ME 133b Project Report: Pacman!

Nishka Arora, Shevali Kadakia, Krish Mehta

I. INTRODUCTION

After a term of learning about path-finding, mapping, and Kalman filters, we thought it would be interesting to integrate all three pieces into our final project. We decided to implement the Pacman game with a few variations. Through this project, we explored and modified the A* and D* algorithms to help Pacman find its way through the map. We also used a modified version of occupancy maps and Kalman filters to track the locations of the ghosts and obstacles for Pacman to dodge.

II. DESCRIBING THE TASK

The objective of the Pacman game is to have Pacman, the yellow object, collect as many pellets on the grid as possible while dodging the ghosts along the way. Pacman is usually controlled by the user, and there are typically pellets placed in every grid box. However, in our version of Pacman we made some modifications. Instead of having pellets in every grid box, we had a select few pellets that Pacman had to find its way to. We also gave Pacman no information about the map, so it had to discover the map's walls and bounds itself. In addition to this, instead of a user controlling Pacman, Pacman finds its way around the map itself by taking one step every 0.5 seconds towards the pellet it deems best at the moment. By making these modifications to the game, we were able to use ideas of path-finding (to the closest pellet), mapping (building a map for Pacman), and filtering (to estimate the position of the ghosts). To stay true to the original game, we recreated the original Pacman map, and we also programmed the ghosts to operate following a very similar algorithm as in the original Pacman game, and added the retro background music.

We made it Pacman's objective to collect each pellet across the map while dodging the ghosts. The information that Pacman was given was the location of the pellets and sensors that could detect obstacles and ghost pings. Ghost pings are signals that each ghost sends out, and the ghost ping contained information about the ghost's current location. Each ghost has its own ping that is broadcasted every few seconds. In an attempt to win the game, Pacman finds a path to the pellets using either A* or D* (discussed further later). As Pacman moves, since it has no prior knowledge of the map, Pacman uses its sensors (distance) to learn about free spaces and walls as it moves through the map. It uses its sensors to also detect ghost pings and uses Kalman filters to predict the ghost positions given their ping, and takes this into consideration while pathfinding.

III. DESCRIBING THE GENERAL SYSTEM

Our thought process for figuring out how to implement this game is as follows.

First, we knew that the Pacman map was a discrete grid and we had a select few pellets that the Pacman had to find a path to. Because of this, we decided to try using A* and D* where each grid box was associated with a cost. When trying to implement these algorithms, we saw that A* worked well, but D* did not perform well.

For our final product, we decided to stick with A* with a slight modification. We decided to not give the Pacman AI information about the walls on the map. Instead, we had Pacman localize itself on the map using sensors. The four sensors would gather information about obstacles in the up, down, left, and right directions. The width of the path in Pacman is only one unit, so at every point in the map, the blocks diagonal to Pacman are walls. So, distance sensors on diagonals may not be as valuable. We update the occupancy map, accordingly.

The ghost AI was programmed to be very similar to the ghosts in the original game.

One goal of the game is for Pacman to avoid the ghosts. In order to do that, we have ghosts emit a ping every few seconds. These pings contain true information about the ghost's current position. This information is given to Pacman. Pacman has a predicted ghost position map for each ghost. When a ping is received, the position of the ghost is "known" to Pacman with probability 1. At every subsequent step, Pacman estimates the ghost's future position by spreading the ghost probability to the possible places that the ghost could be. Initially, this was programmed with the true wall-map, i.e. the predicted ghost position would only be in non-wall areas. Then, we realized that Pacman shouldn't have access to the true wall map. So, we programmed another version where the ghost's future predictions are made using the estimated occupancy grid map.

IV. IMPLEMENTATION

A. General Map

The general map was the manager that was in charge of the rendering of the map and facilitating the communication between Pacman, its sensors, and the pathfinding.

