2018 Annual American Control Conference (ACC)
June 27–29, 2018. Wisconsin Center, Milwaukee, USA

A virtual bug planning technique for 2D robot path planning

Nishant Sharma, Shivam Thukral, Sandip Aine, and P.B. Sujit

Abstract— **We present a path planning technique inspired by the bug algorithm to quickly compute paths in an obstacle rich environment (or to report that no such path exists). In our approach, we simulate virtual bugs that upon sensing an obstacle splits into two bugs exploring the obstacle boundary in opposite directions, until the bugs find the goal in the line-of-sight. Then the bug leaves the obstacle and proceeds towards the goal. The process of splitting a bug into two continues until all the bugs reach the goal. The algorithm is simple to implement and it rapidly finds a solution if one exists. We provide worst case bounds on the path length with provable guarantees on convergence and develop heuristics to minimize the number of active bugs in the environment. We compare the performance of our algorithm with different planners from Open Motion Planning Library (OMPL) and visibility graph methods. The results show that the proposed algorithm delivers lower cost paths compared to other planners with lower computational time and rapidly indicates if a path does not exist.**

I. Introduction

Path planning is an essential component for a robot to traverse autonomously in an environment filled with obstacles. Given a source and a goal location, the objective for the robot is to find a (near) minimum length path which is collision free. Finding an optimal path is computationally intensive [1], and hence there is a need to develop path planning algorithms that can determine paths having near-optimal performance with low computational complexity, especially, for real-time robotic applications. Ideally, a real-time path planning algorithm should i) be very fast, (ii) have some bounds on the solution quality, and (iii) ensure determinism in the solution. Over the years, several types of path planning algorithms have been developed that perform well in different scenarios addressing (i), (ii) and (iii).

One of the earliest path planners are sensor-based planners that uses a tactile sensor or range sensor to sense the presence of an obstacle. The Bug1 and Bug2 algorithms [2] are the simplest sensor based path planners with provable guarantees on the performance of the algorithm. However, their paths can be highly sub-optimal, as these algorithms do not explore the space systematically to make informed planning. There have been several improvements based on this initial work over past two decades [3]–[8], however, the sub-optimality issue remains.

Alternatively, heuristic search-based algorithms can be used that provide theoretical guarantees like completeness/optimality and does not rely on randomization. The classical planning techniques are based on A* [9], which is a provably optimal algorithm. There are several variants of A*, like D* [10], Anytime A* [11], Theta* [12], and Multi-heuristic A* [13], that improves upon A*. However, the main issue with these planners is that the environment needs to be discretized beforehand which is i) tedious, and ii) may exclude solutions.

On the other hand, sampling based algorithms adopt an orthogonal approach. They randomly sample the environment and construct an exploration tree to find a path from the source to the goal [14]–[16]. There have been some recent efforts to add deterministic behavior for sampling-based algorithms [17], [18].

Another set of planners are based on generating a visibility graph (VG) of the region, and determining a shortest path on VG. The VG provides a road-map of the search space and the shortest path algorithm gives the optimal path on that road-map. However, with increase in number of obstacles, the computational effort required to create the VG increases [19], [20].

A simple bug algorithm uses (a) goal seeking behavior – moving towards the goal until an obstacle is detected, and (b) obstacle follow behavior – follow the obstacle boundary until the goal is in line-of-sight (LOS), as the basic behavioral primitives. These primitives can be used in tandem until the bug reaches the goal. The main issue with this simple approach is that i) a bug may select a longer route and ii) it may need to explore the complete obstacle boundary before leaving the obstacle. These deficiencies occur because the bug is a robot which is exploring an unknown environment to find a path to the goal. However, we are interested in using the simple bug based algorithms for known environments and hence we can mitigate the above issues caused by the original bug algorithm by using virtual bugs.

A virtual bug splits into two virtual bugs upon detection of an obstacle. These two bugs explore the obstacle in opposite directions, and hence discover the shorter route to the goal, thereby mitigating the route selection problem. Once the goal is detected by a bug in the line-of-sight, it travels towards the goal. Along its way, if it intersects another obstacle, then the bug will again split into two. Since each bug splits itself into two, the exploration automatically forms a binary tree structure. This process of moving towards the goal, splitting into two bugs and following the boundary continues until all the bugs reach the goal. Since the bugs flood an obstacle rich environment to find paths, we call the algorithm BugFlood. An interesting property of this planner is that it can quickly inform whether a path exists or not in a given environment.

The main contributions of this paper are: (i) we develop a fast path planning algorithm that generates near-optimal

Nishant Sharma is with University of Nebraska–Lincoln.
Shivam Thukral and Sandip Aine are with IIIT-Delhi.
P.B. Sujit is with IIIT-Delhi, New Delhi – 110020, India.
This work is partially funded by EPSRC project EP/P02839X/1.

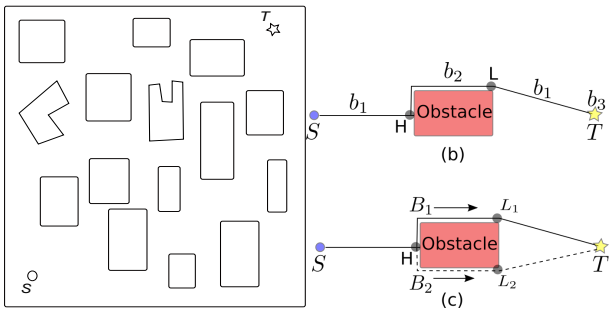


Fig. 1: (a) The S is the starting location of the robot, T is the goal, and the rectangles in the bounded region are the obstacles (b) Different behaviors of the bug from S to T. H is the hit point and L is the leave point (c) The bug splits into two bug B_1 and B_2 at the hit point H.

paths but with less time (ii) provide theoretical guarantees on the path length and completeness of the algorithm, (iii) quickly determine if a path does not exist, and (iv) evaluate against state-of-the-art algorithms from OMPL library for varying number of obstacle density. A preliminary idea of BugFlood without extensive validation and theoretical results was introduced in [22]. The rest of the paper is as follows. In Section II, we describe the problem. In Section III, we describe the BugFlood algorithm, analyze its properties in Section IV, and validate through simulations in Section V and the conclusions are presented in VI.

