J Intell Robot Syst (2007) 50:73–84
DOI 10.1007/s10846-007-9157-6

Performance Comparison of Bug Navigation Algorithms

James Ng · Thomas Bräunl

Received: 22 January 2007 / Accepted: 11 March 2007 /
Published online: 25 April 2007

Abstract The Bug algorithm family are well-known robot navigation algorithms with proven termination conditions for unknown environments. Eleven variations of Bug algorithm have been implemented and compared against each other on the EyeSim simulation platform. This paper discusses their relative performance for a number of different environment types as well as practical implementation issues.

Keywords Mobile robot · Navigation algorithm · Bug algorithm · Performance comparison

1 Introduction

Navigation in an unknown 2D environment is one of the standard problems in robotics. A mobile robot is being placed at a starting position (S) and has to find its way to a target position (T) that is specified by distance and direction relative to the starting position; no other information about the environment is known to the robot.

The robot only has to reach the target position – or terminate if the target is unreachable – it does not have to map its environment. Therefore, a particular navigation algorithm can have a statistically better performance than another, but may not be better for any possible environment setting.

The algorithms in the Bug family approach the navigation from the theoretical side, postulating termination (detection whether a goal is reachable or not) for every possible case, while making significant simplifying assumptions, such as point objects and noise-free perfect sensor data and navigation.

Since every algorithm in the Bug family has to have the termination property, subsequently published Bug algorithms try to improve the algorithm performance, e.g. the path length or time required to either reach the target or to detect that the target is unreachable. The aim of this paper is therefore to identify the navigation techniques that work best by conducting

J. Ng · T. Bräunl (✉)
EECE, M018, UWA, 35 Stirling Hwy, Crawley, Perth, WA 6009, Australia
e-mail: thomas.braunl@ieee.org

unbiased performance comparisons of various Bug algorithms, based on empirical data from experiments conducted in different environments.

Section 2 introduces algorithms from the Bug family and their navigation techniques.

Section 3 discussed theoretical differences between Bug algorithms, as well as practical implementation issues.

Section 4 presents simulation results from eleven Bug algorithms in four different environments and also discusses algorithm implementation complexity.

Section 5 presents conclusions and also touches on fault tolerance issues in noisy environments.

2 The Bug Algorithm Family

Bug algorithms solve the navigation problem by storing only a minimal number of way points, but without generating a full map of the environment. If no solution path exists, the algorithm is able to recognize this situation and terminates reporting that the target is unreachable, instead of endlessly wandering about.

The Bug model makes three simplifying assumptions about the robot [1]. First, the robot is a point object. Second, the robot has perfect localization ability. Third, the robot has perfect sensors. These three assumptions are unrealistic for real robots, and therefore Bug algorithms are usually not directly applied for practical navigation tasks, but can be considered as a higher-level supervisory component of a system that incorporates all three assumptions. Bug algorithms can be seen as a first logical step towards solving a robotic 2D navigation task.

The following algorithms from the Bug family have been implemented and evaluated: Bug1 [1], Bug2 [1], Alg1 [2], Alg2 [3], DistBug [4], Class1 [5], Rev1 [6], Rev2[6], OneBug [7], LeaveBug [7] and TangentBug [8]. OneBug and LeaveBug are variations developed by the authors.

Please note that this is not a complete list of Bug algorithms in existence. There are more, e.g. VisBug-21 [9], VisBug-22 [9], HD-1 [10], Ave [11], RoverBug [12], WedgeBug [12] and CautiousBug [13]. Those were not included for brevity and because they are quite similar to some of the included algorithms. For instance, RoverBug, WedgeBug and CautiousBug are similar to TangentBug.

Below we present a simplified pseudo-code version of Bug1 as a reference for other algorithms.

```
1) Drive directly towards target T, until either:
    a) The target is reached. Algorithm stops.
    b) An obstacle is encountered.
   Define hit point $H_j$. Goto step 2.
2) Perform clockwise circumnavigation while continuously updating $Q_m$, the closest
   position to the target so far, until either:
    a) The target is reached. Algorithm stops.
    b) $H_j$ is encountered again (circumnavigated obstacle). Test whether the line segment
       [$Q_m$, T] is obstructed by this circumnavigated obstacle:
        i) If yes, the target is unreachable. Algorithm stops.
        ii) If no, return to $Q_m$ clockwise or counter-clockwise, whichever is shorter. Goto
            step 1.
```

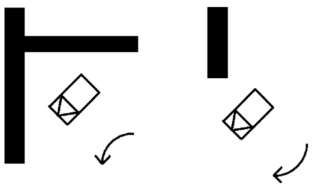


Fig. 1 *Left*: Robot rotates on the spot *Right*: Robot drives in a circle

LeaveBug is similar to Bug1, except that instead of circumnavigating the entire obstacle before evaluating the line segment [Q_m,T], the robot evaluates this condition after completing each path segment that does not prevent movement towards the target.

