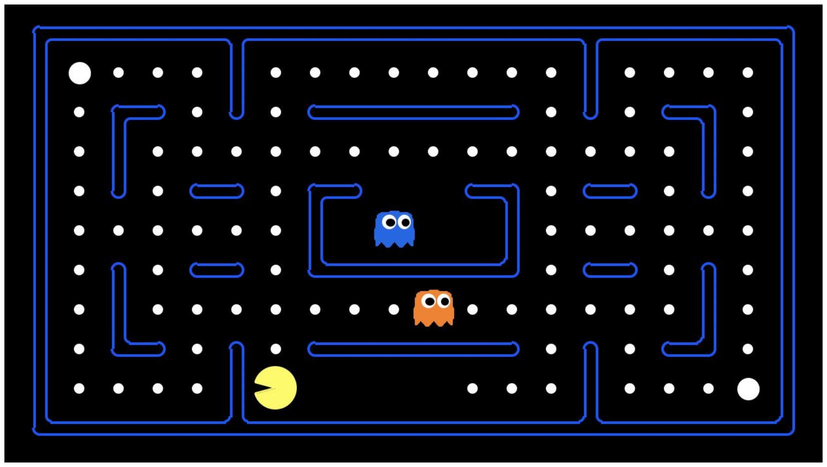
Monte Carlo Tree Search vs. Minimax for Real Time Pac-Man

Durga Harish
Matthew Meyn
Jon Dodge

March 20, 2017



Abstract

In this report, we compared results of different search strategies for controlling the Pac-man Agent in real time. We considered Minimax search, alpha-beta search and Monte Carlo Tree Search (MCTS) in partially observable and fully observable environments. We established different evaluation functions for various kind of Pac-Man states. Pac-Man is an arcade game, in which the protagonist has several goals, but no conclusive terminal state. Instead, the game has two subgoals, 1) surviving and 2) scoring as many points as possible. This report enhances the ideas of existing MCTS algorithm by 1) considering ghost actions to be strategic 2) making intelligent decision for backing up score after rollout process. In order to evaluate the search performance, we ran simulations of the game for 50 times in two different environments, (mediumClassic, shown on the cover sheet, and smallClassic, shown in Figure 0), and recorded the score and win/loss status. We found that unmodified MCTS performs poorly in this real time environment, as compared to Minimax. However, when elements of Minimax were added to MCTS, it performs relatively well in both environments. When Pacman was able to observe the environment fully, performance was similar or better than Minimax. In the partially observable environment, however, modified MCTS clearly outperformed Minimax alone.

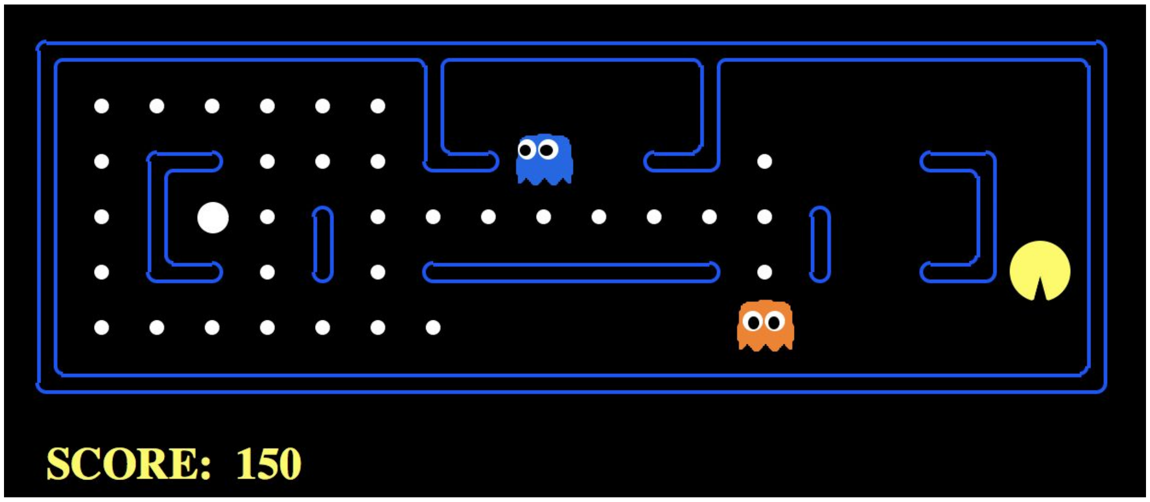


Figure 0: The Small Classic game and medium classic board. Pacman can be seen as the yellow circle object, while the ghosts are orange and blue and appear as blobs with eyes.

