Final Project: PacMan with Blind and Heuristic Search

Benjamin Friedman
friedmab@oregonstate.edu
CS 531
Project on Github: https://github.com/montymxb/pacman

March 13, 2020

Abstract

In this paper we (the author) describe findings regarding the effectiveness of heuristic search and blind search in Pacman. The goal was to identify whether the game of Pacman could be played by searching the board blindly. In addition, we wanted to see if Pacman could be played with heuristic search as well, taking into account the relative positioning of the eponymous main character to ghosts and food. In our work we found that blind search of sufficient depth produces solutions approximately 40% of the time. We explore how this may vary significantly on the implementation, the heuristic used, and the time taken to produce a solution. For our cases A^*, $Minimax$, $DepthFirst$, and $BreadthFirst$ search were all compared together. In addition, varying heuristics were designed for usage with A^* and $Minimax$ search to see how performance could be improved. The rest of this paper explores our approach and analysis into this topic, and provides our findings.

1 Introduction

This paper details research, implementation, testing, and findings regarding blind and heuristic solution techniques for the game of Pacman. We choose Pacman in particular because it is an internationally recognized game, it has a large history behind it, and existing research for comparison was plentiful. As such, we were able to derive some approaches to Pacman that we believe to be novel with regards to our heuristic techniques. Again, there is extensive work done in this area, so it is possible that research may have been done on some of the compound heuristics we designed.

Pacman was chosen also because it poses a distinct search challenge. Although the search space is large (298 pathable quantized spaces in our implementation, 246 of which can have food, 4 ghosts and 1 Pacman gives us $(2^{246} * 5^{298})$ states, without factoring in the afraid state for each ghost), we hypothesized that there were ways to tackle the problem such that it was solvable in smaller portions. In particular, we were interested in looking at how A^* and $Minimax$ search could be used to approximate the best moves at given states. To compare against this, we also wanted to check with 2 blind search algorithms, Depth First and Breadth First search. These were each implemented in our test setup, along with various heuristics. This gave us a way to run observable simulations, and to collect the results to make our conclusions.

2 Background

Pacman was first introduced in May of 1980 [**?**]. The game was revolutionary at the start, and quickly became an international sensation. The fact that this paper is about Pacman is a testament to its persistence in contemporary culture. With this in mind, the game itself is still quite simple. The game involves a simple character, 'Pacman', who navigates a maze via 4 inputs: up, down, left, and right. Without input Pacman will continue in the last direction he was taking until he hits a wall. In the maze, there are capsules that Pacman can eat, which add to a score. Upon eating all the capsules in the maze, Pacman moves onto another maze full of capsules again, which can be described as advancing to the next level. Besides the capsules, there are also cherries and power pellets; special capsules that lie in the 4 corners of the map. These capsules are used to help mitigate the next factor of the game, the Ghosts [8].

Four ghosts roam the maze of Pacman. Should Pacman touch one, he loses a life. If he loses 3 lives, the game ends. The only exception being a short time duration after Pacman eats a power pellet, where he can temporarily eat the now 'afraid' ghosts, which move slower during this duration. However, even an eaten ghost is in a temporary state, as the ghost will quickly respawn in the center of the map, and give chase again.

The attractiveness of trying to apply search to solve the game of Pacman lies in the complexity of the goal. Although you are trying to get as many capsules as possible, you must also not lose lives. The ghosts in the original Pacman were deterministic in their actions, and that version of the game is considered solved (such as in the case of the Cherry Pattern) [12]. In the version of Pacman that we were testing on, the actions of the ghosts were purely stochastic, with the ghosts randomly deciding to give chase or to turn onto a different route. This posed a unique challenge to searching, as the successive states can be known, but the outcome is not predictable.

Additionally, the domain of the game is discrete, as Pacman is moving along various points amongst the maze, but never ventures in-between points. The same goes for the ghosts, which even when moving slowly still move in fixed discrete amounts. In our implementation this

was a bit tough to say the same, as the project we based our work on was implemented in an HTML5 canvas. It was still discrete in terms of movements, but the incrementing amounts were so small that it appeared continuous. In our presentation we indicated this was continuous, which was a mistake, and we clarify here that it is in fact a discrete domain (just a very large discrete domain).

Historically, Pacman has been considered a good testing bed for learning AI algorithms since approximately 2010, as noted in the text [1]. Many examples online exist of applying search with learning to solve the game. There are also additional examples of $Minimax$ and A^* being used to solve Pacman in courses at UC Berkeley, so the application of the algorithms themselves is not novel [5]. It has been documented that Pacman can be effectively solved with Monte Carlo Tree Search and Minimax search, as well as POMDP; which can be considered a more effective approach than the prior two [1] [10]. Material exists online showing that Pacman has even been adopted as the primary method of teaching AI in introductory level courses [5] [4].

