Path Planning Algorithm for Package Deliverer*

Brandon Feng, Tommy Rogers, Seungjae Lee

Abstract— **This paper addresses the problem of robot-driven package delivery. Robots are now used extensively for moving and delivering objects, whether they are deployed in Amazon warehouses or traversing sidewalks to deliver DoorDash orders. Oftentimes, these robots have limited fuel and weight capacities. Consequently, not only must the robots be able to reach goal locations without crashing, but they must be able to path-find optimally given these weight and fuel constraints. This paper describes the use of the Dijkstra and Bug2 algorithms in determining optimal, shortest paths that will allow the robot to deliver its required packages undamaged. Experimental results show the efficacy of the combined algorithms in allowing the robot to perform its given delivery task.**

I. Introduction

With the field of online retail growing rapidly, companies such as Amazon, Alibaba, many more are constantly looking for ways in which they can become more efficient in order to quickly and effectively deliver products. As a result, robots are becoming increasingly prevalent in warehouse environments as a means of transporting merchandise. Warehouses are ever changing environments and therefore obstacle avoidance is crucial in creating a successful robot. In addition, robots face constraints in carrying capacity, complicating path optimization.

This paper seeks to address these two main challenges and create a robot that will pick up and deliver packages of different weight in an optimal manner to limit total distance travelled while taking into account that the robot has limited capacity. The robot will receive a list of packages with a specified start location, end location, and weight for each item. Similar to most real life examples, such as warehouses, obstacles are ever changing and are unlikely to be fully known by the robot. Therefore, as obstacles are encountered on the optimized route, the robot must be capable of avoiding obstacles and returning to the desired path.

This paper will be structured as follows: Part II will describe previous work that is related to this paper. Part III will discuss the problem statement for this paper. Part IV will describe our implementation and part V will show our experiments and testing. Finally, Part VI will conclude by discussing the successes and limitations of this project.

II. Related work

Within any implementation of a delivery robot, path planning is vital for successful optimization and completion of the task. Simmons, Goodwin, Haigh, Keonig, and O'Sullivan [1] at Carnegie Mellon University created an office delivery robot, Xavier, with the purpose of delivering files to a variety of locations within a building. Simmons *et al.* [1] used a modified A* search algorithm to create a sequence of paths using topographical maps of the environment. When identifying the optimized path, Simmons *et al.* [1] recognized locations of possible obstructions, such as a closed door, and added to the given path length the additional length of the path given an obstruction, weighted by the probability that the obstruction is encountered. Due to potential actuator error, path planning also took into account the potential for the robot to miss a turn or stray from the proposed path. The shortest overall path was then chosen for the robot. Later on, researchers at Carnegie Mellon updated the path planning of Xavier to incorporate Partially Observable Markov Decision Process (POMDP) to estimate the location of the robot at any given moment [2]. Koenig and Simmons [2] found that the use of POMDPs outperformed the A* search algorithm which was previously used for Xavier in terms of efficiency and successful outcomes.

Simmons *et al.* [1] allowed Xavier to receive real time requests and alter path planning accordingly to optimize distance travelled. Goals with lower priority or greater distance would be placed on hold when more convenient tasks were given and the paused tasks would be resumed later. Coltin, Veloso, and Ventura [3] updated the approach of Simmons *et al.* [1] by allowing the robot to take web based commands regarding a variety of possible tasks, including guiding visitors, delivering coffee, or reciting an audio message [3]. These requests could be given with multiple time frame requests such as "as soon as possible", a specified time, or a time frame window [3]. The scheduling agent then created a schedule which would take into account time requests and would create the most efficient path which would result in no overlapping tasks and optimal efficiency [3].