1. Introduction

The arcade game Pac-man is a popular stochastic, real-time environment for tree-search game playing agents. The game is basically a combination of path-finding and pursuit-evasion problems. The hero, Pac-man, must clear all food from a maze while avoiding ghosts, which end the game if they collide with Pac-man. In our version of the game, ghosts generally try to move toward Pac-man when they are able to eat him, with a small probability of other randomized moves. To the problems presented by the classic game, we add a version in which Pac-man cannot see the entire maze (nor the ghosts). This adds a further dimension of stochasticity to the game.

To survive and win, Pac-man must be able to consider more than one move in advance. Thus, algorithms like Minimax and Monte Carlo Tree Search (MCTS) seem logical choices to plan Pac-man's moves. The challenge, however, is that in a real time game like this, there is very little time for computation between moves. On an average machine, it is not feasible to explore more than perhaps 2-4 plies of a Minimax tree, even when using alpha-beta pruning. Thus, any Minimax agent for Pac-man ends up relying heavily on static evaluation functions to guide it. Minimax tends to face other difficulties in this domain, which will be discussed further on.

MCTS offers a more computable alternative to pure Minimax that might seem somewhat more suited to the stochastic, multiagent aspect of the game. In MCTS, the tree of possible paths through the game is built only partially, using randomly simulated playouts from nodes chosen by a selection policy. The four steps in MCTS (Selection, Expansion, Playout and Backpropagation) are performed for a fixed number of iterations or until a time limit is exceeded. This ability to easily terminate computation when time is up can be useful for real time games. In order to use MCTS in real time, we adopted some ideas presented by Pepels, Winands, and Lanctot [1] for enhancing MCTS search in Pac-man.

Nonetheless, computational time limits present a major challenge in applying MCTS to arcade games like Pac-man. Our initial approach to MCTS was composed of two modifications to the standard algorithm:

1. Modify the random playouts to account for the likelihood of rational behavior on the part of Pac-man's adversaries, thus effectively pruning branches from the game tree that are not likely to occur.
2. Depending on the proximity of ghosts, choose from two policies for move selection: a policy that prioritizes ghost avoidance, and one that prioritizes finding food.

Given the work that has been done on Pac-man with Monte Carlo methods, we hypothesized that MCTS would outperform Minimax in general. But in particular, we expected MCTS to be better suited to handle the stochasticity of the partially observable case. We wanted to test this hypothesis in both a small maze and a medium-sized maze, typical of the classic game. We suspected that Minimax would perform better in the smaller environment than in the larger one, because of the severe depth limitations of the algorithm in a real time game.

For the test-bed for our agents, we used the Pac-man for AI framework created by John DeNero, Dan Klein, and others at UC Berkeley [3]. This framework includes the mechanics of the game, graphics and ghost agents. To this we added agents to generate moves for Pac-man: a Minimax agent (with and without Alpha-Beta pruning), and a MCTS agent.

Upon experimentation with the MCTS agent, we found that our initial modifications to the algorithm were not sufficient to improve performance over Minimax. In fact, Minimax was outperforming MCTS, for reasons that will be explored later in Section 6.1.1. It appears that each of the two search algorithms has its own strengths and weaknesses for this domain. So, for the final version of our Pac-man agent, we added elements of Minimax search to the MCTS framework.

2. Related Work

Among variants of the game, Ms. Pac-man seems to have attracted the most attention from AI researchers, because of its addition of randomized ghost moves to the original game. In 2011, Ikehata and Ito [2] found that the Upper Confidence Method applied to Trees (UCT) was helpful in preventing Pac-man from getting trapped between two ghosts approaching from opposite sides. Pepels, Winands, and Lanctot [1] added several improvements to the MCTS Pac-man agent. The first is a variable-depth search tree to reflect accurately the actual length of Pac-man's path, given that there are more decision nodes at some physical points than others. Second, they introduce three distinct tactics corresponding to different policies, which the agent switches among based on threat level or edibility of ghosts. Third, the already-built search tree is partially reused, subject to rules that throw out the existing tree if a game-changing event has occurred, or continuous decay of stored values otherwise. Fourth, Pac-man's choices in the playouts are limited by certain rules, and the ghosts' moves are simulated according to a greedy strategy designed to trap Pac-man. Finally, Pac-man is given long-term goals to augment the agent's behavior as dictated by MCTS.

