AI through Ms. Pac-Man

Abstract

Ms. Pac-Man - The Game:

Ms. Pac-Man is an arcade video game released in the early 1980s that introduced a female protagonist to the original Pac-Man game including several improvements like new maze designs that led to the popularity of the game. The motto of the game is fairly simple, that is for Ms. Pac-Man to earn as many points as she can by eating pills and power-pills all the while avoiding being eaten by four ghosts that exist within the maze. As the rounds increase, the speed increases, and duration of the ghosts' vulnerabilities reduces. Due to its increasing popularity, Ms. Pac-Man became quite an interesting subject for research in the field of Artificial Intelligence.

The Inspiration:

The co-relation between games (especially video games) and AI goes back a long way and Ms. Pac-Man has played a pivotal role in encouraging machine learning and development of computational intelligence. The official Ms Pac-Man Competition and its open framework lay a great foundation for AI enthusiasts to try a hand at developing best software controllers that could play Ms. Pac-Man. Throughout the course of this class we were given access to this framework and asked to implement various algorithms that are considered to lay the foundations of AI.

The Results:

Implementation of each algorithm is manifested into a controller that takes the current game state as an input where game state is a collection of information corresponding to the game such as maze layout, number of pills available, ghosts' positions etc. Each controller must return an output in the form of a move indicating the direction chosen by Ms. Pac-Man. The remainder of the paper is a comparative study of seven algorithms that were implemented along with a new algorithm implemented independent of any standard algorithms. The seven algorithms are namely Depth First Search, A * Search, Simulated Annealing, Evolution Strategy, Genetic Algorithm, MiniMax and K Nearest Neighbors.

Uninformed Search - Depth-first search (DFS)

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking. Often DFS can lead searches into infinite depths and therefore for all practical implementations it is reasonable to limit the depth of the tree search to a certain level for optimization.

Structurally, DFS operates on a stack as a base data structure. Using a stack child nodes of the current node that are found are pushed onto the stack and they are popped out in a last-in-first-out (LIFO) matter upon being visited. In a Depth Limited version of DFS, once a specified depth is reached, the current node is popped out and then their adjacent (sibling) nodes are traversed if found otherwise the traversal traces back to the parent.

In Pac-Man's context, the search begins with the given game state as the first node of the tree and all available moves from this node form its child nodes. All available moves from each child further form child nodes of the tree and this tree formation continues till the specified depth is reached. In this case the depth is limited to 7. For each child node found from the current node, scores of advancing the game to the child are formulated. Each node found is pushed into the stack and while the stack is not empty the nodes are explored and pushed on the stack. Once the max depth is reached the high score encountered during the search is returned as a result which is then compared with the last recorded highest score (resulting from traversal into other childe nodes). If the new high score is higher than the current highest score, a new move corresponding to the new

highest score is returned as a move for the controller. Therefore in DFS, tree traversal in search for highest score results in the best possible move for each game state.

In performance, it was found that this search doesn't result in great moves for Pac-Man. It was found that an average score of 350 points could be achieved using this algorithm. The images and code snippets for this controller can be found in the Appendix Section.

Informed Search – A *

In computer science, A* is a computer algorithm that is widely used in pathfinding and graph traversal, the process of plotting an efficiently traversable path between multiple points, called nodes. Noted for its performance and accuracy, it enjoys widespread use. A* to be superior to other tree traversal approaches as in concentrates on minimizing the effective cost of each operation. A* achieves better performance by using heuristics to guide its search. A * also bases the search on a destination node to which path costs are monitored.

At each iteration of its main loop, A* needs to determine which of its partial paths to expand into one or more longer paths. It does so based on an estimate of the cost (total weight) still to go to the goal node. Specifically, A* selects the path that minimizes $f(n) = g(n) + h(n)$ where n is the last node on the path, g(n) is the cost of the path from the start node to node n, and h(n) is a heuristic that estimates the cost of the cheapest path from n to the goal. The heuristic is problem-specific.

In Pac-Man's context, A * is implemented by first generating a graph very much similar to the kind used for DFS. Here each node in the tree is not a simple Pac-Man node, but also contains other details such as the node's parent, whether the node is visited or not, the cost incurred up to this node and well as the cost of a path leading to the destination. Two list 'open' and 'closed' are kept track of to monitor set of nodes to be evaluated as well as the ones that have already been evaluated. As mentioned above the heuristic to choose a move is based on the minimal value of the sum of cost incurred so far and cost of path to the destination from the current child node. Assume each available move from current node is a child node, the choice of the best move is based on the least value of f(n) that can be determined. The open and closed lists of nodes help move the nodes between the two lists to ease the process of decision making and avoiding revisiting of the nodes.

In performance, it was found that this search gives out much better results as compared to DFS with the score going up to 600 points using this algorithm. The images and code snippets for this controller can be found in the Appendix Section.