As the workspace of our deliverer focuses on environments with unknown dynamic obstacles, obstacle avoidance is essential for successful delivery of packages to their destination. Numerous approaches have been researched to implement reactive local planning, which avoids obstacles and reaches towards a goal with only local information. Some of the extensively studied algorithms are bug algorithms and potential fields algorithms [4,5]. In potential fields algorithms, a goal and obstacles are treated as attractive force and repelling force, respectively. Thus, a movement of a robot is decided by the relative strength of attractive and repelling forces. In the bug algorithm, a robot moves towards a target until it detects a nearby obstacle with sensors. Then, it uses information of an

*Final Project for COSC 81 Fall 20. Dartmouth College

Brandon Feng is an undergraduate student at Dartmouth College majoring in computer science and mathematics. (brandon.y.feng.22@dartmouth.edu)

Tommy Rogers is an undergraduate student at Dartmouth College majoring in computer science. (thomas.w.rogers.23@dartmouth.edu)

Seungjae Lee is a B.E. candidate at Thayer Engineering School at Dartmouth College with focus on computer engineering. He graduated Colby College with B.A in Computer Science and Applied Mathematics. (Seungjae.Lee.TH@dartmouth.edu)

obstacle such as edges to move around it and move towards a target. Some of the variations of bug algorithms that have been studied are Bug1 and Bug2 [6], DistBug [7], and TangentBug [8].

More recent work on delivery robots has been focused on multi-robot delivery. Mathew, Smith, and Waslander's work implemented a two robot system, one of which was a truck filled with packages and the other a quadrotor aerial delivery robot [9]. The two worked cooperatively to deliver a series of packages in an optimal fashion [9]. This paper will focus on a single robot implementation and will not discuss a multi-robot implementation.

III. Problem Statement

A single, autonomous, differential-drive robot (modelled after the Husarian ROSbot 2.0) moves within a map. The map is a two-dimensional occupancy grid consisting of free space and static obstacles. The map also consists of various "locations", some of which are "recipients" and some of which are "warehouses". Each warehouse contains some set of packages, each of which are to be delivered to some specified recipient. The goal for the robot is to deliver the packages of all the warehouses to their designated recipients such that its distance travelled is minimized. That being said, the robot is small and has limited capacity. As such, it will likely have to make multiple trips back and forth to the various warehouses in order to deliver all the packages. Thus, the robot will have to factor in which set of packages to pick up and deliver while planning how to optimize its travel distance.

The assumption is that the robot has a high level understanding of the map, in that it knows the (x, y) Cartesian coordinates of the recipients, warehouses, and its own initial pose. Consequently, it knows the Euclidean distance between each of these different locations as well as their orientations with respect to one another. Moreover, the robot knows the set of packages and their respective states (if they have been delivered, are currently being carried by the robot, or are waiting in a warehouse). Finally, the robot has a laser range scanner for use in local path-finding.

Consequently, there are two problems that must be addressed in order for the robot to achieve its goal of delivering all the packages:

1. *Search:* The robot should find the optimal sequence of actions, such that each action is either moving to a different location or picking up/dropping off a set of packages. Optimality is measured by the distance the robot travels (rotations and picking up/dropping off packages do not entail any cost).
2. *Local Planning:* Once the robot determines an optimal sequence of actions, it must complete them using local planning to move from one location to another. It knows the orientation of the target location given its current location and is equipped with a laser scanner but has no other knowledge of paths or obstacles.

IV. Algorithm and Implementation

To solve two subproblems stated in the previous section, we used Dijkstra's algorithm to find the optimal path and Bug2 algorithm for local planning and obstacle avoidance.

A. Dijkstra's Algorithm

Since we are optimizing for distance travelled, we use Dijkstra's algorithm. We use Dijkstra's rather than A* because the search is not done continuously in real-time, but rather, performed once before the robot begins its movement. Thus, speed is not of the essence and we save ourselves the trouble of finding a suitable heuristic.

