Assignment 1
SC627

Bhavini Jeloka
18D100007

I. Introduction to Bug 1

The flowchart given below summarizes the bug 1 algorithm. From the figure it is clear that the algorithm is capable of avoiding obstacles in order to reach the goal as long as such a path exists.

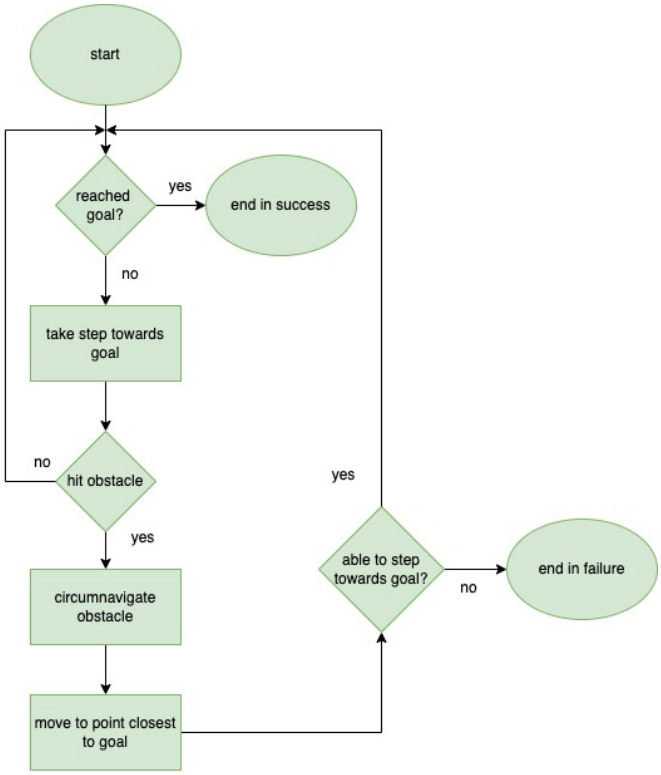

Fig. 1. Bug 1

II. BugBase vs. Bug 1

The BugBase algorithm as mentioned in the text does not have any logic for avoiding obstacles and collision. As a result, the appearance of such an obstacle will immediately result in failure as seen in Figure 2 (where the goal is marked by the point in blue), with the environment described in Figure 3. Naturally, such an algorithm is inefficient and we need an improvement.

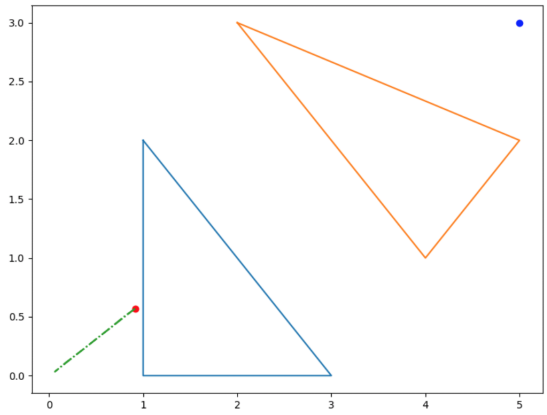

Fig. 2. BugBase does not perform well in the presence of obstacles

Under the purview of the Bug 1 algorithm introduced earlier, the bot tries to look for a point on the obstacle boundary that is closer to the goal and does not offer any resistance. In order to achieve this ambition, it must first explore the entire obstacle and then move towards the point that is closest to the goal and takes a step. If this step offers no resistance, then the algorithm can move forward. In a contrasting scenario, it results in failure.

A. Algorithm Initiation

We use the functions in the helper code to implement the logic explained above. When the algorithm is initiated it takes as its input, the obstacles, the goal and start point. Then, the distance of the bot from the goal is calculated and if it is greater than the step size, the bot is expected to move. At each instant, the distance of the bot from the obstacles is calculated to ensure that the bot is still in the free configuration space. In order to calculate these distances, we invoke the functions built earlier.