Informed Search – Simulated Annealing

Simulated annealing stems as an improvisation on the Hill Climber Algorithm that takes into account a highest scoring neighbor as the best possible option from among a range of neighboring nodes. It is a probabilistic technique for approximating the global optimum of a given function. Specifically, it is a metaheuristic to approximate global optimization in a large search space. It is often used when the search space is discrete. For problems where finding the precise global optimum is less important than finding an acceptable local optimum in a fixed amount of time, simulated annealing may be preferable to alternatives such as brute-force search or gradient descent.

Simulated annealing interprets slow cooling as a slow decrease in the probability of accepting worse solutions as it explores the solution space. Accepting worse solutions is a fundamental property of metaheuristics because it allows for a more extensive search for the optimal solution.

In Pac-Man's context, all available moves from the current game state are considered possible neighbors. For each neighbor to current game state, a score is evaluated and the acceptance of this score is decided based on the probability given by Math.*exp*((highScore - score) / temperature); Here temperature is an important parameter that decides the acceptance probability of the given node. When the temperature is high the probability of acceptance would be higher giving the algorithm to choose a bad neighbor with a high probability. But as temperature decreases on each iteration via annealing process, the acceptance probability would decreases there by helping the algorithm reach its local maxima at a faster rate. The advantages Simulated Annealing has over Hill Climbing includes the omission of having to look at all the neighboring nodes of the given node to find the best possible neighbor. However downsides of general local searches still exist in Simulated Annealing that involve the possibility of being limited to a local maxima and never being able to search the entire state space to find the global maxima.

In performance, it was found accepting nodes that are not the best with a certain probability bettered the odds of Pac-Man to advance in the game and scores up to 800 points were achieved. The images and code snippets for this controller can be found in the Appendix Section.

Evolutionary Search – Evolutionary Strategy

In computer science, an evolution strategy (ES) is an optimization technique based on ideas of adaptation and evolution. It belongs to the general class of evolutionary computation or artificial evolution methodologies. Most evolutionary searches involve generation of a population from a given population that tends to shift closer to the desired population of individuals that are generally considered to be fitter that their previous counterparts. Evolutionary Strategy in particular employs a technique of mutating each individual to a certain degree to be able to generate one off spring that is considered fitter than the parent. So given a population of size n (that is n individual exist in the population), n new off-springs are generated via mutation and thereby a population tending towards a goal population is evolved.

In Pac-Man's context, a set of all possible moves from the current node are considered to be members of the initial population. This initial population is iterated through and each individual is mutated to create a new offspring from it. The mutation is done one the basis of advancing a copy of the game using this individual and then choosing at random, one move from the set of new available moves from the new game state. Therefore, for each original available move a new move is selected at random from the moves that result by advancing the game using the given original move. From one to one mutation a new set of individual are generated of the same size as the original population. From the new population a fitness function is implemented on each individual to find the one that is most fit and that become the best fit move for the current game state that is passed onto the controller.

In performance, this controller seemed to better the chances of Pac-Man, aiding her to reach a score of 1000 points. The images and code snippets for this controller can be found in the Appendix Section.

Evolutionary Search – Genetic Algorithm

A genetic algorithm (GA) is a method for solving both constrained and unconstrained optimization problems based on a natural selection process that mimics biological evolution. The algorithm repeatedly modifies a population of individual solutions. A genetic algorithm (GA) is a search heuristic that mimics the process of natural selection. This heuristic (also sometimes called a metaheuristic) is routinely used to generate

useful solutions to optimization and search problems. Genetic algorithms belong to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems using techniques inspired by natural evolution, such as inheritance, mutation, selection and crossover.

In Pac-Man's context, a set of all possible moves from the current node are considered to be members of the initial population. This step is essentially same in both GA and ES. But the process of generating child population varies in both. In GA, the controller uses the technique of crossover to generate off-springs from two parents that are selected from the given population based on a heuristic. The heuristic here is a simple score function based on the resulting game state after advancing the game with the given individual move. This forms the crux of the fitness function. Now based on this fitness function two fittest parents from the population are selected for the process of crossover. The crossover takes place by generating more possible moves by advancing the game using selected parents and finally the resulting population which has a strength greater than that of the original population of size n, truncation is performed to limit the population back to a size of n. From the newly generated population, running the fitness function on each individual quickly determines the fittest individual of the new population which now becomes the best possible move for the current game state.

In performance, this controller seemed to better the chances of Pac-Man, aiding her to reach a score of 1000 points. The images and code snippets for this controller can be found in the Appendix Section.

Adversarial Search – MiniMax Algorithm

Minimax is a decision rule used in decision theory, game theory, statisticsand philosophy for minimizing the possible loss for a worst case (maximum loss) scenario. Originally formulated for two-player zero-sum game theory, covering both the cases where players take alternate moves and those where they make simultaneous moves, it has also been extended to more complex games and to general decision-making in the presence of uncertainty. Simply put MiniMax Algorithm follows a strategy of considering the state space as a game played by two opponents where the state space is explored using a search tree where at each layer of the tree a player tried to maximize his chances of winning based on who the player is. Taking an example of a two-ply game, where there are two players namely Max and Min, Max always tries to maximize his chances of winning say by maximizing his score while Min always tried to hinder Max's attempt by minimizing his score and chances.

