**Pathfinding with Multiple Agents**

**Karl Farhat and John McCrummen**

University of Colorado at Colorado Springs, USA

{kfarhat, jmccrumm}@uccs.edu

**Abstract**

Pathfinding is an important element which exists in nearly all video games with an AI component that involves movement. When an AI entity must travel from one place to another, it is rarely the case that it can make it there in a straight line. This is where pathfinding algorithms come in handy. There are many different types of pathfinding algorithms which consider a variety of elements such as obstructions, finding quicker paths, or multiple conflicting AI agents in the same environment. In this research paper we discuss different algorithms for cooperative pathfinding as well as adversarial pathfinding. We then attempt to apply what was learned to the classic game Pac-Man, by programming the actual Pac-Man in the game in addition to the ghosts, then evaluating our results by comparing the number of pacdots collected when playing against each ghost alone.

**Introduction**

In its simplest sense, pathfinding accomplishes the goal of creating a path in which an AI agent (typically computer player/character) can go from its current position to another specific position. However, this is never as easy as it sounds due to the fact that other outside variables must be considered. For instance, the AI agent must also avoid any obstacles that may be in the way, and frequently, must also avoid other AI agents trying to accomplish the same or similar tasks. In the best scenario, the distance from a current point to a goal point is a straight line. This is known as the Euclidean distance, and is calculated disregarding any possible walls or obstructions. Millington (2006) claims that the “Euclidean distance is always either accurate or an underestimate”. In other words, the actual travelling distance can’t ever be any better than the Euclidean distance, it can only be worse. In situations where there are very few obstructions, the Euclidean distance can actually be a pretty accurate estimate of how far an agent is from the goal, but in a complicated environment it becomes quite poor.

The case in which there is more than one AI agent involved is known as multiagent pathfinding. Multiagent pathfinding has two distinct cases, cooperative pathfinding and adversarial pathfinding. Cooperative pathfinding doesn’t necessarily mean that the AI agents are working with one another. In fact, this is hardly ever the case. The cooperative aspect of the term simply refers to the fact that the AI agents must not conflict with one another, and can sometimes work together to each find their own paths. For example, a solution must be avoided which “requires two agents to occupy the same space at the same time, or more abstractly, a solution in which agents access more of a resource than is currently available” (Standley, 2010). This issue clearly complicates the pathfinding problem and work must be done to avoid such conflicts.

Adversarial pathfinding refers to cases where the multiple agents are actual opponents in a game, and where one agent’s goal is to avoid the opponent, while the opponent’s is to catch the agent. This means that consideration of agents controlled by the adversary is to be included in the algorithms (Ivanova and Surynek 2014).

**Related Work**

**The A\* Algorithm**

Although the A\* algorithm can be used to solve many different types of problems, not just those of which deal with artificial intelligence, it has been applied to the problem of pathfinding consistently.

In each step of the algorithm, it looks at all of its successor nodes (possible locations to go to next) and determines in one way or another the ideal choice amongst them and goes there next. It also keeps track of the cost it has taken to get to the current node as well as an estimate of the remaining cost to get to the goal. This can be represented by the function *f(n) = g(n) + h(n)* where *f(n)* is the total cost, *g(n)* is the cost from the starting point to the current point *n*, and *h(n)* is the estimated cost from the current point *n* to the goal state (Cui and Shi, 2011). If at any point the algorithm ends up in a repeated state, the new path to that state is compared with the old one, and whichever one is shorter remains while the other is discarded. Two collections are also created called *open* and *closed* which hold the nodes that are to be seen next and the nodes that have already been looked at, respectively.

The A\* algorithm has received much praise for its effective problem solving compared to classic uninformed search techniques such as breadth-first and depth-first search. It has proved to be quite useful for many applications involving pathfinding with a single AI agent. However, when multiple AI agents get involved, the A\* algorithm is not sufficient enough and certain adjustments need to be made to comply with the extra AI agents.

**Local Repair A\***

One of the main variations of A\* is the Local Repair A\* (LRA), first described by Silver (2005) whom claims it is widely used in the video game industry. In LRA, paths for each of the agents are computed the same as the normal A\* algorithm, and conflicts between the agents are not discovered until execution. This means that the agents will follow their paths until a collision is imminent, then the agent that is about to move into an occupied position will recalculate the remainder of its route. However, this approach does have a couple of drawbacks. For instance, it can cause cycles and deadlock, where two agents can no longer make any progress. Silver (2005) proposed several other variants of A\* to overcome these problems by the use of cooperative search, including Cooperative A\* (CA), Hierarchical Cooperative A\* (HCA), and Widowed Hierarchical Cooperative A\*. Standley (2010) proposed as well other variants of A\* to solve these problem with cooperative search like A\* with Operator Decomposition in which each of the agents can “either move towards the goal, stay idle (wait action), or move away from its goal” (Goldenberg et al 2012).