II. Problem Description

Consider a scenario as shown in Figure 1, where, a robot is located at start point S and needs to find a path to the goal T in the presence of obstacles whose information is known apriori. The obstacle space is denoted as $\mathcal{O}$ and there are n obstacles denoted as $O_i, i = 1, \ldots, n$. The obstacles can be of any shape and size. Each obstacle O_i is represented by a set of nodes $v_j \in V_i, j = 1, \ldots, |V_i|$. Let p_i be the perimeter of the i^{th} obstacle. The robot has a sensor that can detect the obstacles within range of r_s meters. We assume that the bugs have a shared memory to store and update information. We use a point mass model for the robot and its motion is modeled as a single integrator model,

$$\begin{aligned} \dot{x} &= u_x, \\ \dot{y} &= u_y, \end{aligned} \tag{1}$$

and hence we assume that the robot is a holonomic robot. Given the environment details, the robot capabilities and assumptions, the goal is to develop a near-optimal path planner that finds a path π quickly from S to T. An essential property of the planner must be to inform quickly if a path does not exist.

III. BugFlood Algorithm

The bug has three behaviors - (b_1) move towards goal in a straight line, (b_2) follow obstacle boundary, and (b_3) stop. We will describe these behaviors using Figure 1(b) and the following definitions.

Definition 1 (Hit Point): When a bug moves in a straight line towards the goal and comes in contact with an obstacle at point P, then we define the point P as a hit point H.

Definition 2 (Leave point): The point where the bug detects the goal in LOS and leaves the obstacle boundary towards the goal is defined as the leave point L.

Definition 3 (Obstacle follow behavior b_2): The robot follows the obstacle boundary from hit point H to leave point L.

Assume a bug starts from S and moves towards T as shown in Figure 1(b) (behavior b_1) and when it detects an obstacle within its sensing range, it creates a hit point H. If we follow, the $\mathrm{P}_{\mathrm{com}}$ algorithm [23], then the bug follows a default behavior, say turn left. Therefore, the bug will start following the obstacle boundary from the left side (behavior b_2), until it reaches a point where it can see the goal (leave point L). At this point, the behavior changes from b_2 to b_1 and the bug moves towards the goal. When the bug reaches the goal location, then the bug stops (behavior b_3). The behaviors b_1 and b_2 cycle until all the bugs reach the goal. The drawback of using $\mathrm{P}_{\mathrm{com}}$ is that the bug can enter into an infinite loop (as shown in Figure 2(a)) or meeting infinite number of obstacles. However, this algorithm has an interesting property of goal seeking behavior that we utlize in our paper.

A. Virtual bug splitting

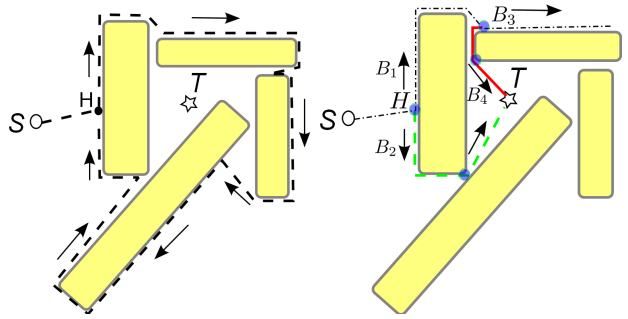
In the traditional bug-based algorithms, when the bug detects an obstacle, it uses a default strategy to always turn "right" or always "left". However, we use virtual bugs to explore both in the right side as well as on the left side of an obstacle, by splitting one bug into two bugs upon detection of an obstacle as shown in Figure 1(c). Due to this novel way of splitting one bug into two, we create a binary tree structure and it is easy to find which bug has reached the goal and trace its parents to find the route from T to S. In Figure 1(c), we can see that, at H, the bug splits into two bugs B_1 and B_2. The bug B_1 explores on the left of the obstacle, while B_2 explore on the right side of the obstacle. Once the bugs reach T, their paths are computed and the least cost path is selected using the bug splitting technique, the infinite loop scenario in Figure 2(a) is mitigated as shown in Figure 2(b).

B. Algorithm

We will now use this idea of bug splitting to explore the search space for finding potential solutions. The BugFlood algorithm is described in Algorithm 1, however, to explain the algorithm we need the following definitions.

Definition 4 (Path): We define a path $\pi = \{L_0, H_B, \hat{V}_i, L_{B'}, H_{B''}, \ldots, T\}, i \in \mathcal{O}$ as a combination of hit points, obstacle nodes and leave points.

The path between a hit point H_B created by bug B and leave point $L_{B'}$ created by bug B' (which was spawned by B) may consists of several nodes $\hat{V}_i \in V_i$ of obstacle O_i.



(a) (b)

Fig. 2: (a) The bug uses default strategy of moving left upon detecting an obstacle. (b) The bug splits into two bugs (B_1 and B_2) at the hit point H traveling in opposite directions exploring the obstacle.

The last component of the path is the goal T, while the first leave point L_0 is the start location S. In order to make the bug indexing simpler, we assume that all the bugs with odd index numbers will explore towards the left side of the obstacle from the hit point, while the bugs with even index number will explore on the right side of the obstacle. The active bugs in the environment are denoted as $\mathcal{A}$. If $\mathcal{A} = \emptyset$, then there are no active bugs in the environment.

Definition 5 (Bug Path): Let $\pi(B)$ represent the path of bug B from its current location to S. The path is obtained using **retrace path(B)** function in Algorithm 3.

Definition 6 (Bug Path at node v_i): Let $\pi(v_i, B)$ represent the path of bug B from v_i to S.

Definition 7 (Shortest Path): Let $\bar{\pi}(v_i)$ represent the shortest path from node v_i to the start location S and $\ell(\bar{\pi}(v_i))$ be the length of the shortest path from S to node v_i.