In Pac-Man's context, all available moves from the current game state are considered possible options for each player in the game. While it is Max's turn, the heuristic of getScore is tried to be maximized. On the other hand while it is Min's turn, the getScore value is tried to be minimized for each possible move. In a recursive fashion Max and Min decided moves alternatively. Both player keep track of the lowest and highest score encountered so far and update these values which each exploration in the search tree to maximize their chance of winning. To avoid searching the state space indefinitely we limit the number of turns to 8.

MiniMax implements an adversarial search pattern which depicts a competitive environment. It was found that the performance of this algorithm increased the chances of Pac-Man scoring well. However it was found that the decision making was not as smooth as preferred since the Pac-Man would be stuck at a few points in the maze. The images and code snippets for this controller can be found in the Appendix Section.

Supervised Learning – K Nearest Neighbors

In pattern recognition, the k-Nearest Neighbors algorithm (or k-NN for short) is a non-parametric method used for classification and regression. In both cases, the input consists of the k closest training examples in the feature space. The output depends on whether k-NN is used for classification or

regression. Traditionally a training set is assumed that classifies data into various classes. Each entry in the set is then assumed to be plotted on a graph along with the new entry which is yet to be classified. K closest elements to the new entry are then observed and the class for the new entry is decided by the classifier based on the classes of the K nearest neighbors found in the training data.

In Pac-Man's context, k-NN controller takes into account a training set from an external file and converts them into a 2D array of information. The information held by this array includes the closest pill found for a given game state and whether or not a ghost was found within the path determined to the aforementioned closest pill. The data also contains the decision of whether Ms. Pac-Man decided to move towards or away from the pill detected. Given the new entry for the current game state, first the required information is gathered. That is the closest pill, its path, whether a ghost exists in this path or not are all determined. Once this info is gathered, we loop through the training data to find the k nearest neighbors. In this case k is set to be 5 and the closest 5 neighbors are found based on the distance calculated using the difference in the individual distances between the Pac-Man and the closest pill for each instance in the training set. Once the k nearest neighbors are determined, their decisions are observed and the current game state's decision is based on the majority of the decisions found in the 5 nearest neighbors.

In performance, it was found that this algorithm performed very well emulating a good supervised learning behavior. Ms. Pac-Man could now score up to 2000 point and sometimes cross over to level 2 with 3000 points. The images and code snippets for this controller can be found in the Appendix Section.

Custom Implementation

Throughout the implementation of above algorithms, some recurring patterns in features that influence Ms. Pac-Man's decision were found. The reliance on pill positions and ghost positions was noticed to be quite high. Therefore taking hints from the above mentioned k-NN algorithm, essential information was gathered from the game state before determining any moves or decision.

The custom implementation of the algorithm can be explained in the following steps. Firstly similar information like the kind gathered in k-NN is gathered here including the closest pill, closest pill distance and also the path to the closest pill. These values are all stored in global variables. Once ready, calculations are made to find if there is an edible ghost within the vicinity of Ms. Pac-Man. Since eating edible ghosts is rewarded with more points this steps helps Ms. Pac-Man score higher points. There are constraints on how far Ms. Pac-Man is willing to go to eat a ghost. Once an edible ghost is found in the maze and it is within the range for Pac-Man to eat, Pac-Man makes a move towards the edible ghost. If no edible ghosts exist, the decision is based on the closest pill found and whether or not there is a ghost either in the path of the pill or in the vicinity of the path determined. If a ghost exists in such a scenario, it is risky for Ms. Pac-Man to advance to the closes pill, therefore it moves away from any such ghost detected. At last if no ghost is ever determined (edible or not) the Pac-Man is free to move towards the pill detected without any limitation.

In performance it was found that relying the decision on three parameters as mentioned above help Ms. Pac-Man advance the game fairly well. It was found that she could now move to level 2 with close to 4000 points almost most certainly and in some cases it was found that she could cross over to level 3 as well.

Overall, the implementations of each algorithm helped explore a lot of features of the game state and proved to be a good exercise in gaining practical experience of implementing the generic algorithm as well as gain interest in motivating critical thinking.

Appendix

Uninformed Search - Depth-first search (DFS)

