# Bullet-Hell Automation using Artificial Intelligence

Chelsey Flores (Chjflor), Zachary Hall (Hallzj), Makena Smith(Maklsmit), Ashley Won (Aswon)

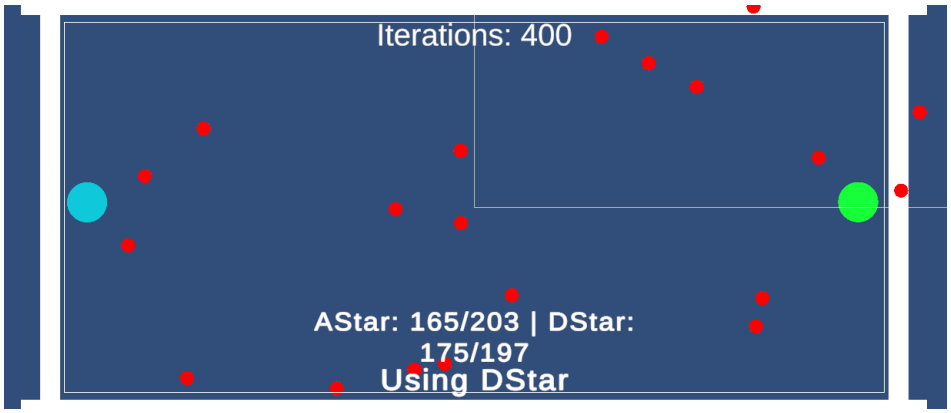
May 4, 2026

## 1 Introduction

Pathfinding plays a fundamental role in video games, and many attempts have been made throughout history to use search algorithms such as Dijkstra's algorithm, Greedy best-first Search, and A* to find the most efficient paths.

In machine learning, pathfinding generally refers to finding the shortest route between two end points. For our project, we wanted to investigate the efficiency of search algorithms in a dynamic environment, where the constraints are consistently changing with respect to time, and find if the changing constraints has an impact in the efficiency of the algorithms, and find which algorithm works the best in such cases.

For our case, we opted to use a bullet-hell game. A game in which a player attempts to complete a goal, in our case get from a start point to and end point, while numerous amounts of projectiles attempt to destroy the player. It has a similar goal to a traditional path finding problem, but we must account for the constraints that the bullets move, so the optimal path will change with respect to time. We have also recognized the efficiency of the D* algorithm in dynamic environments. In this project, we investigate the performance of the A* algorithm and the D* by implementing a simple bullet-hell style game that uses each algorithm as players to beat the game. We evaluate its efficiency by collecting various numerical measurements across different game parameters.




### 1.1 Project Description

We focused on creating a simple bullet-hell style game autoplayer using pathfinding algorithms, namely A* and D*, to play and beat the game and record the performances. The gameplay features the algorithm attempting to cross the screen while avoiding projectiles that fall from the top of the screen (figure 1).

#### 1.1.1 Game Rules

The spawn point of the player and position they must reach to "win" are static and do not change between iterations. The projectiles are randomly generated above and within the left and right bounds