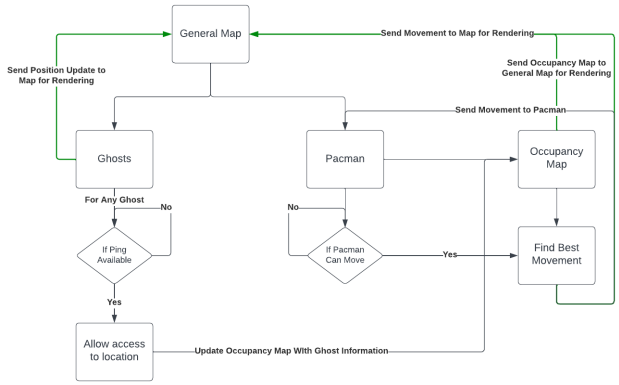

Fig. 1. Information Flow

The above diagram shows the information flow. There are three distinct cycles that are being run: One controlling the ghost's movement (not depicted in the diagram), one controlling Pacman's movement, and one controlling if the ghosts are sending pings.

The generalized map then collected information from the three "nodes" and then renders it on demand.

B. Occupancy Map

As discussed before, Pacman AI is not given the true map of the world. That is, the true positions of the walls are not known. Pacman must map the world using sensors. We do this using Occupancy Grid Maps. We use log-odds ratios to compute the probability for increased numerical stability (as opposed to multiplying out joint probabilities directly). The mathematical expressions that define our algorithm to find the probability of the map $p(m|z_{1:t}, x_{1:t})$:

1) The random variable in question is $m = m_i$ where

$$m_i = \left\{ \begin{array}{ll} 1, & \text{if } m_i \text{ is wall} \\ 0, & \text{if } m_i \text{ is free} \end{array} \right\} \quad (1)$$

Each map position has a binary value defining whether it's occupied. The true values are stored in the true map that Pacman does not have access to.

2) $l_{t,i} = \log \frac{p(m_i|z_{1:t},x_{1:t})}{1-p(m_i|z_{1:t},x_{1:t})}$ Here, $l_{t,i}$ defines the log-odds of the probability that m_i is occupied. These are usually set constant value that are added or subtracted from the grid points that are in the path of a laser. If the laser/ray from the sensor is over a box where there is no wall, a constant L_{FREE} is subtracted from the log-odds ratio of that block i. If the sensor ray passes over a walled block $L_{OCCUPIED}$ is added to the log-odds ratio i. These constants are set at the beginning of the program and can be varied. A larger magnitude for L_{FREE} and $L_{OCCUPIED}$ means that Pacman is more confident in its sensors (in Bayesian terms, updates are easier).

3) The probability $p(m|z_{1:t}, x_{1:t})$ is recovered from the log-odds ratio by the following expression:

$$p(m|z_{1:t}, x_{1:t}) = 1 - \frac{1}{1 + \exp\{l_{t,i}\}}$$

Pacman has 4 sensors (up, down, left, right). Each sensor has a search radius (the maximum number of grids it can see in one direction). The $search\_radius$ is a constant that can be edited. A laser ray from a sensor can go up to a maximum distance of the search radius. A larger search radius makes mapping faster (especially for parts of the map that are free). If the ray is blocked by a wall, the ray cannot go past the wall.

Note that all that Pacman knows beforehand is the width and height of the true map. Since we need to simulate the sensor rays (have each ray be blocked where a wall exists), our script does use the true wall map as an input. The pseudocode used to update the log-odds ratio map is written out in Algorithm 1. The update method is executed every time Pacman moves. That is, every time Pacman moves, it collects more data using its sensors and updates what it knows about the world. In order to use the log-odds ratio maps, they must be converted back to probability maps. That is done by applying Equation 3 to the log-odds ratio $l_{t,i}$ of every grid position i. This probability map is accessed by Pacman to do its path planning (described in part D). It is also used to predict the future positions of the ghosts (described in part C).