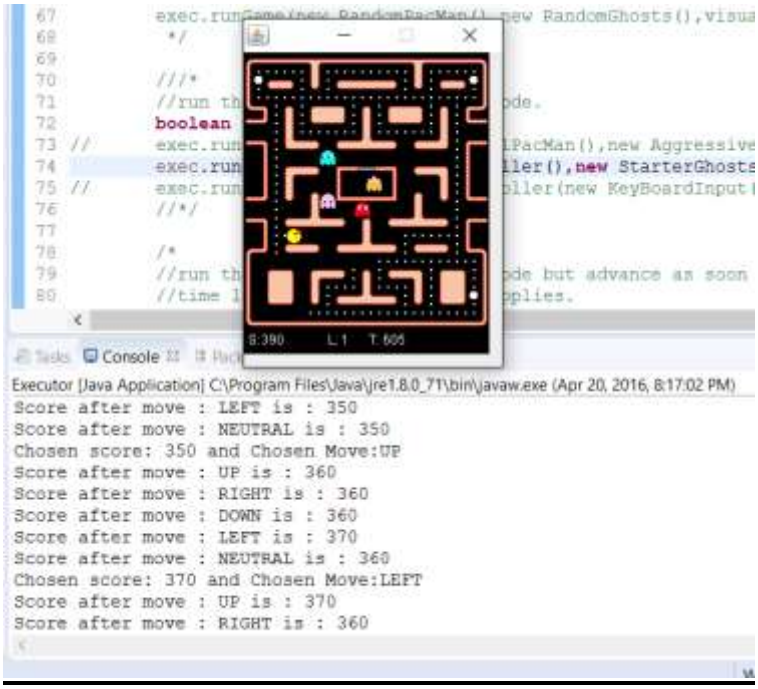


Figure showing Ms. Pac-Man for DFS.

```
public int getScoreFromNode(PacManNode gameState, int maxdepth)
{
    MOVE[] allMoves=Constants.MOVE.values();

    int highScore = -1;

    Stack<PacManNode> stackOfNodes = new Stack<PacManNode>();
    stackOfNodes.push(gameState);

    // while stack is not empty, find new nodes resulting from operations on the node at the top of the stack
    while(!stackOfNodes.isEmpty())
    {
        // Pop the stack and replace with list of child nodes
        PacManNode currentNode = stackOfNodes.pop();

        if(currentNode.depth >= maxdepth)
        {
            int score = currentNode.gameState.getScore();
            if (highScore < score)
                highScore = score;
        }
        else
        {
            // For each of the available moves, generate the resulting nodes from current node
            // push the new node to the top of the stack
            for(MOVE m: allMoves)
            {
                Game copyNode = currentNode.gameState.copy();
                copyNode.advanceGame(m, ghosts.getMove(copyNode, 0));
                PacManNode node = new PacManNode(copyNode, currentNode.depth+1);
                stackOfNodes.push(node);
            }
        }
```

Code Snippet showing implementation for DFS.

Informed Search – A *

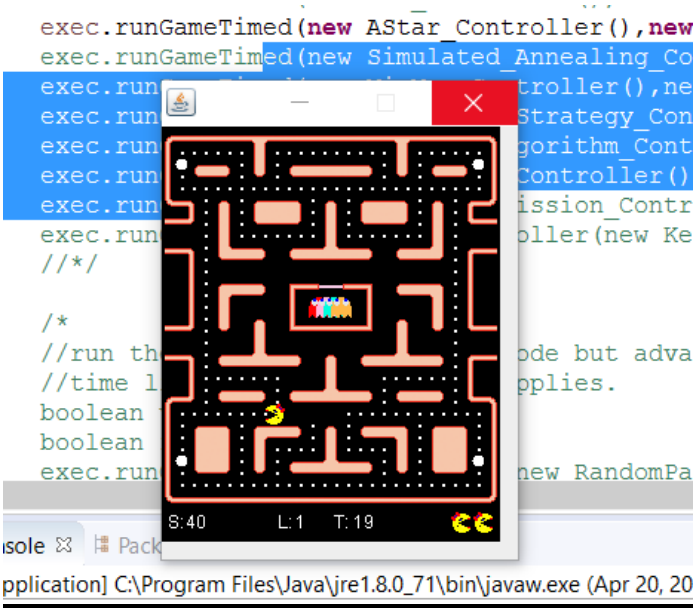


Figure showing Ms. Pac-Man for A * Search.

```
while(!open.isEmpty())
{
    N currentNode = open.poll();
    closed.add(currentNode);

    if (currentNode.isEqual(target))
        break;

    for(E next : currentNode.adj)
    {
        if(next.move!=currentNode.reached.opposite())
        {
            double currentDistance = next.cost;

            if (!open.contains(next.node) && !closed.contains(next.node))
            {
                next.node.g = currentDistance + currentNode.g;
                next.node.h = game.getShortestPathDistance(next.node.index, target.index);
                next.node.parent = currentNode;

                next.node.reached=next.move;

                open.add(next.node);
            }
            else if (currentDistance + currentNode.g < next.node.g)
            {
                next.node.g = currentDistance + currentNode.g;
                next.node.parent = currentNode;

                next.node.reached=next.move;

                if (open.contains(next.node))
                    open.remove(next.node);
```

Code Snippet showing implementation for A *

Informed Search – Simulated Annealing

Figure showing Ms. Pac-Man for Simulated Annealing.

```
            {
                int score = currentNode.gameState.getScore();
                double prob = getProbability(highScore, score, temp);
                // Decide if we should accept the neighbor
                if (prob > Math.random())
                    highScore = score;
            }
            else
            {
                // For each of the available moves, generate the resulting nodes from current node
                // push the new node to the top of the stack
                for(MOVE m: allMoves)
                {
                    Game copyNode = currentNode.gameState.copy();
                    copyNode.advanceGame(m, ghosts.getMove(copyNode, 0));
                    PacManNode node = new PacManNode(copyNode, currentNode.depth+1);
                    queue.add(node);
                }
            }
        }
        // reduce temperature
        temp *= 1-coolingRate;
        return highScore;
    }

    private double getProbability(int highScore, int score, double temperature) {

        // accept better solution
        if (score < highScore) {
            return 1.0;
        }
        // get probability of acceptance
        return Math.exp((highScore - score) / temperature);
    }
```

Code Snippet showing implementation for Simulated Annealing.