3. Minimax Search

As a basis of comparison to MCTS, we use Minimax, with and without Alpha-Beta pruning. We implemented standard Minimax, with Pac-man being the Maximizer and the ghosts together being treated as a single agent, the Minimizer. Note that Alpha-Beta pruning involves a slight modification of MinValue() and MaxValue():

```
function MinimaxDecision(GameState, Depth) returns Action
      legalActions <- GetLegalActions(GameState, PACMAN)
      Choices <- empty list
      for a in legalActions
            State <- Successor(GameState, PACMAN, a), a
            Choices.append(State, MinValue(State, Depth))


function MaxValue(GameState, Cutoff) returns value
      if Cutoff == 0 or GameState.IsWin() or GameStat.IsLose()
            return StaticEvaluation(GameState)
      V <- -inf
      for a in GetLegalActions(GameState, PACMAN)
            v <- max(v, MinValue(Successor(GameState, PACMAN, a), Cutoff-1))
      return v
```

```
function MinValue(GameState, Cutoff) returns value
      if Cutoff == 0 or GameState.IsWin() or GameStat.IsLose()
            return StaticEvaluation(GameState)
      V <- inf
      states <- GenerateResultingStates([GameState], 1, NUM_AGENTS)
      for s in states
            V = min(v, MaxValue(s, Cutoff - 1)
```

```
function GenerateResultingStates(GameStates, FirstAgentIdx, LastAgentIdx)
      if FirstAgentIdx == LastAgentIdx
            return GameStates
      else
            newStates = []
            for s in GameStates
                  for a in GetLegalActions(s, FirstAgentIdx)
                        newStates.append(Successor(s, FirstAgentIdx, a)

            return GenerateResultingStates(
                        newStates, FirstAgentIdx + 1, LastAgentIdx
                        )
```

```
function MaxValue(GameState, Cutoff, Alpha, Beta) returns value
      if Cutoff == 0 or GameState.IsWin() or GameStat.IsLose()
            return StaticEvaluation(GameState)
      V <- -inf
      for a in GetLegalActions(GameState, PACMAN)
            v <- max(v, MinValue(Successor(GameState, PACMAN, a), Cutoff-1))

            if v >= Beta
                  return v

            Alpha <- max(Alpha, v)

      return v
```

```
function MinValue(GameState, Cutoff, Alpha, Beta) returns value
      if Cutoff == 0 or GameState.IsWin() or GameStat.IsLose()
            return StaticEvaluation(GameState)
      V <- inf
      states <- GenerateResultingStates([GameState], 1, NUM_AGENTS)
      for s in states
            V = min(v, MaxValue(s, Cutoff - 1)

            if v <= Alpha
                  return v
            Beta <- min(Beta, v)

      return v
```

3.1. Static Evaluations

Although evaluating non-terminal nodes is important for both Minimax and MCTS, it is particularly crucial for Minimax. Since so few plies can be computed in the available time between moves, the static evaluation function becomes a major part of the algorithm, and was one of our major challenges in implementing an effective Minimax agent for Pac-man. In particular, the challenge is to find a function that not only assesses both Pac-man's life expectancy and his efficiency in gathering food, but also weighs the one measurement against the other. Whenever an evaluation function measures both of these goals, it is implicitly (or explicitly) assigning some weight or importance to each goal against the other.

Our approach for balancing risk against reward in the Minimax static evaluations is to calculate a "survival score" and a "reward score" separately, assigning weight to the reward score dynamically according to the survival score. Survival scores are often negative unless ghosts are in an edible state. After calculating the survival score of a game state, a constant threshold value θ is subtracted from the survival score. If that difference is less than 1, no further computations are done, and the valuation of that state is determined entirely by the survival score. Otherwise, the reward score is computed to reflect how much food Pac-man has collected and how near he is to remaining food. Before it is added to the survival score, this reward score is weighted by how high Pac-man's survival score is above threshold:

$$w = log_{\theta}(s - \theta)$$

where s is the value returned by the survival evaluation function. The threshold is computed based on the size of the maze, the number of ghosts and the search depth horizon:

$$\theta = \ -N_s / (N_g \, h^2)$$

where N_s is the total size of the game grid, and N_g is the number of ghosts in the game.

