Comparison of Dijkstra & A* Path Planning Algorithms for Simulated Forced Path Re-Planning

Felipe Valdez[1], Jason Dekarske[2], and Jonathan Dorsey[3]
Department of Mechanical and Aerospace Engineering
University of California Davis
Davis, CA 95616 USA
[1]fdvaldez@ucdavis.edu, [2]jdekarske@ucdavis.edu, [3]jtdorsey@ucdavis.edu

Abstract— Real life scenarios present complexity in unexpected obstacles that must be considered with implementation of a path planning scheme in mobile robotics. In this paper, the merits of Dijkstra and A* Search algorithms in path re-planning are compared. The former depends on finding the shortest path between nodes in a graph (explores all possibilities), while the latter uses a heuristic function which gives priority to nodes that are better than others. The use of Simultaneous Localization and Mapping (SLAM) is crucial in this scenario because the robot must discover these unexpected obstacles. In this work, several performance metrics are compared with respect to the following: online re-planning, sensor imperfections, and changes in the environment inside a GPS-denied environment typical for an academic building with dynamic obstacles.

Keywords- autonomous mobile robots, dijkstra's algorithm, A* search algorithm, path planning, ROS

I. Introduction to problem, motivation & past work

Automation is reaching further into the lives of ordinary people by the day. For every second of work delegated to an autonomous system, we see an increase in productivity for more advanced work done by people. This is especially true in an office-like setting. Simple tasks like mail or food delivery can be accomplished by mobile robots to increase efficiency and reduce error in the logistic system. In this paper, we aim to design a robot that will accomplish the task of moving about an academic building where the floorplan is only partially known. That is, the robot knows, generally how to navigate to a specific room, but may not know who or what is blocking its path. These hazards may not be as mundane as they first appear; a heavy robot would cause considerable damage to its interactors if not cautious. Whether it's the people themselves or the various objects around the building, damages caused by the robot could have drastic consequences. Our robot will introduce two similar path planning algorithms to inform the actual design of itself. That is, we will test these algorithms and vary the system setup to improve performance based on these metrics.

Considerable effort has been expended to design similar robots across a variety of industrial, manufacturing, and agricultural sectors in order to exchange laborious manual tasks for technological solutions, with the promise of increasing productivity and reducing operating costs. However, the inclusion of mobile robots that can freely move from one point to another is still in its infancy, and many challenges are yet to be faced. One of the many advantages of deploying mobile robots in real world environments across all developmental sectors is that, compared to humans, robots can operate in non-air conditioned spaces with exposure of chemicals without having to worry about potentially life threatening conditions. One of the main challenges in this field is the autonomous navigation in unstructured or semi-structured environments. Whereas some industrial or manufacturing operations can design facilities built around the needs of mobile robots, many applications of robotics still require the robot to operate in a dynamic human environment with little pre-existing knowledge of environment it has been put in.

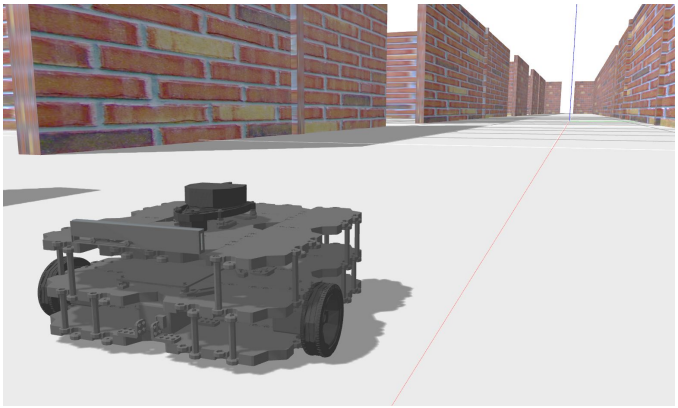

Figure 1. Semi-structured environment of our proposed robot simulation inside a school building.

While performing inspection, mapping is an important step when a mobile robot wants to explore the unknown environment, and the robot should be able to move and find the pathways to avoid obstacles and achieve the goal. SLAM (Simultaneous Localization and Mapping) is a commonly used strategy to determine the goal position with respect to the

current position of the robot in an area that has not been recognized previously [2-3]. In application, a mobile robot will gather information about the environment via distance sensors (LIDAR, in this case) and construct a point cloud corresponding to the configuration of the environment. This point cloud is assembled with knowledge of the current state of the robot and dynamics which integrate other states. In the end, SLAM produces a map of the environment and the location of the robot as seen in figure 2.