OneBug is similar to Alg2, except that no stored points are used. Instead, the robot completely explores a segment along the blocking obstacle that prevents movement towards the target.

3 From Theory to Implementation

Since Bug algorithms are usually published as pseudo code, they do leave some room for interpretation. Therefore, it is important to specify all adaptations required to transform them into proper executable algorithms. We have used the RoBIOS application programmer interface [14], which is compatible with real SoccerBot mobile robots as well as the EyeSim simulation system. Below is a discussion of some of the some issues encountered during the implementation phase:

3.1 Update Frequency

In theory, Bug algorithms continuously update their position data. This is required, e.g. for calculating the robot's position relative to the M-line in Bug2. While this may be sound in theory, it is not possible to achieve a continuous update in practice. It was found through experiments that a distance of 40 mm between updates achieved the optimal balance between updating too frequently and too infrequently on the EyeSim simulator.

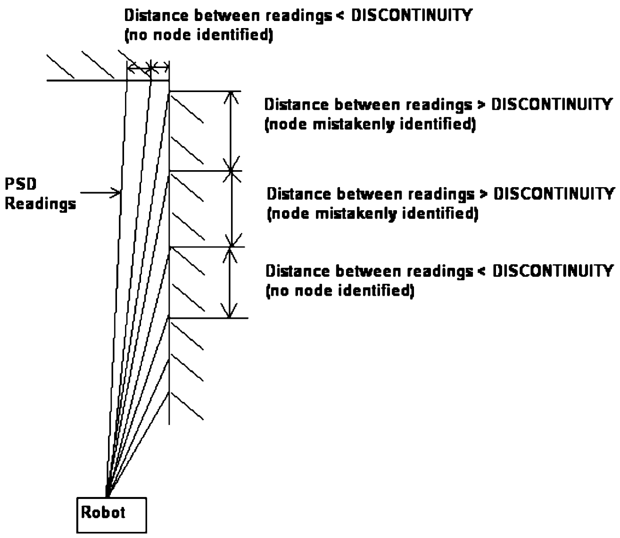
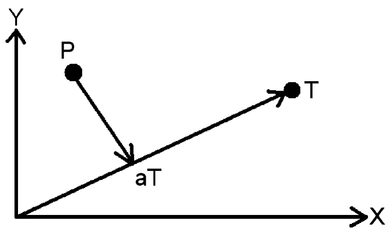

Distance between readings < DISCONTINUITY
(no node identified)
Distance between readings > DISCONTINUITY
(node mistakenly identified)
Distance between readings > DISCONTINUITY
(node mistakenly identified)
Distance between readings < DISCONTINUITY
(no node identified)
PSD Readings
Robot
Fig. 2 Finite angular resolution may cause incorrect node identifications

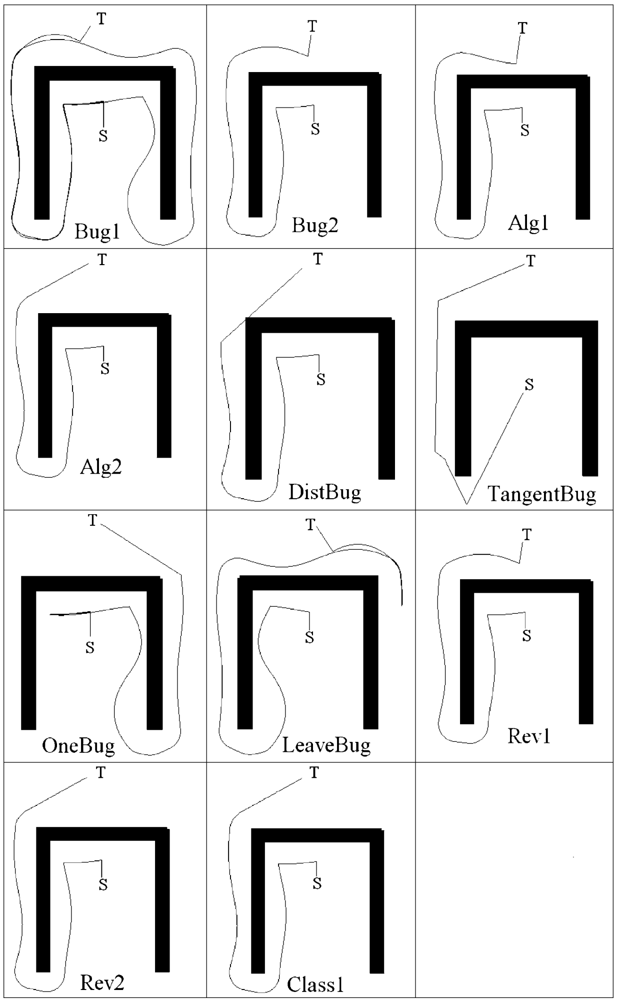
Fig. 3 The vectors "P-aT" and "T" are perpendicular



