Zero Robotics Competition Checkpoint Code Writeup

Arav Chand, Adittya Nischal, Krishang Gupta, Disha Chakraborty, Shohom Chakraborty, Lavanya Purwar, Ray Tang

January 24, 2025

Mentors:
Dr. Anurag Purwar, Mr. Arthur Kalish

Affiliations:
Stony Brook University, Institute of Creative Problem Solving

Abstract

This is the checkpoint code write-up document for the ICPS team in the MIT Zero Robotics Competition. It contains documentation on our code development process, algorithms, and debugging process, as well as our plans for the rest of the competition. Please note that this document does not contain extensive summaries of the game manual. The reader is expected to have a basic understanding of the game in advance.

Acknowledgements

We express our sincerest gratitude to the Institute of Creative Problem Solving and Dr. Arthur Kalish for his ongoing support for our team. We also thank Dr. Anurag Purwar, whose guidance and expertise has helped us immensely through the competition.

Contents

1 Objective and Strategy

This section introduces the problem, provides background information, and shows our initial strategy discussions.

1.1 Initial Discussion

Immediately, we noticed that there were three main constraints:

- Time limit
- Battery usage limit
- Debris variables

All these constraints are accounted for in the score function, which combines them as follows:

$$\text{Total Score} = 0.8 \cdot T + 1.0 \cdot B + 1.2 \cdot \text{ImQ} - \text{Penalties}$$

Time Score (T): $T = 180 - \text{time used in seconds}$

Battery Score (B): $B = 100 - \sum(12 + m) \cdot D$

Image Quality (ImQ): Based on num obj rmv from remaining in the imaging field.

However, we note that to maximize the score, some metrics should be increased and some decreased. For example, we want to increase the number of debris that we remove, but decrease our time used. It would be helpful if we had a metric to calculate this, such that we can compare the relative efficiency of different paths when we are writing algorithms.

From this we can create the reward function, which is defined as:

$$\text{Reward} = \frac{\text{ImQ}_{\text{num obj removed from field}}}{\Sigma[(12 + m)D] + T_{\text{time used (in seconds)}}}$$

Note how the reward increases when ImQ increases and also increases when distance and time decrease.

Thus, our initial strategy will be to maximize the reward across all paths that our algorithm will produce, hence maximizing our points.

2 Key Algorithms and Logic

This section describes our algorithms. It includes our thought processes that led to our algorithms, as well as explanations for how they work and their advantages and disadvantages.

The following are some of our initial algorithm ideas.

Algorithm 1 Simple Greedy Algorithm

```
Initialize environment
while $game$ is not over do
    for $smallDebris$ in $alldebris$ do
        if $smallDebris < minDebris$ then
            $minDebris \leftarrow smallDebris$
        end if
    end for
    Go to closest debris $minDebris$
    Pick up $minDebris$
    Find closest path to edge
    Drop off $minDebris$
    Move to second closest debris
    if $field$ does not contain small debris then
        Remove all large debris and special debris
    end if
end while
```

2.1 Greedy Algorithm

One of our earliest ideas was a greedy algorithm. This would take the closest debris at each step and move it outside of the imaging zone, minimizing the time and distance. This is by far the simplest algorithm out of the one shown. Below is the generalization:

The above algorithm is a generalization, it does not go into detail about parts such as the algorithm to find the most efficient path (this will be explored in the next subsection).

The above algorithm is very basic and has a lot of pitfalls; for one, it is not the most efficient; it does not get the most points in the least amount of time. It can be shown that, to do this, an algorithm must prioritize removing the larger debris, as they are worth more points. This algorithm prioritizes getting the small debris out first, then goes back and gets the larger debris.

2.2 Directed Graph Algorithm

Another approach we explored involved constructing a directed graph representation of all the debris. In this algorithm, each piece of debris is represented as a node in a graph. Directed edges are created between nodes that obstruct each other in the horizontal or vertical directions. This structure provides a systematic way to determine which debris pieces must be cleared first to allow access to others.

The algorithm above integrates a directed graph to prioritize debris extraction based on dependency relationships. A "dead-end" node, defined as a node with no outgoing edges, represents a piece of debris that can be cleared immediately without any obstructions. Once extracted, the graph is updated to reflect the new environment, dynamically adjusting the dependencies.

This approach improves upon the simple greedy algorithm by considering adjacency and dependency. For example, if a piece of large debris blocks access to smaller debris, this method ensures that the blocking debris is removed first, allowing efficient clearing.

However, this algorithm comes with additional computational complexity. Constructing and maintaining the directed graph in real time introduces overhead, especially in scenarios with large numbers of debris nodes. Despite this, the directed graph algorithm provides a structured and systematic approach to address the inefficiencies of simpler methods.