We will now explain the BugFlood algorithm given in Algorithm 1 using the scenario in Figure 3(a). The algorithm is carried out in three steps. In the first step, the bug moves from a leave point (S is also a leave point), to create a hit point or to reach the goal. In the second step, the bug moves from a hit point to create a leave point. In step three, we trace the bug paths that have reached the goal. Let $\mathcal{A}$ be the set of active bugs, and the bug starts from S towards T (Step 1). The first bug has the counter $k = 0$. The bug moves in a straight line towards T and intersects the obstacle creating a hit point H_0 as shown in the Figure 3(a). At H_0, bug B splits into two bugs B_1 and B_2 (line 3:b.(2)). The data structure Π is used for tracing the path back and hence the hit point and the previous bug are recorded. The bug counter k is updated to $k = k + 2$. Each bug $B \in \mathcal{A}$, moves towards the goal until it hits either an obstacle or reaches the goal. For the scenario in Figure 3(a), the Step 1 closes at H_0.

In the second step, the bugs in $\mathcal{H}$ ($\mathcal{H}$ is an array to hold the new bugs created after the bug creates a hit point) are used to find their respective leave points. Each bug, starts from the hit point and performs the obstacle follow behavior (Step 2, line 1-2) until T is in LOS. At this point, B is flagged as an active bug and added into $\mathcal{A}$ and removed from $\mathcal{H}$. After

Algorithm 1 BugFlood algorithm

Step 0: $k = 0$, bug B, $\mathcal{A} = \{k\}$, $\Pi(k).hitPoint = \{0\}$, $\Pi(k).leavePoint = \{S\}$, $\Pi(k).prevBug = \{\}$, $\mathcal{O}, S, T, \mathcal{H}$

Step 1:

1: **for** each bug $B \in \mathcal{A}$ **do**
2: Move towards T until
3: a. T is reached. $\Pi(B).hitPoint = T$, $\mathcal{B} \leftarrow B$,
- Remove B from $\mathcal{A}$, If $\mathcal{A} == \emptyset$ **then** Go to Step 3.

OR

b. (1) Obstacle O_i is detected, then generate a hit point H_B at (x_i, y_i),
- (2) $\Pi(B).hitPoint \leftarrow H_B$, $\mathcal{H} \leftarrow [k+1, k+2]$,
- (3) $\Pi(k+1).hitPoint \leftarrow H_B$,
- (4) $\Pi(k+2).hitPoint \leftarrow H_B$,
- (5) $\Pi(k+1).prevBug \leftarrow B$,
- (6) $\Pi(k+2).prevBug \leftarrow B$,
- (7) $k = k + 2$, remove B from $\mathcal{A}$.

4: **end for**
5: Goto Step 2

Step 2:

1: **for** each bug $B \in \mathcal{H}$ **do**
- Follow the obstacle boundary until T is in LOS
- Create leave point L_B. $\Pi(B).leavePoint \leftarrow L_B$
- $\mathcal{A} \leftarrow B$, $\mathcal{H} = \mathcal{H} \setminus B$

2: **end for**
3: Goto Step 1

Step 3:

1: **for** each bug $B \in \mathcal{B}$ **do**
2: retrace_path(B)
3: **end for**

Algorithm 2 Algorithm to reduce number of bugs

Step 2:

1: **for** each bug $B \in \mathcal{H}$ **do**
2: Follow the obstacle boundary until
3: **if** T is in LOS **then**
4: Create leave point L_B. $\Pi(B).leavePoint \leftarrow L_B$
5: $\mathcal{A} \leftarrow B$, $\mathcal{H} = \mathcal{H} \setminus B$, Goto line 1.
6: **else if** reached a vertex $v_j \in \mathcal{V}$ **then**
7: **if** $vIndex(j) == \emptyset$ **then**
8: $vIndex(v_j) = B, vDistance(v_j) = \ell(\pi(B))$. Goto line 2
9: **else**
10: **if** $\ell(\pi(B)) < vDistance(v_j)$ **then**
11: $vIndex(v_j) = B, vDistance(v_j) = \ell(\bar{\pi}(v_j, B))$.
12: **end if**
13: $\mathcal{H} = \mathcal{H} \setminus B$. Goto line 1.
14: **end if**
15: **else**
16: Goto line 2
17: **end if**
18: **end for**
19: Goto Step 1

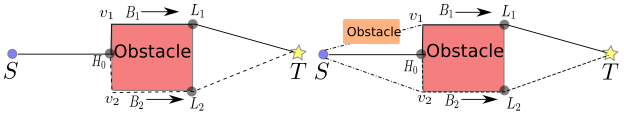


Fig. 3: (a) The new path after pruning is $\pi = \{S, v_1, L, T\}$. (b) During pruning phase, the new path segment may intersect with another obstacle.

performing this process for all the bugs in $\mathcal{H}$, the Step 1 is carried out again. The Step 1 and Step 2 cycle until the condition $\mathcal{A} = \emptyset$ (line 3:a) occurs and the Step 3 begins to recover the paths. Whenever a bug B reaches the goal, it is recorded in the set $\mathcal{B}$ before removing from $\mathcal{A}$ (Step 1, line 2:a). For the example scenario, the bug B_1 will move from H_0 to L_1 through obstacle node v_1, while the bug B_2 will move from H_0 to L_2 through node v_2. When the bugs are at L_1 and L_2 respectively, the Step 2 ends and Step 1 begins. In the Step 1, the bugs B_1 and B_2 find the goal (Step 1, line 2:a), $\mathcal{B} = \{B_1, B_2\}$ and $\mathcal{A} = \emptyset$. Since there are no active bugs, the third step begins.

In the third step, the associated path of each bug $B \in \mathcal{B}$ is traced using Algorithm 3. The result of this Algorithm 3 is the set of collision free paths from S to T.

C. BugFlood heuristics

The Algorithm 1 can generate several paths. However, the algorithm has two deficiencies (i) the obtained paths are not optimal and (ii) the number of bugs generated can be high due to large number of obstacle resulting in higher computational complexity of the algorithm. We will now describe techniques to mitigate these two aspects.

