1. Bug 1 is an exhaustive search algorithm. It circumnavigates any obstacle it encounters and as such looks at all possible choices before making a decision. The plus side of this is that it has a predictable performance overall. The trade-off is that it takes a longer time to find a potential path. Bug 2 is a greedy search algorithm. It goes around an obstacle until it returns to the m-line (direct line to the goal). As such it is faster than Bug 1 and will usually outperform it. However, the trade-off is that it’s performance can be unpredictable, in fact, if the m-line has more than 3 interactions with an obstacle, then Bug 1 will outperform Bug 2.

2. In order to modify Bug 2 so that it can work without perfect positioning, we can use a method called boundary following.

Algorithm:

1. Robot follows the m-line (direct line to the goal) to the goal until it encounters an obstacle.
2. When an obstacle is encountered, begin to go around the obstacle.
   1. Create a value called dmin which will hold the value of the shortest distance observed so far between the sensed boundary of the obstacle and the goal
   2. Create a value called dleave which is the shortest distance between any point in the currently sensed environment and the goal
3. Continue to go around the obstacle until dleave < dmin, in which case terminate the boundary following.
4. Continue going towards the goal in a direct line and Repeat Steps 2 and 3 if an obstacle is encountered.
5. Terminate Bug 2 Algorithm once goal is reached.

3.

a.

1. Define a set number of vertices you will create as well as the maximum length between two vertices. Define the starting point as the initial position of PR2’s arm and the goal as the desired final position.

2. Randomly select an initial vertex.

3. Randomly select another vertex and see if a path exists between this vertex and an existing vertex that is closes to it. The path must go through free space and be within the maximum allowed path distance. If such a path exists, then create a path and by doing so a new vertex is added to the tree. If no such path exists, then randomly select another vertex.

4. Repeat step 3 until the minimum number of vertices have been created.

5. Add the start and goal and vertices if they are not already part of the tree.

b. Each configuration represents a path that goes through free space between a vertex and the nearest vertex to it.

c. We can simply check and see if the line created by a new configuration intersects with a previously existing configuration. If so, then this configuration is not feasible. If not, then this configuration can be added to the tree.

d. As stated in part (a), we can randomly generate vertices until a valid path between another vertex is found. We can also modify our program to be more biased by adding a probability that a vertex is generated in a specific area. This may help us better find a path between start and goal.

4. Let us assume RRT has already been implemented and a path has already been computed. Let us say that the path was computed using the A\*star algorithm.

Pseudo-Code:

Closed list -> Vertices that make up the path from start to goal provided by A\*

Vertex robot -> the vertex the robot is currently at (the starting vertex to begin with)

For(i=0; i<size of closed list - 1; i++)

{

Vertex current -> vertex from closed list at index ‘i’

Vertex next -> vertex from closed list at index ‘i+1’

If(No obstacle between ‘current’ vertex and ‘next’ vertex)

{

Draw path from ‘current’ vertex to ‘next’ vertex

robot = next

}

Else

{

Call A\* method and give current closed list

/\*When giving the closed list to A\*, instead of starting from the original start location, A\* need only be performed from the current vertex ‘robot’

Accept the new closed list from the A\* method and continue

/\* The new closed list will start from Vertex ‘robot’ and hence the i-value of the loop must be changed such that is starts from the beginning of the new closed list\*/

i = -1

}

}

If this algorithm is followed it should find a path between start and goal even if there are unknown or changing obstacles.

Pros: This algorithm does not create unnecessary new configurations and works with the originally computed tree.

Cons: This algorithm may take a while to find an eventual path between start and goal as it needs to run A\* star every time it encounters an obstacle on its original path.