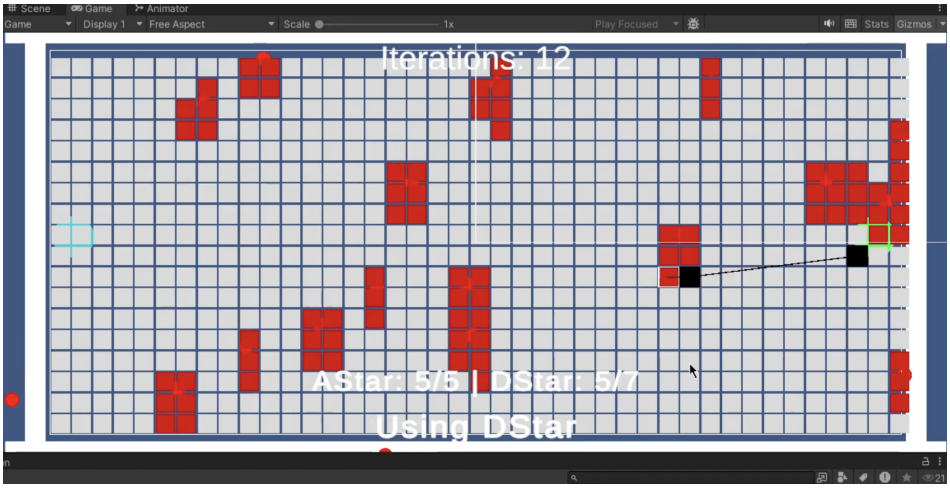



Figure 1: Grid UI Overlay

of the play space which then fall down to attempt to hit the player, and if the player is hit, they lose and the game resets, updating win/loss with each iteration. The goal of the algorithm is to find a path to the goal and make it there without getting hit under different sets of conditions. The conditions are the speed of the projectiles, with a base case of 20mps, and how often projectiles are spawned where 5 projectiles/second is the base case. For each iteration, the game randomly chooses between the A* player and D* player, and notes it at the bottom of the game interface.

### 1.1.2 Project Objectives

The main objective is to be able to test the performance of A* and D* to play and complete the game under different conditions and recording the results. The algorithm will be tested over a number of trials, recording us the number of wins and losses. In addition to this, UI has been included to show the grid or path dynamically updating as the player moves across the screen and the path gets interrupted (figure 2).

### 1.1.3 Project Constraints

The primary constraints of our demo come from the bullet-hell game itself. The player is restrained within the bounds of the created level, and must make it to the other side in order to count a completed run under the different assigned conditions of bullet speed and bullet spawn rate.

Another constraint is the computing power in the platform the game runs in. A* in particular has a high space complexity, and we observed that it runs slower for higher density gameplay.

### 1.1.4 Modeling Human Thought

The usage of informed search algorithms models the human thought of making an informed decision on what the best course of action could be in a given scenario. For our game, the algorithm is modeling human thought through dynamically updating its path or movement based on the proximity of the projectiles.

## 1.2 A* Algorithm

A* is an informed search algorithm to find the shortest path using a heuristic as a weight to determine which node to explore next. It extends Dijkstra's algorithm by introducing a heuristic approach. Common heuristics for A* include using Euclidean Distance or Manhattan distance. By using the heuristic, we reduce the number of nodes to be searched, and ideally decrease the search time to the goal.

$$f(s) = g(s) + h(s)$$

Where $g(s)$ is found cost to reach the current node from the start and $h(s)$ estimated cost from the current node to the goal, this is using octile distance (grid distance).

Some properties of A* is that, first, A* is guaranteed to find a path from start to goal if there exists a path. Second, if the heuristic of A* is admissible, A* is guaranteed to find an optimal solution. Third, A* makes the most efficient use of the heuristic. That is, no search method which uses the same heuristic function to find an optimal path examines fewer nodes than A*.

The time complexity for A* is highly dependent on its heuristic. The space complexity for A* is $O(b^d)$ where $d$ is the depth of the shallowest solution, and $b$ is the branching factor.

Considerable effort has been made to optimize this algorithm over the past decades and dozens of revised algorithms have been introduced successfully. Examples of such optimizations include improving heuristic methods, optimizing map representations, introducing new data structures, reducing memory requirements, and simplifying the problem and approximating.

```
function reconstruct_path(came_from, current)
    total_path := {current}
    while current in came_from.keys:
        current := came_from[current]
        total_path.prepend(current)
    return total_path

// A* finds a path from start to goal.
// h is the heuristic function. h(n) estimates the cost to reach goal from node n.
function a_star(start, goal, h)
    // The set of discovered nodes that may need to be (re-)expanded.
    // Initially, only the start node is known.
    // This is usually implemented as a min-heap or priority queue rather than a hash-set.
    open_set := {start}

    // For node n, came_from[n] is the node immediately preceding it on the cheapest path from
the start
    // to n currently known.
    came_from := an empty map

    // For node n, g_score[n] is the currently known cost of the cheapest path from start to
n.
    g_score := map with default value of Infinity
    g_score[start] := 0

    // For node n, f_score[n] := g_score[n] + h(n). f_score[n] represents our current best
guess as to
    // how cheap a path could be from start to finish if it goes through n.
    f_score := map with default value of Infinity
    f_score[start] := h(start)

    while open_set is not empty
        // This operation can occur in O(Log(N)) time if open_set is a min-heap or a priority
queue
        current := the node in open_set having the lowest f_score[] value
        if current = goal
            return reconstruct_path(came_from, current)

        open_set.remove(current)
        for each neighbor of current
            // d(current,neighbor) is the weight of the edge from current to neighbor
            // tentative_g_score is the distance from start to the neighbor through current
            tentative_g_score := g_score[current] + d(current, neighbor)
            if tentative_g_score < g_score[neighbor]
                // This path to neighbor is better than any previous one. Record it!
                came_from[neighbor] := current
                g_score[neighbor] := tentative_g_score
                f_score[neighbor] := tentative_g_score + h(neighbor)
                if neighbor not in open_set
                    open_set.add(neighbor)

    // Open set is empty but goal was never reached
    return failure
```