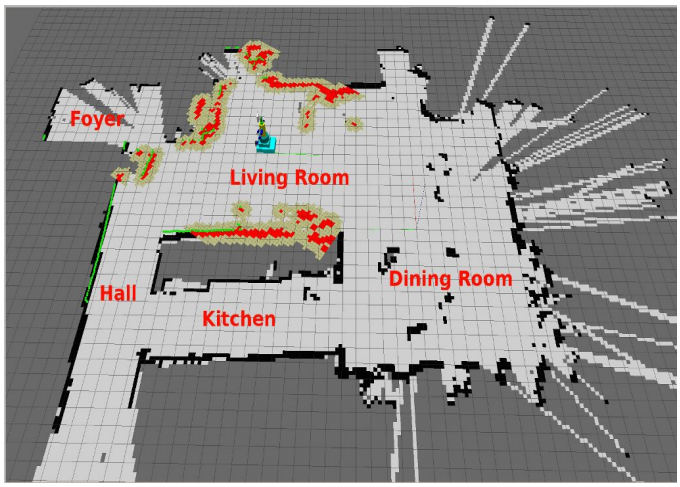

Figure 2. An example of a SLAMed environment. The robot (blue) traversed the home while mapping walls and obstacles (black) [4].

Even while computing and manufacturing technologies have greatly increased the ability of small and inexpensive robots to perform processor intensive tasks like SLAM, the ability of a robot to identify where it is with respect to its surroundings is only a partial solution to a more encompassing challenge. For advances in SLAM techniques and sensor technology to be of any great utility, robots must be provided with a robust and tractable means of generating a path from a known starting point to a desired final location, and they must handle unexpected events or obstacles which commonly occur in real life. Throughout decades of robotics research, this need has driven the development of many different path planning algorithms and methods, all of which approach the problem from a slightly different perspective or make different fundamental assumptions to achieve the task of planning and replanning typically with the objective of finding a way to improve specific characteristics of existing techniques. This variety in the fundamental principles, methods, and assumptions for robotic path planning provides an extremely diverse toolbox from which a variety of known, proven, or even experimental algorithms can be selected [5].

Planning a path for navigation can be cast as a search problem on a graph. A number of classical graph search algorithms have been developed for calculating least-cost paths on a weighted graph; two popular ones are Dijkstra's algorithm (Dijkstra 1959) and A* search (Hart, Nilsson, & Rafael 1968; Nilsson 1980). Both algorithms return an optimal path, and can be considered as special forms of dynamic programming. A* operates essentially the same as Dijkstra's algorithm except that it guides its search towards the most promising states, potentially saving a significant amount of computation [6].

As will be explained later in this paper, Dijkstra & A* planning algorithms have been chosen because of their distinct algorithmic differences. Without diving into great detail, we are interested in comparing the performance of these path planners along with the situations and parameters which limit or enhance their ability to successfully or optimally complete an assigned task.

Although many methods have been used to plan the path of a mobile robot, some benefit from saving data in the world map to use if it is found that the map has changed since planning time. The following methods have been chosen as potential strategies for overcoming blocked paths when the environment is mostly known.

A. Dijkstra's algorithm

Dijkstra's algorithm is one of the simplest algorithms. Starting from the initial vertex where the path should start, the algorithm marks all direct neighbors of the initial vertex with the cost to get there. It then proceeds from the vertex with the lowest cost to all of its adjacent vertices and marks those with the cost to arrive via itself if this cost is lower. Once all neighbors of a vertex have been checked, the algorithm proceeds to the vertex with the next lowest cost as seen in figure 3 [7].