The search problem is modelled as follows: each state holds the state of the robot (its location and the packages it holds) as well as the state of each of the packages (if they have been delivered). Each state also holds the action to be performed next. Each action entails the robot either moving to a different location or picking up a set of packages (TRAVEL or PICKUP). Which action the robot takes at each state is dependent on a kind of finite state machine. It is assumed that if the robot moves to a warehouse then it will pick up a set of packages next. Similarly, if a robot just picked up a package, it is assumed it will move to a new location next. It is also assumed that if a robot moves to a recipient location then it will drop off the packages belonging to that recipient. Finally, cost is measured by the Euclidean distance between locations. If the robot does not move to a different location (which means it is picking up a set of packages), then no cost is entailed. The goal state is, of course, having all the packages delivered.

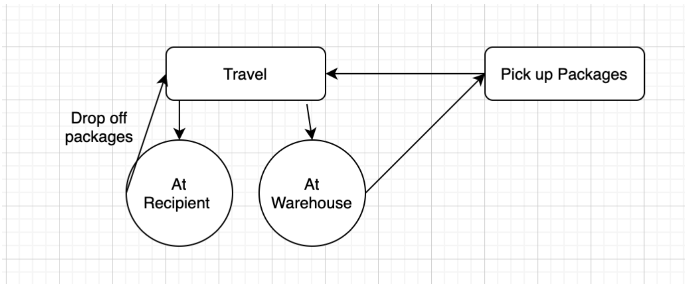

Figure 1. Flowchart of actions of the deliverer

The search problem is implemented using two classes: Node and Search.

1. *Node*: Node essentially models the aforementioned state. It holds information about the robot's location, the packages belonging to the robot, the set of delivered packages, and the action to be taken. It also holds information about the total cost of the node, the parent node for use in backchaining, and a comparator function for use in implementing the priority queue for Dijkstra's.
2. *Search* : Search implements Dijkstra's algorithm using Node. The main methods are as follows:
   - *goal_reached*(): boolean that states whether goal state has been reached. Goal state is reached when all the packages have been delivered.

- *get_distance*(): returns the Euclidean distance between 2 locations. Used for calculating the cost of each state.
- *deliver_packages*(): updates the set of delivered packages as well as the packages held by the robot (and consequently, the robot's remaining capacity).
- *find_all_subsets*(): used to find all subsets of packages the robot can pick up at a given warehouse.
- *get_neighbors*(): finds neighboring states of parent state according to the diagram described above.
- *backchain*(): used to output the final path. Makes use of the Node class, as each node stores its parent node.
- *search*(): follows the general pseudocode for Dijkstra's. It makes use of the heappush and heappop functions to implement its priority queue.

B. Bug2 Algorithm

Our general approach for obstacle detection and avoidance was to implement a Bug2 Algorithm using LiDAR to sense potential obstacles. The robot will travel linearly between locations, and when an obstacle is detected by the LiDAR, a PD controller is activated and the robot will follow the obstacle's wall to its right at a specified distance. Once the line between locations is once again reached, the robot will reorient to face the goal location and will continue on the line.

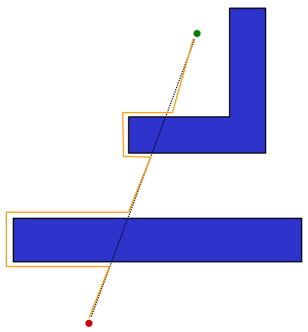

Figure 2. Example of Bug2 Algorithm. *(from COSC81 Lecture Slides)*

The Bug2 algorithm and PD Controller are implemented using a single class: Movement. The main methods are as follows:

- *create_path*(): implements the search class and given information regarding the packages and their respective destinations, finds an optimized path of locations for the robot.
- *main*(): handles the robot movement using a finite state machine. The state is updated as a result of the robot's pose and its proximity to obstacles. It makes use of helper functions for basic mathematical functions such as distance_to_point() and angle_to_2pi().

C. Finite State Machine

Our implementation of the local path planning and Bug2 algorithm uses finite state machines. At the initial step, with the given starting location and the goal location, the algorithm computes the angle that faces the goal location from the starting location. Then, the robot rotates until it reaches the computed angle. The robot moves towards the goal until it detects an obstacle in front of it. Upon detection, it first drifts leftward to follow the wall of the obstacle. Once the robot moves away from the tangential line, it uses PD controller algorithm to move around the obstacle until the robot reaches the tangential line. Upon reaching the line, the robot rotates until it faces the goal and moves forward until it either reaches the target or faces another obstacle. Table 1 describes parameters used in this algorithm. Figure 3 shows a visual representation of the finite state machine.