1) Path pruning: The obtained paths can be pruned by constructing new path segments between two nodes sequentially along the path. Since this technique uses the triangle inequality, the new path obtained will be shorter than the original path. For example, consider the path (S, H_0, v_2, L_2, T) obtained for a bug in Figure 3(b). This path can be modified by introducing a segment between S and node v_2. The new path $\pi = \{S, v_2, L_2, T\}$ is shorter than the original path $\pi = \{S, H_0, v_2, L_2, T\}$. In addition to reducing the path length, the pruning also helps in smoothing the path. During pruning, if the new path segment $\pi = \{S, v_1, L_1, T\}$ intersects with an obstacle as shown in Figure 3b then the new path segment is not considered. The path through v_1 remains the same as $\pi = \{S, H_0, v_1, L_1, T\}$. We can increase the complexity by creating bugs for the new obstacle but for simplicity we do not consider this option.

2) Bug cleaning: Bugs, while exploring the environment can encounter an obstacle that has been previously visited by other bugs. We can use this information, to reduce the computational time of Algorithm 1 by introducing few constraints in Algorithm 1. When a bug B reaches a node $v_j \in V_i$, then it checks if v_j was previously visited or not. If v_j was previously visited by some other bug B' then we check if $\ell(\pi(v_j, B')) > \ell(\pi(v_j, B))$. If false, then we remove B from the list of active bugs, otherwise, we link $\ell(\pi(v_j, B))$ to v_j as the shortest path route to v_j and remove B from the list of active bugs. In order to implement this pruning, we use two arrays $vIndex$ and $vDistance$ in Algorithm 2. The pruning procedure is given in Algorithm 2. Note that, the bug B' may have already visited v_j and is currently exploring at some other location. As the bug propagation is sequential, we do not kill B' at v_j but instead kill B as B will follow the same route as that of B'.

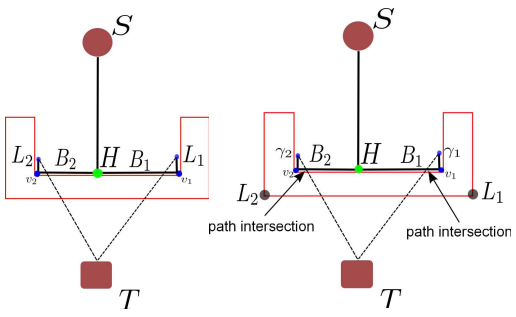
Algorithm 3 Function retrace_path(B)

1: $\pi = \{\}$, $cbug = B$
2: **while** $\pi(last) \neq S$ **do**
3: $\quad \pi \leftarrow [\Pi(cbug).hitPoint(2) \;\; \Pi(cbug).leavePoint]$
4: $\quad cbug = \Pi(cbug).prevBug$
5: **end while**
6: return(π)

D. Non-convex environments

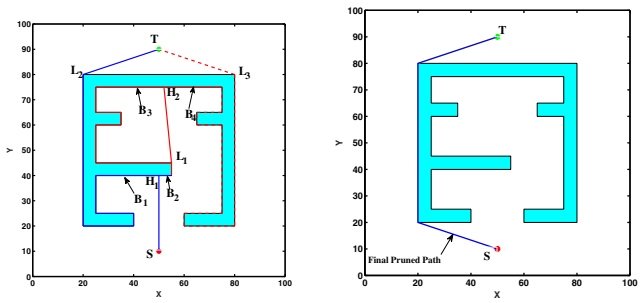
The Step 2 of Algorithm 1 works well for environments having only convex obstacles. For example, consider the scenario in Figure 4a, where the bug starts from S and creates two bugs (B_1 and B_2) at H. The bug B_1 moves along the right side of H and at L_1, T is in LOS, so it tries to move towards T. However, it again intersects with the obstacle, creating additional bugs. These two additional bugs will in turn be killed at v_1 and v_2 respectively as the nodes have already been explored. In a similar fashion, at L_2, B_2 tries to move towards the goal and it intersects with the obstacle creating two bugs which will be again killed at v_2 and v_1 respectively. The bugs originating from B_1 and B_2, and their subsequent bugs do not leave the environment and they are terminated. This condition is similar to the Bug0 algorithm. We mitigate this problem by using the history of the path $\pi(B)$.

When a bug B detects T in LOS, it projects a line ($\mathbb{L}$) from its current location to T. If the projected line intersects with $\pi(B)$, then the bug continues to follow the boundary until the intersection is null. This simple extension is included in the Step 2 of Algorithm 1 so that BugFlood algorithm can generate collision free paths to the goal in any kind of environments. The updated algorithm of Step 2 is given in Algorithm 4. In Figure 4b, bug B_1 at γ_1, projects a line to the goal (dotted line from points γ_1) that intersects its own path $\pi(B_1)$. Since, the intersection is not null, the bug continues the follow obstacle boundary behavior until L_1. Similarly, γ_2 intersects with $\pi(B_2)$ and it continues to use obstacle follow behavior until L_2. Thus, the bugs avoid the unnecessary termination condition. An example scenario showing the bugs finding a path for environments having non-convex obstacles is shown in Figure 5 along with the corresponding pruned paths.



(a) (b)

Fig. 4: (a) At L_1 and L_2, bugs B_1 and B_2 find the goal T in LOS and move towards T creating a local minima (b) Checking the path from points γ_1 and γ_2 to ensure there is path.



(a) (b)

Fig. 5: (a) The paths taken by the bugs for different non-convex environments (b) Pruned shortest path

IV. BugFlood Algorithm Analysis

In this section, we analyze the theoretical properties of the BugFlood Algorithm. We first show bounds on the path obtained from BugFlood Algorithm.

Theorem 1 (Path length): The path length of any bug produced by Algorithm 1 will never exceed

$$\ell(\pi(T)) = D + \sum_i p_i, \tag{2}$$

where p_i is the perimeter of the i^{th} obstacle that the bug meets and D is the Euclidean distance between S and T.