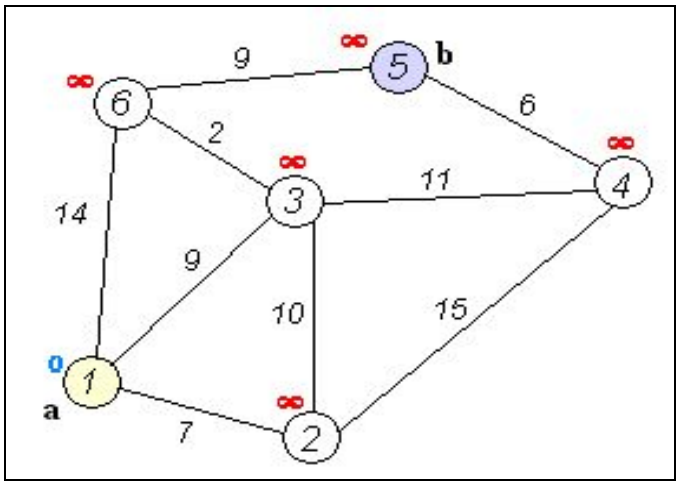

Figure 3. Dijkstra's algorithm to find the shortest path between *a* and *b*. It picks the unvisited vertex with the lowest distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller. [8].

The major disadvantage of the algorithm is the fact that it does a blind search which consumes more resources and consequently, time. Another disadvantage is that it cannot

handle negative edges. This leads to acyclic graphs and most often cannot obtain the right shortest path.

Yizhen Huang [9] presented an improved Dijkstra shortest path algorithm. The improved algorithm introduces a constraint function with weighted value to solve the defects of the data structure storage, such as redundancy of space and time. The number of search nodes is reduced by ignoring reversed nodes and the weighted value is flexibly changed to adapt to different network complexity. The simulation experiment results showed that the number of nodes on search shortest path and computation time were significantly reduced. Thus, the improved algorithm can be faster to search out the target nodes.

Faisal Khamayseh [10] presented an efficient and improved heuristic algorithm for finding shortest paths between a given source and destination using candidate subgraphs in a weighted directed graph with weights as a function of edges. The proposed heuristic determines the candidate subgraphs by marking the ancestors of the destination node using reverse matrix representation to enable backward traversals. Comparing with current conventional algorithms, this heuristic improves the shortest path algorithm significantly.

B. A* SEARCH ALGORITHM

Incremental search algorithms discover the best route through a maze by minimizing a certain cost function. In most cases, the maze is set up on a grid with discrete locations separated by a constant displacement. Further, travel from one grid box to an adjacent one yields a cost of 1 while an obstacle may not be traversed. These simplifications bode well for a robotic navigation problem because the complexity of a real life environment would increase online computation time.

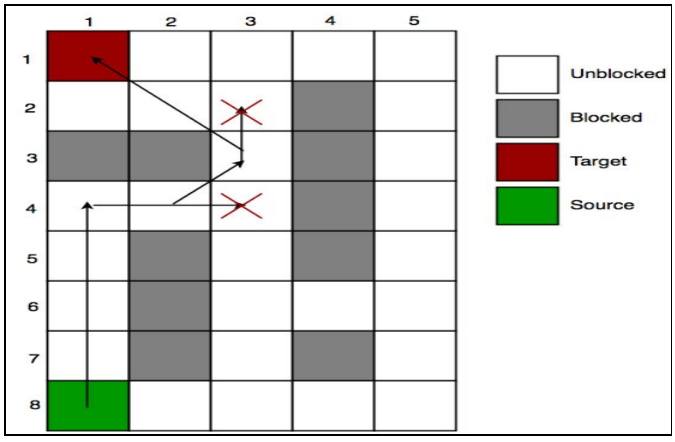

Figure 4. A* search algorithm makes the most intelligent choice at each step. Hence you can see that algorithm goes from (4,2) to (3,3) and not (4,3) (shown by cross). Similarly the algorithm goes from (3,3) to (2,2) and not (2,3).

Depending on the environment, A* algorithm might accomplish search much faster than Dijkstra's algorithm. These algorithms are based in graph theory, which connects nodes and assigns costs to such nodes. In the path planning implementation of graph theory, nodes represent locations on a map. Incrementally, the nodes are scanned and the costs are calculated. A* is a variant of dijkstra's algorithm which adds a heuristic function which artificially weights the graph in order to prioritize solutions which are computed to be in the direction of the goal point. This behavior is dictated by the heuristic function. The functional benefit of this approach is that it vastly reduces the number of nodes that need to be considered for the final path. However, since this path is calculated from the start point, if the grid changes in any way, the path must be completely recalculated [11].