3.2. Evaluating Reward

The problem is more complex than just balancing the reward of eating against the danger of getting eaten, however. Constructing an effective reward evaluation is itself a substantial challenge. We can use the score as an easy heuristic, of course, but it is often not a sufficiently complete basis to judge a move rational or irrational. Pac-man's score often grows only by small increments as he eats at most one unit of food in a single move, yet doing so is the only way to win the game. The small incremental reward of eating the nearest food often can be overwhelmed by the substantial risk associated with that move if doing so will move Pac-man closer to a ghost. In short, it is necessary for Pac-man to take risks in order to score points or win the game, but the risks are often catastrophic while the rewards are usually incremental.

Moreover, if Pac-man is not within two moves' distance of the nearest food (as often occurs at mid-to-advanced points in a game), the two-ply Minimax tree will not contain any nodes with food rewards, even though those rewards are available just "over the horizon," as shown in Figure 1.

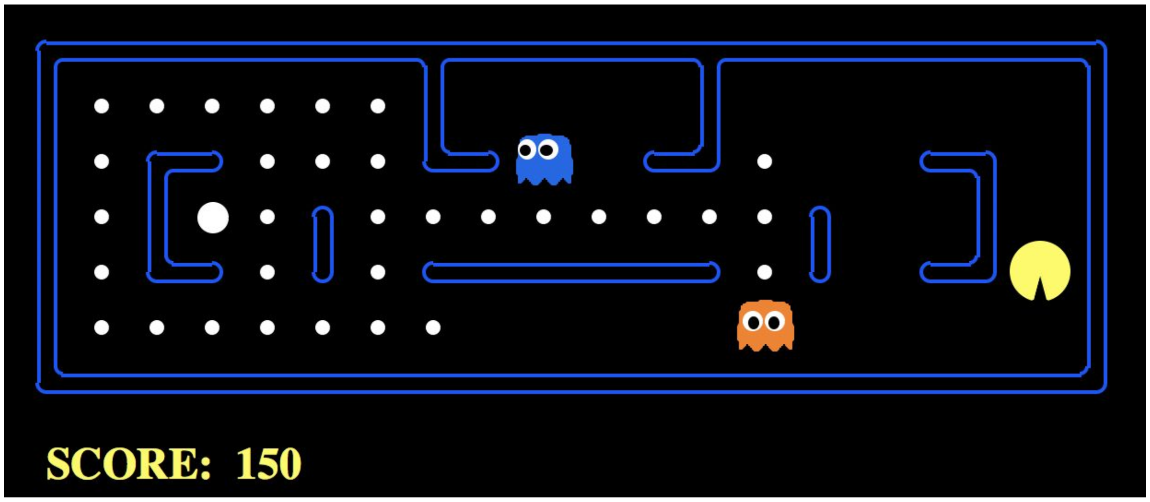


Figure 1: Pac-man is not close enough to food to reach any within the horizon of the Minimax tree.

It is necessary, then, to reward Pac-man for being *near* food as well as for eating it. This introduces another balancing act, because if Pac-man is too greatly rewarded for mere proximity to food, he will simply go near it and admire it without touching, because it may be worth more to him on the board than in his stomach! So, we need to reward Pac-man for each empty square on the board (because this represents food he has eaten), but also to reward him for being near to as many squares as possible that do contain food. We reward proximity to food in proportion to

$$\lambda_f = 1/4^{(ManhattanDistance(PacmanLocation,\, FoodLocation))}$$

To address these problems, our reward evaluation function tries to amplify the value of incremental progress towards the goal of clearing the level of food. It assesses a substantial reward (200 points normalized by the total number of squares on the board) when Pac-man makes a move to a square that contains food. It also divides the score for food eaten by the total number of food remaining, so that each successive unit of food eaten is worth more than the last. On the other hand, we penalize excessively conservative play by multiplying the overall reward score calculated for each state by a "pacing" factor:

$$\lambda_p = log_h(log_h(N_T))$$

where h is a horizon equal to the depth of the Minimax search (in our case 2), and N_T is the number of turns that have led up to this game state. The pacing multiplier penalizes the agent for inefficiency and offsets the tendency of reward scores to increase as the game progresses and Pac-man has eaten more of the food.

