns the direction needed to get from the current location to the next.

    2. If self.newCorner == False and self.strategy != 'Corners' then we are on the current corner but the last move wasn't a corner strategy. Pacman needs to reinitialize a new A* route with the current corner, we do this by calling initializeRoute() and then returning the direction from chaseCorner(). Note if this is the option chosen a check is made to make sure we haven't landed on the corner location through a different strategy.

3. If self.newCorner == True then we need to get a new A* route with a new corner and return chaseCorner() to get the direction of travel.

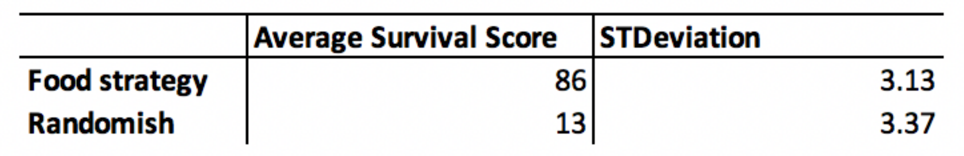
Testing

Below are my tests that I carried out to test the performance of different strategies. For tests where I have compared strategies, I have left the code in the codebase with it commented that it's not used in the final implementation. For statistical significance, each test consists of 10 independent trials of 100 games. I then took an average of the survival rate. Each test analyses this survival rate.

Food

Test 1

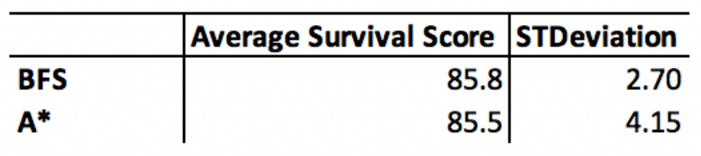
This test is on the standard map against two ghosts. This test was designed to illustrate that implementing a food strategy is better than just getting Pacman to move randomly. As we can see from the results the Food strategy has a significantly higher survival score. This is because Pacman makes less moves that don't result in getting food.



	Average Survival Score	STDeviation
Food strategy	86	3.13
Randomish	13	3.37

Test 2

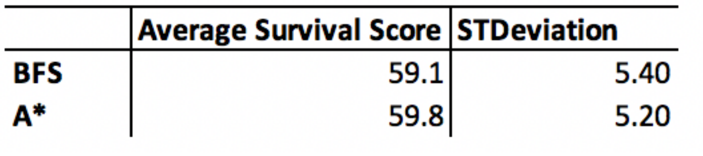
This test is on the standard map against two ghosts. I tested the agent with both an A* and BFS search algorithm for the food strategy. This test is designed to see which searching algorithm has the best performance for searching for food. The results below show that is nearly no difference between the two algorithms. This is as we would expect as both should return optimal routes. The difference here would manifest itself in the time and space complexity of the program, with BFS being costlier given it does not use a heuristic and checks all possible routes



	Average Survival Score	STDeviation
BFS	85.8	2.70
A*	85.5	4.15

Test 3

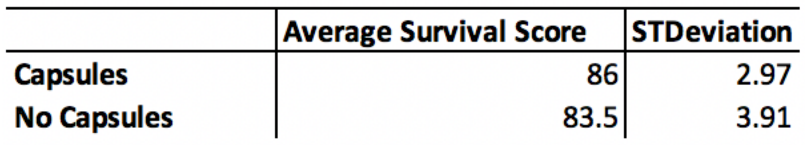
I also tested BFS and A* on the trickyClassic map. This test is designed to see if a more complex map (size and ghosts) would affect the performance of the two searching algorithms. This map is larger than the standard one and has four ghosts instead of two. Similarly, to test 1 the results between BFS and A* are very similar and both are as good on a more complex map.



	Average Survival Score	STDeviation
BFS	59.1	5.40
A*	59.8	5.20

Test 4

This test is on the standard map against two ghosts. The average survival rate with capsules included in the food locations that Pacman targets is higher than without. Intuitively I expected that capsules included would help Pacman evade danger more easily, since there are periods when the ghosts would not be able to eat him. Despite this, given how close these results are and their respective standard deviations I cannot conclude that including capsules is better. It is likely that Pacman will accidently come across the capsules whilst searching for food anyway and therefore even with them not included they will be eaten.