3 Problem

For Pacman, the search problem is in identifying a path that eats all the capsules, without losing all of Pacman's lives. As mentioned before, in this version of Pacman the ghost's actions can be considered stochastic. This poses a problem, as there is no way to always predict the exact move a ghost will take next, and so the optimal path for eating all capsules in the least moves cannot be followed reliably. If it was followed blindly, it would result in Pacman failing to account for the movements of the ghosts, and losing his lives far too quickly. This issue can be seen in our conclusions, which also brings up an additional point of solving a maze quickly over slowly.

So, the probem can be summarized neatly into two goals:

- eat all the capsules in a level.
- avoid losing lives to the ghosts.

Looking at the problem like this, we saw that we needed to devise an approach that allowed Pacman to eat as many capsules as it could safely, whilst avoiding the ghosts. If a decision comes between eating a capsule and losing a life, the priority should go on the life.

Another aspect of this problem is in how the ghosts are implemented. They have differing levels of aggression, and as such they move in different ways. The red ghost (Blinky) will actively chase you more often than any of the other ghosts. The pink and blue ghosts (Pinky and Inky) both will move in a way that they will get in front or behind Pacman, resulting in a pincer maneuver (a move that traps Pacman with no escape on either side of a path). The orange ghost (Clyde) is essentially random, and will not actively target the player [9]. Some approaches view the ghosts as being on separate teams that compete to beat Pacman independently, but we assumed that all agents acting on the same team with the same purpose [1].

With the varying levels of aggression, Pacman needs to be able to avoid becoming trapped (via a pincer) which is a common maneuver performed by the ghosts. This can be mitigated if Pacman is near a power pellet, which allows a means to eat the ghosts and quickly break out of an inescapable situation. In our approach we did not design heuristics or search around prioritizing the use of power pellets, but we suspect this may have positive implications if used correctly.

To note some implementations prioritize achieving the best score possible for a given level or game. In our approach we did not factor this in as a priority, and instead focused on

survivability and completion of levels alone. To this extent, we ignored the Cherries that show up in the middle of the map, which provide points but do not serve a function in completing levels.

4 Hypothesis

Our primary hypothesis was that heuristic search, using singular or compound heuristics, can be used to plan a path to complete levels in Pacman, whilst minimizing the loss of lives, and in a way that was superior to blind search. An auxiliary hypothesis that we would like to make is that good positioning is likely an important heuristic to succeeding with heuristic search.

5 Our Approach

Our approach used A^* and $Minimax$ to attempt to solve Pacman. Since there are many possible solution paths (many ways to ultimately end up with eating all capsules) we designed each of these search algorithms to indicate the best direction for Pacman to take, considering the current state of the board.

To make this decision process we put the board through a quantization phase, where we take the state of the board, and quantize it to a simpler representation (see figure 1). As mentioned before, this version is not continuous in state, but it is close enough that the number of states on the actual board is far too many to search effectively. Instead, we quantize the positions based on the grid used to lay out the capsules across the board. These are evenly spaced at discrete intervals, and so we used these same intervals to quickly create a grid based representation of the entire board. Any given space is either open, occupied by a capsule, or is a wall and cannot be occupied by anything. With the grid in place, the positions of Pacman and the ghosts are quantized as well, and stored as part of our quantized model state. For all of our search, successor states are generated off of this quantized model, producing additional quantized models. We found that this approximate approach was faster, allowed us to search farther, and gave us results that were nearly identical to what the search would do in the actual board space.

The implementation of A^* we used closely resembles IDA^*, as we wanted to take advantage of setting a depth limit on our searches. The pseudocode can be seen below:

```
A*():
  path := []
  model := quantize_current_state()
  path.push(model)
  result := _A*(path, 0, 0)
  return result.direction


_A*(path, cost, depth):
  cur := path.pop()

  h := heuristic(cur)
  cost += heurstic
```

Figure 1: A visualization of the quantized grid that is actively searched by the A* algorithm, with green spaces being high cost and red spaces low cost. Slightly askew due to converting the quantized coordinates back to the local ones for visual tracing.

```
  if depth exceeds limit
        return cur cost and direction taken to gen this node

  min := MAX
  bestDir := -1

  successors := genSuccessors()

  for each successor as s
    if s not in path
      path2 := path
      path2.push(s)

      result := _A*(path2, cost + 1.0, depth+1)

      if result.min < min
        min = result.min
        bestDir = s.direction
  return min and bestDir
```