3.2 Recognition of Stored Positions

In theory, Bug algorithms use infinitesimally small points to represent the start, target, latest hit point and other significant positions in Alg1, Alg2, Rev1 and Rev2. This does not work in practice, because of the limited update frequency and subsequent deviations during wall-following. Hence, in our implementation, each significant position is represented by a

Fig. 4 Paths for Bug algorithms in environment A



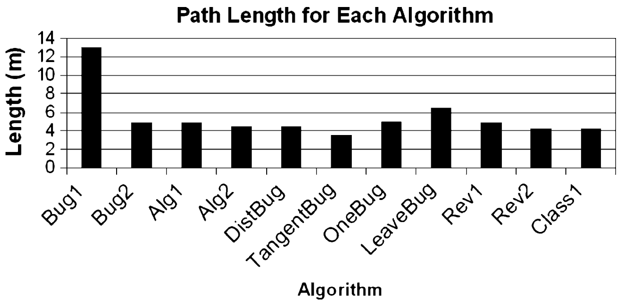


Fig. 5 Path lengths for environment A

square of side length 50 mm. A square was chosen because it is computationally efficient to check if the robot is inside. The size of the square was chosen such that it is slightly larger than the robot's driving distance during an update cycle, but not so large that would lead to frequent false positives.

3.3 Robot Sensor Equipment

Some Bug algorithms require range sensors whereas others do not. For instance, DistBug and TangentBug require range sensors, while Bug2 and Bug1 require only tactile sensors. For continuity and simplicity, we used a robot model with eight range sensors and no tactile sensors for implementing all algorithms. Therefore range sensors are also used if only tactile information is required, e.g. for wall-following or checking whether the robot is able to drive towards the target.

3.4 Moving Towards Target

In all Bug algorithms, the ability to check if the robot can move towards the target at its current location is essential. For instance, Bug1 requires Q_mT to be checked in its test of target reachability. Also, Bug2, Alg1, Alg2, Rev1 and Rev2 all require this check to be made on any prospective leave points.

In theory, the robot is able to use its tactile sensor to evaluate this check. In our implementation, this check is performed by obtaining the free-space in the target's direction and comparing it to a predefined value. Through experiments, it has been found that a value of 270 mm works adequately. To obtain the free-space in the target's direction, the robot points one of its eight range sensors in the target's direction such that rotation is minimized.

3.5 Wall Following

Lumelsky [1] notes that special algorithms beyond the scope of Bug algorithms are required to follow a wall. In our implementation the robot uses a simple PD controller to follow the wall. The input is the distance from the range sensor closest to the followed wall and the output is the curve that the robot drives until the next update. This works well if the robot is following a "gentle" curve, but an obstacle's perimeter can be arbitrary. Hence, there are situations where the robot has to perform special movements.

More specifically, if there is a wall directly ahead of the robot and to its right, the robot will rotate counter-clockwise on the spot until it can drive forwards again. This situation is illustrated in Fig. 1, left. If there are no walls surrounding the robot, the robot drives in a circular pattern until it detects a wall on the right or it detects a wall ahead of it. This is illustrated in Fig. 1, right.