**Dynamic A\***

In dynamic A\*, or D\*, it is no longer assumed that the environment or paths within it remain constant. In certain scenarios, the path that an agent discovers to reach a goal can change, either in the sense that the weight of certain paths are inconsistent during game play, or the available paths themselves can be altered. In addition to this, the location of the goal can even change, forcing the agent to recalculate its path multiple times along the journey.

Similarly to the case of Local Repair A\*, Dynamic A\* can work as normal until it is forced to make a recalculation. This doesn’t pose too much of a problem and is relatively effective as long as the number of recalculations stays somewhat low. If, however, there is so much replanning that it overwhelms the agent, it can prevent progress from being made (Millington 2006). For instance, if the agent is forced to recalculate its route before it has even finished determining the previous route it was supposed to take, it will simply not work and the agent will be stuck.

**Minimax Algorithm**

The main algorithm used for adversarial pathfinding is the minimax algorithm. This algorithm takes into account an adversary, or an opponent against the main agent that we control. In games, the minimax value refers to the smallest value that the opposing agents can force the agent to receive, without knowing his actions. The algorithm consists of computing each node’s minimax value.

Alpha-beta pruning is an algorithm that aims to decrease the number of nodes that are evaluated by the minimax algorithm. It stops completely evaluating a move when a possibility of a move worse than a previously examined one is found. The terms alpha and beta refer to the current best maximum and best minimum choices so far, respectively. These values are looked at to determine which nodes can be safely pruned. In alpha-beta pruning, it is often the case that entire subtrees of the search tree can be removed in addition to individual leaves (Russell and Norvig 2010). Although the search is still exponential, the time to search can essentially be cut in half with this process.

There are multiple things to take into account when performing pruning. One of these is the move order, which is the order in which the possible moves to make are presented in the search tree. If the better moves are presented before the worse ones, the bad ones are more likely to be pruned, making the pruning process more effective. However, the better moves aren’t typically known in advance, hence the need for the search in the first place. To get an idea of which moves are in fact better, the results from previous minimax searches can be used (Millington 2006).

Another property to consider during alpha-beta pruning is the size of the search window. The search window is the interval between the alpha and beta search values (Millington 2006). It is typically started off as (-∞,+∞) and is shrunken down as the tree is traversed, but allowing for a smaller search window from the start can make for more pruning and a faster algorithm. Knowing safe limits for alpha and beta helps to shrink the search window effectively without overdoing it.

**Expectimax Algorithm**

There are several factors that the minimax algorithm does not consider which makes it not very effective sometimes. Games that include explicit randomness like rolling dice, unpredictable opponents where the opponent responds randomly, and the fact that actions can fail, where for example a moving robot has dysfunctional wheels.

A solution to this problem is the expectimax algorithm.

The expectimax is a brute force, depth first search algorithm that generalizes the minimax algorithm concept to games with randomness and chance (Veness 2006). Instead of computing each node’s worst case value or minimax value, the algorithm computes the average case or expectimax value of each node by considering the probability of each event occurring.

Incorporating alpha-beta bounds to expectimax algorithms is not possible. At non-chance nodes, the best value is determined by the value of one move. It is then sufficient to compute the value of only one successor to fall outside the search window. At chance nodes, the algorithm is computing the weighted sum at every node, and it needs to be shown that this weighted sum falls outside the search window (Veness 2006).

A solution to the alpha-beta pruning is the Start1 algorithm first introduced by Ballard in 1983. Further research needs to be done to further understand how this algorithm works.

**Our Approach**

For the Pac-Man scenario, the A\* algorithm, or any cooperative improved version of it, could in fact still work in the case of ghosts, even though there are a total of four ghost AI. This is because the game allows for ghost to overlap (pass through) one another as they move across the board. If this was not the case, Pac-Man would probably be a lot different with ghosts bumping into one another and constantly changing directions or ‘getting confused’.

The paths in the Pac-Man game are also exceptionally limited compared to one of today’s 3-dimensional open environment games, lowering the scope of possible movements greatly. At any point in time, a ghost is essentially limited to 3 possible moves to choose from next: go left, go right, or go straight. This also only occurs at one of the intersections of the game board rather than whenever/wherever. This takes into consideration the fact that in the original Pac-Man, a ghost won’t just change directions all of a sudden and start going the other way (unless, of course, the Pac-Man eats a power pellet).