TABLE I. PARAMETERS AND DESCRIPTION

Parameters	
FREQUENCY	Frequency of publishing velocity message (Hz)
VELOCITY	Initial linear velocity in x for *move_straight* state (m/s)
VELOCITY2	Initial linear velocity in x for *obstacle_turn, obstacle_move* states (m/s)
ERROR1	Error bound for distance between two points (m)
ERROR2	Error bound for distance between a point to a line (m)
ANG_ERROR	Angular error bound for $\theta_{goal} - \theta_{robot}$ (rad)
K_P	Potential gain for PD controller
K_D	Derivative gain for PD controller
SET_DISTANCE	Target distance from wall to the right side of robot for Bug2 (m)
OBSTACLE_DIST	Distance for detecting obstacle in front (m)

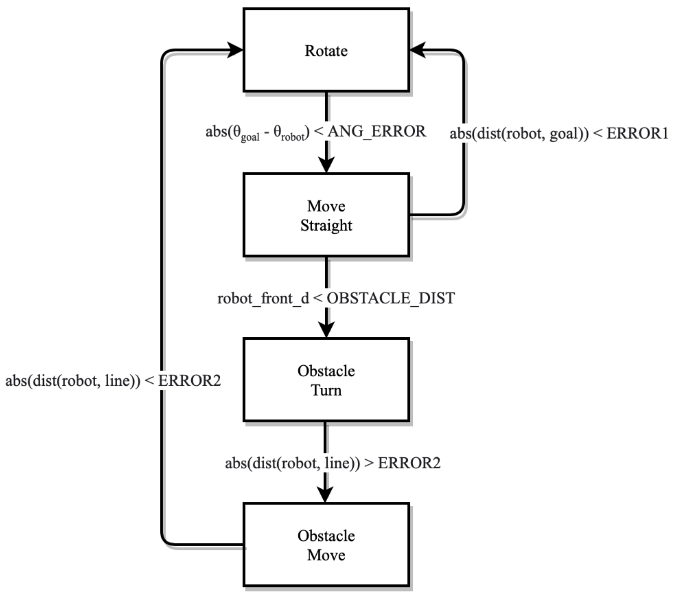
Rotate
abs(θgoal - θrobot) < ANG_ERROR
abs(dist(robot, goal)) < ERROR1
Move Straight
robot_front_d < OBSTACLE_DIST
abs(dist(robot, line)) < ERROR2
Obstacle Turn
abs(dist(robot, line)) > ERROR2
Obstacle Move

Figure 3. Finite state machine for local path planning

The actual states used in the algorithm are the following:

- *Rotate*: The initial state of the robot. If the difference of angles (θ_{goal} - θ_{robot}) is less than pi, it moves with angular velocity of the difference in angles. If the difference is greater than π, it moves with angular velocity with the difference - 2π. This implementation allows the robot to move faster when the difference is large and slower as the robot approaches θ_{goal}, which allows the robot to accurately reach the goal within the error bound. Once the robot reaches θ_{goal} within ANG_ERROR, the state changes to *move_straight* state.

- *Move_straight*: In this state, the robot moves straight towards the goal. The robot moves with linear velocity in x of VELOCITY * dist(robot,goal) / dist(start,goal), which allows the robot to reach the target accurately as it slows down near the target. Once the robot reaches the target, the algorithm updates the next goal location, and the state updates to *rotate* state. If the robot detects an obstacle in front, then the state changes to *obstacle_turn* state.

- *Obstacle_turn*: This state is the preliminary state for Bug2 Algorithm. The robot turns leftward and moves away from the tangent line. The obstacle moves with the angular velocity of ANG_VEL and the linear velocity of VELOCITY2. Once the distance of the robot to the tangent line is larger than the user given error bound (ERROR2), the state changes to *obstacle_move*.