Finally, the mapping is visualized in Figure 1.

Algorithm 1 Update Occupancy Map $p(m|z_{1:t}, x_{1:t})$:

```
search_radius, L_FREE, L_OCC
cx, cy ← current true Pacman location
log_odds = [0]_{width×height}
for move ∈ [(0,1),(1,0),(−1,0),(0,−1)] do
    dx, dy ← move
    for step ∈ {0,1,...,search_radius} do
        cur_x, cur_y ← cx + step*dx, cy + step*dy
        if (cur_x, cur_y) ∈ valid map positions then
            if (cur_x, cur_y) is true wall then
                log_odds[cur_x, cur_y] += L_OCC
                break inner for loop (obstacle reached)
            else
                log_odds[cur_x, cur_y] −= L_FREE
        else
            break inner loop (ray is outside the map)
```

C. Ghost Localization

Not only does Pacman have to make his way to the Pellets, but he also must avoid the ghosts that are heading toward him. The way we set this up is every few time-steps each ghost broadcasts a ping to Pacman with its true location. At that point Pacman knows with probability 1 that the ghost is at the broadcasted location. Then, at each time step, the ghost can take either an up, down, left, or right step. So, the probability distribution of the predicted ghost position to be will spread accordingly. This is exemplified in Figure 2 and visualized on the original Pacman map in Figure 3.

Each ghost has its own prediction map. The general idea for the algorithm is that when a ghost ping is received,

Fig. 2. On the left is the true game map with the occupancy map overlayed in alpha. On the right is the occupancy map. In this example, Pacman has explored the top part and the left part of the map. Color maps probabilities from red to green where red represents wall and green represents free. Yellow represents unexplored/unsure parts of the map.

Pacman throws away the current estimate it has for the ghost and starts over with probability 1 at the location in the ping. Then, at each time step the pinged estimate "spreads" to different places the ghost could be. To achieve this, at each grid point that is in the support of the probability distribution of ghost location, we divide the probability of that point between all valid next moves.

On implementing this we realized that this indirectly gives Pacman direct access to the map because in order to predict the next moves from a ghost position in the support, Pacman will require the true wall map. In order to rectify this we made a modification to the algorithm. Now, if the probability of a grid in the support being the current ghost location is $p[i, j]$, then $p[i, j]$ will be distributed among all possible next moves $(i, j + 1), (i + 1, j), (i - 1, j), (i, j - 1)$ according to the probability that Pacman thinks that position is a wall in the occupancy map from Section C (because ghosts can't be on walls). This would prevent Pacman from having any more information about the probability that a grid square is a wall than was acquired through the lasers in the occupancy map part.

Figure 4 can be compared to Figure 3 visualizes how the probability distribution of the predicted position of ghosts now includes walls but proportionally to the probability that Pacman believes it is a wall.