A final complication presents itself: if the agent is rewarded equally for each unit of food it has eaten, and is rewarded as well based on proximity to food-containing squares, it can easily clear areas of the board only partially, leaving behind isolated bits of food that are hard to get later in the game [1]. Our approach to minimizing this behavior is inspired by image processing: we run a "filter" over the grid of food positions to detect and penalize areas of isolated units of food. Thus, we reward clearing squares of food, but penalize clearing them unevenly.

```
for square in GameState.Grid
    if square is wall
        continue
    else
        if not Contains(Square, Food)
            Score += 100 / GameState.NumFood
        else
            for g in range(Square.x - m, Square.x + m)
                for h in range(Square.y - m, Square.y + m)
                    if not wall and not Contains(Square, food)
                        Score -= 100 / numSquares
```

4. Monte Carlo Tree Search

The standard implementation of Monte Carlo Tree Search first proceeds from the root node (that is, the current game state), and descends the game tree according to a selection policy until it reaches a node that has not yet been expanded (i.e. number of visits is zero).

That node's children are added to the tree, and rollout begins. This is a simulation wherein we choose Pac-man's moves at random in our study but one can use heuristic function to choose actions, but the ghosts' moves are selected according to a simulation reflecting their tendency to move toward Pac-man. Finally, once simulation reaches a terminal state or the depth limit is exceeded, a score (assigning 1000 or -1000 for a win or loss, a static evaluation estimation in between for non-terminal states) is calculated and backed up the tree, with the score being added to each parent node's score, and the number of visits to the state in question being incremented.

```
function getAction(GameState,num_of_iterations,simulation_depth)
        Visits <- 0
        Reward <- 0
        Children[GameState] <-None
        Parent[GameState] <- 'Root'

        Intelligent_decision <- decision_index(GameState)
        for  iteration in range(1,num_of_iterations) do
                tree_traversal(GameState,simulation_depth,intelligent_decision)

        ## After above iterations, Visits and Reward will be updated
        legalAction <- getLegalActions(GameState)
        Average_Reward <- -inf
        for action in legalActions do
                childGameState <- GenerateSuccessorState(gameState,action)
                cur_Avg_reward <- reward[childGameState] / visits[childGameState]
                If cur_Avg_reward > Average_Reward
                        Best_Action <- action
        Return Best_Action



function choose_tactic(GameState)
        pacPos = GetPacmanPosition(GameState)
        numGhostsNear = 0

        for ghostPos in GetGhostPositions(GameState)
                d = ManhattanDistance(pacPos, ghostPos)
                if d < ALERT_RADIUS
                        return SURVIVAL     ## 0
                if d < 2 x ALERT_RADIUS
                        numGhostsNear += 1

        if numGhostsNear > 1
                return SURVIVAL
        else
                return FOOD         ## 1
```

```
function decision_index(GameState,radius_search=3)
    pacmanPosition <- GameState.Pacman.position

    ###Case 1 when pac-man is really close to the ghost, it has to survive
    Total_sum <-0
    for ghost in GameState.ghosts do
        Total_sum<-Total_sum+ManhattanDistance(pacmanPosition, ghost.position)
    If Total_sum < radius_search do
        Decision <-  0
    ####Case 2 when pacman is far from ghost then it has to grab food
    Decision <-1
    Return Decsion


function tree_traversal(GameState,simulation_depth,intelligent_decision)
    currentGameState <- GameState
    Rollout <- False
    While not Rollout do
        Rollout,currentGameState <- find_leaf_node(currentGameState)


    rolloutGameState <- currentGameState
    scores<- simulate_node(rolloutGameState,simulation_depth,intelligent_decision)
    Visits[rolloutGameState] <- Visits[rolloutGameState] +1
    Reward[rolloutGameState] <- Reward[rolloutGameState]+scores
    backpropagate_score(rolloutGameState,scores)


function find_leaf_node(currentGameState)
    legalActions <- getLegalActions(currentGameState)
    childActions <- getchildActions(currentGameState)

    legalActions <- set(legalActions) - set(childActions)

    If legalActions!=[] do
        for action in legalActions do
            Temp <- GenerateSuccessorState(currentGameState,action)
            If visits[Temp] == 0 do
                Rollout<-True
                break
    Else do
        Temp <- ucb_selection_policy(currentGameState)
        Rollout <- False

    Return Rollout, Temp
```

```
function ucb_selection_policy(currentGameState)
      legalAction <- getLegalAction(currentGameState)
      Ucb_value <-  -inf # A large negative number
      for action in legalActions do
            childGameState <- GenerateSuccessorState(currentGameState,action)
            Temp_ucb_value <- ucb_calculate(childGameState,currentGameState)
            If Temp_ucb_value > ucb_value do
                  select_action <- action
                  Select_state <- childGameState
      Return select_action,Select_state


function ucb_calculate(childGameState,currentGameState,C=2)
      Return Reward[childGameState]+C*Sqrt(
            ln(Visits[currentGameState])/ Visits [childGameState])



function simulate_node(rolloutGameState,simulation_depth,intelligent_decision) do
      curGameState <- rolloutGameState
      While simulation_depth!=0 do
            legalActions <- getLegalAction(curGameState)
            RandomAction <- chooseRandom(legalActions)
            curGameState <- GenerateSuccessorState(curGameState,RandomAction)
            simulation _depth <- simulation_depth-1
      Return getScoresTerminal(curGameState)

function getScoresTerminal(curGamestate)
      If curGamestate.isLose()
            Survival <- 0
      Else
            Survival <-1

      Food = curGameState.getFood()
      Return [Survival,Food]



function backpropagate_score(currentGameState,score)
      If parent[currentGameState]=='Root' do
            Return
      Else do
            currentGameState <- parent[currentGameState]
            visits[currentGameState] <- visits[currentGameState]+1
            Reward[currentGameState] <-  Reward[currentGameState]+score
            Return backpropagate_score(currentGameState,score)
```

