# Pathfinding Algorithms in 2D Games

Andrew Spittlemeister

May 2017

## 1 Abstract

This paper will focus on the decades old problem of developing well-suited pathfinding algorithms for use in video games. A significant amount of research is focused on evolving a definitive search algorithm that can be applied to the mass of pathfinding problems; however, this paper will show the lucrative results of defining a search algorithm that is specific to the game's rules state space. We use the popular game of PacMan as the pathfinding algorithm testing grounds to compare classical search algorithms to our algorithm specific for the PacMan state-space. Moreover, we use pathfinding algorithms to program the Ghost movements in the game (to move towards PacMan). The classical algorithms that we use are Breadth First Search and A Star Search that is utilizing the Manhattan Distance as its heuristic. We compare both of these to an algorithm we are calling Context Dependent Subgoaling A* (CDSA*). Given the proper expansion limits, this algorithm performs as well as the classical algorithms but in a fraction of the time.

## 2 Problem Description

In this project we (Andrew Spittlemeister and Jordan Opdahl) have produced a solution for developing an artificial intelligent agent to be used in a game such as PacMan. The game is played on a grid based maze in which the user must navigate a sprite through all the traversable grid locations to collect tokens. While the user is trying accomplish this task multiple computer controlled sprites (typically called ghosts) are also navigating the maze towards the user sprite, the player wins the game if he/she successfully avoids all ghosts and collects all tokens before being touched by a ghost. The artificial intelligence problem at hand is to program a rational agent for all ghost sprites to efficiently navigate the maze towards the user controlled sprite. What makes this problem particularly interesting is that the user controlled sprite is not always at a fixed location on the grid which means that with every movement of the user's sprite the ghost agents will need to reevaluate their path. The nature of the grid-based maze implementation of the map also means that the state-space representation of the ghost agent will be a graph with cycles. Furthermore, this means that

the search function that will aid in agent navigation will have to accommodate for this attribute. We should also note that each ghost sprite does not pay attention to or care about the whereabouts of the other ghosts. With this in mind there is the possibility that two or more ghosts will cross over the same grid location at the same time. To go by general convention of this game and to avoid dealing with these special cases, we allowed the coexistence of two or more ghosts in the same location without any repercussions.

To build a successful agent for each ghost we will need to come up with a way for the ghost agent to compute an optimal path towards PacMan within the interval of each timestep and then be prepared to move to the first position of that path upon the end of the timestep. We should note here that since the PacMan agent is user controlled, the ghosts do not have any knowledge of which direction the PacMan sprite will move (or if it will move at all) when computing their move action for the next time step. We chose to write the PacMan game from scratch using the Python 3 language primarily using the Pygame library to handle loading game graphics to the screen. Using a high level language such as this, it was very important to our project to use efficient pathfinding algorithms for each ghost when the computation time is constrained. To keep this project within the achievable realm of delivering a fully functioning game by the defined due date, we have not explore any predictive measures that the ghost sprites could take into consideration in movement (i.e. if PacMan is continually moving in the same direction, one could make a prediction that it will continue in the next time step). Instead we have calculated a path that is using PacMan's current location as its goal state in movement for the next time step.

# 3 Background and Literature Review

The problem of pathfinding (generally defined as finding the shortest route from one place to another) has been a very highly sought after problem to solve in the game industry as it has parallels and analogous models present in the simplest of games (i.e. like PacMan) to the most modern, complex, and three-dimensional games as well. Typically a good pathfinding algorithm is used to efficiently and cleverly traverse a player character or non-player character over some terrain that is most likely filled with obstacles or various objects of hindrance. This algorithm must be clever as to obey the rules of the game (i.e. a character who doesn't normally walk through walls should not walk through walls) so that the user can play the game fairly and without noticing blatant irrational actions in the products of the algorithm. This algorithm must be efficient because this calculation may have to be done often, as in our implementation of PacMan it must be done at least 3 times between every frame (that is 192 times per second). If these calculations could not be completed in that time it would result in significant error handling or simply an unplayable game. In research and collaboration with game production company BioWare Corp. done by Vadim Bulitko et al., it was found that the company holds strict limits of 1 - 3 ms to compute paths for all agents in its games. Previous alternative methods usually

included the use of variants of Dijkstra's algorithm, which is a breadth first search algorithm that has the possibility of taking an extremely long time to finish and has little to no control over which path options to explore over others.