For our research, we analyzed all the previously performed pathfinding algorithms to determine the strengths and weaknesses of each in order to decide what would work best for the Pac-Man game. We then applied these to the ghosts and compare the behavior to the original game in attempt to make them perform better. Considering the pathfinding aspect of Pac-Man is quite different than in a typical game, we created our own algorithm for Pac-Man himself which was more relevant to the 2-dimensional limited movement environment and the fact that all agents move simultaneously.

For the actual Pac-Man in the game, an adversarial multiagent pathfinding algorithm was originally considered. The minimax algorithm discussed briefly in the previous section seemed like the perfect solution. The minimax algorithm with alpha-beta pruning would have to be used with Pac-Man as our main agent, and all the other ghosts as his opponents. The algorithm would have to be slightly modified so it handles several opponents instead of just one. Another possible solution was the expectimax algorithm. The expectimax is implemented in a similar way to minimax algorithm, but with probability for each node occurring. Unfortunately, both of these ideas were scrapped, considering that they were meant for turn-based implementations, while in Pacman all agents are making decisions simultaneously.

We used Unity, a game development platform, for our project. It helped us build the layout of the game Pac-Man, and the ability to program each of the ghosts and the Pac-Man individually in the way we wanted to. For evaluation purposes, we compared how long our programmed Pac-Man lasted compared to an average player.

**Implementation**

**Ghosts**

In order to compare the different pathfinding algorithms and their levels of success within the game Pac-Man, we applied a different algorithm/approach to each of the four ghosts separately.

Blinky, the red ghost, was implemented in a fashion very similar to the Dynamic A\* algorithm. Seeing as the goal (Pac-Man himself) was not constant, it was clear that recalculations were necessary to perform particularly often in order for him to successfully hunt Pac-Man down. To accomplish this, a collection of decision points were created on the game board, each of which was at a location where Blinky was capable of making a move other than just turning around. There were 80 of these total, and every time Blinky came across one, he would recalculate his next decision based on the current location of Pac-Man from there. Rather than calculate the entire path leading to Pac-Man, Blinky would simply decide which move (up, down, left, or right) would bring him closer to Pac-Man at the next immediate step, since the unpredictable movement of the player would prove it pointless to do all that extra calculation. Contrary to the original game, this allows Blinky to turn around abruptly at a decision point rather than just going left, right, or straight.

The movement of Pinky was created similarly to the basic A\* algorithm, where the goal (Pac-Man’s location) was calculated and the shortest of each of Pinky’s moves was chosen at each point until the goal was reached. Once Pinky reached her goal, she would note the next location of Pac-Man and repeat the process. This was different than Blinky in the fact that Pinky would wait until she reached Pac-Man’s last location before calculating the next path. Much like the actual behavior of Pinky in the original game, however, Pinky didn’t actually go straight to Pac-Man himself, but made an attempt to cut ahead of Pac-Man and block him. This was applied by setting the goal for Pinky to be calculated by noting the direction Pac-Man was headed relative to his current location, and would set the closest intersection (decision) point to him at that time as Pinky’s next goal. In the case where Pac-Man wouldn’t be moving, his current location itself would simply be the goal. Seeing as Pinky would explicitly choose the shortest path at each decision point, this could cause Pinky to get stuck at a wall going back and forth on it and to never actually go around it. Furthermore, since Pinky wouldn’t calculate her next goal until she reached the current one, she would simply be stuck in this back and forth motion for the rest of the game until Pac-Man got caught by someone else. To avoid this, a variable was created which would count the number of moves that Pinky made so far for the current goal, and once it reached a certain value she would simply recalculate.

For the final two ghosts, Clyde and Inky, their behavior was implemented using the Cooperative A\* algorithm discussed earlier. The two ghosts work together in a sense that they are both looking for Pac-Man at the same time. Their behavior is similar to Blinky’s (dynamic A\*). They first look for all possible decision points closest to their goal, Pac-Man’s position, then chose randomly with the ones with lower heuristics having higher probability. That way they have a lower chance of crossing paths, and a higher chance of catching Pac-Man. Their path would be recalculated each time a decision point was reached. A variable, ghostStrength, was attached to these two ghosts which represented the likelihood of them choosing the shortest path at each decision point vs another longer one. A higher ghostStrength meant a more accurate path to Pac-Man, but also limited the randomness in decision making which could make Inky and Clyde have movements which were too similar. Another behavior in their movement was that they were not allowed to cross over each other, so whenever an intersection was imminent, their paths would change (they’d turn around). This behavior might have reduced their performance, however.