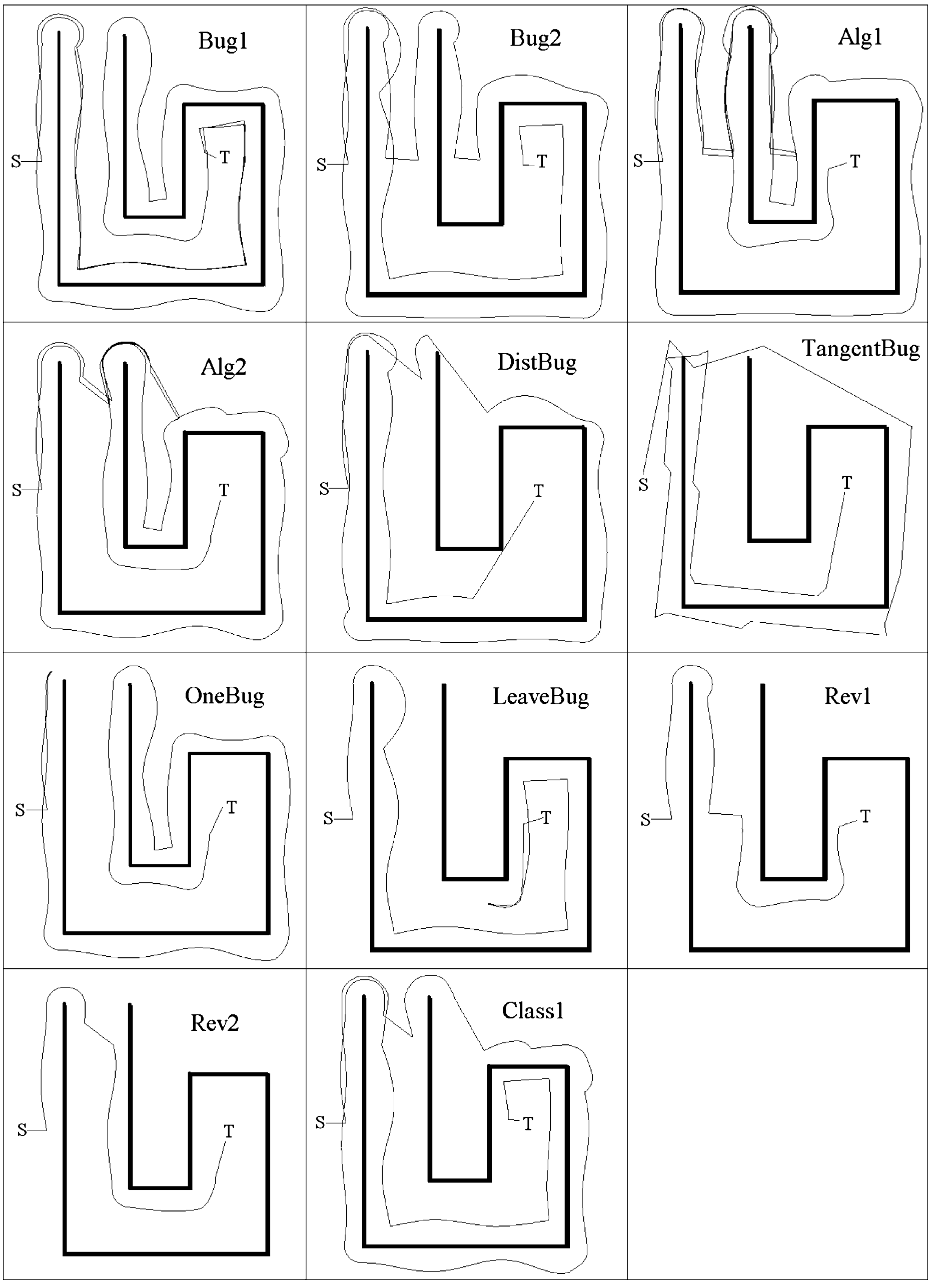


Fig. 6 Paths for Bug algorithms in environment B

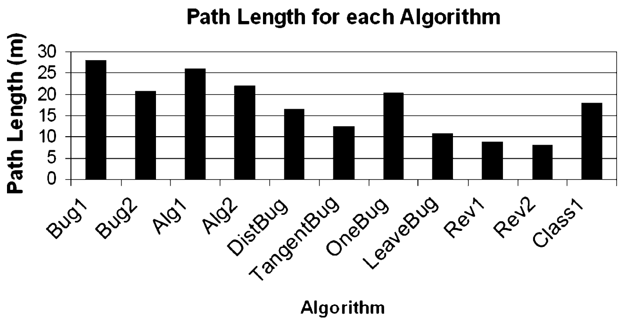


Fig. 7 Path lengths for environment B

3.6 Limited Angular Resolution for the LTG

In theory, the Local-Tangent-Graph (LTG) should be continuous. In practice, range sensors have a finite angular resolution. For our robot model this resolution is 1 degree between each sample. To identify nodes, successive values are compared against a discontinuity threshold. If the difference is larger, a node is identified. This can lead to an error where a node is mistakenly identified, as illustrated in Fig. 2. To reduce the possibility of these errors, the sensor range is restricted and the robot is programmed to move away from an obstacle if it comes too close to it.