Algorithm 4 Modified step 2 of Algorithm 1

Step 2:

for each bug $B \in \mathcal{H}$ **do**
  Follow the obstacle boundary
  if T is in LOS **then**
    Create a temporary leave point L'_B.
    Draw a line ($\mathbb{L}$) between L'_B and G.
    If $\mathbb{L}$ intersects with $\pi(L'_B)$ **then** to line 2
    else Create L_B. $\Pi(B).leavePoint \leftarrow L_B$
  end if
  $\mathcal{A} \leftarrow B$, $\mathcal{H} = \mathcal{H} \setminus B$
end for
Goto Step 1

Proof: When a bug does not meet any obstacle, then the distance traveled by the bug is the Euclidean distance between S and T which is equal to D. If the bug intersects an obstacle O_i, then the bug splits into two, say, B_1 and B_2, then one of the bug can follow the perimeter up to $p_i - \epsilon, \epsilon > 0$, while the other may follow only ϵ distance. Therefore the upper bound on the maximum additional distance travelled by a bug is p_i. In the worst case, where the bugs intersects all the obstacles in the environment then the additional path length traveled is $\sum_i p_i$. Therefore, the worst case path length is from S to T is $D + \sum_i p_i$. ■

Any algorithm should find a path if there exists one, otherwise it should rapidly inform that no path exists for the given configuration. We will first show that a path does not exists between S and T when the obstacles are convex and then extend the result to environments with non-convex obstacles.

Lemma 1 (Bug Kill): A bug B_k is terminated if it visits a node v_j that has been previously visited by some other bugs.

Proof: The $vIndex$ array in Algorithm 2 records if a bug has visited a node v_j or not. If $vIndex(v_j) \neq 0$ implies that the node is visited by some bug, otherwise the node is not visited by any bug hence $vIndex(v_j) = 0$. If bug B_k visits v_j and $vIndex(v_j) \neq 0$, then the bug B_k is terminated according to Algorithm 2. However, if $vIndex(v_j) = 0$, then $vIndex(v_j)$ is updated to 1 and the bug B_k continues its course. □ ■

Theorem 2 (No path): If a bug B_k is the first to visit a convex obstacle O_i, and bugs B_{k+1} and B_{k+2} formed by B_k are terminated while performing the obstacle following behavior on obstacle O_i, then no path exists from S to T.

Proof: Assume that a bug B_k departs from a leave point $L_{k'}$ towards T and it detects a convex obstacle O_i while traversing towards the goal. Up on detection, B_k creates a hit point H_i and splits into two bugs B_{k+1} and B_{k+2} that follow the obstacle boundary in opposite directions. The following situations can happen to B_{k+1} and B_{k+2}:

Case 1 – Obstacle O_i is visited by some other bugs: In this case, based on Lemma 1, B_{k+1} and B_{k+2} will be terminated. Although B_{k+1} and B_{k+2} are terminated, they are not the first bugs to visit O_i, hence this case does not satisfy the condition given in the theorem.

Case 2 – B_{k+1} and B_{k+2} are the first bugs exploring O_i: In this case, $vIndex(v_j) = \emptyset, v_j \in V_i, j = 1, \ldots, |V_i|$ (from Algorithm 2) and there are two scenarios that can occur (a) B_{k+1} and B_{k+2} find the goal in LOS and (ii) do not find goal in LOS. Let the set of nodes visited by bug B_{k+1} of obstacle O_i be V_i^{k+1} and B_{k+2} be V_i^{k+2}, such that $V_i^{k+1} \bigcup V_i^{k+2} = V_i$ and $V_i^{k+1} \bigcap V_i^{k+2} = \emptyset$. This implies that the bugs are moving along the obstacle carrying out the follow behavior in search of the goal.

Case 2(a) – Assume bug B_{k+1} have visited nodes $v_j \in V_i^{k+1}$ and did not detect the goal in LOS. Once it detects the goal, then it will create a leave point L_{k+1} and proceeds towards the goal. Similarly, if B_{k+2} detects the goal in LOS then it will create a leave point L_{k+2} and move towards the goal.

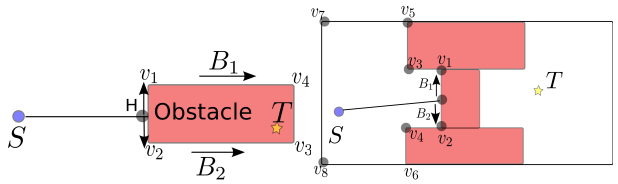


Fig. 6: (a) A scenario where the the goal is inside the obstacle and there is no path from S to T. (b) A scenario where the obstacles are blocking in a way that there is no path from S to T.

Since B_{k+1} and B_{k+2} are not terminated on the obstacle O_i, hence the condition for no path is not satisfied.

Case 2(b) – B_{k+1} and B_{k+2} do not find the goal in LOS. As the bug B_{k+1} could not detect the goal in line-of-sight for all the nodes $v_j \in V_i^{k+1}$, it will now move towards a node $v_j \in V_i^{k+2}$. When the bug B_{k+1} reaches $v_j \in V_i^{k+2}$, then by Lemma 1, B_{k+1} is terminated. Similarly, B_{k+2} will traverse from a node $v_j \in V_{k+2}^i$ to a node $v_{j'} \in V_i^{k+1}$ and it will be terminated at $v_{j'}$ from Lemma 1. ■

Lemma 2 (All bugs are killed): If the condition in Theorem 2 occurs, then all the bugs are killed.

