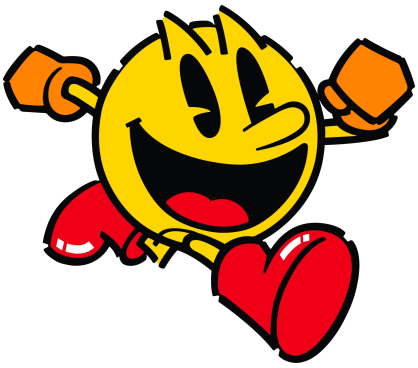
Max Horowitz, mgh68
Sophie Lopez, shl98
Jacob Ball, jab869



AI-Powered Pac-Man

Introduction:

The game of Pac-Man is known by many to be a classic, being played as an arcade game since 1980, and has since made its way over to being played on computers. The objective of Pac-Man is for the user-controlled agent, aptly named Pac-Man, to navigate an enclosed maze and eat all of the pellets while avoiding four ghosts who are also roaming the maze. Collecting pellets increases score, and once all of the pellets are collected the user graduates to the next level. Each level has a fruit that appears and disappears at random and can be collected while visible for points. Score can be increased by eating “power pellets” which, when consumed by Pac-Man, briefly change the states of the ghosts, slowing them down and allowing them to be eaten by Pac-Man for extra points. A user has three lives to collect as many pellets and complete as many levels as possible, and lives can only be lost when the Pac-Man is caught by a ghost. The game is typically played with arrow keys on a keyboard, indicating the next direction the user wants the Pac-Man agent to turn in the maze. We decided to turn the game of Pacman into an AI decision making problem by modifying the action input from user input– such as the press of a key– to an AI algorithm.

We set out to accomplish creating an AI decision making algorithm that controls Pac-Man’s actions in order to navigate the maze and maximize the score while avoiding the ghosts. There are many components that must be taken into consideration when constructing and implementing a heuristic algorithm like this, specifically the locations of the ghosts in respect to the main agent (to which we will be referring Pac-Man as), the locations of the remaining (unconsumed) pellets and power pellets, the state and location of the fruit, and the possible directions the agent can move from its current state. As it follows, we aimed to find a heuristic that determines the best move Pac-Man can make from the current game state in order to avoid ghosts and raise the score as much as possible. The key problem we had to solve was figuring out how to optimize each of the aforementioned components without sacrificing too much of another, as well as determining which could be sacrificed at the expense of another. Deciding which aspect to place more importance on and quantifying this importance was difficult, but we ultimately decided to focus our heuristic on the proximity of the closest ghost to the Pac-Man, as well as the proximity of the nearest uncollected pellets and power pellets. So,

we define the heuristic as a function of the weighted sum of these two factors: the proximity of the closest ghost to the main agent and the proximity of the pellets to the main agent. Primarily, if Pac-Man consumes a fruit, the ghosts go into a state for a few seconds which we will call *freight* mode where they slow down, and change direction and color to indicate that they can now be eaten by Pac-Man for extra points. Thus, when the ghosts are in freight mode, the Pac-Man should move to a game state closer to the nearest ghost. To implement this, we set the heuristic to be positive and rise by a greater factor with increasing proximity of Pac-Man to the nearest ghost when the ghosts are in *freight* mode. If the ghosts are not in *freight* mode, the heuristic value must reflect that the Pac-Man wants to move as far from the ghosts as possible. So we set the heuristic score to be negative, and decrease by an increasing factor as proximity of the nearest ghost to the main agent increases. So the score will be highest (i.e. closest to zero for a negative number) when the closest ghost is furthest from the main agent and lowest when the closest ghost is closest to the main agent, indicating that the most optimal state for the Pac-Man to move is that to maximize distance between itself and the nearest ghost. Next, pellet proximity is a weighted sum, with weight determined by locality to the main agent, allocating more importance if the pellet is closer and/or if it is deemed a power pellet. The final heuristic is a weighted sum of the calculated pellet proximity and the heuristic values determined by the closest ghost proximity/the current state of *freight* mode. We implemented the heuristic so that if the closest ghost is more than a fixed distance from the main agent, the heuristic solely depends upon the locality of the pellets. Otherwise, the proximity to the closest ghost is deemed more important and has a higher weight in determining the heuristic.