	Average Survival Score	STDeviation
Capsules	86	2.97
No Capsules	83.5	3.91

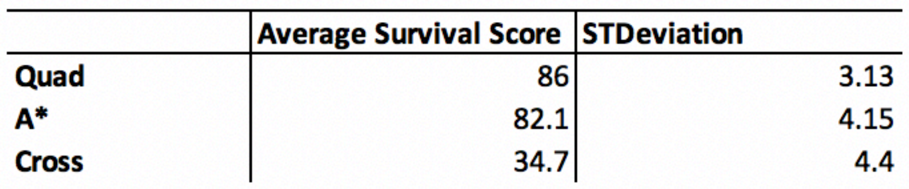
Survival

Test 1

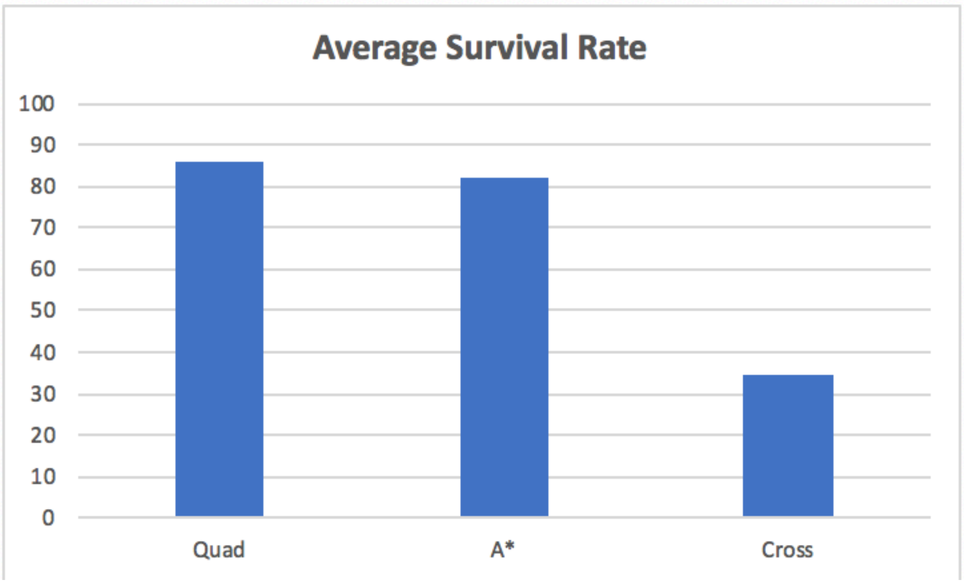
This test is on the standard map against two ghosts. This test is designed to see which of the three strategies (Cross, Quad or A* survival) led to the best performance by comparing the average survival rate.

We can see the Cross strategy is significantly inferior to the other two. As mentioned previously in the strategy section, this strategy assumes that the ghost is only a danger when directly north, east, south and west. Since Pacman's safe area is reduced, Pacman's ability to evade danger is greatly limited. Furthermore, there will be many instances when the ghost is not detected as it is located diagonally to Pacman, but it is moving along a different direction of travel but towards Pacman. In this scenario they could meet, leading to a loss.

The average survival rate of the Quad strategy is better than the A* by 4%, however, given this difference is close to the standard deviation of both it's difficult to say with certainty that this is the case and not just randomness. I had a look at the strategies on a harder map, and the test results are show below.