4.1. MCTS with Minimax Ghost Avoidance

In experimentation, we discovered that while a Minimax Pac-man agent does not seek rewards efficiently, and cannot (in the real time game) plan effectively, a MCTS Pac-man agent does not perform reliably in ghost-avoidance mode. Our initial experiments showed the agent behaving erratically when ghosts were nearby. We believe this is due to the combination of the randomized nature of the simulated playouts with the relatively small number of playouts we can do in real time (our limit was 20). Therefore, we decided to try to marry the best of each algorithm in our approach, to which we will affectionately refer as "MCTS with MGA," or Monte Carlo Tree Search with Minimax Ghost Avoidance.

In this modification, inspired by the tactics-switching approach of Pepels, Winands and Lanctot, we choose Pac-man's actions by MCTS as long as the Manhattan distance between Pac-man and the ghosts is sufficiently large. Any time the distance threshold is crossed, the agent enters "survival" mode, wherein the only priority is to avoid ghosts that can eat Pac-man. Specifically, if one ghost is within a specified Manhattan-distance "radius," or more than one ghost is within 2 times that radius, MCTS is suspended and a Minimax search begins to find the most promising route to escape the ghosts.

The logic for deciding which algorithm to apply is the same as for choosing tactics:

```
function ChooseSearch(GameState)
      pacPos = GetPacmanPosition(GameState)
      numGhostsNear = 0

      for ghostPos in GetGhostPositions(GameState)
            d = ManhattanDistance(pacPos, ghostPos)
            if d < ALERT_RADIUS
                  return Minimax
            if d < 2 x ALERT_RADIUS
                  numGhostsNear += 1

      if numGhostsNear > 1
            return Minimax
      else
            return MCTS
```

The rationale here is that if one ghost is very close, Pac-man needs to take the optimal path to get away from the ghost. If there are two ghosts nearby, even if they are not as close, the agent still needs to be especially careful, because it is likely to get trapped between the ghosts otherwise.