3.7 M-line Identification

Bug2, Alg1 and Rev1 use the concept of the M-line that links start and target position. Checking if the robot is positioned on the M-line is essential. Consider the situation where "S" is at the origin, "T" is a vector to the target, "P" is a vector to the robot's current location and "a" is a scalar such that the vectors "P-aT" and "T" are perpendicular. Fig. 3 illustrates this situation.

It follows from the dot product that:

$$(T_x)(P_x - aT_x) + (T_y)(P_y - aT_y) = 0$$

Rearranging for "a" gives:

$$a = \frac{T_x P_x + T_y P_y}{T_x^2 + T_y^2}$$

If $0 \leq a \leq 1$ and the Euclidean distance between "P" and "aT" is smaller than a threshold value, the robot is on the M-line.

4 Experiments and Results

For these experiments, a simulation setting without sensor or actuator noise has been selected. Figure 4 illustrates the Bug algorithms on environment A, featuring a local minimum. Several early navigation techniques such as the potential field method [15] had difficulties overcoming local minimums. In theory, no Bug algorithm should have difficulty overcoming a local minimum and this is verified in our implementation.

Bug1 has the longest path length, followed by LeaveBug, while all other algorithms have a similar small path length. This is due to the fact that Bug1 does not check for target

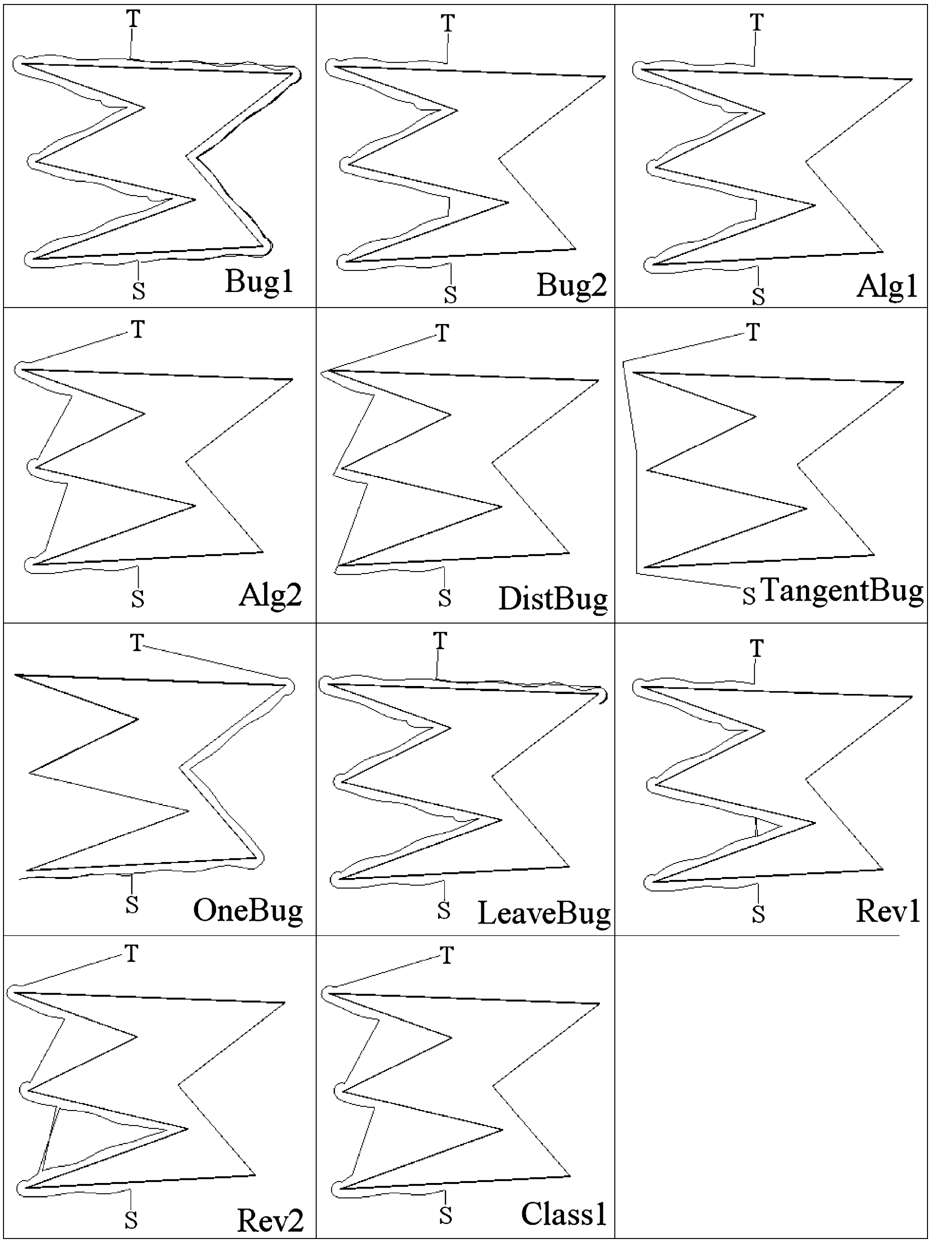


Fig. 8 Paths for Bug algorithms in environment C