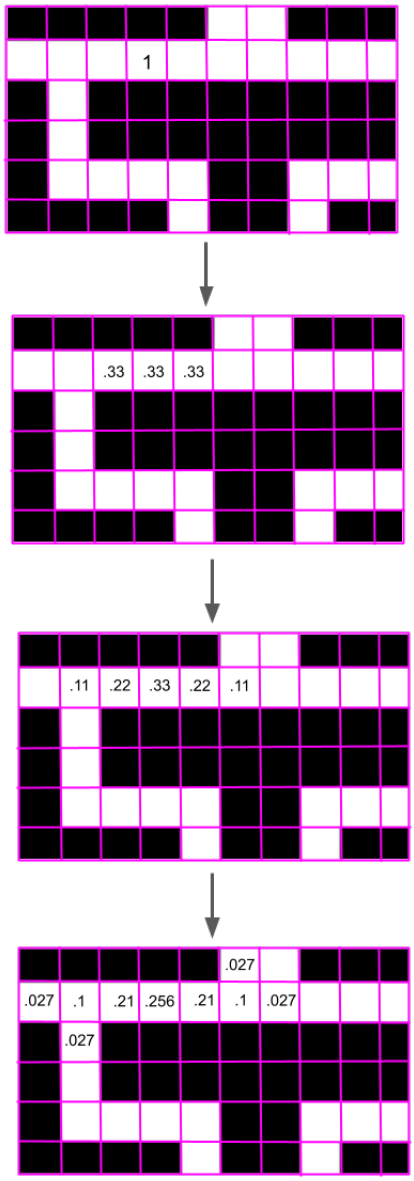

Fig. 3. This figure shows the probability distribution update step for a single ghost. Initially, the ghost is in the ping broadcasted location. With each non-ping time step, the distribution's support is spread to valid locations on the graph.

D. A* / D*

1) Setup of A/D*:*

In order to run the A*/D* algorithm, we needed to create a map such that each grid space was a node that had the following information:

- node cost
- node parent
- node neighbors
- node status
- node location

The node cost was calculated using the cost to go to the node, the cost of the node spot, and the cost to go to the goal. Since A*/D* builds the graph such that the root is the final destination and the tree searches for the start, we set the goal to be the starting position of Pacman, and the root of the tree as the pellet location. The cost to go to the goal for each node was the distance from the node's position to Pacman's position. We used the following equation to calculate the distance:

$$\text{cost} = |\text{node row} - \text{Pacman row}| + |\text{node col} - \text{Pacman col}|$$

Since Pacman cannot move diagonally, finding the number of rows and columns that Pacman would need to traverse is a more accurate representation of distance. For the cost to go to the node, we calculated this by counting the least

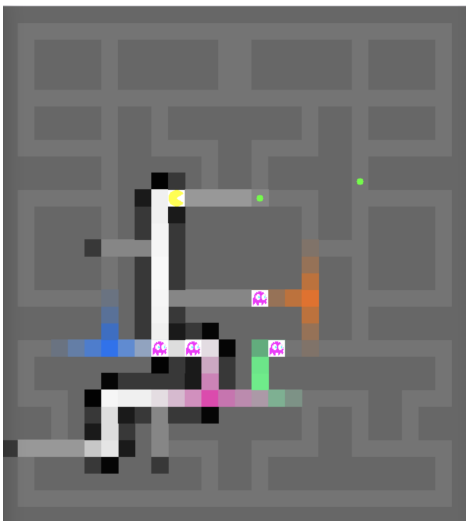

Fig. 4. Each color - pink, orange, blue, green - represents the probability distribution of the location of a specific ghost.

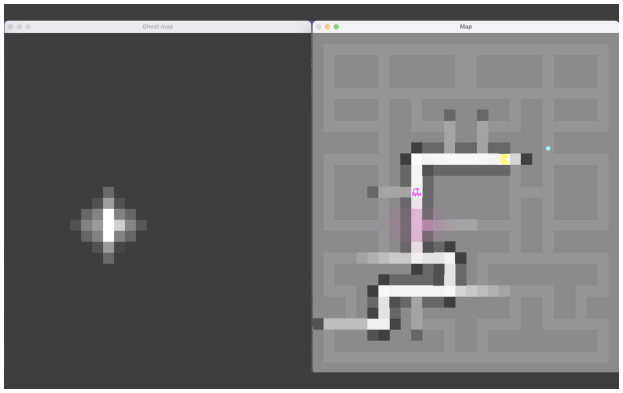

Fig. 5. Distribution for a single ghost using the occupancy map. Non-zero values even in wall positions. Value in non-wall positions is higher than wall positions.

number of steps required to get to that spot. Each step had a value of 1. For the cost of the node spot, this cost was determined by the probability of the spot being a wall or free space. We wanted the cost of the wall to be greater than the cost of a free space. To do this we initially scaled the probabilities by 10. However, some issues we encountered was that when Pacman got close to the pellet, the path finding algorithm would choose the path through the wall because it was cheaper than walking around the wall. To take care of this issue, we decided to set the cost of the wall to be exponential such that the cost of a node with a probability of 0.2 would be significantly less expensive than cost of a node with a node with a probability of 0.9. Thus, we calculated the cost of the node by doing $e^{10 \cdot \text{probability}}$. This produced the graph in Figure 1.