Proof: A bug B_k interacts with obstacle O_i if and only if O_i is in line-of-sight between the leave point of B_k and the goal. The split bugs of B_k, B_{k+1} and B_{k+2} could not detect the goal in LOS even after exploring the complete perimeter of O_i (required condition for Theorem 2). This implies that the goal is inside O_i and it is not reachable. Since the goal is in O_i, other bugs will also visit O_i. Since B_{k+1} and B_{k+2} visited all the nodes of O_i and updated $vIndex(v_j) \neq 0$, for all $v_j \in V_i$, then by Lemma 1, all the bugs that are visiting O_i after B_{k+1} and B_{k+2} are terminated. Therefore, all the bugs that must visit O_i and are killed. ■

Consider the scenario in Figure 6(a), where B splits into B_1 and B_2 at hit point H. A bug B_k is killed only when it visits a node $v_j \in V_i$ that has been previously visited (Algorithm 2, line 14). Assume that B_2 travels to v_3 and B_1 travels to v_4 in search of T. Now, B_1 will travel from v_3 to v_4 in search of T, but v_4 is already visited by B_2 and B_1 is killed (by Algorithm 2). Similarly, B_2 moves from v_4 to v_3 that was previously visited by B_1 and B_2 is killed. Since there was no path from S to T as T was inside the obstacle, the bugs visited nodes that were previously visited. A similar scenario when all bugs are killed is shown in Figure 6(b). The scenario presented in these examples can be extended to n obstacles without losing generality. Therefore, if all the bugs are killed then no path exists from S to T.

The time taken by the algorithm to find a condition where all the bugs are killed is the time that the algorithm informs that no path exists from S to T. This is an important property for many applications, where the robot can perform some other action like informing the operator about no path or perform some other safe maneuver.

Definition 8 (Children of a bug): When bug B_k detects an obstacle, it splits into two bugs B_{k+1} and B_{k+2}. B_{k+1} and B_{k+2} further split into two bugs when they detect obstacles respectively. The process of splitting into two bugs recursively creates a binary tree with bug B_k as the root node. We define all the bugs formed from bug B_k due to splitting, as the children of bug B_k.

The bugs B_{k+1}, B_{k+2}, and the subsequent bugs formed when B_{k+1} and B_{k+2} split, and their on, are the children on B_k.

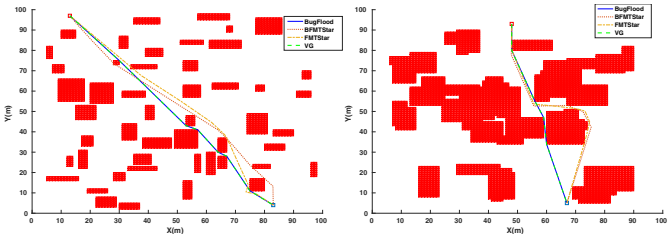
Lemma 3: If a bug B_k is the first to visit a non-convex obstacle O_i, and all its children are terminated while performing the obstacle following behavior on obstacle O_i, then no path exists from S to T.

Proof: The bug B_k detects a non-convex obstacle O_i and splits into two bugs (B_{k+1} and B_{k+2}). The bug B_{k+1} can create a leave point on O_i and move towards goal T. However, it can split again into B_{k+3} and B_{k+4} in O_i itself. This can happen due to non-convex nature of the obstacle. Assume that B_{k+3} moves towards the leave point as it follows the obstacle, while B_{k+4} moves away from the leave point exploring the obstacle. The bug B_{k+3} will be killed as it approaches the nodes explored by B_{k+1} by Lemma 1. A similar process can happen to B_{k+2} and its children. As the bugs traverse the obstacle, let $v_j \in \hat{V}_i^{k+1}$ be the set of nodes that B_{k+1} and its children have explored and let $v_j \in \hat{V}_i^{k+2}$ be the set of nodes that bug B_{k+2} and its children have explored such that $\hat{V}_i^{k+1} \bigcup \hat{V}_i^{k+2} = V_i$ and $\hat{V}_i^{k+1} \bigcap \hat{V}_i^{k+2} = \emptyset$. Similar to the conditions in Theorem 2, if any of the bugs found the goal in LOS and depart from O_i, then the theorem condition is not satisfied. Assume that the bug $B_{k'}$, the leaf node of bug B_{k+1}, moves towards a node $v_{j'} \in \hat{V}_i^{k+2}$, if $B_{k'}$ detects goal in LOS while moving towards $v_{j'}$, then this condition is similar to case 2(a) of Theorem 2 and hence does not satisfy the condition. However, if $B_{k'}$ reaches $v_{j'}$, then by Lemma 1, $B_{k'}$ will be killed. Similarly, bug $B_{k''}$ the leaf node of B_{k+2} moves towards a node $v_{j''} \in V_i^{k+1}$. Since $B_{k'}$ and $B_{k''}$ are traversing along the same segment of the obstacle boundary and $B_{k'}$ could not detect the goal in LOS, $B_{k''}$ will also not detect the goal and will reach $v_{j''}$. The bug $B_{k''}$ will be killed as $v_{j''}$ has already been visited. Since, all the children of B_k are terminated in O_i and they cover the complete perimeter of O_i and they are unable to detect the goal, hence no path exists from S to T. ■

We will now prove that the BugFlood Algorithm is complete by showing that it does not enter infinite loops or when all the bugs are killed. Hence if there is a path then the algorithm will always find it.

Theorem 3 (Completeness): BugFlood algorithm described in Algorithm 1 with the modified step 2 given in Algorithm 4, always returns a path if one exists.

Proof: The algorithm cannot return a path only when the bugs enter into infinite loops or when all the bugs are killed. A bug enters into an infinite loop only when (a) it reaches the same node v_i multiple number of times or (b) continues to intersect the obstacles and move towards infinity. Since, we assume that the environment is bounded and has finite number of obstacle, (b) cannot happen. If (a) happens then from Lemma 1, the bugs are killed after visiting a node that has already been visited, therefore infinite loops cannot



(a) (b)

Fig. 7: (a) Paths for convex environments for 4 planners - BugFlood, VG, BFMT*, and FMT*. (b) Path for non-convex environments for four planners - BugFlood, VG, BFMT*, and FMT*.

occur. From Lemma 2, if all the bugs are killed then no path exists from S to T. Therefore, the bug algorithm will always find a path if one exists. ■

Note that, since the bugs explore the obstacles in both the directions from the hit point, a situation where some of the bugs are terminated with the condition given in Theorem 2, and the rest reaching the goal does not occur.

V. Results

The validation of the proposed model is carried out using simulations on ROS (Robot Operating System) [24]. We assume each bug to be a point mass holomonic robot. The robot can go straight and change it's direction instantaneously.