Our implementation of *Minimax* was as follows:

```
minimax():
  path := []
  model := quantize_current_state()
  path.push(model)
  result := minimax_min(model, 0, -MAX)
  return result.dir


minimax_min(cur, limit, last):
  if limit >= depthLimit
    return heuristic(cur) and cur.direction

  successors := genSuccessors(cur)
  v := MAX
  dir := undefined

  for each succcessor as s
    result := minimax_max(s,limit+1,v)
    if result.cost < v
      v := result.cost
      dir := s.direction

    if v < last and useAlphaBetaPruning
      return v and dir
  return v and dir


minimax_max(cur, limit, max):
```

```
if limit >= depthLimit
  return heuristic(cur) and cur.direction

successors := genSuccessors(cur)
v := -MAX
dir := undefined

for each succcessor as s
  result := minimax_min(s,limit+1,v)
  if result.cost > v
    v := result.cost
    dir := s.direction

  if v > last and useAlphaBetaPruning
    return v and dir
return v and dir
```

Following the implementation of our algorithms, we also have the implementations for our heuristics. These are as follows:

- Food Count: Returns the number of capsules left on the board, expressed as ($foodCount$), a common heuristic.
- Closest Food Distance: Returns the Manhattan distance of the closest food to Pacman, expressed as ($foodDistance$), a common heuristic.
- Closest Ghost Distance: Returns the Manhattan distance of the closest ghost to Pacman, expressed as ($ghostDistance$), a common heuristic.
- Closest Food or Ghost: Returns the distance of the closest food, unless the closest ghost is < 10 moves away (via Manhattan distance from before), at which point this heuristic then returns the distance of the closest ghost to Pacman instead. This heuristic prioritizes food when it is safe, and when it is not it encourages running. This is a compound heuristic that we did not find in our research, so we believe this to be novel.
- Closest Food or Ghost, or Chase: *Uses Observation.* This is a test heuristic that we wanted to use to see how the agent would improve *if* we were allowed to make observations on whether ghosts were eatable or not. It's nearly the same as the prior heuristic, but it allows Pacman to chase ghosts down when it is safe to do so. This is a compound heuristic that we did not find in our research, so we believe this to be novel.
- Closest Ghost + Distance to Board Center: This heuristic uses the distance of the closest ghost combined with the distance from the center (horizontally) of the board. This is expressed with a decreased emphasis on vertical positioning by ($vertDistance * 0.25 + ghostDistance$), and is intended to test whether positioning can help keep Pacman out of pincering maneuvers; which often occur in corners on the top and bottom of the maze. This is a compound heuristic that we did not find in our research, so we believe this to be novel.
- Ghost Distance + Food Count + Food Center: Adds distance to the nearest ghost with the current food count, and finally adds the food 'center'. This center is computed as the average of all the Xs and Ys for all capsules on the board, roughly giving the average center of food on the map. This was intended to also help with positioning, by helping Pacman prioritize movement towards regions of higher capsule density, expressed as

($ghostDistance * 0.4 + foodCount * 0.3 + foodCenter$). This is a compound heuristic that we did not find in our research, so we believe this to be novel.

Out of the above heuristics only 2 are admissible with regards to the minimal number of moves to get all capsules on the board: food count and closest food distance. So only these two heuristics would have a chance of producing an optimal solution, and the rest will not. We considered this acceptable, as an optimal game of Pacman (in a stochastic domain) is not an outcome that can be expected. Instead, we focused on the goal of producing heuristics that allowed an agent to win games in realtime by taking the most likely moves to produce a win, and also those moves that would prevent a loss; depending on the heuristic.

6 Baseline Approach

The baseline method we used to compare against our algorithms was *BreadthFirstSearch* (BFS) and *DepthFirstSearch* (DFS). We searched on the same quantized models as before, and implemented them with logic to terminate on any goal state (having a capsule on it), and to terminate searching on successor nodes that have a ghost on them.

BFS was implemented as follows:

```
bfs():
  lastMaxVal := 0.0
  m := quantize_world()
  frontier := [m]
  visited := []
  goal := undefined
  depth := 0

  while frontier.length > 0
    cur := frontier.first()
    cur.nodes := []
    remove first from frontier
    depth++
    visited.push(cur)
    successors := genSuccessors(cur)
    for each successor as s
      s.parent := cur
      if !containedIn(visited, s)
        if s.didEatFood
          goal = s
          break
        else if !s.areGhostsOnPacman()
          frontier.push(s)

    if goal
      break

  if goal
    step := goal
    while step.parent != m
      step = step.parent
```

```
    return step.direction
```

DFS was implemented as follows:

```
dfs():
  lastMaxVal := 0.0
  m := quantize_world()
  depthLimit := 50
  frontier := [m]
  visited := []
  goal := undefined
  depth := 0

  while frontier.length > 0
    cur := frontier.pop()
    cur.nodes := []
    if depth > depthLimit
      break
    depth++
    visited.push(cur)
    successors := genSuccessors(cur)
    for each successor as s
      s.parent := cur
      if !containedIn(visited, s)
        if s.didEatFood
          goal = s
          break
        else if !s.areGhostsOnPacman()
          frontier.push(s)

    if goal
      break

  if goal
    step := goal
    while step.parent != m
      step = step.parent
    return step.direction
```