B. Dealing with Obstacles

This is where Bug 1 deviates from BugBase. If the bot nears an obstacle, we compute the tangent to the obstacle to provide a direction for circumnavigation. For this as well, we have written a function. We want the bot to completely to around the obstacle and come back to the starting point. To achieve this, we exploit the angle between the vector representing the centroid and the current position and the vector from the centroid to the start position. We run the circumnavigation loop till these vectors have an ϵ (tolerance) angle between them. One might wonder, will not the bot start with such an ϵ angle? This might prevent circumnavigation from taking place at all. To tackle this, we impose this condition only when the bot is reasonably away from the hit point. In addition to this, we also store the distance from the goal at each point during circumnavigation.

Once circumnavigation finishes, we recall from memory, the point closest towards the goal and move towards it. Once again, we catch up to the point by the centroid method where we now use the vector between the desired point (in place of the hit point) and the centroid. This becomes are leave point and we move in the direction of the goal. The algorithm ends in failure if this step leads us closer to the object. If it moves away from the goal, we have successfully found a method to overcome the obstacle.

C. Termination

The algorithm terminates if the distance between the goal and the bot is within a tolerance limit. Due to numerical errors, we cannot impose an exact matching of the goal and the bot, hence such a tolerance becomes necessary.

III. Numerical Results

We test our algorithm in a 2D environment with two obstacles shown in Figure 3. Some key features of the successful run are given below along with plots and visualisations.

1) Time taken for the algorithm to run: 1140 seconds (19 minutes)
2) Total path length travelled: 26.57 units

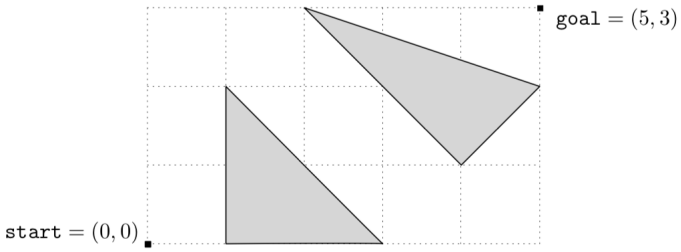

Fig. 3. Environment set up

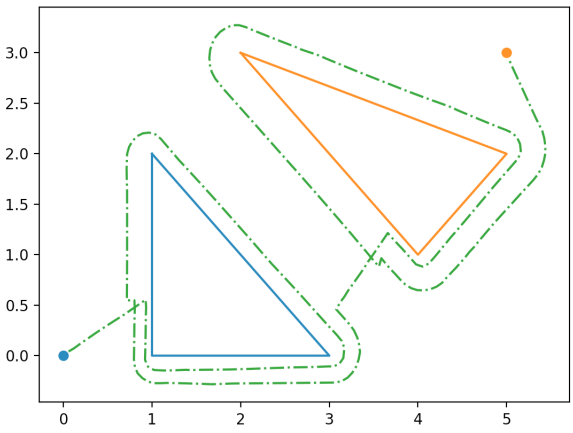

Fig. 4. Bug 1 successfully reaches the goal

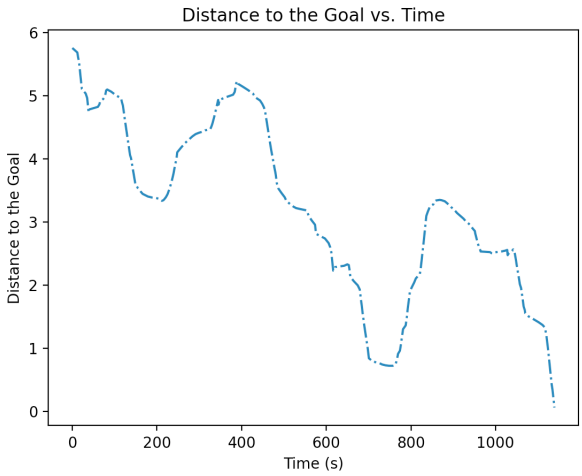

Fig. 5. Distance remaining to the goal as a function of time (Bug 1)