Evolutionary Search – Evolutionary Strategy

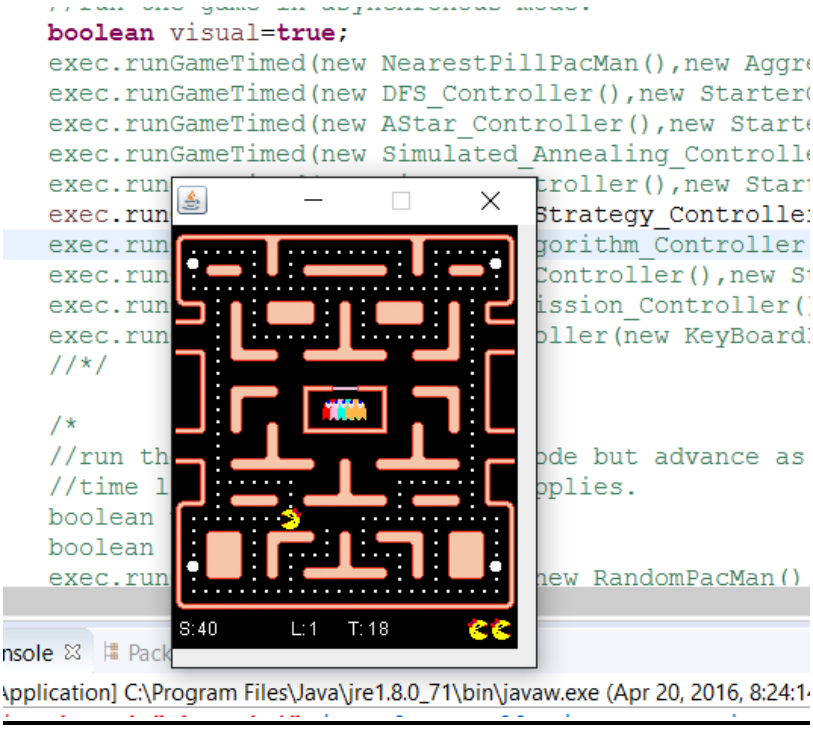


Figure showing Ms. Pac-Man for ES.

```
private MOVE[] esFunction(Game state,long timeDue) {

    MOVE[] newPopulation = null;
    int i = 0;
    for (MOVE m : population)
    {
        newPopulation[i] = mutate(state,timeDue,m);

    }
    return newPopulation;
}

private MOVE mutate(Game state,long timeDue, MOVE move) {

    MOVE[] newPopulationSet;

    Game gameCopy1 = state.copy();
    gameCopy1.advanceGame(move, ghosts.getMove(gameCopy1, timeDue));
    newPopulationSet = gameCopy1.getPossibleMoves(gameCopy1.getPacmanCurrentNodeIndex());
    MOVE individual = getRandom(newPopulationSet);
    return individual;

}

public static MOVE getRandom(MOVE[] array) {
    int rnd = new Random().nextInt(array.length);
    return array[rnd];
}

private int fitnessFunction(Game state,long timeDue,MOVE m) {

    Game gameCopy = state.copy();
    gameCopy.advanceGame(m, ghosts.getMove(gameCopy, timeDue));
    int fitness = 0;
    int currentIndex = gameCopy.getPacmanCurrentNodeIndex();
    fitness = gameCopy.getClosestNodeIndexFromNodeIndex(currentIndex, gameCopy.getPowerPillIndices(),DM.PATH);
    return fitness;
}
```

Code Snippet showing implementation for ES.