7 Experimental Setup

Our experimental setup was based on an existing open source implementation of Pacman from Github, which was developed by Lucio Panepinto [2]. This implementation uses numerous instances of an HTML5 Canvas and Javascript. This was *not* a fast implementation, and additionally was difficult to integrate with, but it proved helpful for visual testing and debugging. Using this implementation, we integrated in our agent, algorithms, heuristics, and logic to help get the agent into playing the game itself instead of a human. Once integration was completed, we setup a test rig that automatically played 20 games using a pre-set algorithm and a heuristic (if used) combination. The speed of each game was accelerated by 5x, greatly

increasing the number of games we could test. However, because of this some of our results may not be consistent with out measurements at normal speed, as many of the timings being adjusted created slight discrepancies in the game (such as ghosts not making it back to the spawning box before re-spawning in the maze itself).

8 Generation/Acquiring of Test Data

The results of each run were collected into a working report while the games were played, and upon completion these results were then collected for analysis. All games were played on the original Pacman maze layout, and with the same starting state for each new game. As mentioned above, for each Algorithm/Heuristic combination 20 games were run.

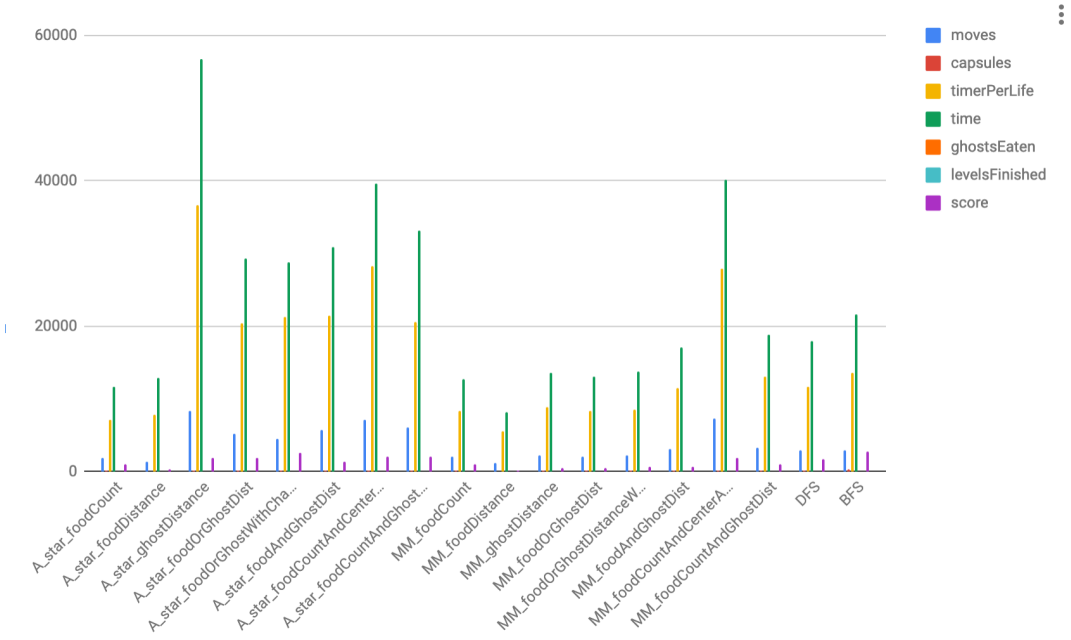
We focused on collecting the following data for each game:

- Moves taken
- Capsules eaten
- Time per life
- Time per game
- Number of ghosts eaten
- Number of levels finished
- Score