### 1.3 D* Algorithm

D* is an informed search algorithm similar to A*, but starts from the goal and searches for the start. For each node of the graph, its cost is tracked and any changes to it, whether it's raised or lowered, are then propegated to its neighboring nodes towards the start.

A property of D* is that for each run of the algorithm, only nodes that have been changed are reevaluated. This makes it ideal for dynamic graphs since it allows for only parts of the graph updated as needed.

The time complexity of D* is $O(n)$ with $n$ being the number of nodes in the worst case, but as mentioned previously since it only updates nodes as needed, so it can be better depending on the situation of the game.

The two heuristics that D* works under are g-value and rhs. The g-value of any given node is the the current best known node to take from this position. This is known since we are pathing from end to start. RHS looks at looking ahead 1 step on what the next best node is after this on. The algorithm is constantly comparing these two values of the node we are looking to move to. In the case that the g-value and rhs values are the same, we move to said next node, if they are not, we recheck the node and rebuild the path.

### 1.4 Dodging Algorithm

The dodging algorithm connected with both A* and D* is built on the idea that as long as the player moves away from nearby projectiles. The player object has a collider that checks for collisions with projectiles. This range is greater than the collision box of the player so that it can detect projectiles before getting hit. Once a projectile enters the outer collider, the player will attempt to move away from it by getting the normal vector of the collision, essentially returning a vector pointing in the opposite direction of the projectile relative to the player. Ex: if the player is moving right, and the projectile is in front of it, the normal vector will be to the left. Once the player has this direction, they will attempt to move in said direction until the projectile is outside of detection range. If multiple projectiles are in the detection range, both normal vectors will be taken into account.

## 2 Research Methods

The game was created and tested in Unity version 2022.3.36f1 using C#, and the figures were created with Matplotlib library in Python. Each trial consisted of an individual iteration of the game. Every iteration one of the two algorithms, A* and D*, were chosen randomly for simplicity of implementation. The successful runs for each algorithm are tracked against the total runs. This gives us a percent success rate for each algorithm.

## 3 Results

From our trials and the results, we have concluded that A* has a higher error rate of 18 percent for both the base gameplay and the higher density gameplay. However, the error rate of D* has increased for higher density case, 11 percent vs 16.6 percent. These values show that D* seems to be more effective in this environment which intuitively makes sense based upon the idea that D* should perform better in a dynamic environment where the grid is constantly updating due to the projectiles

### 3.1 Limitations

The primary limitation of our method is having a limited amount of testing runs, in particular for the higher density gameplay. More trials for both base and higher density cases would yield more accurate failure rate results, and very well could lead to differing results. Along with this the algorithm used for dodging could possibly be further improved. There would be times where a projectile would fall

```
while openList is not empty:
    // pick node with lowest k value
    point = openList.getFirst()
    expand(point)
void expand(currentPoint):
    // check if cost increased
    boolean isRaise = isRaise(currentPoint)
    double cost
    for each neighbor in currentPoint.getNeighbors():
        if isRaise:
            // neighbor depends on this path
            if neighbor.nextPoint == currentPoint:
                neighbor.setNextPointAndUpdateCost(currentPoint)
                openList.add(neighbor)
            else:
                // check if neighbor offers a cheaper alternative
                cost = neighbor.calculateCostVia(currentPoint)
                if cost < neighbor.getCost():
                    currentPoint.setMinimumCostToCurrentCost()
                    openList.add(currentPoint)
        else:
            // cost unchanged or lower
            cost = neighbor.calculateCostVia(currentPoint)
            // found a better path to neighbor
            if cost < neighbor.getCost():
                neighbor.setNextPointAndUpdateCost(currentPoint)
                openList.add(neighbor)
// detects if a node needs repair due to increased cost
boolean isRaise(point):
    double cost
    // cost has increased above the previous minimum
    if point.getCurrentCost() > point.getMinimumCost():
        for each neighbor in point.getNeighbors():
            // try to find a cheaper route via this neighbor
            cost = point.calculateCostVia(neighbor)
            if cost < point.getCurrentCost():
                point.setNextPointAndUpdateCost(neighbor)
    // true if still in raise state after checking neighbors
    return point.getCurrentCost() > point.getMinimumCost()
```