A primary motivation for improving pathfinding algorithms is the almost exponential increase in traversable terrain complexity and user environment. Popular games like World of Warcraft not only have a massive map for a character to efficiently navigate but is also an online game that must take into account a lot of unknown variables produced by every user on the server. Another genre of games called Real Time Strategy (RTS) games typically has to tackle the task of calculating navigation paths for a huge amount of separate units simultaneously [1]. In our PacMan game we only need to control 3 units at a time but in an RTS game this could easily be on the order of thousands of units. Additionally, our implementation of PacMan Ghost sprites does not take into account any other Ghost sprites as part of the environment (i.e. they can pass over one another); but it is very common to want to navigate multiple units on a terrain that dynamically manages and deals with where other units might be moving to as well. Coping with these new and complex types of problems while wanting to maintain the benefits and fitness of the A star search algorithm have opened a completely new field of study; that is, the study of environment and agent heuristics.

Without a heuristic the A* algorithm will still be complete (in other words it will find the shortest path) but will perform no better than any other breadth first search algorithm. With a particularly bad heuristic it may perform even worse than traditional breadth first search algorithms. But the important thing to note is that we now have an opportunity to give our algorithm more knowledge about its (possibly changing) environment. It is also useful to acknowledge the portability of the A* pathfinding algorithm. Given its generic nature of being founded around a graph based state-space representation, we can easily manipulate the algorithm to find paths in many different types of environments. Popular implementations of game environments include two-dimensional square and hexagonal grids, three-dimensional cubic grids, and mesh graphs [4].

The A* algorithm does not innately handle finding the optimal path in a dynamic environment in real time. If a constant heuristic is applied to it, it has a very high probability of losing completeness. To retain completeness in a real time search algorithm, we need to make the assumption that at any given state there is an explorable action that is "safe". In other words an agent should be able to reach the goal state from any state that is reachable from the agent's starting state [2]. The first requirement of a new heuristic function is simply that it should be able to modify itself given a change in the environment. With this construct in mind we could imagine a variant of A* search that computes a path towards the goal state that is of a set length, then the agent executes that path, reevaluates is heuristics, and repeats until the goal state is reached. This allows us to not spend too much time traversing through a length of path that at time of execution is less optimal than it was at time of searching (i.e. avoid getting stuck in local minima). Another requirement is that we should have an idea of the quality of our heuristic at any given state. In general, heuristic functions become

more accurate when they are closer to the goal state (generally due to the lack of variation between the current state and the goal state when they are close). So if we have measure of quality of the current heuristic we can dynamically fine tune the length of the path that we are computing every iteration. If the heuristic is known to be too inaccurate for use then the algorithm can choose to compute a longer distance until a certain criteria for accuracy is met. Conversely, if we are finding that the length of the path we need to compute is on the order of the total path length then the algorithm may choose to instead to move the goal closer to the current state, thus improving search heuristic accuracy to the new subgoal [2]. These algorithmic approaches are the backbone of the Dynamic Learning Real-Time A* (DLRTA*) algorithm [3].

# 4 Problem Approach

As previously stated we chose to build our PacMan game from scratch so as to be able to have more direct control over the game's state-space. The way that Pygame library deals with moving images around the screen is based on mapping the images to certain pixel values. To simplify the search space we overlaid a grid map that defined traversable and non-traversable square cell locations. This map can be seen in Figure 1. Blue grid locations represent non-traversable spaces (the walls of the maze) and black grid locations represent traversable spaces (the paths of the maze). Both Pacman and the Ghosts are only able to navigate via the traversable grid locations and were restricted to only making horizontal and vertical movements (i.e. no diagonal movements). Assigning pixel groups to grid locations in this way reduced the amount of possible locations for the sprites from 369,664 pixels to 163 traversable grid cells.