Evolutionary Search – Genetic Algorithm

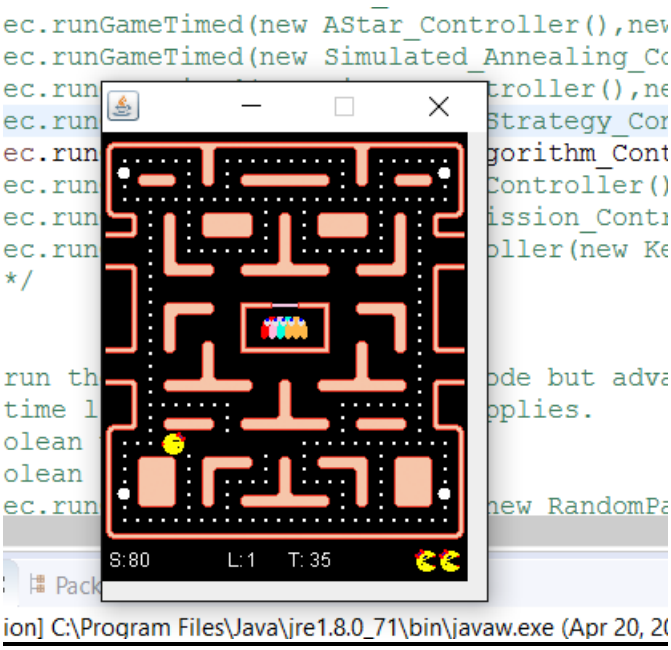


Figure showing Ms. Pac-Man for GA.

```
private MOVE[] gAFunction(Game state,long timeDue) {

    MOVE[] newPopulation = population;
    MOVE parent1 = selectParent(state,timeDue,population);
    MOVE parent2 = selectParent(state,timeDue,removeIndividual(population,parent1));

    newPopulation = crossover(state,timeDue,parent1, parent2,population.length);
    return newPopulation;
}

private MOVE[] crossover(Game state,long timeDue, MOVE parent1, MOVE parent2, int populationSize) {

    MOVE[] newPopulationSet1, newPopulationSet2;

    Game gameCopy1 = state.copy();
    gameCopy1.advanceGame(parent1, ghosts.getMove(gameCopy1, timeDue));
    newPopulationSet1 = gameCopy1.getPossibleMoves(gameCopy1.getPacmanCurrentNodeIndex());

    Game gameCopy2 = state.copy();
    gameCopy2.advanceGame(parent2, ghosts.getMove(gameCopy2, timeDue));
    newPopulationSet2 = gameCopy2.getPossibleMoves(gameCopy2.getPacmanCurrentNodeIndex());

    MOVE[] newPopulation = merge(newPopulationSet1, newPopulationSet2);
    return elite(state,timeDue,newPopulation, populationSize );
}

private MOVE selectParent(Game state,long timeDue,MOVE[] population) {
    MOVE individual = null;
    int highestScore = -1;
    int score = 0;
    for (MOVE m : population)
    {
        score = fitnessFunction(state,timeDue,m);
        if (score > highestScore) {
            highestScore = score;
            individual = m;
        }
```

Code Snippet showing implementation for GA.

Adversarial Search – MiniMax Algorithm

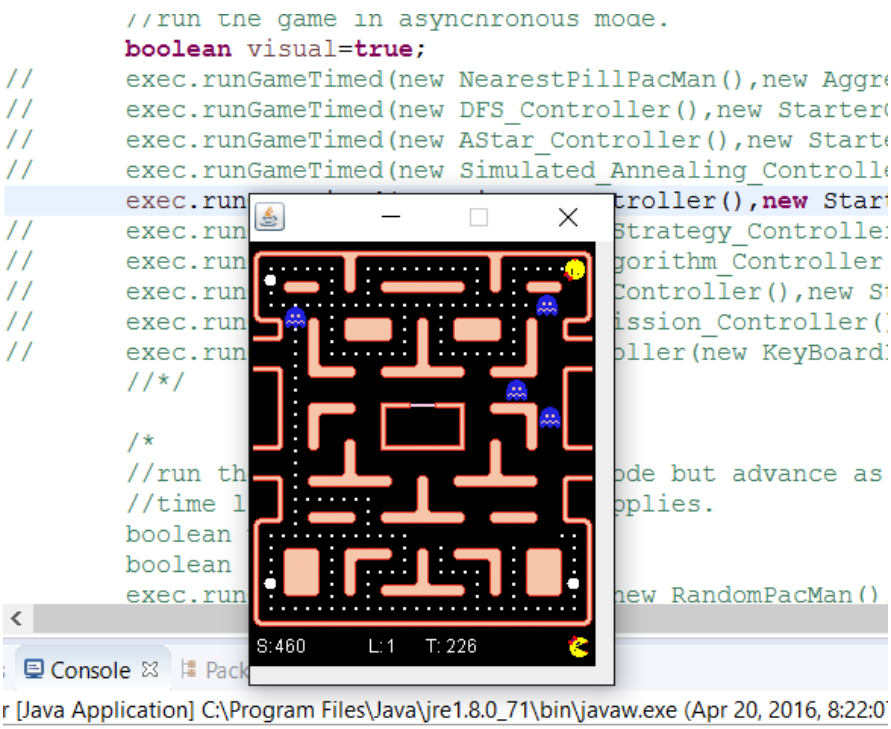


Figure showing Ms. Pac-Man for MiniMax.

```
        return lowestScore;
    }

    // min player move and score

    private int max(Game state,long timeDue,int depth) {
        if (depth == MAX_DEPTH)
            return eval(state);

        int highestScore = Integer.MIN_VALUE;
        int score = 0;

        depth++;
        MOVE[] moves = state.getPossibleMoves(state.getPacmanCurrentNodeIndex());

        for (MOVE move : moves) {

            Game stateCopy = state.copy();
            stateCopy.advanceGame(move, ghosts.getMove(stateCopy, timeDue));
            score = min(stateCopy,timeDue, depth);

            if (score > highestScore)
                highestScore = score;
        }

        return highestScore;
    }

    /**
     * Returns -1 if game is over or else 1
     */
    private int eval(Game state) {
        if (state.gameOver())
            return -1;
        else return 1;
    }
```

Code Snippet showing implementation for MiniMax.