The main drawback of A* algorithm and indeed of any best-first search is its memory requirement. Since at least the entire open list must be saved, A* algorithm is severely space-limited in practice, and is no more practical than best-first search algorithm on current machines.

Several modifications have been suggested ever since the first version of the sequential A* algorithm was introduced. Evidence suggests that there has been a remarkable progress in the performance of the A* algorithm due to these; however, there is room for its improvement. Mentioned below are the contributions of well-known A* algorithms proposed for different search domains [12].

1. Sequential A* Algorithm

Liang Zhang et al. in [13] proposed a novel hybrid method to find the optimum path planning for mobile robots. The A* algorithm and the genetic algorithm form the basis for this method. The workspace of the robot in this paper is previously known and consists of a grid map of its indoor environment. At first, the method find the shortest path by the A* algorithm using Manhattan distance as heuristic function. Next, it uses the genetic algorithm to optimize the path found.

Barnouti et al. [14] built a GUI application system that implements the A* search algorithm, which addresses the pathfinding problem to find the shortest path between source and destination. The problem of pathfinding in commercial computer games has to be solved in real-time, usually requiring memory and CPU resources. The algorithm is tested by using images that represent either a map that belongs to strategy games or a maze. The system performance is tested using 100 images for each map and maze. The overall performance of the system is acceptable; for more than 85% of the images, the shortest path between the source and destination could be found.

2. Parallel A* Algorithm

Luis Henrique and Luiz Chaimowicz in [15] aimed to improve the A* algorithm performance by implementing the bidirectional search algorithm as a parallel version for

robotics, digital games, and DNA alignment. The sequential version of New Bidirectional A* (NBA*) uses two processes that execute in a sequential manner to find the path. In this paper, the researchers proposed PNBA* (Parallel New Bidirectional A*). In PBNA*, both search processes execute in parallel and that enhances its performance. They compared their algorithm with A* and NBA* for fifteen puzzles and grids pathfinding scenarios. They used random mazes with uniform and non- uniform costs. As a result, the execution time decreased significantly when using PNBA*.

Phillips et al. [16] presented PA*SE, a parallel implementation of A* search algorithm, another version based on a relaxed independence check that allows a trade-off between higher parallelization at the cost of optimality, and finally a parallel implementation of weighted A*. The experimental results on up to 32 processors demonstrated a linear speedup over A* and weighted A*.

In our proposed approach, rather than applying the algorithms (Dijkstra's and A*) to each current robot state, it will be applied in a planning stage in order to develop the path and to recalculate in the event of an unexpected obstacle or avoid being trapped in local minima.

II. Specifics of the problem to be studied

A. Research Question:

To lay a firm foundation about the direction and focus of this paper, it is valuable to establish a primary research question which encapsulates the overarching objectives & philosophy. Our research question can be summarized as follows.

How do different path planning algorithms such as Dijkstra & A* perform when forced to re-plan a path in realtime?

Determining a well thought out and testable solution to this primary inquiry, prompts deeper fundamental questions whose answers provide substantial insight into the scenario we propose testing. Two such questions could be defined as follows.

Which environmental conditions arise where one methodology provides significant improvement over the other?

What changes to physical design of the robot benefit the performance of one or both planning algorithms.

B. Embodiment & Situatedness:

One of many challenges involved in simulating robotic systems is approaching the problem from the perspective of a computer scientist, rather than a roboticist. This can include undue emphasis on algorithm performance, theoretical complexity, speed of implementation, computational power, and so on. While these are valid concerns in the domain of robotics, they do not reflect the physical presence and being of an actual robotic system. As such, special care must be taken to address *embodiment* and *situatedness* of the simulated robotic system.

The qualities of embodiment and situatedness infer concepts such as a robot's knowledge of its location in an environment, its ability to collect environmental data through realistic sensors, and its experiences as it interacts with the world. These qualities are vital to the formulation of this paper since they provide the dividing line between a mere computer science/technology problem versus a robotics problem which investigates the paradigms of physical robotics.

III. The robot simulation

In a commercial setting, the development of a robot would undergo many separate stages of research, development, simulation, prototyping, and iteration. This workflow permits a methodical and strategic approach to fully explore the design space and algorithmic options available to a new robotic system. However, the condensed timescale available during a single academic quarter, only permits a cursory investigation into the research question stated above.