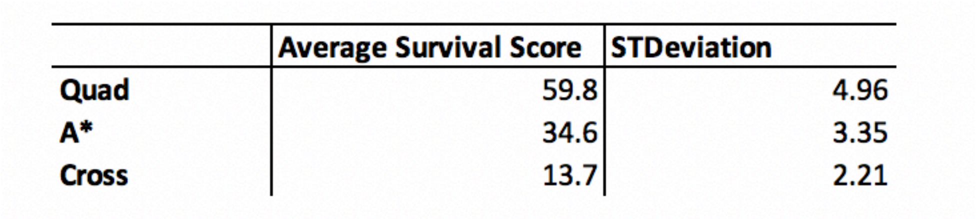


	Average Survival Score	STDeviation
Quad	86	3.13
A*	82.1	4.15
Cross	34.7	4.4

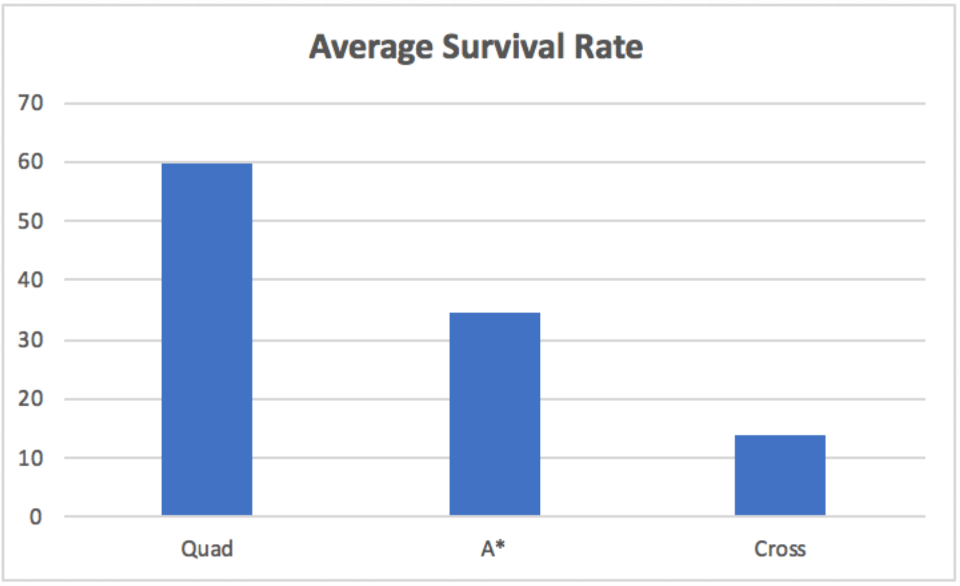


Test 2

I tested the survival strategies on the trickyClassic map. This test is designed to see if a more complex map has an effect on the strategy's performance in relation to one another. This map is larger than the standard one and has four ghosts instead of two. The Cross strategy is still much worse as expected but there the Quad strategy is significantly better than the A*. This strategy outperforms A* materially on this map because it is a larger more open world. The A* strategy was useful when there were lots of walls and corridors because it would tell you how a ghost would have to move to get around these walls and meet Pacman. In this world there are more ghosts and less walls and because it is open there is a high chance that a ghost won't be forced to take the shortest route. The assumption that the ghost will approach you from the shortest route does not hold as much and as Pacman only removes one direction when using the A* method it's likely that this could be the wrong direction. In contrast, the Quad method is a very conservative and removes at least two directions if the ghosts appear in one of the quadrants of the safe area. It is much more likely to evade danger and correctly guess which way the ghosts will approach from.



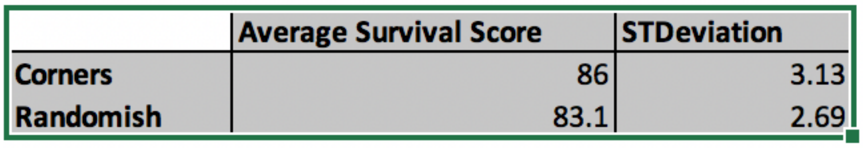
	Average Survival Score	STDeviation
Quad	59.8	4.96
A*	34.6	3.35
Cross	13.7	2.21



Corner

Test 1

This test is on the standard map against two ghosts. This test is designed to see which corner strategy (Randomish and Corners) led to the best performance by comparing the average survival rate. The results show that the Corners strategy led to a better average survival rate, however, with a small difference of around 3%. Because of Pacmans map of the world and the size of the map this final strategy is rarely used, because Pacman can find most of the food without using corners or moving randomly.



	Average Survival Score	STDeviation
Corners	86	3.13
Randomish	83.1	2.69