Algorithm 2 Directed Graph Algorithm

```
Construct directed graph $G$ with debris as nodes
for each $node_i$ in $G$ do
    for each $node_j$ blocking $node_i$ (horizontally or vertically) do
        Add directed edge $node_i \to node_j$ in $G$
    end for
end for
while $game$ is not over do
    Identify all dead-end nodes (nodes with no outgoing edges)
    for each dead-end node $d$ do
        Move to $d$ and extract
        Remove $d$ from the graph
        Update graph by recalculating edges
    end for
end while
```

2.3 Modified Dijkstra's Algorithm

A more advanced approach involves utilizing a modified version of Dijkstra's Algorithm to optimize debris removal. This method guarantees a globally optimal solution by minimizing a total cost function, unlike the greedy approach, which may get stuck in local optima. Below is the general process for this algorithm:

Algorithm 3 Modified Dijkstra's Algorithm

```
Initialize priority queue $Q$ with elements of the form $(TotalCost, CurrentPosition, DebrisRemoved)$
Add initial state $(0, StartPosition, \emptyset)$ to $Q$
while $Q$ is not empty do
    Pop element $(TotalCost, CurrentPosition, DebrisRemoved)$ with the smallest cost from $Q$
    if all debris is removed then
        Terminate and output $TotalCost$
    end if
    for each $debris$ reachable from $CurrentPosition$ do
        Calculate $RemovalCost$ using debris score and total battery consumption
        Compute $NewCost = TotalCost + RemovalCost$
        Update $NewPosition$ to the position after removing $debris$
        Add $debris$ to $DebrisRemoved$
        Push $(NewCost, NewPosition, DebrisRemoved)$ to $Q$
    end for
    Sort $Q$ by $TotalCost$ in ascending order
end while
```

The above algorithm works by exploring all possible paths and calculating their respective costs. A priority queue is used so that the paths with the smallest costs are prioritized.

This approach improves upon the greedy algorithm by considering the global optimal solution, rather than evaluating only immediate returns. Each debris piece is evaluated using a cost function based on its score relative to the battery consumption required to retrieve and remove it. The battery consumption accounts for these two distances:

- The distance traveled to reach the debris (without additional mass)
- The distance traveled to remove it (with the additional mass of the debris)

Although computationally more expensive than the greedy approach, the modified Dijkstra's Algorithm is better for more complicated field configurations and ensures a global optimal solution.

2.4 Optimal Remove-Return Geometry

All of these algorithms involve this repetitive process:

- Getting to the first debris
- Picking up the first debris
- Flying to a location outside the imaging zone
- Dropping the debris
- Flying to the second debris and picking it up

It can be shown that there is an optimal geometry for this process, which will be discussed below.

Consider the following example, where we are near a vertical section of the border of the imaging area. We will discuss this with vertical sections first, then we will extend it into horizontal sections. As we will see, the mathematics is simple for both of them, but it is easier to explain when we consider vertical sections first.

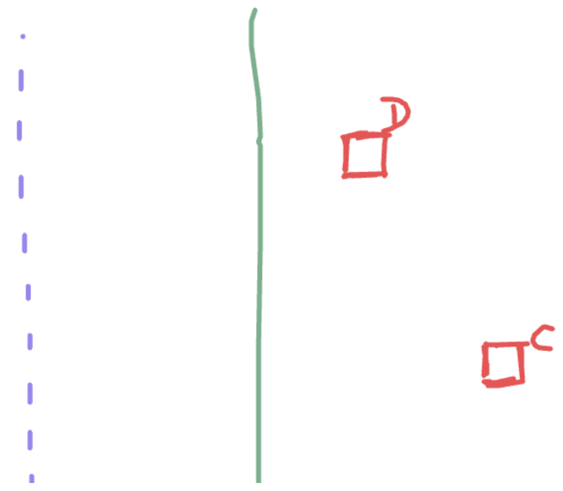

Figure 1: Two debris.

Let Debris C be the current location of the Astrobee, and debris D be the next debris to be collected. What is the optimal path geometry to go outside the imaging zone, then to debris D? It can be seen that it will follow some V-shape, with a point that "bounces" off the border of the imaging zone. All we need is to find this point.

The method to get the optimal geometry is simple. Draw line $\overline{D'C}$. Notice that this line will always intersect one point on the border of the imaging zone.

We will denote this intersection point as B. Draw line $overline BD$. Now, lines $\overline{D'C}$ and $\overline{BD}$ make up the optimal path.