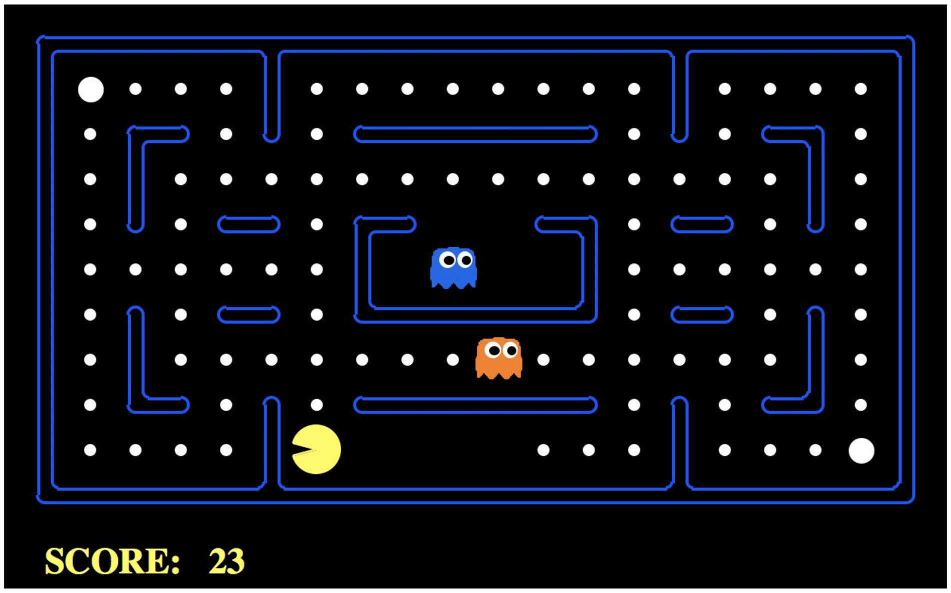


Figure 1: Danger not immediate, but Pac-man is likely to get trapped if he stays here too long.

5. Experimental Setup

We tested each agent--Minimax (with and without Alpha-Beta pruning) and MCTS with Minimax Ghost Avoidance--on two different boards, Small and Medium size, for 50 runs of the game. Each of these agents was tested in the standard Pac-man fully-observable environment, and in a “partially observable” environment in which Pac-man can only see North, South, East and West as far as there is no intervening wall, and can always see everything a radius of one unit of distance away from him vertically, horizontally and diagonally.

For each run, the game framework records a final score, and whether Pac-man won or lost. We present these scores and the number of games won, along with averages and win rates, in the Appendix. Table 1 shows average scores and number of wins for each agent in both board sizes, out of 50 runs each. Observability is specified as total, meaning the classic version of the game where the entire board is visible to Pac-man, and “line-of-sight” (LOS) plus radius 1, meaning Pac-man can see in all four orthogonal directions up to a wall, and can see whatever is in a diagonal square.

6. Results Summary

Table 1 : Average score of the game and number of wins using different search Agent and observability.

Nature of Search	**Observability**	**Game layout**	**Average Score**	**# of Wins**
AlphaBeta Search	LOS* + Radius 1	Medium Classic	-213.46	0
AlphaBeta Search	LOS* + Radius 1	Small Classic	-330.54	0
Minimax Search	LOS* + Radius 1	Medium Classic	-239.5	0
Minimax Search	LOS* + Radius 1	Small Classic	-317.12	0
MCTS with MGA#	LOS* + Radius 1	Medium Classic	38.72	1
MCTS with MGA#	LOS* + Radius 1	Small Classic	-12.2	4
AlphaBeta Search	Total	Medium Classic	124.04	2
AlphaBeta Search	Total	Small Classic	-77.14	4
Minimax Search	Total	Medium Classic	197.46	5
Minimax Search	Total	Small Classic	-122.94	1
MCTS with MGA#	Total	Medium Classic	129.92	3
MCTS with MGA#	Total	Small Classic	187.54	1

***LOS: Line of sight #Minimax Ghost Avoidance**

6.1. Results Analysis: Fully Observable Environment

6.1.1 Minimax

Minimax performed similarly with or without Alpha-Beta pruning, as would be expected since both variations on the algorithm should return the same min/max values.

In the fully observable environment, we found the Minimax agent to be unexpectedly unsuited to the Small game board. The Minimax tests (with and without Alpha-Beta pruning) had a combined average score of -100.4 points, and a total of 5 wins out of 100 combined runs. By comparison, the same agents scored much better on the Medium board: an average of 160.75, with 7 wins out of 100 runs.