**Pac-Man**

Our original idea on how to implement Pac-Man’s movements was to use the Minimax or Expectimax algorithms. In the Pac-Man scenario, this would mean there would be alternating moves between Pac-Man and one of the ghosts, which would look strange and not be natural to the traditional feel of the game. Furthermore, it would be difficult to implement for more than one ghost at a time. With these considerations, we decided to go a different route.

To start off, we implemented Pac-Man’s behavior by having him calculate which ghost is closest to him before each move, then picking the immediate direction which is farthest away. Essentially, he would just escape the ghosts with the optimal path to get away each time, disregarding the pellets. Although it was effective in keeping Pac-Man from getting caught, this didn’t allow for many pellets to be collected.

We then decided to have Pac-Man focus on collecting pellets as long as there wasn’t a ghost within his near vicinity. This distance was represented by a variable, FEAR\_DIST, which was experimented with in order to determine the optimal distance from a ghost Pac-Man should be before being concerned about it. When Pac-Man was far enough to not worry about ghosts, he would look each direction for pellets and go to the closest one. This made him much more effective, but would still get stuck not knowing what to do if there weren’t pellets in his line of ‘sight’. An attempt was made to then have Pac-Man locate the nearest dot, even if it wasn’t in sight, but since his decisions were updated so frequently, he would just move back and forth quickly behind a wall and never actually went around it. Even still, Pac-Man was able to perform better than the average human.

**Results**

**Ghosts**

The effectiveness of each ghost was measured by testing the game on a collection of people and comparing how many pellets they could collect under each circumstance. For the testing of Inky and Clyde, they were given ghostStrength values of 10 and 5, respectively, meaning Inky was a little more inclined to make the correct decision each time (shortest path).

The game was tested on 10 different participants with 10 runs on each ghost (with inky and clyde counting as one). The averages of their runs were then computed and compared in attempts to discover the best pathfinding solution. Overall, Inky & Clyde performed the best with 7/10 of the participants succumbing to them the most. In other words, the runs with these two ghosts allowed for the fewest number of pacdots to be collected compared with Blinky and Pinky for 7 of the participants. Furthermore, the total average number of pacdots collected when facing these ghosts was ~94 while the averages for Blinky and Pinky were ~111 and ~115 respectively, showing that Blinky and Pinky performed relatively similarly with Blinky overcoming Pinky by a hair.

According to these results, the Cooperative A\* algorithm showed to be the best performer in capturing Pac-Man. However, this could very well have been due to the simple fact that in the case of Inky and Clyde, the participant was up against two ghosts at once rather than just one, and despite their performance individually, this advantage of numbers could have been the only reason for their success. Other factors that may have skewed the results in general are the idea that participants could have changed their strategies mid-game, either causing for an increase or decrease in their performance half-way through. Also, each participant naturally improves a little as they progress and get a hang of the game. An attempt to avoid this was made by testing the participants with a different order of ghosts each time, but it still could have made a difference.

|  |  |  |
| --- | --- | --- |
| **Blinky** | **Pinky** | **Inky and Clyde** |
| 121.1 | 134.9 | 112.7 |
| 56.6 | 112.7 | 67.2 |
| 82 | 104.5 | 79.6 |
| 93.5 | 84 | 130.4 |
| 104.1 | 106 | 118.4 |
| 129.4 | 138.4 | 95.1 |
| 129.1 | 108.8 | 80.8 |
| 195 | 137.9 | 113.3 |
| 77 | 87.4 | 68 |
| 121.1 | 134.9 | 70 |

Figure 1: Avg. Comparisons of Each Ghost for 10 Players

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Blinky** | **Pinky** | **Inky and Clyde** |
| **# Wins** | 2 | 1 | 7 |
| **Averages** | 110.89 | 114.95 | 93.55 |
| **Sums** | 1108.9 | 1149.5 | 935.5 |

Figure 2: Total Averages and Win Count for Each Ghost

**Pac-Man**

Several different groups of tests were performed in order to create the ‘right’ formula for Pac-Man’s behavior. The first of these experiments was to determine the optimal value for FEAR\_DIST. In these tests, Pac-Man was faced against all four ghosts and ten tests were ran on four different values for the fear variable (2, 3, 5, & 7). The averages of these tests were compared with one another, the results of which are shown in Figure 3.