A. Simulation setting

We consider a search space consisting of source and goal locations, and obstacles. The search space is of length $A_L = 100$m and width of $A_W = 100$m. We assume that each obstacle can be represented as a polygon. Each obstacle is a rectangle whose length,width and position are randomly generated. The maximum width of the obstacle is given by $W_i = \gamma_w \lceil \frac{A_W}{N} \rceil$, and length of the obstacles is $L_i = \lceil \gamma_l \frac{A_L}{N} \rceil$. The notations $\gamma_w > 0$ and $\gamma_l > 0$ are constants and in the simulations it was set to 6 and 4 respectively. Non-convex environments are generated by placing rectangular obstacles of random size randomly in the environment. Due to random placement some obstacles have overlaps creating the combined obstacle as a non-convex obstacle. We used only the nodes that constitutes the obstacle boundary and removed all the nodes inside the obstacle. This process is carried out to ensure all the planners have the same input environment to determine paths. The simulations were carried out on a desktop having configuration of 8GB RAM, and Intel Core i7-3610QM CPU (4 cores) at 2.3GHz. A sample environment with 50 obstacles (convex and non-convex) is shown in Figure 7 along with paths for four different planners.The results show that BugFlood performs close to VG.

B. Comparison of algorithms

The proposed BugFlood planner was tested using Monte-Carlo simulations for non-convex environments and was compared with other well known path planners from the OMPL library [21], namely, PRM* [27], RRT [28], RRT* [29], BFMT* [25], BIT* [30], CForest [31], FMT* [26], and IRRT* [32]. The number of simulations were 100 with different environments and obstacle densities. The comparison metrics are: path cost (length) in meters, time taken to generate a path from source to goal in seconds, and time taken to determine when no path exists. Note that, for OMPL, we used default parameters with time to terminate as ten seconds. We also carried out path pruning using shortcutPath method for all the OMPL planners to improve the path cost.

No. Obs	PRM*	RRT	BFMT*	BIT*	CForest	FMT*	IRRT*	RRT*	VG	BFld
10	120.1	144.4	118.2	122.0	128.1	118.2	130.5	129.6	112.4	118.1
20	118.7	146.6	116.9	122.0	128.2	116.6	131.0	129.4	112.5	116.2
50	112.9	144.8	107.6	116.1	121.6	107.3	127.8	124.3	104.9	107.7
100	110.9	141.9	102.9	117.4	123.2	102.7	125.4	123.9	97.8	102.3
250	111.9	143.4	99.9	112.3	124.8	99.4	123.5	121.0	94.9	99.1

(a) Average path length in meters for environments consisting of different density of non-convex obstacles

No. Obs	PRM*	RRT	BFMT*	BIT*	CForest	FMT*	IRRT*	RRT*	VG	BFld
10	3.3	1.2	8.4	2.8	1.1	12.6	1.2	1.3	99.2	0.3
20	3.3	1.2	7.9	2.8	1.2	14.8	1.2	1.3	510.6	0.4
50	4.1	1.2	7.7	2.4	1.2	15.1	1.4	1.4	3.1s	0.7
100	3.9	1.2	7.0	2.8	1.2	17.3	1.4	1.4	9.7s	1.1
250	3.5	1.2	6.3	2.4	1.2	16.6	1.6	1.6	28.8s	1.4

(d) Time to compute a path (msec) for non-convex obstacles environment

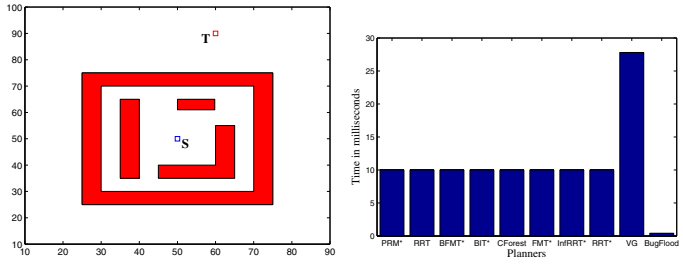
TABLE I: Comparison of average path length and average time taken to compute a path for different path planners under various obstacle density scenarios

Path length comparison: The Table I (a) shows the average path cost for ten different planners. From the table, BFMT*, FMT*, BugFlood and VG techniques are close.Although the average cost of BFMT* and FMT* is very close to that obtained using BugFlood, but the time required to find the path is higher than BugFlood(see Table I(b)). For example, an environment consisting of 20 obstacles, BugFlood is 11x and 18x times faster than BFMT* and FMT* respectively. On the other hand, BugFlood is 5% less than the VG path cost, while being much faster than VG.

Time to compute a path: We compare the performance of different planners based on the time taken to compute a path as shown Table I(b). From the table, we can see that the planners RRT, RRT*, IRRT*, and CForest are the closest to that of BugFlood. The BugFlood is computationally superior than the other techniques except for the 250 obstacle case. However, BugFlood algorithm produces paths of much smaller cost compared to the other techniques (see Table I(a)). Thus, from the computational complexity and the path cost perspective, BugFlood offers advantages than the planners in OMPL.

1) Time taken to inform no path exists: The BugFlood planner is deterministic and hence we can determine if a path exists for a given environment. We created a new set of simulations for this purpose, where the source is enclosed inside a rectangular shaped obstacles whose length, width and the location is randomly placed. A sample environment is shown in Figure 8a.

Figure 8b shows the time taken by different algorithms to provide the information whether a path exists or not. The planners from OMPL use time as a parameter. If a path exists within the predefined time then the planner quits after providing the first path. However, in the current case, there



(a) (b)

Fig. 8: (a) An example environment that shows no path exists from source to goal. (b) Average time taken to determine if no path exists in the environment.

is no path and the planner tries to generate paths until the prescribed time. We use 10 milliseconds as the prescribed time for the OMPL planners. From the Figure 8b, we can see that the OMPL planners continue to search the space until 10 milliseconds, while on average the BugFlood planner determines that no path exists in 0.4 milliseconds. This information can be used in changing the behavior of the robot and can be crucial in autonomous vehicles.