Although more study would be needed to determine for sure why this was the case, it appears that despite our efforts at balancing the static evaluation functions, the agent had difficulty weighing the priorities of ghost-avoidance and reward-seeking against each other. Although we experimented with many different static evaluation algorithms, we always found the agent oscillating between objectives, especially when ghosts were relatively near (within a Manhattan-distance of perhaps 5-10 spaces), but not near enough to end Pac-man's life within the horizon of the 2-ply Minimax game tree. Thus, whenever ghosts were at an intermediate distance such as 5-10 units, the agent tended to perform inefficiently because of the conflicting goals of safety and food.

6.1.2. Monte Carlo Tree Search with Minimax Ghost Avoidance

Our modified MCTS with MGA algorithm was able to overcome some of these shortcomings of Minimax on the Small board. Because ghost avoidance only became a priority when the ghosts were quite close (we used a threshold Manhattan distance of 5), the agent was able to gather food efficiently when ghosts were too far away to be an immediate threat. When the ghosts did become an immediate threat, the Minimax search appears to have been reasonably effective in guiding Pac-man away from the ghosts. On the Small board, the agent scored an average of 120.92 over 50 runs, winning the game 3 times.

On the Medium board, however, the MCTS with MGA agent did not outperform the purely Minimax agent, averaging 187.54 over 50 runs and winning only once. We suspect that our Monte Carlo implementation's advantage over Minimax disappeared on the larger board because the MCTS was not able to perform enough simulations/expansions per turn. As the physical space grows, more different playouts are possible, but our iterative limit remains 20, and our maximum search depth remains 4, so our Minimax agent becomes less reliable at finding good paths to explore.

6.2. Results Analysis: Partially Observable Environment

The greatest advantage of our MCTS with MGA algorithm over pure Minimax was seen in the partially observable Medium size environment. The MCTS with MGA agent scored an average of 38.72, as compared with -239.5 for Minimax. This is consistent with our expectation that an algorithm designed for stochastic environments would perform better than Minimax, which is going to have less utility when the adversary's actual state is unknown.

On the Small board, the score advantage of MCTS with MGA was larger, averaging -12.2 for its 50 runs, as compared to -317.12 for Minimax. MCTS with MGA also won 4 times on the Small board, as compared to zero for Minimax. This is consistent with our observation that Minimax did more poorly than expected in the Small fully observable environment.

7. Conclusion

In the course of our experiments with MCTS and Minimax, we found that one was not always more effective in Pac-man than the other. The constraints of real time play cause problems for both algorithms, but fortunately, those problems are somewhat different for each search strategy. Since each has strengths and weaknesses in this domain, it is possible to use them selectively, alternating dynamically to best suit the current situation.

MCTS proved better at directing our agent toward rewards when the adversary was not directly threatening Pac-man, but failed to reliably guide him away from imminent danger because its selection and simulation was not exhaustive. Minimax was more effective in exactly those situations, in which a ghost was one or two moves away. Minimax is more suited for pursuit-evasion because it is able to systematically plot out the possible ways the game could unfold from the given state, when the critical time horizon was within its own maximum tree depth.

Given that Minimax tends to be easier to apply when the depth of the game tree is limited, we were surprised to find that Minimax on its own fared more poorly in the Small game environment than in the Medium environment. We suspect this is due to more frequent conflicts between ghost avoidance and reward seeking, conflicts that make our Minimax agent less efficient and more “indecisive” as it shifts back and forth between competing goals.

One shortcoming of our Pac-man agent is its inability to avoid being caught between two ghosts. We hope in future work to design an agent that can avoid this fate. This would require either some way of extending the search tree deeper to consider such eventualities, or possibly better heuristics that indicate the likelihood of being trapped.

Another idea for future work is to make the moments when policy selection switches more seamless. As it stands, we observe poor results when the agent oscillates between behaviors. It is possible that parameter tuning could help this, but it is worth further examination.

8. References:

1. Tom Pepels, Mark HM Winands, Marc Lanctot, “Real-Time Monte Carlo Tree Search in Ms Pac-Man, *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 6, No. 3, September 2014
2. N. Ikehata and T. Ito, “Monte-Carlo tree search in Ms. Pac-Man,” in Proc. IEEE Conf. Comput. Intell. Games, 2011
3. http://ai.berkeley.edu/project_overview.html