- *Obstacle_move*: This state is the Bug2 algorithm state. The robot moves around the target using the PD controller algorithm to compute the angular velocity. The error is computed by taking the difference of SET_DISTANCE to the measured distance. For the linear velocity, the robot moves VELOCITY2 when the dist(robot,line) is bigger than 1. If not, the robot moves by VELOCITY2 * dist(robot,line) so that the robot does not overshoot from the line. Once the distance of the robot to the tangent line is less than the error, the state changes to *rotate* state.

V. Experiment

Based on trials and errors, we were able to find values (Table 2) for our parameters so that we can achieve high accuracy of package delivery. To test our algorithm and chosen parameters, we made a map that has static obstacles of various shapes such as circles, rectangles, stars and other random polygons. Then, we made sets of packages (start location, goal location, weight) that the robot needs to deliver, shown in Table 3. The goal of the experiment is to check if our system chooses the optimal path with a given set of packages and if the robot correctly navigates with a given path while avoiding obstacles.

TABLE II. PARAMETER VALUES USED IN EXPERIMENT

Parameter Values			
FREQUENCY	100 Hz	ANG_ERROR	0.000278π radians
VELOCITY	2.5 m/s	K_P	6
VELOCITY2	0.5 m/s	K_D	2
ERROR1	0.05 m	SET_DISTANCE	1 m
ERROR2	0.02 m	OBSTACLE_DIST	1.2 m

TABLE III. TESTED SET OF PACKAGES WITH ROBOT CAPACITY = 5

#	Start	Goal	Weight
1	(16,10)	(4,4)	1
2	(16,10)	(9,0)	2
3	(4,5)	(-16,0)	3
4	(4,5)	(5,-15)	4

For packages in Table 3, our algorithm was able to find the optimal path shown in Table 4. Upon the close scrutiny of the result, we see that the robot decides to pick two packages at once to reduce repetitive travels, which shows that the algorithm did choose the shortest path possible. Then, the local path planning algorithm was able to effectively navigate through obstacles and pick up and deliver packages to their locations. Figure 4 shows the path that the robot took to pick up and deliver packages while avoiding obstacles. Both results show that our algorithm was able to find an optimal path and navigate without any obstacle collisions.

TABLE IV. OPTIMAL PATH FOR THE PACKAGES

Optimal Path			
Start	**Goal**	**Action**	**Packages**
(0,0)	(4,5)	+P4	P4
(4,5)	(5,-15)	-P4	None
(5,-15)	(16,10)	+P1, +P2	P1, P2
(16,10)	(9,0)	-P2	P1
(9,0)	(4,4)	-P1	None
(4,4)	(4,5)	+P3	P3
(4,5)	(-16,0)	-P3	None

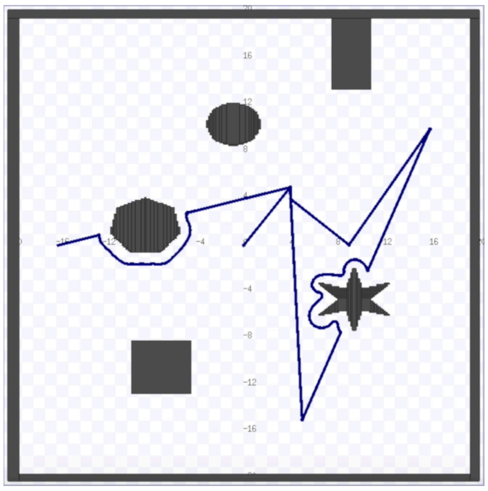

Figure 4. Path taken by the robot to deliver packages.

Although our algorithm worked effectively for our main experiment, we also observed that it can fail due to the limit of Bug2 Algorithm. In Figure 5, we demonstrate a case where our algorithm fails to deliver a package to destination. The robot attempts to deliver a package from (0, 0) to (14, 17); however, it faces a rectangular obstacle near the top right corner. Then, it uses Bug2 to move around the obstacle and eventually follow the wall. Once the robot reaches the tangent line, it goes towards the starting point again, and thus, repeats the same process infinitely.