To program the movement for either a Ghost or PacMan sprite we used the notion of assigning a goal cell that is laterally or vertically adjacent to the sprites current cell. This therefore can define a direction that the sprite can move in. When a direction is specified the sprite moves 1 pixel per frame which corresponds to exactly 2 grid locations per second. When a sprite has arrived at its goal grid cell, the direction is set to null until it is set again either by user input or search algorithm results. Ghost search algorithms will run whenever the ghosts have fully achieved moving into their previous goal cell. Again, this will occur twice every second. It is useful to note here that the Ghosts do not need to know the entire path to the goal cell that PacMan is in, just the first direction they need to step in along that path. The PacMan grid location is always known to the Ghosts when computing paths.

Our PacMan game has 3 different Ghost sprites that attempt to catch Pac-Man: a red, green, and orange ghost. To compare the different search algorithms we decided to implement a different search algorithm for each ghost.

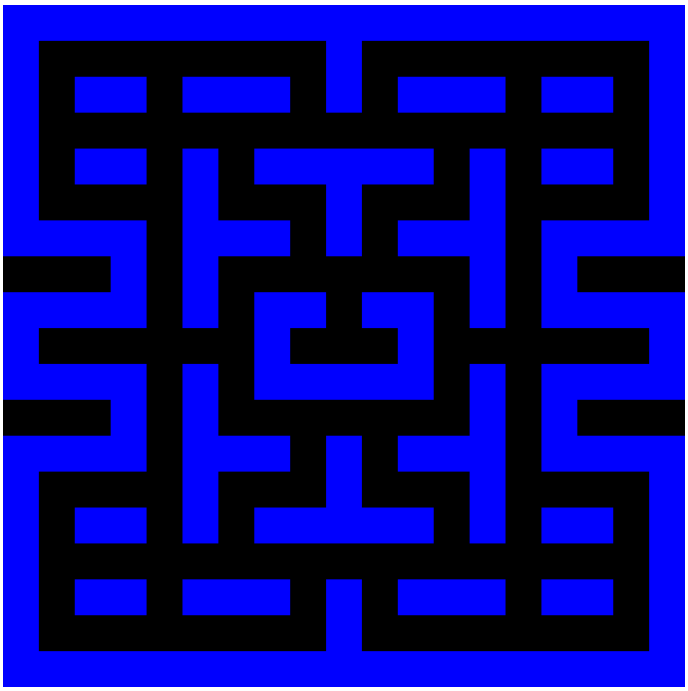

Figure 1: Grid based PacMan map.

### 4.1 The Orange Ghost

Our orange ghost utilizes the classical Breadth First Search pathfinding algorithm to fully compute the minimal path from its current grid location to the PacMan's grid location. The algorithm then traces back the path to the beginning. From here the ghost now knows the first cell it needs to move to, so it sets that cell as its goal cell; and by extension the ghost now knows the direction in needs to move in for the upcoming time step. In this implementation of breadth first search we used the actions of moving vertically and horizontally to explore the state-space that represents our grid maze. We denoted the cost of moving to a traversable grid cell as 1 and the cost of moving to a non-traversable grid cell as infinity. With these assignments the breadth first search algorithm is able to find the optimal path from the orange Ghost to the PacMan agent. The algorithm uses an Open and Closed list to keep track of nodes already expanded and nodes that have yet to be expanded upon. The Closed list was also used as a safeguard to check that the optimal path did not traverse a grid cell more than once.

## 4.2 The Green Ghost

Our green ghost utilized the slightly more efficient A* pathfinding algorithm which also computes the optimal path from its current grid location to the PacMan's grid location. This algorithm also traces back the path from the goal cell back to the first step to figure out the direction in which the Ghost should move in for the next time step. This algorithm runs similar to breadth first search but expands nodes in order of the sum of the least cost and heuristic function output. The costs for moving into cells is calculated in the same way as done in breadth first search (i.e. 1 for moving to a traversable cell and infinity for moving into a non-traversable cell). Since the ghost always knows the cell location of PacMan, A*'s heuristic function, h(n), is simply found by using the manhattan distance from the ghost's current location to PacMan's current location. Since this heuristic is admissible and consistent, then similar to breadth first search, the A* algorithm is complete and will find the optimal path.