The proof is relatively straightforward and will be summarized here rather than be written in a formal matter because the author does not have time. Note that when we reflect D to get D', the line $\overline{D'C}$ is the shortest path between D' and C. When we reflect $\overline{D'B}$ over to get $\overline{BD}$, the length of this line does not change because reflections are rigid geometrical transformations; they do not scale objects.

Next, we need to determine the intersection between the line $overline D'C$ and the border of the imaging zone. Because the border is a square, we could be working with a vertical or horizontal section. This is why we are not able

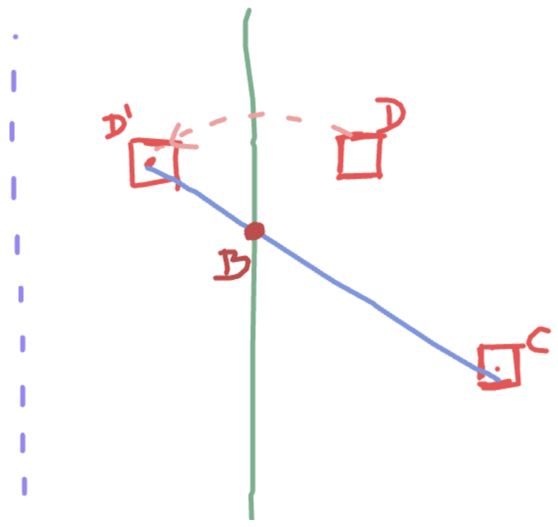

Figure 2: The line is drawn and point B is marked.

to simply use $y = mx + b$ to solve for the intersection; vertical lines have undefined slope. However, there is another option.

Let us define a perpendicular line from C to the border as Δx, and then the line from the intersection of Δx and the border to B as Δy. See Figure 3 below.

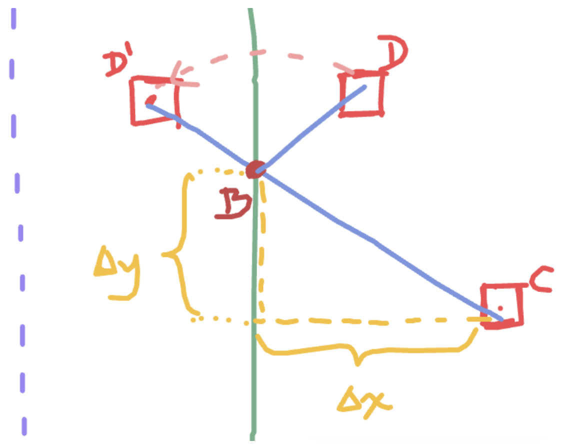

Figure 3: Δx and Δy are shown.

Recall that if we have the slope of a line, we can get the y-coordinate by multiplying the slope by the x-coordinate, since slope is

$$\frac{\Delta y}{\Delta x}$$

The slope of this line can be defined as:

$$m_{\overline{D'C}} = \frac{D'_y - C_y}{D'_x - C_x}$$

Therefore,

$$\Delta y = m_{\overline{D'C}} \cdot \Delta x$$

and we can derive the location of point B.

2.4.1 Horizontal Lines

The mathematics is very similar for horizontal lines, we just need to rearrange the equation to solve for Δx instead.

$$\Delta x = \frac{\Delta y}{m_{\overline{D'C}}}$$

3 Code Structure

Due to the limited time to complete something before the checkpoint code due date, we decided to go with a simplified version of the greedy algorithm.

We first wrote many helper functions to perform subtasks in the algorithm. The subsections below outline them.

3.1 GetCurrentLocation and GetDebrisLocations

These two functions return arrays of coordinates. GetCurrentLocation returns an array of the x and y coordinates of the Astrobee, and GetDebrisLocations returns a 2D array of all of the coordinates of the debris, with each debris' number corresponding to the index of the sub-array in the array.

3.2 GetDistance

A very simple function that returns the Euclidian distance between two points. This is used in other functions, such as GetClosestDebris.

3.3 GetClosestDebris

A function to return the index of the closest debris to the Astrobee. The function does this by iterating through the list of debris, using the GetDistance function to get the distance from the Astrobee to each debris, and then finding the minimum.

The GetSecondClosestDebris function is similar, it just returns the second closest debris instead.

3.4 GetClosestWall

This function returns the closest wall to the Astrobee by comparing the lengths of perpendicular lines from the Astrobee to each of the four walls.

3.5 Get Path Geometry "Bounce" Point

This function calculates the coordinates of the "bounce" point discussed in section 2.4.

4 External Tool and Resource Use Statement

4.1 Use of GitHub/Git and Local Development Systems

We wrote a C++ header file for use in Visual Studio Code for easier development, as VS Code offers IntelliSense and autocomplete. The header file included the basic API functions given to us on the global game object in the IDE with no function definitions.