Figure 2: D* psudocode

on the player directly from above and due to the algorithm the bot would not be able to know that dodging horizontally is the best course of action as the algorithm is built on dodging in the opposite direction of where the projectile is relative to the player.

It is also important to note that we only measured the failure rate for each algorithm. We have not considered computational power used for our trials, which is a major factor in evaluating an algorithm.

### 3.2 Further Research and Improvements

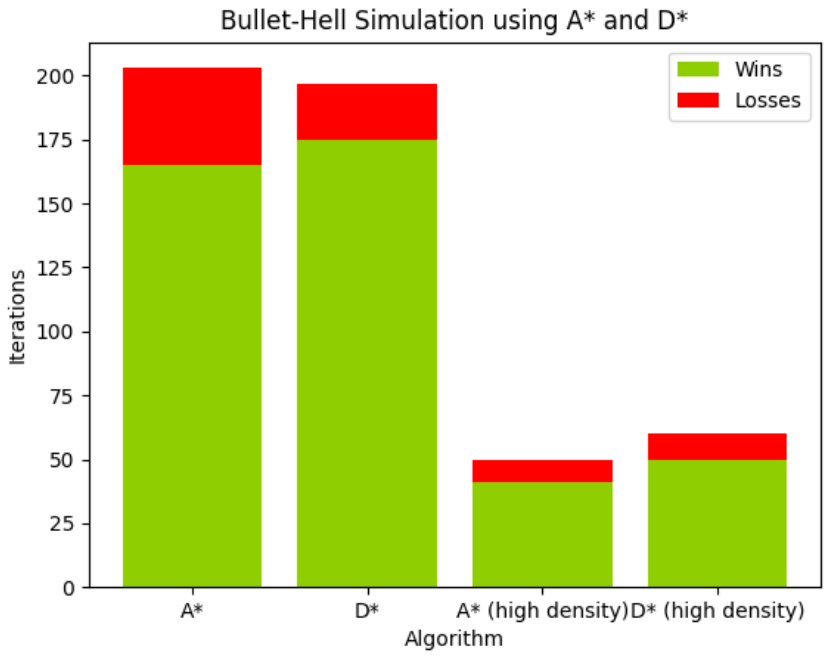
For future work, we would test our algorithms for a higher number of trials, and across more diversified gameplay, such as different bullet size, movement patterns, and speed.

## 4 Data and Analysis

We conducted about 200 trials for A* and D* algorithms for our base game. In addition, we conducted over 50 trials for our game at higher density. Then, we found the error rate for each algorithm at different game difficulty classes.

### 4.1 Results

Using base gameplay, A* showed a 18.7 percent error rate across 203 trials, and D* showed 11.2 percent error rate across 197 trials. With higher density gameplay, A* showed 18 percent error rate across 50 trials. D* showed 16.67 percent error rate across 60 trials.




## References

[Cui(2011)] Cui (2011) A*-based Pathfinding in Modern Computer Games IJCSNS International Journal of Computer Science and Network Security, VOL.11 No.1 https://www.researchgate.net/profile/Xiao-Cui-12/publication/267809499_A-based_Pathfinding_in_Modern_Computer_Games/links/54fd73740cf270426d125adc/A-based-Pathfinding-in-Modern-Computer-Games.pdf

[Lague (2017)] Lague (2017) A* Pathfinding Tutorial (Unity) https://youtube.com/playlist?list=PLFt_AvWsXl0cq5Umv3pMC9SPnKjfp9eGW&si=m9bULo1nDZ4JRoYE

[Wikipedia(2026)] Wikipedia contributors (2026) D* *Wikipedia, The Free Encyclopedia* https://en.wikipedia.org/wiki/D*$0