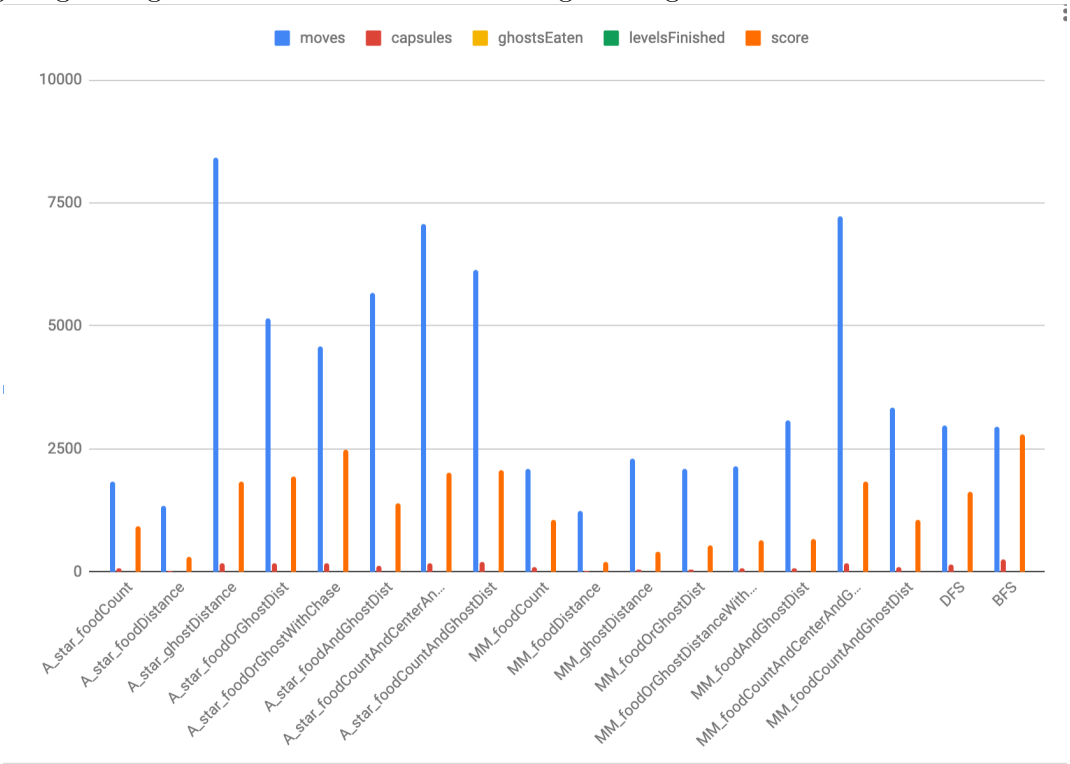
These results were then exported into a series of Google Sheets graphs for us to compare and contrast the data.

9 Results

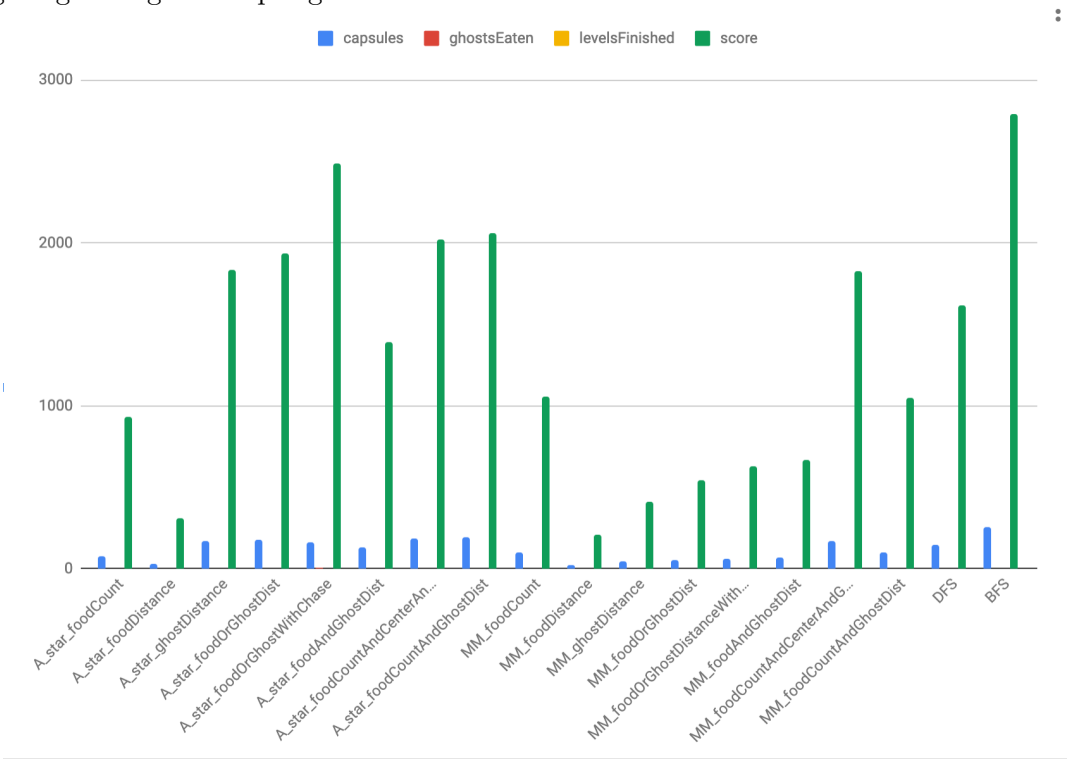
Highlighting average of overall time and time per life.