Due to the presence of the heuristic function, the time complexity of this algorithm is now dependent on the accuracy of this function as well as the state-space representation of the current time step. This differs from breadth first search which only depends on the state-space representation of the current time step. If the heuristic is admissible, the A* time complexity is no worse than that of breadth first search (and is often times much better). Our A* algorithm also uses an Open and Closed list to keep track of nodes already expanded and nodes that have yet to be expanded upon. The Closed list once again is used as a safeguard to check that the optimal path did not traverse a grid cell more than once.

## 4.3 The Red Ghost

The red ghost employs the use of a slightly modified A* like algorithm, called Context Dependent Subgoaling A*. This algorithm utilizes the unique 3 facets of our game to improve the time in which it takes to compute the ghosts next direction. These 3 aspects include that our map is finite, the ghost only needs to determine the initial direction in which it needs to move, and the unique aspect of our game in which the ghost moves at the same speed as the PacMan sprite. This last fact allows us to know that if a ghost moves in the correct direction along the optimal path towards PacMan, then even if PacMan moves in the opposite direction, the optimal path will be no further than it was previously in the last time step. In many cases, since the PacMan agent is trying to avoid 3 ghosts, the PacMan agent may be inclined to not move to a cell that is in the opposite direction of this particular ghost. The context dependent subgoaling A* algorithm is identical to the classical A* algorithm but defines a limit to how many nodes in the state-space can be expanded. When this limit is surpassed, it uses the next best node in the Open list as the final subgoal to the path. For the defined depth and a consistant heuristic function, this node (which represents a cell) is considered the best estimate of a portion of the optimal path.

Since we know that when the ghost chooses the correct direction towards PacMan the complexity of search stays relatively the same, then given any movement of PacMan we can use the same expanding node limit in the search algorithm to correctly predict the next direction the ghost should move in along the optimal path to PacMan. Note that this algorithm is using the same heuristic function (the manhattan distance) as the classical implementation and is also using the Open and Closed list in the same manner. The limit that this algorithm uses is specific to the problem it is being applied to. In our specific case we needed to determine the limit by which the algorithm could still execute the optimal path from the ghosts initial starting position at the beginning of the game to PacMan's initial starting position at the beginning of the game. Once this is found we know that we have locked that particular ghost within a range of search that produce the correct direction along the optimal path at every time step. Figure 2 below shows the initial starting positions for PacMan and each ghost sprite.

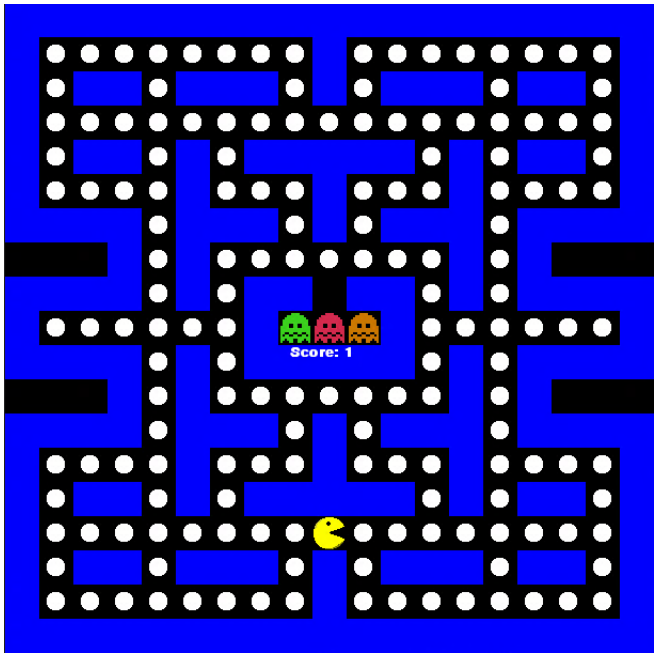


Figure 2: Starting Positions of PacMan and Ghost Sprites.

This algorithm offers a large increase in efficiency compared to the classical A* approach. However, this increase in efficiency is only found when a search for the optimal path requires the expansion of more nodes than the limit defined by CDSA*. Once the path is short enough to require less nodes than the defined limit, both the classical A* and CDSA* will perform identically.