VI. Conclusions

We have presented a novel path planning algorithm from the bug family called BugFlood. The algorithm is fast, easy to implement and generates paths whose cost is near-optimal. We have compared the performance of BugFlood with OMPL planners and visibility graph method under different obstacle density scenarios. The results show that BugFlood performs close to near-optimal with very low computational time requirement. Further extensions of BugFloodcould include – adding kinematic constraints to determine smooth paths, and extending the algorithm for multi-robot path planning.

References

[1] S. M. LaValle, *Planning algorithms*. Cambridge university press, 2006.

[2] V. J. Lumelsky and A. A. Stepanov, "Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape," *Algorithmica*, vol. 2, no. 1-4, pp. 403–430, 1987.

[3] R. A. Langer, L. S. Coelho, and G. H. C. Oliveira, "K-bug, a new bug approach for mobile robot's path planning," in *IEEE International Conference on Control Applications*, Oct 2007, pp. 403–408.

[4] K. Taylor and S. M. LaValle, "I-bug: An intensity-based bug algorithm," in *IEEE International Conference on Robotics and Automation*, 2009, pp. 3981–3986.

[5] Y. Gabriely and E. Rimon, "Cbug: A quadratically competitive mobile robot navigation algorithm," *Robotics, IEEE Transactions on*, vol. 24, no. 6, pp. 1451–1457, 2008.

[6] I. Kamon, E. Rivlin, and E. Rimon, "A new range-sensor based globally convergent navigation algorithm for mobile robots," in *IEEE International Conference on Robotics and Automation*, 1996, pp. 429–435.

[7] J. Ng and T. Bräunl, "Performance comparison of bug navigation algorithms," *Journal of Intelligent and Robotic Systems*, vol. 50, no. 1, pp. 73–84, 2007.

[8] T. Nayl, G. Nikolakopoulos, and T. Gustafsson, "Real-time bug-like dynamic path planning for an articulated vehicle," in *Informatics in Control, Automation and Robotics*. Springer, 2015, pp. 201–215.

[9] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no. 2, pp. 100–107, 1968.

[10] A. Stentz, "Optimal and efficient path planning for partially-known environments," in *Proc. of the IEEE International Conference on Robotics and Automation*. IEEE, 1994, pp. 3310–3317.

[11] M. Likhachev, D. I. Ferguson, G. J. Gordon, A. Stentz, and S. Thrun, "Anytime dynamic a*: An anytime, replanning algorithm." in *ICAPS*, 2005, pp. 262–271.

[12] K. Daniel, A. Nash, S. Koenig, and A. Felner, "Theta*: Any-angle path planning on grids," *Journal of Artificial Intelligence Research*, vol. 39, no. 1, pp. 533–579, 2010.

[13] S. Aine, S. Swaminathan, V. Narayanan, V. Hwang, and M. Likhachev, "Multi-Heuristic A*," in *Proceedings of the Robotics: Science and Systems (RSS)*, 2014.

[14] J. J. K. Jr. and S. M. LaValle, "RRT-Connect: An Efficient Approach to Single-Query Path Planning," in *ICRA*. IEEE, 2000, pp. 995–1001.

[15] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic Roadmaps for Path Planning in High-dimensional Configuration Spaces," *IEEE T. Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[16] S. M. Lavalle, J. J. Kuffner, and Jr., "Rapidly-exploring random trees: Progress and prospects," in *Algorithmic and Computational Robotics: New Directions*, 2000, pp. 293–308.

[17] S. Karaman and E. Frazzoli, "Incremental Sampling-based Algorithms for Optimal Motion Planning," in *Robotics: Science and Systems*. Zaragoza, Spain: The MIT Press, June 2010.

[18] L. Janson, B. Ichter, and M. Pavone, "Deterministic sampling-based motion planning:optimality, complexity and performance," in *International Symposium on Robotics Research*, 2015.

[19] A. Hussein and A. Elnagar, "A fast path planning algorithm for robot navigation with limited visibility," in *Systems, Man and Cybernetics, 2003. IEEE International Conference on*, vol. 1, 2003, pp. 373–377.

[20] T. Siméon, J.-P. Laumond, and C. Nissoux, "Visibility-based probabilistic roadmaps for motion planning," *Advanced Robotics*, vol. 14, no. 6, pp. 477–493, 2000.

[21] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, December 2012.

[22] N. Sharma, J. Pinto, and P. Sujit, "Bugflood: A bug inspired algorithm for efficient path planning in an obstacle rich environment," in *Proc. of the AIAA Infotech @ Aerospace, San Diego, California, USA*, AIAA 2016-0254, 2016.

[23] V. J. Lumelsky and A. Stepanov, "Dynamic path planning for a mobile automaton with limited information on the environment," *Automatic Control, IEEE Transactions on*, vol. 31, no. 11, pp. 1058–1063, Nov 1986.

[24] "Open source robotics foundation," http://www.ros.org/.

[25] J. A. Starek, J. V. Gomez, E. Schmerling, L. Janson, L. Moreno, and M. Pavone, "An asymptotically-optimal sampling-based algorithm for bi-directional motion planning," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2015, pp. 2072–2078.

[26] L. Janson, E. Schmerling, A. Clark, and M. Pavone, "Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions," *The International journal of robotics research*, vol. 34, no. 7, pp. 883–921, 2015.

[27] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *Robotics and Automation, IEEE Transactions on*, vol. 12, no. 4, pp. 566–580, 1996.

[28] J. J. Kuffner and S. M. LaValle, "Rrt-connect: An efficient approach to single-query path planning," in *IEEE International Conference on Robotics and Automation*, vol. 2, 2000, pp. 995–1001.

[29] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[30] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "Batch informed trees (bit*): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs," in *IEEE International Conference on Robotics and Automation*, 2015, pp. 3067–3074.

[31] M. Otte and N. Correll, "C-forest: Parallel shortest path planning with superlinear speedup," *IEEE Transactions on Robotics*, vol. 29, no. 3, pp. 798–806, 2013.

[32] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "Informed rrt*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sept 2014, pp. 2997–3004.