Supervised Learning – K Nearest Neighbors

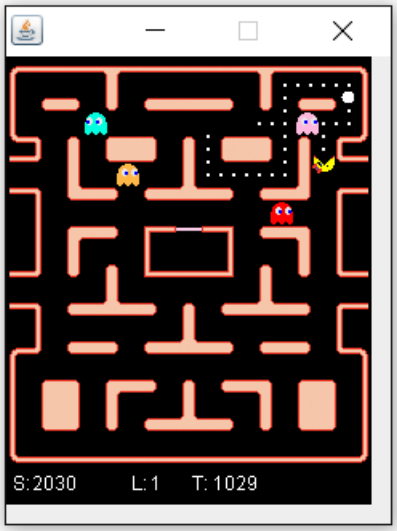
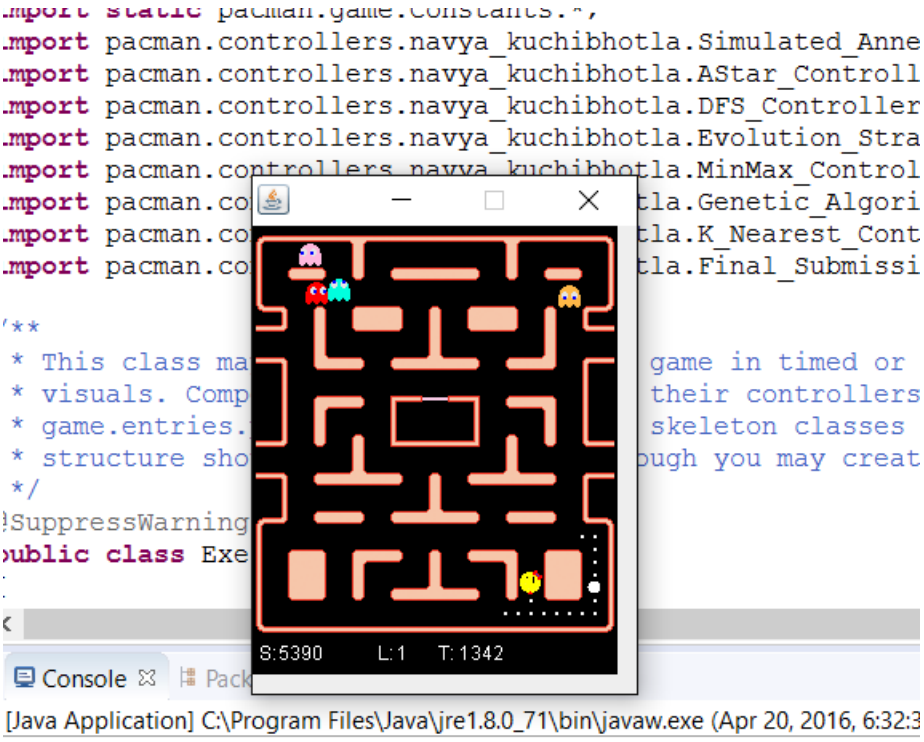


Figure showing Ms. Pac-Man for k-NN.

```
private void analyzeState(Game game) {

    int current = game.getPacmanCurrentNodeIndex();
    int[] pills = game.getActivePillsIndices();
    int[] powerPills = game.getActivePowerPillsIndices();

    int[] targetsArray = new int[pills.length + powerPills.length];
    System.arraycopy(pills, 0, targetsArray, 0, pills.length);
    System.arraycopy(powerPills, 0, targetsArray, pills.length, powerPills.length);

    c_pill = game.getClosestNodeIndexFromNodeIndex(current, targetsArray, DM.PATH);
    c_pill_path = game.getShortestPath (current, c_pill);
    c_pill_dist = game.getShortestPathDistance(current, c_pill);

    ghostInPath = ghostFoundInPath(game,game.getShortestPath(current, c_pill));
}

private int ghostFoundInPath(Game game, int[] path) {
    int ghostFound = 0;
    for (GHOST ghost : GHOST.values()) {
        int ghostIndex = game.getGhostCurrentNodeIndex(ghost);
        for(int i= 0; i < path.length; i++) {
            if(path[i] == ghostIndex){
                ghostFound = 1;
                break;
            }
        }
    }
```

Code Snippet showing implementation for k-NN.