The biggest hurdle in efficiently implementing this heuristic was the dynamic nature of the game state and its dependence upon not only the current state of the Pac-Man agent and the pellets in the maze, but the state and possible actions of four other unpredictable agents (the ghosts). Given five agents and the up to six possible next actions they can each take, it was hard to determine how to keep track of each possible next gamestate. We needed to converge on a target gamestate in a minimax tree and isolate the “best direction movement” all within one frame of gameplay. More specifically, we are bounded by up to 0.03 frames per second and need to make decisions within that bound. Because minimax gamestate trees can be fairly deep (we experimented with depths ranging from 3 to 8), and each gamestate (as we implemented it) can have up to 6^5 combinations of successor gamestates (which is the number of permutations that the pacman and each ghost can move combined), we needed our heuristic evaluation to be quick. We decided the best way to optimally implement our algorithm was using a multi-agent minimax algorithm with alpha-beta pruning. With multiple minimizer agents as the ghosts and

Pac-Man as the maximizer, this algorithm allows us to predict the game states of the ghosts and act in the most optimal way according to the heuristic. Given the heuristic described above, the minimax algorithm determines the importance value for each character (via the heuristic function) and each next possible game state: The Pac-Man game state is the maximizer, so if the state is in *freight* mode, it will aim to move towards the highest heuristic, i.e. towards the closest ghost in freight mode. The minimizers (the ghosts) will aim to move farther away from the main agent. If not in *freight* mode, Pac-Man picks the negative heuristic closest to zero, indicating it wants to move to the game state furthest from the nearest ghost. The ghost wants to minimize the heuristic so it aims to move closer to the main agent. Note that the importance of the proximity of pellets does not change whether or not the game state is in *freight* mode. A major challenge was converging on a target gamestate in the minimax tree gamestate search to isolate the best direction movement all within one frame of gameplay. In our game, we used `clock.tick(30)/1000,` which means that in each second, at most (30/1000) frames should pass. This limits the depth we can traverse in our gamestate search and thus we needed our heuristic to have low time & space complexity.

Our heuristic is therefore limited by low time and space complexity, which is why we decided to constrict it to the two aspects of the game we deemed most important– proximity to the nearest ghost and proximity to the pellets. Without this strict restraint of time and space complexity, however, we may have been able to include other aspects of the game into our heuristic to further improve our AI's gameplay. For example, we were not able to include the distance to the fruits (which give bonus points) or proximity to the other ghosts instead of just the closest in our heuristic. Although our heuristic was bounded by these complexity constraints, we were able to improve efficiency by implementing Alpha-Beta pruning which allows us to search up to two times more game states than the standard minimax.

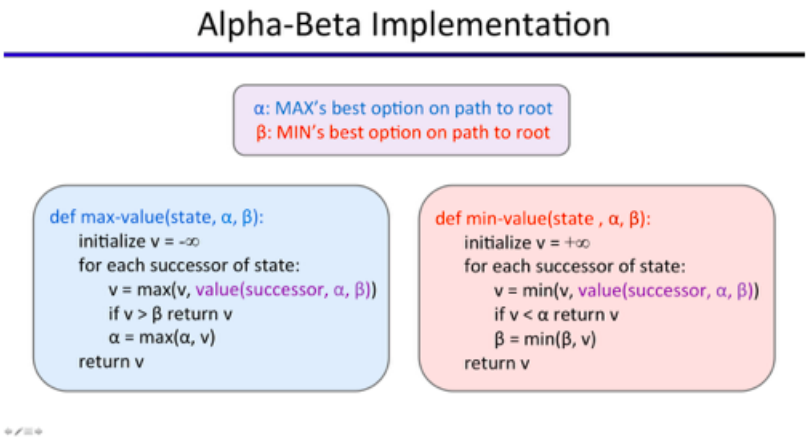
To familiarize oneself with the gameplay and interface, the user can run *make user* to play the game with user input. To watch the AI engine play on its own, the user can run *make ai*.