From Figure 1 can see that if the proability is near 1 then the cost is significantly higer than if the cost were closer to 0.2. This scaling allows us to ensure that Pacman does not find a path through the wall regardless of the additional distance required to travel to get to the pellet. Thus, the total

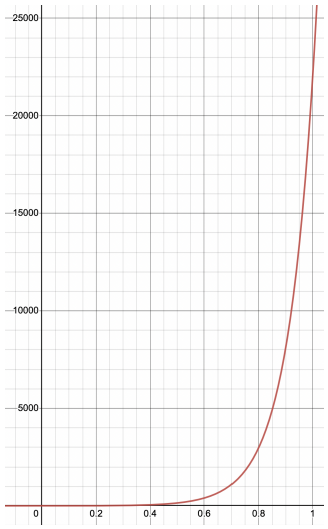

Fig. 6. e^{10x} Scaling

cost of a path is the sum of all of the nodes' cost to go to the goal, the nodes' cost from the root, and the cost of the nodes themselves.

In regards to the node's parent and neighbors, the parent was important to keep track of because it helped find the exact path that would lead to the goal, and the neighbors were important to keep track of to learn how to grow the tree for A*/D*. The node status kept track of the status of the node in the onDeck queue. Since we were trying to implement D*, the different possible statuses for the nodes were processed, ondeck_remove, ondeck_expand, and unknown. Lastly, the node location was the coordinates of the node. Since each grid box had a node associated with it, we mapped nodes to their grid box equivalents through their locations.

2) Issues With D:* We initially believed that D* would work for this project because the essence of D* is updating a path as you learn more about it. However, when we followed the basic implementation of D* we saw that it did not work. We initially tried to troubleshoot by fixing the ordering of the onDeck queue. Since we knew that multiple walls were changing based on the sensor readings, not because Pacman ran into a wall, we knew that the walls' costs were constantly fluctuating. This would mean that we would have to reorder the onDeck queue based on the new probabilities of each node. This, however, did not work as the path was not the shortest path to the pellet. Additionally, we tried to insert the start node back into the onDeck queue, but this caused a loop in the tree. We hypothesize that this loop was created because of the fluctuating costs. Moving outwards does not correlate to a constant increase in costs because walls cost significantly more than free spaces. Thus in some cases, because of the fluctuating costs, the parent of the a node was its neighbor, and the parent of the neighbor was the original node, hence the loop. We also tried not placing the start node back into the onDeck to prevent this forever looping, but this resulted in the onDeck queue sometimes being emptied before the goal node was found, and thus an error.

After all of this, we realized that there was a fundamental difference in the way that we wanted D* to run and the way that we learned how the D* algorithm works. The difference is that the D* algorithm that we learned only takes into account costs increasing at a node along one of the branches of the tree. Thus, D* adjusts the paths of the current branches to look for a better path. This works in the case of increasing costs because the branches have already found the best paths, thus finding an invalid node along a branch does not mean the entire tree needs to be recalculated. Only the edge of the branch needs to be adjusted, which is why the onDeck queue can be saved and reused. However, in our case, we have that the costs increase and decrease. Thus, if we find a better path because the cost along that path is now less than the current path that we're on, the new path does not get accounted for because those nodes were not processed when making the D* tree. One way we thought of potentially resolving this was by adding any node which decreased its cost onto the onDeck queue. However, we realized that adding that node meant that were essentially adding an leaf not attached to the D* tree, so it would cause issues in path finding algorithm. Thus, after all these attempts at trying to get D* working, we realized that the fundamental difference would not be fixed unless we significantly modified the D* algorithm.