Custom Implementation



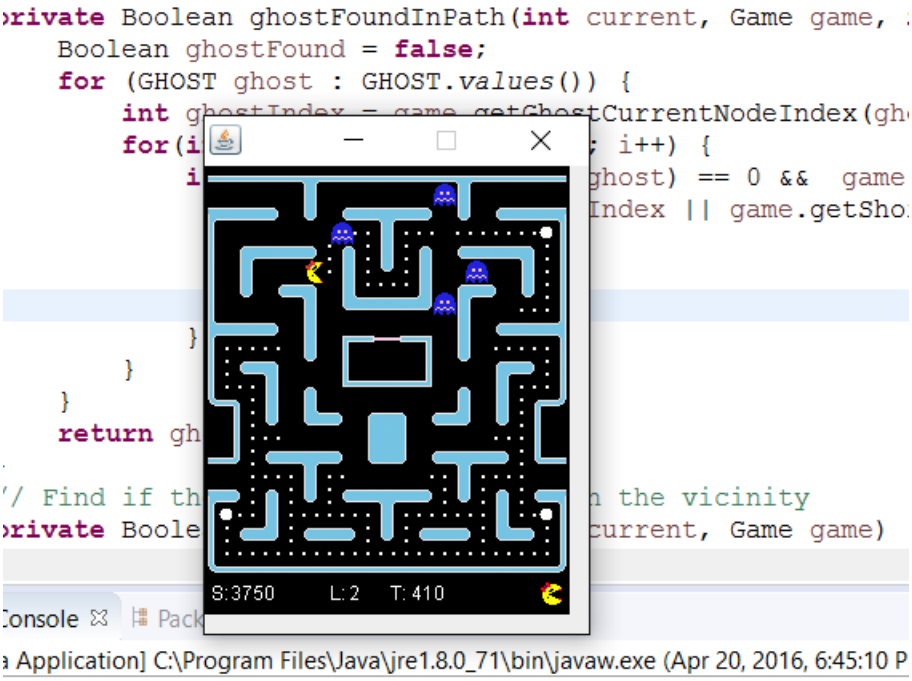


Figure showing Ms. Pac-Man for Final Submission Controller.

```java
public MOVE getMove(Game game, long timeDue) {

    int current = game.getPacmanCurrentNodeIndex();

    gatherData(game);
    int ghostIndex;
    // ghost is found in path so run away unless it is edible
    if(edibleGhostFound) {
         ghostIndex = game.getGhostCurrentNodeIndex(closestGhost);
        return game.getNextMoveTowardsTarget(current, ghostIndex, DM.PATH);
    }
    else if (ghostFoundInPath) {
         ghostIndex = game.getGhostCurrentNodeIndex(closestGhost);
        return game.getNextMoveAwayFromTarget(current, ghostIndex, DM.PATH);
    }
    else
        return game.getNextMoveTowardsTarget(current, c_pill, DM.PATH);

}

private void gatherData(Game game) {

    int current = game.getPacmanCurrentNodeIndex();

    int[] pills = game.getActivePillsIndices();
    int[] powerPills = game.getActivePowerPillsIndices();

    int[] targetsArray = new int[pills.length + powerPills.length];
    System.arraycopy(pills, 0, targetsArray, 0, pills.length);
    System.arraycopy(powerPills, 0, targetsArray, pills.length, powerPills.length);

    c_pill = game.getClosestNodeIndexFromNodeIndex(current, targetsArray, DM.PATH);
    c_pill_path = game.getShortestPath (current, c_pill);
    c_pill_dist = game.getShortestPathDistance(current, c_pill);
    edibleGhostFound = edibleGhostFound(current, game);
    ghostFoundInPath = ghostFoundInPath(current, game,c_pill_path);

}
```

```java
private Boolean ghostFoundInPath(int current, Game game, int[] path) {
    Boolean ghostFound = false;
    for (GHOST ghost : GHOST.values()) {
        int ghostIndex = game.getGhostCurrentNodeIndex(ghost);
        for(int i= 0; i < path.length; i++) {
            if(game.getGhostLairTime(ghost) == 0 &&  game.getGhostEdibleTime(ghost) == 0 &&
                    (path[i] == ghostIndex || game.getShortestPathDistance(current, ghostIndex) <= DIST_MARGIN )){
                ghostFound = true;
                closestGhost = ghost;
                break;
            }
        }
    }
    return ghostFound;
}
// Find if there is an edible ghost in the vicinity
private Boolean edibleGhostFound(int current, Game game) {

    closestGhostDist = Integer.MAX_VALUE;
    int tmp;
    for (GHOST ghost : GHOST.values()) {
        if (game.getGhostLairTime(ghost) == 0) {
            tmp = game.getShortestPathDistance(current, game.getGhostCurrentNodeIndex(ghost));
            if (tmp < closestGhostDist)
                closestGhostDist = tmp;
            closestGhost = ghost;
        }
    }
    if(closestGhost != null && game.isGhostEdible(closestGhost)){
        int ghostPathDist = game.getShortestPathDistance(current, game.getGhostCurrentNodeIndex(closestGhost));
        if( game.getGhostEdibleTime(closestGhost) >= ghostPathDist && ghostPathDist <= (c_pill_dist + DIST_MARGIN))
            return true;
    }
    return false;
}
}
```

Code Snippets showing implementation for Final Submission Controller.

References

1. https://en.wikipedia.org/wiki/ - All definitions are referenced from Wikipedia
2. http://www.cin.ufpe.br/~tfl2/artificial-intelligence-modern-approach.9780131038059.25368.pdf