reachability until it re-encounters the first hit point (collision point) with the U-shaped obstacle. This unnecessarily makes it circumnavigate the complete obstacle, while other algorithms depart towards the target much earlier.(Fig. 5)

Figure 6 illustrates the algorithms on a terrain originally created by Sankar [3]. The beauty of this terrain is that it is complicated enough to reveal the unique characteristics of each

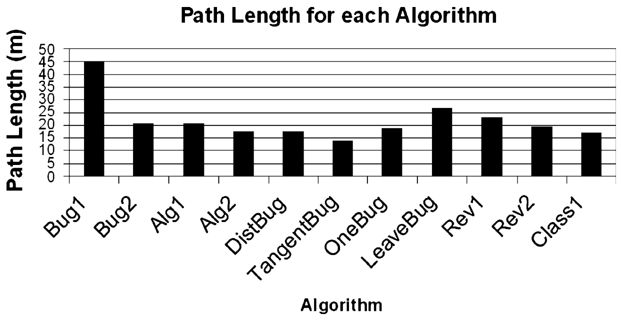


Fig. 9 Path lengths for environment C

algorithm but not so complicated as to be overwhelming. For instance, the M-line is clearly visible in the paths for Bug2, Alg1 and Rev1, as is the stored points concept in Alg1 and Alg2. DistBug's leaving condition allows it to leave slightly earlier than Alg2 and Rev2, resulting in a shorter path length. The overall shortest path was reached by Rev2 followed by Rev1. This can be attributed to their alternative wall following procedure.

Some interesting issues arise when comparing Alg1 against Bug2 and Alg2 against DistBug. Recall that Alg1 is very similar to Bug2 except that Alg1 uses stored points and Alg2 is very similar to DistBug except for stored points and the inclusion of the range-based leaving condition. Interestingly, Bug2 and DistBug produce shorter paths than their stored point counterparts.(Fig. 7)

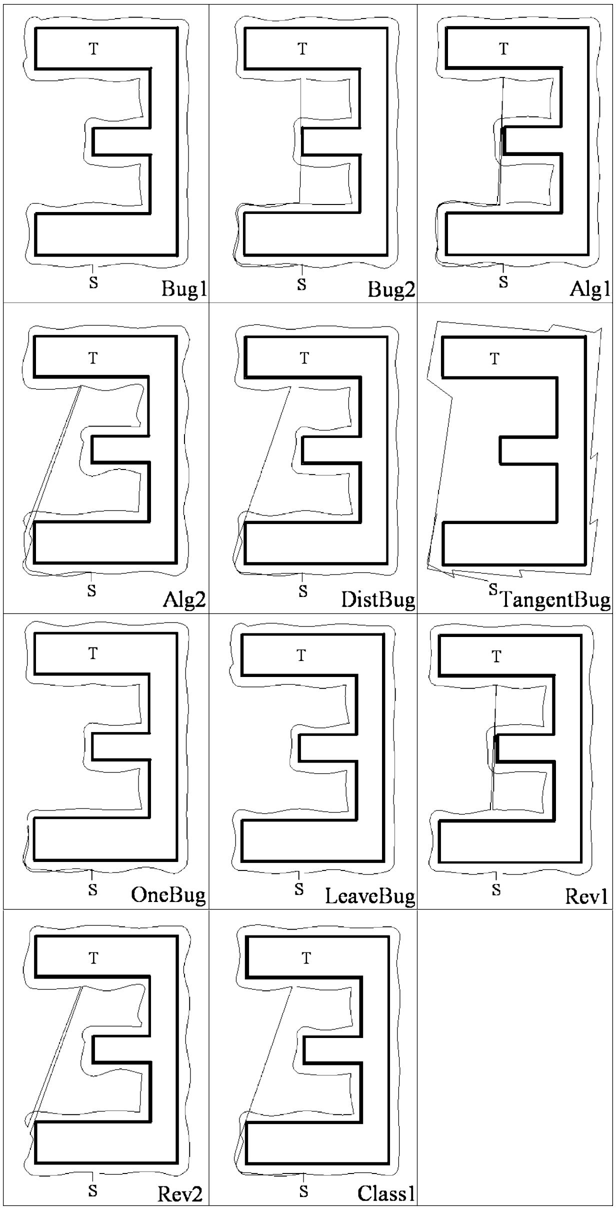
Figure 8 illustrates the algorithms on a terrain featuring only a single semi-convex obstacle. Consider the minimal convex hull associated with any obstacle. If all differences between the obstacle and the minimal convex hull are convex, then the obstacle is called semi-convex.