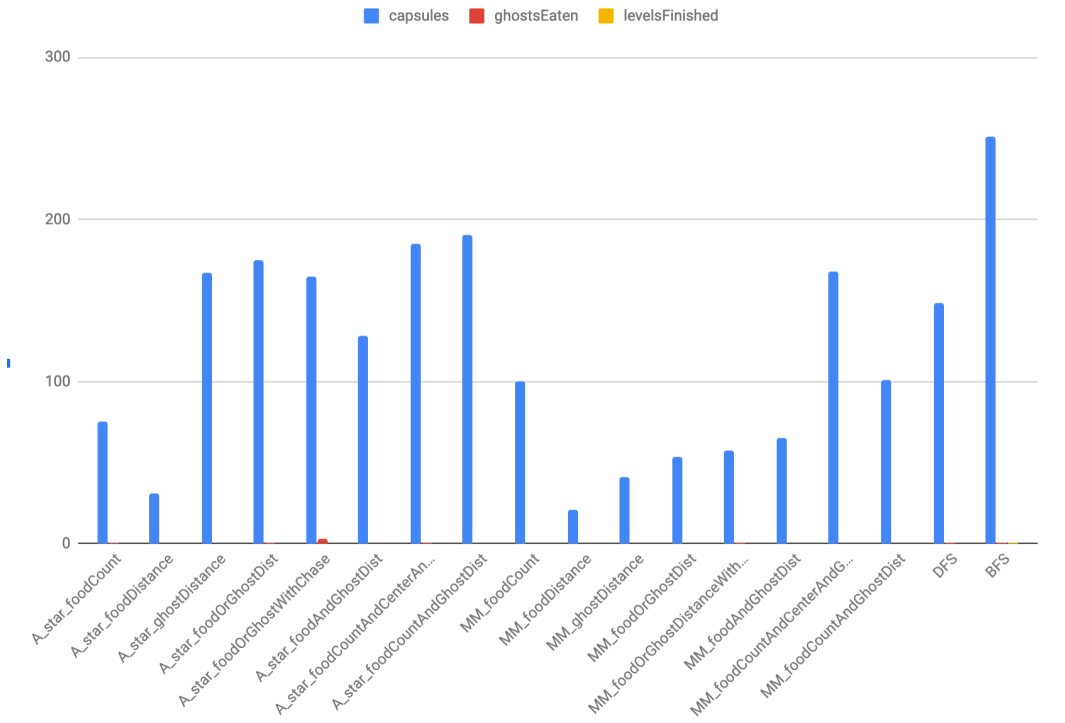
Highlighting average number of moves taken throughout a game.



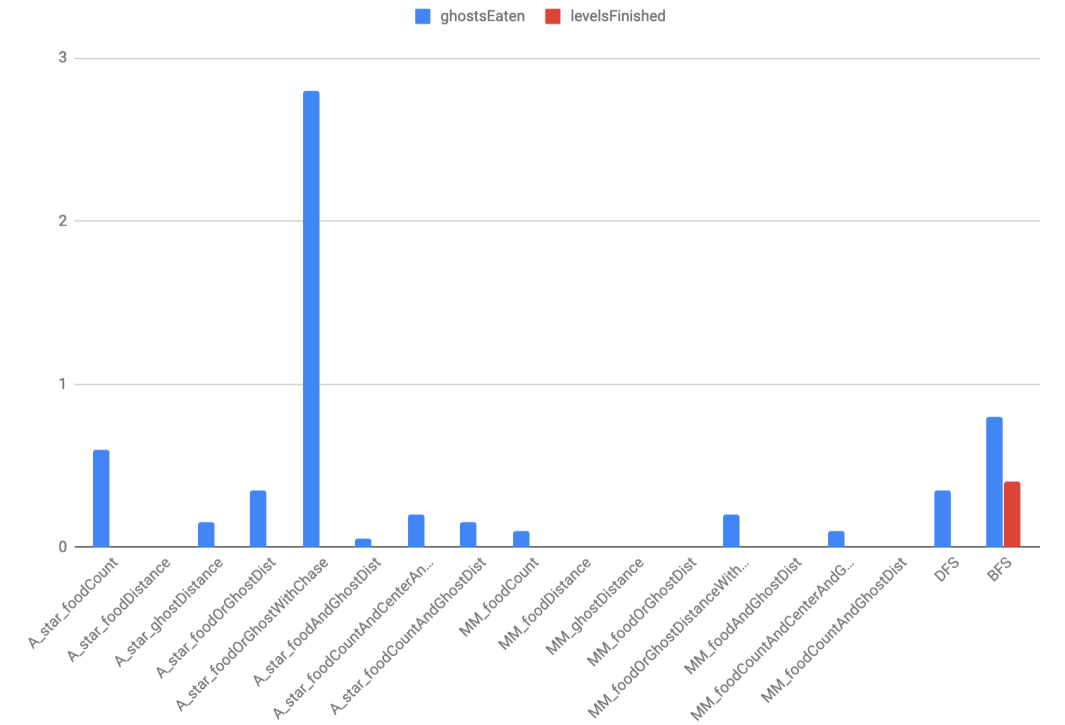
Highlighting average score per game.