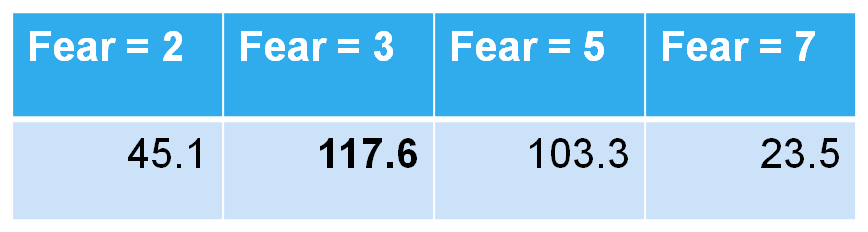


Figure 3: Avg. Pacdots Collected Against All Ghosts with Different FEAR\_DIST values

After an optimal value of 3 for FEAR\_DIST was determined, tests were performed on each ghost separately with this value. Pac-Man was never caught by Blinky or Pinky, but he got stuck after collecting 339 and 230 pacdots, respectively. For Inky & Clyde, he collected an average of 191.7 pacdots over ten tests.

Finally, the results for each of the ghosts was compared with the results obtained from the previous tests done on human players. These values are shown in Figure 4. In every case, the programmed Pac-Man performed significantly better than the average human (more than double pellets collected).

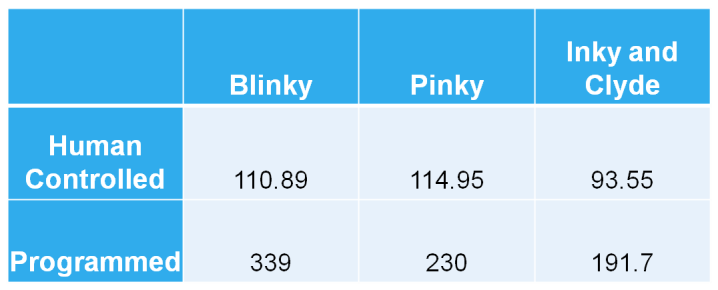


Figure 4: Comparison of Human-Controlled vs. Programmed Pac-Man

**Future Work**

Although this project was quite successful, considering the drastic improvement of our programmed Pac-Man compared with human players, things can still be done to make it better. Pac-Man is still capable of getting stuck in certain areas and stopping when he could be going after more pacdots. A proper implementation of pathfinding to one of the remaining dots could be done so that Pac-Man actually reaches the dot before deciding to do something else. This could perhaps be accomplished by pausing all of Pac-Man’s other movement mechanisms until that specific path was performed.

Another more powerful improvement would be applying weights to each of the decision points on the board and basing Pac-Man’s movements on these weights and essentially applying reinforcement learning to his behavior. This would especially enhance the performance of Pac-Man when faced with either Blinky or Pinky, whose movements and decisions are consistent rather than random, creating more effective learning for Pac-Man. This may also help with the issue of not being able to catch every pacdot.

**Conclusion**

In this paper, we briefly talked about known pathfinding algorithms that are most used in video games. These algorithms can be assembled into two main groups, cooperative pathfinding algorithms, with non-colliding paths for multiple agents plotting a route from their initial and final destinations, and adversarial pathfinding, where in addition to finding non-colliding paths as cooperative pathfinding would, the inclusion of agents acting as opponents are taken into consideration. We then discussed the algorithms we implemented and compared their results. Finally, future improvements for the project were discussed.

**References**

Ballard, W. B. 1983. The \*-minimax search procedure for trees containing chance nodes. *Artificial Intelligence, 21:327-350, 1983*

Cui, X., and Shi, H. 2011. A\*-based Pathfinding in Modern Computer Games. Melbourne, Australia.

Goldenberg, M.; Felner, A.; Stern, R.; Sharon, G.; and Schaffer, J. 2012. A\* Variants for Optimal Multi-Agent Pathfinding. *Proceedings of the Fifth Annual Symposium on Combinational Search.*

Ivanova, M., and Surynek, P. 2014. Adversarial Cooperative Path-Finding: Complexity and Algorithms. *2014 IEEE 26th International Conference on Tools with Artificial Intelligence.*

Millington, I. 2006. Artificial Intelligence for Games. San Franciso, CA.

Russell, S. and Norvig, P. 2010. Artificial Intelligence: A Modern Approach Third Edition. New Jersey, USA

Silver, D. 2005. Cooperative Pathfinding. In Young, R. M., and Laird, J. E., eds., *AIIDE*, 117-122. AAAI Press.

Standley, T. 2010. Finding Optimal Solutions to Cooperative Pathﬁnding Problem. *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, California, USA 173-178*

Standley, T., and Korf, R. 2011. Complete Algorithms for Cooperative Pathfinding Problems. *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence, California, USA.*

Veness, J. 2006. Expectimax Enhancements for Stochastic Game Payers. *The University of New South Wales. School of Computer Science and Engineering. Wales. UK*