In this environment the shortest path is produced by TangentBug. This is because TangentBug can use the LTG (local tangent graph) to sweep all areas of a discrepancy between the minimal convex hull and the semi-convex obstacle, since the discrepancy must be convex. Hence, it can travel along the minimal convex hull. Second best after TangentBug is DistBug. Clearly, its use of range sensors allow it to leave the obstacle earlier than Alg2 and this results in a shorter path. Rev1 and Rev2 did not perform well for this environment, since they conducted unnecessary circumnavigation due to a boundary following direction decision.

Some interesting questions arise as to whether particular algorithms will always perform better than others on semi-convex obstacles. For instance, it is foreseeable that TangentBug will always produce the shortest path since it always travels along the minimal convex hull. Also, does DistBug always produce the second shortest path because of its range-based leaving condition? (Fig. 9)

Figure 10 illustrates the Bug algorithms on an environment where the target is unreachable (here the target is inside the obstacle). An unreachable target implies that at least one obstacle must be fully circumnavigated. TangentBug produces the shortest path because its range sensors allow it to scan along the surface of the obstacle without the robot needing to actually travel there. Amongst the tactile sensor algorithms, Bug1 and LeaveBug are tied for shortest path because Bug1 requires the entire obstacle to be circumnavigated before leaving, and LeaveBug requires the entire enabling segment to be explored. On this particular environment, LeaveBug's path is identical to Bug1. As expected, stored points, alternative wall-following methods and other path-shortening measures are useless on an obstacle that renders the target unreachable.

Fig. 10 Paths for Bug algorithms in environment D



Algorithms with the fewest states were the easiest to implement, i.e. simple leaving condition and storing only the current hit point for circumnavigation detection. These include Bug2 and Class1, which require the least amount of code for implementation. They did not fail in any of the test environments and used the least amount of memory. (Fig. 11)

Then, there are slightly harder to implement algorithms such as Bug1, which requires Q_m to be continuously updated as well as information on whether or not the robot can drive

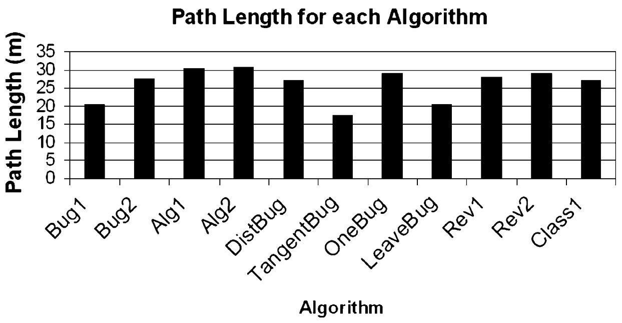


Fig. 11 Path lengths for environment D

towards the target. OneBug and LeaveBug require several states to operate and DistBug requires a range-based leaving condition. Neither of these algorithms failed in any of the test environments, but they required slightly more memory and some more lines of code for implementation.

Algorithms that require stored points are more difficult to implement. Alg1, Alg2, Rev1 and Rev2 all require management of multiple stored points and therefore also more memory. Generally, these algorithms are reliable but a few failures were recorded due to the robot falsely classifying an protruding obstacle point as a previously encountered stored point. Also, additional debugging and testing was needed for the stored points scheme to ensure correct functionality.

Finally, TangentBug was the most difficult algorithm to implement. The requirements of detecting local minima, "corner smoothing" with finite angular resolution and processing nodes were quite complicated in comparison to other Bug algorithms. These requirements also required a lot of computation resources and were a frequent source of failure. Also, the robot has to drive slightly away from the focus node to avoid a collision. One advantage of TangentBug is that there is no need for wall-following since the robot only turns on the spot and travels in straight lines. TangentBug shows what can theoretically be achieved using an omni-directional, always up-to-date LTG.