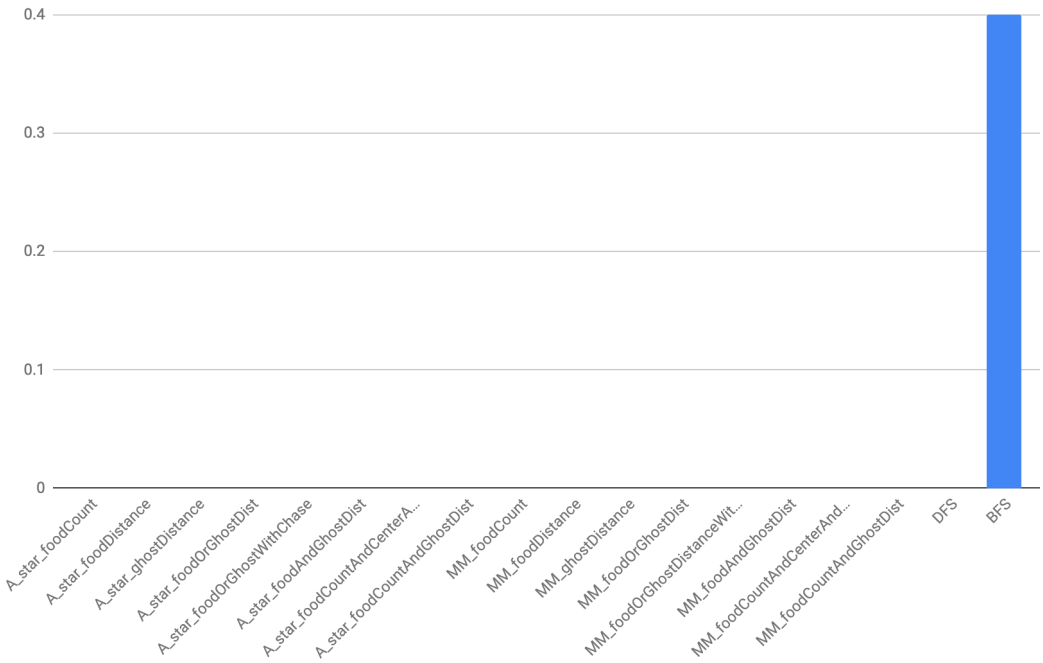
Highlighting average of capsules eaten per game.



Highlighting the average of ghosts eaten per life.



Highlighting the average win rate, where only BFS succeeded with a 40% average win rate.



In our results we were able to see that out of all of our algorithms, and all of our heuristic mixes, only BFS recorded successful wins; with a ratio of 40%.

Out of all the algorithms, the $A\_star\_foodOrGhostWithChase$ heuristic with A^* recorded nearly 3 ghosts eaten per game on average, much higher than any other approach. This was to be expected, as this was the only approach that took advantage of observing the ghost state, which was an experimental observation, but ancillary to our hypothesis.

Overall, the number of capsules eaten per game was dominated by BFS, and followed by $A\_star\_foodCountAndGhostDist$ along with $A\_star\_foodCountAndCenterAndGhostDist$. Beyond these the majority of the A^* algorithms recorded significantly higher average capsules eaten per game, as well as $MM\_foodCountAndCenterAndGhostDist$.

The average scores appear to be closely correlated with the number of capsules eaten, with BFS being the best scoring, and $A\_star\_foodOrGhostWithChase$ being the second.

The average number of moves was found to not correlate with the average score or capsules eaten. Many A^* algorithms took a large number of moves, more than double those taken by BFS, and all with a lower score than BFS. The exception being $MM\_foodCountAndCenterAndGhostDist$, which was in the top 3 of highest number of average moves, next to the two other combinations in A^*.

Finally, we can see a positive correlation with the overall time of a game, time per life, and the associated number of moves taken. In effect, moves associated nearly linearly with time. From this, we can see that all of the A^* instances that factored in ghost distance in some way survived for longer periods of time than most of the other approaches. This follows what we observed with the number of moves taken on average in the one $Minimax$ case as well. We can also see the next best performer being the ghost distance combined with the distance from the average food center.