Prior Work:

The foundation of this project is the open source project code provided by https://pacmancode.com/. The original *run.py* file provides a fully-operational instance of the game Pac-Man, including the *Gamecontroller* class which appropriately initializes the features of the game and updates according to user input. This includes methods to render the environment and all of the agents (the Pac-Man and four ghosts), the pellets, the game background and layout according to game level, as well as methods to start, pause, restart, and

stop the game, update score after collecting pellets and fruits, lives after being caught by a ghost, and more. This source code implements gameplay of the user versus a CPU (the ghosts), in which the user controls the Pac-Man's movement with keyboard input. We found that it was easy to isolate this user-input functionality and replace it with AI-controlled movement, which became the basis of our project.

We started research online about how to incorporate multiple moving pieces into one AI engine that was able to make calculations while pieces were moving. We settled on a multi-agent minimax (with alpha-beta pruning to optimize it). For the implementation of the multi-agent minimax model with Alpha-Beta pruning, we took inspiration from the project descriptions on the UC Berkeley AI course site (http://ai.berkeley.edu/multiagent.html). The site includes an open source code base, along with descriptions of how to implement a multi-agent minimax model for this specific game. More specifically, it explains that the main agent (Pac-Man) should be the maximizer, the ghosts should be the minimizers, and that a single ply in the game state tree represents one Pac-Man move and all of the ghosts' responses. So taking this construction of the game state tree and the interpretation of its depth, we constructed a similar minimax algorithm with multiple minimizer agents and an arbitrary depth parameter. The UC Berkeley AI course site also provides a description of how to augment the multi-agent minimax algorithm to include Alpha-Beta pruning. They provide pseudo-code which we found especially helpful in developing our algorithm and the intuition behind Alpha-Beta pruning with multiple minimizer agents:



The UC Berkeley site went further to implement expectiminimax to take into account suboptimal gameplay, but we wanted to make the assumption that gameplay was optimal (which led us in the direction of minimax).

Methods:

We began by modifying the *run.py* file from the open source code of Pacmancode.com to make user input obsolete. We originally started out with a random algorithm to ensure that the gameflow setup worked correctly. This provided the validation that we properly isolated and removed the user gameplay from the open source code.

Our first attempt at trying to maximize the number of pellets picked up was to implement a “no turning back” AI. The idea is that the algorithm keeps track of the Pac-Man agent’s most recent direction, including changes in direction when turning corners, and won’t turn back around to the opposite direction**.** We discovered a few faults in this implementation which led to a consistently low score, most notably its naive nature which had no regard for the states of the pellets or ghosts, and led to some ignorant errors. For example, when the agent got boxed in and the only possible move was to go the opposite direction it came, it got stuck.

We then moved to a greedy game engine where the idea was to pick up the most pellets before getting caught by a ghost by checking immediate next game states (whichever immediate directions can be moved to) and move one iteration and see how well the score improves or goes down (greedy approach). Although runs of this algorithm lead to unanimously higher scores than the previous “no turning back” implementation due to its greedy nature, it was still not as informed as it could have been. Instead, since it had no account of the proximity to the ghosts, it was simply a race against the clock, trying to raise the score until inevitably getting caught by one of the four ghosts.