5 Conclusions

Bug algorithm performance varies greatly depending on the environment. TangentBug produces the shortest path in environments with wider spaces that allow it to make use of its range sensors (environments A, C and D). Here, TangentBug can drive directly towards a vertex whereas other algorithms have to rely on wall following. The second shortest path in environment A was achieved by DistBug, because it uses range sensors to immediately detect that the target is visible. In environment B, Rev2 produced the shortest path, because the alternative wall following strategy minimized wall-following paths globally. In environment C, DistBug produced the second shortest path because its range sensor allows the robot to leave an obstacle earlier. In environment D, LeaveBug and Bug1 tied for the second shortest path, as the environment required one continuous circumnavigation.

As for implementation complexity, we subjectively ranked the Bug algorithms from simple to complex as: Class1, Bug2, Bug1, OneBug, LeaveBug, DistBug, Alg2, Rev2, Alg1, Rev1 and finally TangentBug.

We conducted simulations both in a perfect, noise-free world, as well as under more realistic noise settings with small errors in sensor reading and localization. In simulation runs with noise, we encountered situations where algorithms did not terminate, terminated incorrectly or terminated at an inadequately large distance from the target. Although performance comparisons under noise were not the main focus of this paper, it needs to be noted that Bug algorithms in general do not exhibit fault-tolerance properties, which are the advantage of probabilistic navigation techniques [16].

References

1. Lumelsky, V.J., Stepanov, A.A.: Dynamic path planning for a mobile automaton with limited information on the environment. IEEE Trans. Automat. Contr. **31**, 1058–1063 (1986)
2. Sankaranarayanan, A., Vidyasagar, M.: A new path planning algorithm for moving a point object amidst unknown obstacles in a plane. Proc. of the IEEE Int. Conf. Robot. Autom. **3**, 1930–1936 (1990)
3. Sankaranarayanan, A., Vidyasagar, M.: Path planning for moving a point object amidst unknown obstacles in a plane: a new algorithm and a general theory for algorithm development. Proc. of the IEEE Int. Conf. on Decision and Control **2**, 1111–1119 (1990)
4. Kamon, I., Rivlin, E.: Sensory-based motion planning with global proofs. IEEE Trans. Robot. Autom. **13**, 814–822 (1997)
5. Noborio, H.: A path-planning algorithm for generation of an intuitively reasonable path in an uncertain 2-D workspace. Proc. of the Japan–USA Symposium on Flex. Autom. **2**, 477–480, (1990)
6. Noborio, H., Maeda, Y., Urakawa, K.: A comparative study of sensor-based path-planning algorithms in an unknown maze. In Proc. of the IEEE/RSI Int. Conf. on Intelligent Robots and Systems **2**, 909–916 (2000)
7. Ng, J., Bräunl, T.: An analysis of bug algorithm termination. (2007, to appear)
8. Kamon, I., Rivlin, E., Rimon, E.: TangentBug: a range-sensor based navigation algorithm. J. Robot. Res. **17**(9), 934–953 (1998)
9. Lumelsky, V.J., Skewis, T.: Incorporating range sensing in the robot navigation function. IEEE Trans. Syst. Man Cybern. **20**, 1058–1068 (1990)
10. Noborio, H., Urakawa, K.: Three or more dimensional sensor-based path planning algorithm HD-I. Proc. of the IEEE/RSI Int. Conf. on Intelligent Robots and Systems **3**, 1699–1706 (1999)
11. Noborio, H., Nogami, R., Hirao, S.: A new sensor-based path-planning algorithm whose path length is shorter on the average. Proc. of the 2004 Int. Conf. Robot. Autom. **3**, 2832–2839 (2004)
12. Laubach, S.L., Burdick, J.W.: An autonomous sensor-based path-planner for planetary microrovers. Proc. of the IEEE Int. Conf. on Robot. Autom. **1**, 347–354 (1999)
13. Magid, E., Rivlin, E.: CautiousBug: a competitive algorithm for sensor-based robot navigation. Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems **3**, 2757–2762 (2004)
14. Bräunl, T.: Embedded Robotics, 2nd edn. Springer, Berlin Heidelberg New York (2006)
15. Latombe, J.C.: Robot Motion Planning. Kluwer, Dordrecht (1991)
16. Choset, H., Lynch, K., Hutchinson, K., Kantor, G., Burgard, W., Kavarki, L. and Thrun, S.: Principles of Robot Motion: Theory, Algorithms and Implementations. MIT, Cambridge, MA (2005)