3) Modified Version of A:* Since the first run of D* is creating a tree from the root, we decided to constantly call the first run of D*, which is equivalent to A* except that there are more statuses for the nodes (when a node is onDeck it can be onDeck remove or onDeck expand). In addition to this, we constantly changed the goal position by choosing the pellet that has the lowest distance from the current position. This distance was calculated using the same formula to calculate the cost to go from the current node to the goal node. The reason we chose this method for distance calculation for finding the next pellet is that our other option was finding the path with the lowest cost. However, given that the map costs are constantly updating, and that it would be computationally expensive to run multiple D*s every time Pacman moves, a better estimation for choosing the next pellet was the estimated number of steps to take.

We thought about implementing a version of D* which would update the status of all the nodes. This means that nodes that are walls would be marked as to be burned so they would not be explored again. We also thought about only updating the path only when the map encountered a change. However, for both cases we realized that since the map is constantly updating because of the locations of the ghosts as well. Thus, the map is constantly changing probabilities so it is best to run the altered version of A*/D*.

E. Ghost AI

The ghost AI was implemented very similar to the AI of the game Pacman. Due to the computational restrictions of Pacman, the ghost AI was extremely simple. The ghost continued to move in the direction it was moving, without backing on its own path, until it has to make a decision. The only spots where the ghost has a choice would be at any 3 or 4-way intersections on the map.

At an intersection, the decision was made simply: Which of the next squares is the closest to Pacman's current location, and it chose that direction to travel in, until a new intersection. This means that sometimes the ghost would take a longer path, which is not optimal. The image below shows the decision-making process.

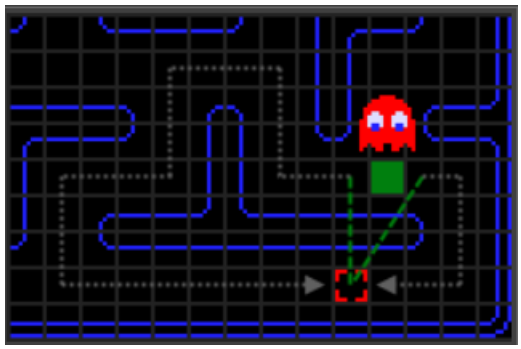

Fig. 7. Ghost Deciscion

In the image, the ghost has to make a decision at the green block, of going left or right. It calculates the distances of the two green lines to the target square (red box), and goes towards the left since that line is shorter. If you notice, this is not the optimal path for the ghost, but it requires $O(1)$ computation which is exceptional for any path-finding in an $n \times n$ grid.

V. Analysis

A. Changing Number of Ghosts

The number of ghosts was increased from 1 to 4, and 5 pellets were used at some set locations. For 1 ghost, Pacman always avoided the ghost, since it managed to get "ahead" of the ghost, and with the speeds of both the ghost and the Pacman being set the same, the ghost can never catch up to the Pacman if it is behind the Pacman.

As the number of ghosts increases to 4, there are more and more instances of Pacman being "cornered". This leads to Pacman having no way of avoiding the ghosts unless all the ghosts end up "behind" Pacman. Since they all target the same location, the ghost, sometimes "stack" up on each other, making one of them redundant. Thus, depending on how the pellets are placed and ghost starting locations, the ghosts sometimes can corner Pacman, and other times they just fall behind and stack up.

B. Changing the Relative Speeds of the Pacman and Ghost

In a one-ghost scenario, if the Pacman has the same speed as the ghost, the Pacman will always be ahead of the ghost, (if the ghost is ahead the Pacman will choose a different route) and this will ensure that the Pacman will never get caught. If the ghost's speed is double that of Pacman, the ghost will always catch Pacman before Pacman can collect a few pellets, unless the pellets are very close together and the ghost starts far away.

