Nikolai Kallhovde

Team: GotoFail (One-person team)

A Post-Mortem of the GotoFail Pac-man Agent

When we were first given the details for this assignment, the idea that immediately came to mind was to implement an agent that chose its moves using the Monte Carlo Tree Search algorithm. I had heard of MCTS from reading about the General Video Game A.I. competition, where it was the most successful algorithm by far for playing a broad range of games. Two advantages of MCTS in particular stood out to me as being useful for a Pac-man AI: it needed no prior knowledge of the game, only the value of the end states, and the results of each playthrough were propagated back to the root node, meaning that the algorithm could be terminated at any time to yield the best option thus far. I started out by writing a basic particle filter to approximate enemy locations. This implementation was the same as the one in Project 4, except for a few changes: the particles would reset if none of them were anywhere close to the fuzzy distance, and the particles would converge on the enemy start location if the game detected that the enemy had been eaten. I then used the most probable locations, as determined by the concentration of particles there, to inform the simulations used by my planning agents. A quick explanation of how MCTS works: Monte Carlo Tree Search is an algorithm that converges to minimax, but chooses more promising nodes first. It starts by traversing the tree of explored nodes, starting from the root and choosing child nodes using an algorithm called UCT that balances the dual goals of fleshing out the most promising nodes and exploring new nodes. Once it reaches a node with unexplored children, it plays out a game from one of those children and propagates the end result back up the tree, then repeating the process if possible.

However, once I implemented this algorithm, I immediately encountered a problem: it ran much too slowly to provide meaningful results. Even when I allowed the agents a full three seconds, it would only be able to run about six playouts per possible move, and those playouts almost always resulted in a tie game. I then learned that many MCTS algorithms use a “heavy playout”, which meant that instead of moves in the playout phase being chosen randomly, they were influenced by some heuristic. But to my dismay, when I made a basic 'proximity to food' heuristic, the algorithm ran even slower due to the computational complexity of evaluating the heuristic at each step, and my results were no better. At that point, I had a few options, including trying to abstract the nodes rather than having one node for each game state, but these solutions proved very difficult to implement, so I decided to explore a different approach.

Being an adversarial game, I wanted to use an adversarial search algorithm like minimax, but I also knew that it would be hard to choose a good depth to use, since the branching factor varied a lot

depending on each agent's position. To address this issue, I came up with an iterative deepening variant of minimax that ran the algorithm multiple times, each time increasing the depth of the search, and broke out once the turn time limit was reached. The agent's move would then be informed by the last completed minimax search. Although this meant that a lot of computational time was wasted doing successive minimax searches at various depths, it also meant that the agent would perform a search to the greatest depth possible in the time allotted without running over. While this idea was sound in theory, in practice it was of limited utility due to the difficulty of writing accurate heuristics for it.

For my final A.I., I took a simpler approach than any of the earlier implementations, but the lessons I learned from each helped me in writing it. At its core, the agent uses a two-ply minimax search with alpha-beta pruning (adapted from my iterative deepening version) and a heuristic which varies depending on the agent's current “strategy”. Keeping the look-ahead depth to two means that the horizon effect is sometimes an issue, but the guaranteed short running time and relative ease of designing heuristics for a fixed-depth node makes up for it. Taking inspiration from ghost behavior in actual Pac-man, the agents can each be in one of four possible strategy modes: Attack, Chase, Scatter, or Reroute (a Defend mode was also planned, but I eventually decided it was not necessary). The Attack mode is the default strategy, and the agents will continue to follow it until an enemy agent is seen. This mode uses the same weights as the baseline offensive agent (100 points for score, -1 for distance to food), but is made much more powerful by choosing which food to target according to a food allocation algorithm. This algorithm runs before the game starts and splits all the food on the map into a top and a bottom group. It does this by putting the food locations from the top and bottom four rows into two lists, distributing the unallocated food in the middle evenly based on how close each one is to the existing lists, and finally moving 'stray' food which borders the other list's food to that list. As long as the lists are somewhat balanced, each agent will only target food from its assigned list. The result is two agents that go for food on opposite sides of the map, something which is both efficient and very hard to defend against.

The other strategies all take effect once an enemy is spotted nearby (the minimum distance is five by default, but it is reduced to one when the game is almost over). The Scatter strategy is activated when the agent is either in Pac-man or scared mode. This strategy gives 30 points for each square the agent is away from its attacker, but -10 points for each square from the nearest power pellet, so the agent will gravitate towards power pellets when being chased. In order to mitigate the effect of the agent getting eaten and gaining points from now being further away from the enemy, the agent detects when it is at its start position and breaks out of the minimax evaluation early with a score of negative infinity. Unfortunately, while relatively simple, the implementation of this strategy is far from perfect – the agent

often goes down dead-ends and gets trapped due to its limited search depth. However, the agent does gravitate towards power pellets most of the time while in Scatter mode, and for the most part it works fairly well. Another strategy, Reroute, is activated when both the agent and their nearby enemy are ghosts. In this situation, the agent considers alternate points from which to cross enemy lines, and chooses one based on the combined distance from the agent to the crossing point and from that crossing point to the nearest relevant food. This strategy serves two purposes: to prevent 'deadlocks', where ghosts are congregated at one bottleneck and it would be suboptimal for either team to cross enemy lines, and also to potentially lure the enemy ghosts across, allowing the agent to then switch strategies and chase them down. Chase is the most straightforward of the strategies to comprehend, but was by far the most difficult to implement. My initial implementation simply gave points to the agent for its proximity to the enemy ghost. This resulted in an agent that would chase the enemy agent, but when it came time to actually eat it, it would just run away. This behavior was the result of a few different factors. One was that once the enemy agent was eaten, its would return to its start position, leading to a much greater distance and lower score. However, even when this was checked for, other factors prevented the correct behavior, namely that the minimax algorithm assumes optimal enemy play, so it is rare for the enemy agent to be considered truly caught, and that there is no penalty for eating the agent on the second turn rather than the first, so it encourages 'procrastinating'. My great insight came when I decided to make the agent chase a target, rather than caring about the enemy position directly. The target was placed directly in front of the enemy agent depending on its direction (just like Pinky's A.I. in actual Pac-man), or, when the enemy was very close, on the opposite side of the enemy from the friendly agent. This resulted in an agent that chased the target at all costs, coincidentally eating the enemy on the way towards reaching that target.

Though my agent did very well overall, there is a lot of room for improvement. Other than addressing specific flaws in my strategies that I mentioned earlier, the most obvious improvement would be to feature more cooperation between the friendly agents, so that when one agent is being chased, the other goes for a power pill, and when one agent is chasing an enemy, the other tries to go around and trap the enemy in a corridor. Another improvement would be to make the particle filters joint rather than individual, have them detect when a food is eaten and adjust accordingly, and use the positions they give more effectively in planning routes. Finally, one major oversight is that the agents have no actual defense strategies other than purely opportunistic ones (actually seeing the enemy), and on defensive maps with bottlenecks and many pellets close to the ghost start location, the agent performs very poorly. If these issues were addressed, the agent would be more balanced and perform much better in a variety of maps and situations, making it an extremely successful agent in tournament play.