As an extra mention, we did record 2 wins for the $A\_star\_foodCountAndGhostDist$ heuristic, and an additional win for $A\_star\_foodOrGhostDist$. Unfortunately, these 3 wins occurred before the test setup was finished, and so they were unable to be officially recorded in our final results.

10 Conclusions

Our results suggest that at least a level of Pacman can be beaten quite easily with blind search, and that although heuristic search comes close, it is not quite able to beat a level.

With regards to the first part of our hypothesis, none of the heuristic search approaches in our recorded results were able to beat a level of Pacman. However, all of the A^* implementations with ghost distance were able to survive significantly longer than BFS, DFS, and almost all other algorithm/heuristic combinations. In regards to our auxiliary hypothesis, incorporating the average food center heuristic seems to have had a significant affect on survivability for both the A^* and $Minimax$ implementations, suggesting that good positioning is an important heuristic to account for.

The results make sense considering the limitations of our algorithm implementations in Javascript. In particular, A^* and $Minimax$ were both implemented recursively. Given the slower execution speed of Javascript, the recursive calls dramatically reduced performance during tests. This made it necessary to reduce the search depth limit to 9 for A^* and 5 for $Minimax$. Although alpha-beta pruning was implemented for $Minimax$, it wasn't enough to remove the extreme costs of a recursive implementation.

This can be contrasted directly with the DFS and BFS algorithms, which were both implemented iteratively. DFS was capped to a maximum search depth of 50, and BFS was not capped at all, as there were negligible performance issues with BFS. As a result, BFS had a dramatic increase in performance over any of the other algorithms, and was capable of solving the puzzle in a closer-to-optimal fashion. Moves were picked with a priority on the closest capsule that wasn't behind a ghost. This did leave the agent to live significantly less than almost all A^* implementations, but it was able to accrue a higher score (and to actually finish levels) in much less time.

We conclude that our heuristics did achieve an excellent lifespan, and in many cases played far better than expected in terms of avoidance (and better than this author in most cases). We further believe that an iterative $A*$ implementation with the heuristics presented here could perform at least as well if not better than a blind BFS. In addition, heuristics that take advantage of power pellets could allow for stronger play against pincers or other trapping moves by the ghosts. Ultimately, we believe that a properly optimized complete search algorithm used with our heuristics, or a variant of them, in a stochastic domain could consistently beat 1 or more levels of Pacman.

References

[1] Russell, Stuart. Norvig, Peter. *Artificial Intelligence: A Modern Approach*. Prentice Hall. 2010.

[2] Panepinto, Lucio. "Pacman". *Github*. June 29th, 2018. https://github.com/luciopanepinto/pacman. Accessed March 5th, 2020.

[3] "List of Atari 2600 Games". *Wikipedia*. March 4th, 2020. https://en.wikipedia.org/wiki/List_of_Atari_2600_games. Accessed March 5th, 2020.

[4] sghost1991. "Adversarial-Search-Pacman". *Github*. Aug. 17th, 2015. https://github.com/sghosh1991/Adversarial-Search-Pacman. Accessed March 5th, 2020.

[5] DeNero, John. Klein, Dan. "Teaching Introductory Artificial Intelligence with Pac-Man". *University of California, Berkeley*. Sept. 6th, 2019. http://denero.org/content/pubs/eaai10_denero_pacman.pdf. Accessed March 5th, 2020.

[6] Partidge, Matthew. "22 May 1980: Pac-Man hits the arcades". *MoneyWeek*. May 22nd, 2015. https://moneyweek.com/392564/22-may-1980-pac-man-hits-the-arcades. Accessed March 5th, 2020.

[7] "Pac-Man". *Wikipedia*. March 3rd, 2020. https://en.wikipedia.org/wiki/Pac-Man. Accessed March. 5th, 2020.

[8] "Pac-Man Fever". *Time*. April 5th, 1982. https://web.archive.org/web/20110122151527/http://www.time.com/time/magazine/article/0%2C9171%2C921174%2C00.html. Accessed March 13th, 2020.

[9] Morris, Chris. "Five Things You Never Knew About Pac-Man". *CNBC*. March 3rd, 2011. https://www.cnbc.com/id/41888021. Accessed March 5th, 2020.

[10] Cassandra, Anthony. "The POMDP Page". *POMDP*. March 13th, 2020. https://www.pomdp.org/. Accessed March 13th, 2020.

[11] "A* Search Algorithm". *Wikipedia*. March 4th, 2020. https://en.wikipedia.org/wiki/A*_search_algorithm. Accessed March 5th, 2020.

[12] Kirbykid. "Pac-Man Design: Deterministic and Random Ghosts". *Design Oriented Blog*. June 9th, 2015. http://www.designoriented.net/blog/2015/06/09/201569pac-man-design-deterministic-and-random-ghosts/. Accessed March 13th, 2020.