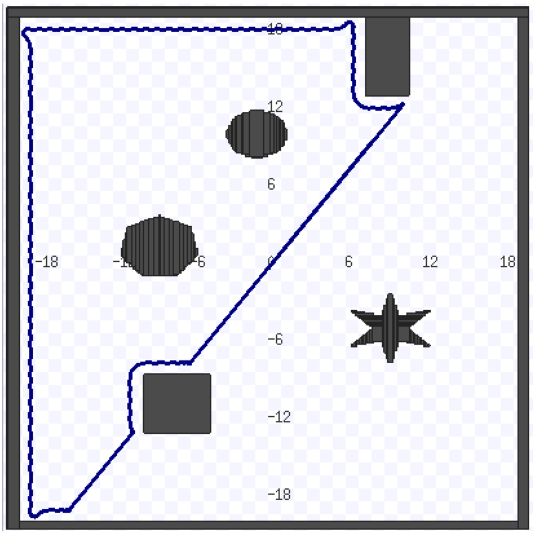

Figure 5. Failed case of Bug2 algorithm

VI. Conclusion

In this paper, we have introduced a method for efficient package delivery using robots. This was done by using a Bug2 algorithm implementation for local path finding and Dijkstra's for global path planning. Consequently, we were able to develop a robot that is able to deliver packages while travelling a minimal distance and avoiding obstacles during travel.

That said, our implementation is relatively limited. For example, we do not deal with dynamic obstacles, which is obviously insufficient for a real-world deployment. Our local path-finding can be improved, perhaps by introducing mapping in order to "remember" paths and obstacles and avoid inefficient Bug2 obstacle traversals. In addition, we could also explore different reactive system such as modified versions of bug algorithm and potential field method to encompass dynamic environment. Moreover, while we optimize for distance travelled, there are other sources of robot fuel expenditure in the real-world, such as having to rotate or pickup/drop off packages. Package weight can also possibly affect fuel expenditure. In all, there are many other variables and challenges to consider to develop a truly efficient working prototype.

References

[1] Simmons, Reid, et al. "A layered architecture for office delivery robots." *Proceedings of the first international conference on Autonomous agents*. 1997.

[2] Koenig, Sven, and Reid Simmons. "Xavier: A robot navigation architecture based on partially observable markov decision process models." *Artificial Intelligence Based Mobile Robotics: Case Studies of Successful Robot Systems* partially (1998): 91-122.

[3] Coltin, Brian, Manuela Veloso, and Rodrigo Ventura. "Dynamic user task scheduling for mobile robots." Workshops at the Twenty-Fifth AAAI Conference on Artificial Intelligence. 2011.

[4] Khatib, O., Real-Time Obstacle Avoidance for Manipulators and Mobile Robots,. 5, 1986. The International Journal of Robotics Research: p. 90- 98.

[5] J.C. Latombe, Robot Motion Planning (Kluwer, Dor- drecht, 1991)

[6] Lumelski, V.J., A. A. Stepanov,, Dynamic Path Planning for a Mobile Automaton with Limited Information on the Environment. IEEE Transactions On Automatic Control,, 1986. 11.

[7] Kamon, I., E. Rivlin, Sensory-Based Motion Planning with Global Proofs. IEE Transactions on Robotics and Automation, 1997. 13((6)).

[8] J. U. Duncombe, "Infrared navigation—Part I: An assessment of feasibility (Periodical style)," *IEEE Trans. Electron Devices*, vol. ED-11, pp. 34–39, Jan. 1959.

[9] S Mathew, Neil, Stephen L. Smith, and Steven L. Waslander. "Optimal path planning in cooperative heterogeneous multi-robot delivery systems." *Algorithmic Foundations of Robotics XI*. Springer, Cham, 2015. 407-423.