With 5 randomly placed pellets, and 4 ghosts (standard Pacman game) having a speed of 80% Pacman, Pacman usually gets cornered after collecting 2-3 pellets. To win a 5 pellet game consistently, Pacman had to have 180% the speed of the ghosts.

C. Changing Number of Pellets

Having a few pellets (1-2) and Pacman having the same speed as the ghosts usually let Pacman win. Any more thean 4 pellets and Pacman fails to escape 4 ghosts consistently.

To see the effect of the pellets, only 2 ghosts were used. Pacman could efficiently collect 5 pellets with only 2 ghosts at the same speed, but any more and it did get cornered sometimes since the mapping led to Pacman choosing to go back to areas it had already explored (since the costs there were lower as walls were better defined), which is usually where the ghosts were trailing.

D. Full Information About Ghosts

We compared to see if Pacman having full information about Map while calculating ghost probabilities would lead to Pacman being better at escaping the ghosts.

Since we set a ping to be emitted every 3 update cycles, this didn't allow for a big change in the probability distribution as it was localized within a relatively small area. This led to no major changes when running the algorithm.

E. Changing Costs

If we did not scale the costs appropriately, the Pacman would sometimes choose a path through the wall instead of around the wall. This would be because the shorter distance to a pellet would have a larger impact on the costs compared to increased costs of going through a wall. Thus, we chose the exponential scaling of the costs of walls so the Pacman never chooses a path that goes through the walls.

VI. Further Work

One potential way to change our work is running D*. This would include finding some potential algorithm that is able to take into account when costs lower. One potential way is to start the root of a tree from the node that lowered its cost and find the shortest path from the node to the goal and the node to the start, then combine the costs of both branches to see if it creates a more optimal path. Additionally, we could adjust our occupancy map to instead start with the assumption that all nodes start with a probability of 0, and the probability of nodes only increase when the sensor detects that it is a wall. If we implement this with a predictive ghost algorithm that uses the ghost velocity to determine where the ghost is going rather than using the ghost pinging, we could potentially not have to change D* to account for lowering costs. Instead we would adjust the algorithm to let the ghost pass the Pacman, or create a path based on where it predicts a certain intersection point.

Currently, ghosts are localized using a distribution of their probability, by assuming that they can move in any given valid direction by a probability proportional to the chance that the space is empty at each step. However, we can also leverage the information about the Ghost's AI, to learn that Ghosts can only turn at intersections, to make a much more accurate representation of the probability distribution, and in certain tight spaces - locate a ghost with perfect accuracy! This can also be extended to reverse engineer the Ghost's pathing AI, and since it is deterministic, we can predict the exact location of a Ghost, without it ever emitting a ping or anything of that sort! That is actually how the person who solved Pacman (completed all the levels until the game ran out of RAM), solved the game itself! By abusing the Ghost's AI!

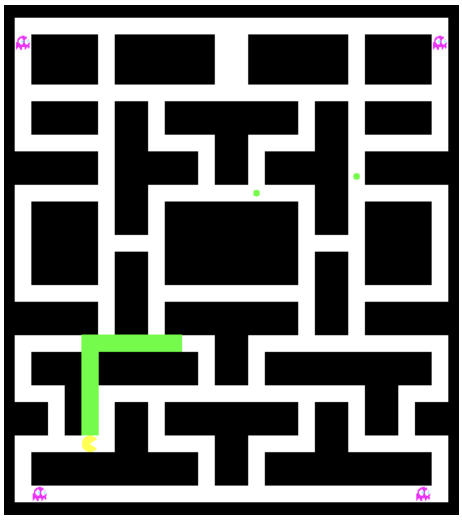

Fig. 8. First Path Pacman Chooses Without Knowledge of Map. It Chooses Closest Pellet and Highlights Path.