To improve the game engine’s performance we had to really strategize. We knew that there are multiple moving parts to the game so the game state is incredibly volatile. The four ghosts move simultaneously to pacman and fruits are temporary, and you want to traverse untravelled areas so you pick up the most pellets in the least amount of time. Due to the complexity constraints, we decided to design a heuristic to optimize the ensuing game state by only taking into consideration the location of the ghost nearest to the main agent and the proximities of the remaining pellets. The key component of our heuristic was figuring out how to optimize each of these without sacrificing too much of another. Deciding which aspect to place more importance on and quantifying this importance was the biggest test. So for our heuristic, we defined the optimality of the next game state as a balance between proximity to the closest ghost as well as proximity to pellets. The balance that we use varies with different gamestate situations because when the closest ghost is very far away from the pacman, it should totally

disregard the ghosts locations and choose its next move only based on pellets. The parameterized *heuristic()* function returns a value *evalGamestate* which is aptly named, since it evaluates the optimality of the input game state. We quantified this using the following rules and intuition:

All distances referenced in our code are in terms of Manhattan distance, which is the sum of the absolute differences between the location vectors of two agents (the main agent and the ghosts or the main agent and the remaining pellets).

ClosestGhost(): For the given input game state, the distances of each ghost from the main agent are kept in an array, with the id of the closest ghost and the distance to the closest ghost being returned.

ProximitytoPellets(): For each pellet that has not yet been consumed (and is therefore remaining on the board), the pellet *proximity* variable is the weighted summation of a function of the distance of each remaining pellet from the current state of pacman. A pellet's *proximity* is weighted more if it's closer to the agent and if it is a power pellet. Proximity is a function of the distance of each remaining pellet from the agent: It is a summation of functions of the respective distances, with the closest pellet (a pellet one unit away from the agent) adding 100 to the proximity, and decreasing as the pellets are further from the agent. Power pellets are handled in the same manner, except their proximity is quantified as 10 times that of the normal pellets. Essentially, the more pellets in the maze and the closer they are to the agent, the more importance is placed on the pellets in the heuristic.

closestGhostImportance: the "importance" of the proximity to the ghosts **is a function of** (or depends on) the distance of the main agent to the closest ghost (determined by the method *ClosestGhost()* above). *closestGhostImportance* is initialized to be 10,000 if in freight mode (meaning Pac-Man ate a fruit and can eat the ghost), and -10,000 otherwise. We quantified *closestGhostImportance* as such: if the closest ghost is less than 50 units (Manhattan distance) away from the current location of the agent (determined by the game state input to the *heuristic* function), *closestGhostImportance* is multiplied by 100 (to be +/-100,000). If it is between 50 (inclusive) and 100 units away, the importance is multiplied by 50. Between 100 and 200 is multiplied by 25, 200 and 300 is multiplied by 10, 300 and 400 is multiplied by 5.

Finally, the *heuristic()* function determines *evalGamestate* like so: If the closest ghost to the main agent is less than 400 units away then *evalGamestate= closestGhostImportance** $\frac{1}{\sqrt{closest\ ghost\ distance} + 0.0001}$ + *proximityToPelletsImportance** *proximity*. Because there are situations where the closest ghost distance is zero, we would encounter a divide by zero error if we didn't include the additional +0.0001 term. Note that the +0.0001 does not impact the value