# 5 Experiments and Results

To determine the efficiency of each search algorithm, we kept track of three variables per algorithm: number of nodes generated, number of nodes expanded, and computation time. The number of nodes generated is how many nodes were added to the open list. The number of nodes expanded is the count of how many nodes actually needed to be expanded and their children added to the open list. The computation time is simply the time it took the algorithm to complete the search from its current node to the goal (or subgoal) node.

We obtained our results by initially starting the game so that each ghost left the ghost-holding area found in the center of the maze. We then monitored the three variables mentioned above for each ghost when they were at the same location on the grid. To keep things consistent, we did not move PacMan to obtain our results. The results below in Table 1 were obtained when the ghosts were still relatively far away from PacMan but on the same square.

Table 1: Results of different Algorithms

| | **A*** | **CDSA*** | **BFS** |
|---|---|---|---|
| **Nodes Generated** | 189 | 114 | 531 |
| **Nodes Expanded** | 64 | 39 | 188 |
| **Computation Time (s)** | 0.00150 | 0.00050 | 0.00900 |

At a first glance, it is clear that Context Dependent Subgoaling A* is the most efficient algorithm in terms of time and space complexity. In order to successfully modify the A* algorithm to be this efficient, we were required to execute multiple experiments on the movement of the red ghost with different limits applied to how many nodes its algorithm was allowed to expand. When the limit was set too low, the red ghost would often oscillate back and forth indefinitely between two cells until the movement of the PacMan agent triggered a different goal cell outcome for the algorithm. When the limit was set very high, the ghost didn't improve its efficiency in search time compared the classical A* algorithm. The 'sweet spot' for the limit was found to be 39. That is, the modified A* algorithm is allowed to expand 39 nodes before it must pick a direction for the ghost to move. When tested, the algorithm moved the ghost on the optimal path to PacMan with no random movements and it paralleled the movement of the ghost that was utilizing the A* search. Given the starting positions at the beginning of the game, it was with this limit that we have ensured that the PacMan agent would not be able to move far enough away from the red ghost where the right direction could not be realized by the CDSA* algorithm. This optimal path for the start of the game can be seen by the red line below in Figure 3.

To ensure that all of these algorithms worked on more than a static goal node, we tested how they reacted to when PacMan moved instead of sitting still. Each ghost followed the movement of PacMan and attempted to close the gap between the two at all parts of the grid. As expected, when the amount of

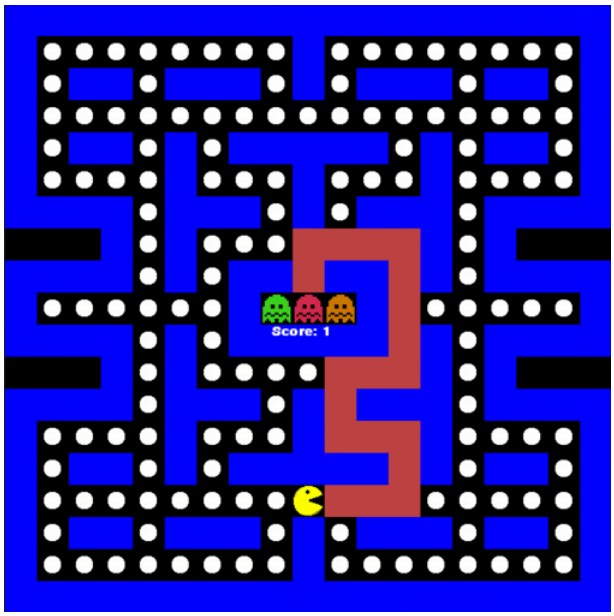


Figure 3: Optimal Path to PacMan

nodes that needed to be expanded to compute the intire path to PacMan was less than the defined limit, CDSA* yielded identical space and time complexity values as that of A*.Additionally, there was no movement that caused any of the ghosts to become lost and lose sight of where Pacman currently was.

## 5.1 Result Analysis

The results in Table 1 above suggests that Context Dependent Subgoaling A* is the most time and space efficient algorithm in our implementation of PacMan. When far from PacMan, this algorithm can determine the ghosts' next move roughly 3 times faster than the standard A* algorithm and 18 times faster than Breadth-First Search. This may not be too important in a game like PacMan where not very many moves need to be made in a short amount of time but if something required that if a thousand or even a million of these searches be done very quickly, then these differences in computation time could be very important.