PICTURES: Path being highlighted Picture to explain changing targets (based on costs, but in practice works w ghosts too)

APPENDIX

Github: Pacman!
Video link: Pacman Wins By Itself

ACKNOWLEDGMENT

Our group is very grateful to Professor Günter Niemeyer and the TAs for putting together a very interesting class, and lots of help and motivation throughout the course of the project!

IMAGES

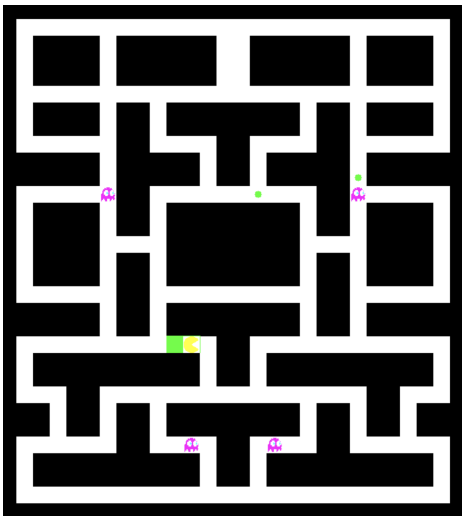

Fig. 9. Pacman Reaches Its First Pellet Successfully

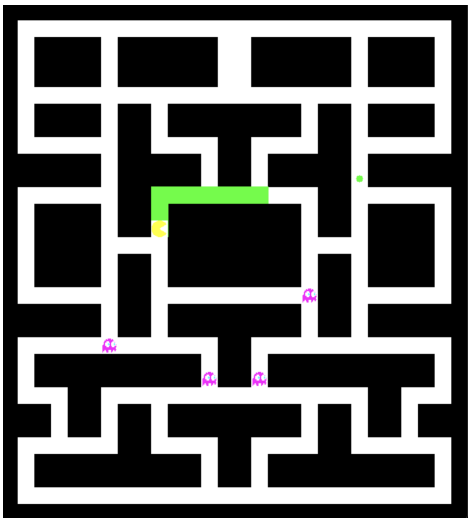

Fig. 10. Pacman Collected First Pellet and Now On Its Path To Second Pellet

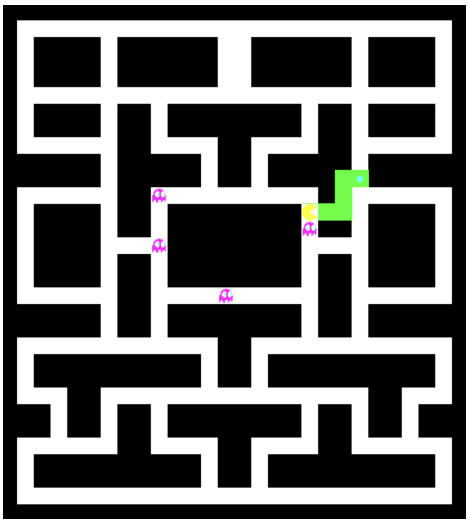

Fig. 11. Pacman's Path Going To Pellet Before Detecting Ghost

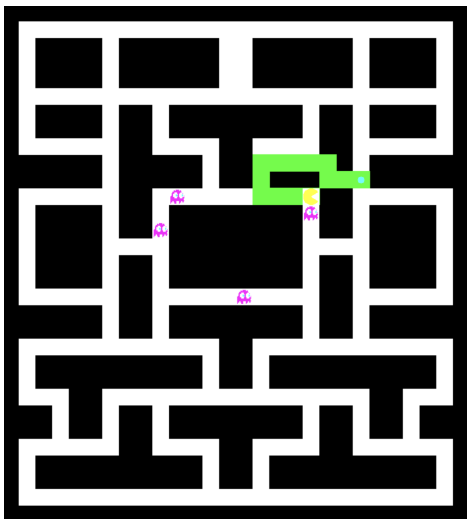

Fig. 12. Pacman's Path Going To Pellet After Detecting Ghost