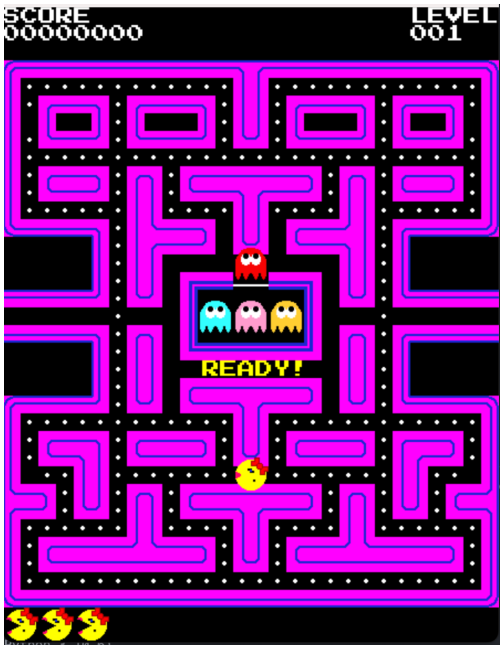
of the ghosts when evaluating gamestates because closestGhostImportance either negates, adds importance, or makes negligible the concept of ghosts in the maze. Otherwise, if the Manhattan distance of the closest ghost to the main agent is farther than 400 units away, the *proximityToPelletsImportance* value (initialized at 50) is multiplied by 10 to become 500 and *closestGhostImportance* is set to 0. So *evalGamestate* is simply equal to the product of *proximityToPelletsImportance* and the *proximity* value determined by *ProximitytoPellets()*. This is to work so that if a ghost is far enough away from the main agent it will essentially become obsolete and the optimality of the next game state will only be evaluated based on the main agent's proximity to the pellets. Otherwise, the heuristic value of the game state depends on the distance of the main agent to the nearest ghost while also taking into account proximity of the pellets. Using this heuristic, we implemented a minimax algorithm with multiple minimizer agents. The maximizer is the evaluation of the next possible state for the main agent (pacman), and there are multiple minimum layers (one for each ghost agent) per max layer. Our code implements this multi agent minimax algorithm for an arbitrary depth. A single layer of the search tree has a relatively large breadth due to the number of minimizers, so even with alpha-beta pruning, multiple iterations can become quite inefficient.

Pacman aims to maximize the heuristic (so get the least negative score or the highest positive score). Ghosts want to minimize (If the ghost is in freight mode, the score will be very high and pacman wants to maximize and ghost wants to minimize. If not, the score will be very low, lowest when the ghost is closest and highest when the ghost is furthest away. Pacman wants to move further away from the closest ghost. Meanwhile, the ghost wants to minimize the score so wants to get closer.

Results:

In order to test the effectiveness of our various heuristics we created a third gameflow in addition to the user and ai game modes. This gameflow called "benchmark" can be run by entering the command "python run.py benchmark" in the powershell as stated in the makefile. When this program is run it will open the pacman pygame environment and stimulate the ai implementation, just as it is run using run.py ai, however the benchmark gameflow keeps track of the scores recorded from the various runs and outputs a benchmark representing the average of the scores. We used this to determine how effective our various heuristics were, and tweaked our algorithm accordingly. In order to compare how effective our algorithm was to an average user we asked a bunch of our friends to play using the arrow keys on the keyboard and recorded their results. I will include screenshots of our GUI as well as some of our data collected

below including the average scores of two of our heuristics as well as the results of our friends playing with user controls.



User Controlled Scores:

Name	**Score:**
Chris	830
Ethan	1370
Phil	1740
Jacob	3790
Dylan	660
Harrison	1260
Max	3650
Talia	4240
Sophie	3020
Lauren	7040
Average Score:	2760

Our Heuristics:

Version:	**Average Score:**
Final Heuristic	1893.3
Old Heuristic	1130
Random Algo	950

Although our algorithm did not perform as well as we wanted to, it did not surpass the average score of our friends, we vastly underestimated how hard it would be to create an effective heuristic with such a small time complexity. In the following section we will discuss some of the factors that may have affected our AI's performance.

Discussion:

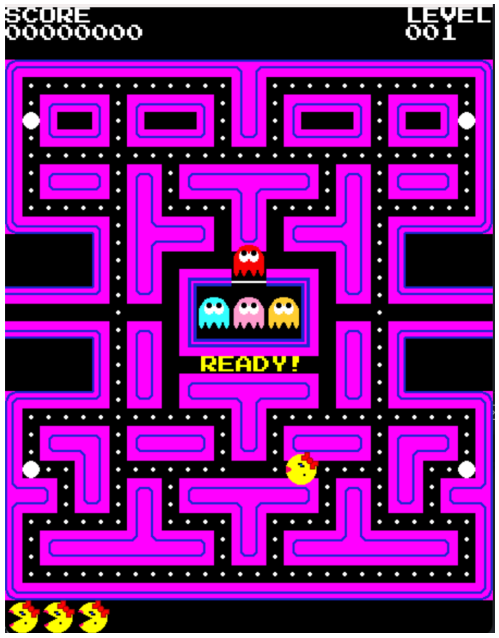
There are several factors that could explain why our algorithm did not outperform the average of the users. Although our heuristic took into account two of the most important factors of the game: the proximity to ghosts and eating pellets, we struggled to incorporate some other factors that a random user may be aware of. Some of these factors include but are not limited to identifying the presence and location of the fruit as well as the location of super-pellets. Eating super pellets is an extremely important aspect of the game as it turns all of the ghosts into freight mode allowing the user to eat the ghosts and gain a point multiplier as a result. It was difficult for us to incorporate these factors into our heuristic as it was important that it maintained a very small time complexity in order for it to choose a move in a matter of milliseconds. Where traditional minimax implementations we have come across such as chess, allow the heuristic to run for a few seconds before choosing each move, our heuristic has to make live decisions forcing us to leave some key features out of our final implementation. However we will discuss some possible solutions to that problem in our conclusion and further delve into how we would overcome that obstacle.

One of the problems we did address between the old and final versions our our heuristic which we included in the results section was addressing a bug that caused the pacman to sometimes get stuck on the left side of the game and just go back and forth between 3 spaces until a ghost came really close. This was due to a flaw with assigning weights to the manhattan distance between the pacman and ghost and caused the pacman to think that ghosts were

close although they haven't even left their starting position. However we addressed thus problem in our current heuristic and the score improved accordingly.

Conclusion:

As touched on previously, because of the number of agents and actions simultaneously updating the game state, we faced a challenge when it came to time and space complexity. Specifically, we found it difficult to determine how to efficiently simulate future game states without influencing the movement of the Pac-Man, causing it to move to invalid positions.



One idea we had to optimize our program was to distinguish game states into groups of similar features. We can then apply different heuristic functions depending on the group the state is in. For example, we may create a group of states where the nearest ghost is within 10 Manhattan distance units from the Pac-Man. The heuristic function applied to this group would likely ignore pellet positions altogether and place the utmost priority on moving to a game state farthest from the ghost. With this method, we can begin to take into account the distances and possible movements of not only the ghost of closest proximity to the Pac-Man, but the other three ghosts as well. We can improve further by keeping track of the temporary fruits as well. This would increase our space complexity of the gamestate evaluation, but if we optimized our gamestate search further, maybe the above additions could be beneficial.

Similarly, among the plethora of game states that are reached within a certain game, many of them are likely to be repeated (to an extent) by a different combination of moves. We

can generalize these similar game states into groups and use a transposition table to cache the heuristic output of a certain state that has already been visited. Thus we can avoid recomputing states that have already been reached. The idea is that the transposition table will save states that either take the most computation time or are repeated the most. Because it is highly unlikely that a gamestate will be repeated exactly (due to floating point errors and preciseness of pacman's & ghosts' positions), we can implement the transposition table to group similar gamestates together and remember an approximation for evaluating the average of those gamestates. This could be a nice optimization to transposition tables to limit the space required to store these tables.

Works Cited

"Artificial Intelligence: Mini-Max Algorithm - Javatpoint." *Www.javatpoint.com*, https://www.javatpoint.com/mini-max-algorithm-in-ai

Berkeley Ai Materials, http://ai.berkeley.edu/multiagent.html

Pacmancode. "Pacmancode." *PACMANCODE*, https://pacmancode.com/

"Part 7 – Transposition Table." *Solving Connect 4: How to Build a Perfect AI*, 24 Apr. 2017, http://blog.gamesolver.org/solving-connect-four/07-transposition-table/

Pettersson, Jonatan. "[Guide] Transposition Tables." *Mediocre Chess*, 1 Jan. 1970, http://mediocrechess.blogspot.com/2007/01/guide-transposition-tables.html