Another important comparison between the three algorithms is space complexity. The size of the array containing the 'open' nodes can become very large and if the grid or playspace is increased, then this array can become even larger. For this reason it is important to consider just how much space an algorithm is able to use and when it is a good idea to try to optimize the space complexity of the algorithm. In Table 1 above, it can be seen that the Context Dependent Subgoaling A* is the most space efficient of the three algorithms as it generates

roughly 40% less nodes than standard A* and 79% less nodes than Breadth-First search. For PacMan on a standard machine, these space improvements may not be important. If, however, PacMan was to be used in an embedded system in a machine at the arcade, then the amount of space available may be limited in which case these improvements would be important. Also, if a larger PacMan game grid was desired, the number of potential nodes generated would increase in which case these improvements become more important. In regards to nodes expanded, Context Dependent Subgoaling A* expands approximately 39% less nodes than A* and 79% less nodes than Breadth-First search to be able to determine what its next move should be. This translates directly into computation time as the less nodes that need to be expanded means the quicker the algorithm can determine what its next move should be.

From Table 1 above, Context Dependent Subgoaling A* seems to be the obvious choice. With less nodes generated and expanded as well as faster computation time than both A* and Breadth-First search, its clearly the best algorithm to use in this implementation of PacMan. What if a larger grid was used? What would be the best algorithm to use for this grid? With the way Context Dependent Subgoaling A* works, using a larger grid and a starting position for PacMan that is further away would require changing the limit on the nodes expanded within this algorithm to ensure that this algorithm still finds the optimal path to PacMan at all times. If the limit was kept the same in this scenario, then the ghost would fairly often become 'lost' in the grid and it would move randomly in some section of the grid rather than move towards PacMan. This could be a desirable trait on larger grids as it would allow the ghost to 'protect' a part of the grid so that when PacMan enters this part of the grid the ghost would begin chasing PacMan from a direction that is different than other ghosts. If it is impossible to change the limit of CDSA* and optimal path finding is still desirable, then A* would be the best algorithm to use. This is the case because A* will always find the optimal path to PacMan and it does not care about how large the grid is because it will just keep expanding nodes until it finds the goal node. This allows it to be flexible and usable for all grid sizes provided the space available on the machine is enough to hold the nodes required for A* to function properly. There isn't a case in which Breadth-First search would be a better algorithm to use than A* (that is using a 'good' heuristic to improve searchign) in a game like PacMan.

# 6 Conclusion

Path finding algorithms are very important to the game industry as they are used in a variety of games. From a simple game like PacMan to a large-scale, interactive game like World of Warcraft, non player controlled agents must be able to find where they need to go for the game to function properly. In our implementation of PacMan, we experimented with three different path finding algorithms: A*, Context Dependent Subgoaling A*, and Breadth-First search. It was found that Context Dependent Subgoaling A* outperformed both A* and

BFS in both space and time complexity. CDSA* is dependent on the grid size of the PacMan game which means that it is not very flexible for varying grid sizes and starting positions unless you change the algorithm itself every time the grid size changes. If a flexible algorithm that is still somewhat space and time efficient is desirable, then A* would be the correct choice as it doesn't need to be modified when the grid size is increased and it also substantially outperforms BFS in both space and time complexity.

# References


[1] Xiao Cui and Hao Shi. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130, 1 2011.

[2] Jonathan Schaeffer Yngvi Bj ornsson Vadim Bulitko, Mitja Lustrek and Sverrir Sigmundarson. Dynamic control in real-time heuristic search. *Journal of Artificial Intelligence Research*, 32(1):419 – 452, 6 2008.

[3] Yngvi Bj ornsson Vadim Bulitko and Ramon Lawrence. Case-based subgoaling in real-time heuristic search for video game pathfinding. *Journal of Artificial Intelligence Research*, 39(1):269 – 300, 9 2010.

[4] Mohd Shahrizal Sunar Zeyad Abd Algfoor and Hoshang Kolivand. A comprehensive study on pathfinding techniques for robotics and video games. *International Journal of Computer Games Technology*, 2015(1):1–11, 3 2015.