We also wrote a small library of utility functions and data abstractions for quickly and easily doing basic tasks, like getting distance to a debris object or finding the nearest debris object.

A GitHub repository (currently private) was created to store our code, including the header file and library.

Algorithm 4 Get Closest Debris

```
Initialize currentLocation[2] with robot's current position:
  currentLocation[0] = GetRobotPositionX()
  currentLocation[1] = GetRobotPositionY()
Debug: curreloc currentLocation[0], currentLocation[1]
Initialize pointer debrisArray to debris locations using getDebrisLocations()
if debrisArray is NULL then
  Print error: "Error:  debrisArray is null"
  return -1
end if
Initialize minDistance = 30.0f and minDistanceDebris = -1
for debrisIndex from 0 to 14 do
  if droppedOffDebris[debrisIndex] is true then
    Continue to the next debris
  end if
  Calculate distance = getDistance(currentLocation, debrisArray[debrisIndex])

  Debug: "Debris debrisIndex:  Distance = distance"
  if distance < minDistance then
    Update minDistance = distance
    Update minDistanceDebris = debrisIndex
  end if
end for
if minDistanceDebris == -1 then
  Debug: "Error:  No debris found within the minimum distance"
else
  Debug:            "Closest debris index:  minDistanceDebris, Distance:
  minDistance"
end if
return minDistanceDebris
```

Algorithm 5 Get Closest Wall

```
Initialize currentLocation with robot's x and y position
Define distances as an array of distances to the four walls:
  distances[0] = Distance to top wall
  distances[1] = Distance to right wall
  distances[2] = Distance to bottom wall
  distances[3] = Distance to left wall
Initialize minDistance as a large value
Initialize minDistanceWall as 0
for wallIndex = 0 to 3 do
  if distances[wallIndex] < minDistance then
    Update minDistance with distances[wallIndex]
    Update minDistanceWall with wallIndex
  end if
end for
return minDistanceWall
```

Algorithm 6 Get Path Point

```
Initialize currentLocation with robot's x and y position
closestWall ← getClosestWall()
Initialize dPrime as the reflected destination point based on closestWall:
if closestWall is 0 (top wall) then
    Reflect destinationPoint vertically across the top wall
else if closestWall is 1 (right wall) then
    Reflect destinationPoint horizontally across the right wall
else if closestWall is 2 (bottom wall) then
    Reflect destinationPoint vertically across the bottom wall
else if closestWall is 3 (left wall) then
    Reflect destinationPoint horizontally across the left wall
end if
Compute slopeOfLine from currentLocation to dPrime
Initialize pathPoint as the intersection of the line with closestWall:
if closestWall is 0 (top wall) then
    Compute intersection of the line with the top wall
else if closestWall is 1 (right wall) then
    Compute intersection of the line with the right wall
else if closestWall is 2 (bottom wall) then
    Compute intersection of the line with the bottom wall
else if closestWall is 3 (left wall) then
    Compute intersection of the line with the left wall
end if
return pathPoint
```

4.2 Artificial Intelligence Usage Statement

Our team recognizes that AI has unparalleled potential, but heavily relying on AI will have negative effects. We understand that relying too heavily on artificial intelligence will leave the student without really understanding or learning of the concept at hand and will always be detrimental in the long run. However, AI has its merits - it can read faster, it has more knowledge that us, and it's blazingly fast. This is why we still used ChatGPT to help with certain tasks.

When we used ChatGPT, it would be for the following reasons:

- To clarify a question regarding code syntax
- To help debug a piece of code (only if the author of the code has tried manually debugging it before and had no success)
- To explain pointers and data structures unique to C++, as many of our members have experience with other languages (e.g., Rust, C#, etc.) but not C++
- To help automatically "clean up" pieces of code (adjust formatting such as indentation so it is easier to read, not actually changing the code itself)

Note that humans are able to execute all these tasks as well; however, the usage of ChatGPT in these instances speeds up the process quite a bit. If a group of our team members was stuck on trying to debug a piece of code for too long, we would ask ChatGPT to help us.

As mentioned above, we do not approve of the use of AI and ChatGPT to completely solve our problems. Below are some examples of tasks we did not use ChatGPT for.

- To create an entire algorithm for the competition
- To write parts of this code write-up
- To write code that implement algorithms for us

5 Testing Iterations and Debugging

We tested the code using the online IDE and went through many iterations. To debug, we mainly used the DEBUG() function and the printf() function.

6 Challenges and Future Improvements

Currently, our code does not handle collisions or failed movements (i.e., a movement function returned false) properly. This will be a vital and necessary feature in future iterations. Furthermore, our algorithm only handles the small debris - we hope to expand it to eventually handle the large and special debris in the future.