Under this time constraint, computer simulation of a physical systems becomes an attractive methodology to pursue since it allows for design and software changes to be rapidly tested and iterated over, increasing the effective throughput over tradition pen and paper design studies.

For the scale of the topic being reviewed, simulation is an appropriate tool for obtaining data and insight into the behavior of systems, architectures, and/or algorithms which we wish to investigate. However, effective simulation must go beyond the face value of the research topic and carefully develop a scenario, system model, and an environment from which meaningful data can be measured and quantified. The goal here is to prove that simplifying assumptions are outweighed by the rapid insight into the design and development which is gained by computer simulation.

To ensure that our research outcomes are founded on a solid footing, we propose the following objectives along with a detailed review of the tools, assumptions, simulation process which were set up and designed to guarantee the result collected reflect a realistic interpretation of the research questions.

A. Simulation Objectives:

The first step in any investigation is to define high level outcomes. By being explicit in the definition of the problem or scenario being researched, a more directed and meaningful simulation to be constructed whose output data will hopefully provide far more applicable and relevant results.

For the simulation presented in this paper, the following statements are intended to define the key behavior(s) of interest.

a) Primary: The main objective of this simulation is to observe and quantify the behavior of a mobile robot when operating under the Dijkstra and A* algorithms when forced to replan an initial path due to partial knowledge of an environment and/or contact or interaction with unexpected obstacles.

b) Secondary: Identify sensor parameters which improve or degrade the performance of either path planning methodology via factors that are possible to control from the perspective of robot embodiment through it's design and implementation from different domains including mechanical, electrical, and software.

B. Simulation Platform:

In general, the term simulation has infinitely many meanings and computational interpretations depending on the field of application which the simulation is presented. For applications within a single domain, such as dynamics and control, the term simulation is intended to mean the process of numerically solving the time evolution of equations that govern a process or the behavior of a specified system. For this intended application, this 'type' of numerical simulation will provide insight into the expected behavior of the physical system.

However, in mixed multi-domain systems like robotics (eg. electronics, mechanical, communications, control, software, computer architecture, etc), simulation becomes a much more involved procedure. While it is often possible to isolate selected domains of interest or features of a robotic system for simulation, the very process of isolating a single region from the entire system posses the potential problem of decoupling what might actually be highly complicated coupled system. This, of course, depends on the problem of interest and overall effect of decoupling highly integrated systems by applying simplifying assumptions.

For this paper, it was felt that the most realistic treatment of robotic simulation could be achieved using Robot Operating System, commonly known as ROS. Unlike Matlab or Python which are effectively general purpose coding languages with a popular following for numerical simulation and visualization, ROS is a completely dedicated robot ecosystem, capable not only of simulation (with integrated physics engines), visualization, code development and validation, but also of actual code deployment on physical robots. Additionally, ROS supports third party plugins which vastly expand its native functionality.

As such, it was felt that ROS would provide the best environment for simulating a path planning robot as close to real world conditions as possible.

ROS is a development platform, framework, and community specifically targeted for robots and robotics applications. The objective behind its development was to create a single unified computing structure which provides a layer of abstraction off of the hardware and away from any single programming language, thus allowing users with different hardware and software needs to seamlessly integrate their own projects and system into the ROS framework. In effect, the ideology behind ROS is to be as general purpose as possible while simultaneously incorporating a wide list of standardized built-in tools and functionality.

As a development tool for robotic systems, ROS is unequaled. It is an open source product of Stanford Artificial Intelligence Laboratory which currently enjoys large commercial success, and possesses a very active and informed online community which is always answering commonly encountered questions. However, the biggest selling point for ROS as the simulation package of choice for this paper is the number of standard features and tools which ship with the default installation, effectively making ROS plug-n-play.

Even with all of these benefits already built into ROS, the main attraction for our group to select ROS as the simulation platform is related to the standardized structure through which ROS processes and sends commands to the robot controller, called the Navigation Stack. The structure of the Nav. Stack effectively allows for certain blocks of code to be simply and easily interchanged. These block are termed "plugins."

Since the objective of this paper is to evaluate several different path planners, the ability to quickly develop and drop in different path planning plugins into the navigation stack which in turn seamlessly integrates with an existing model of the robot and its environment provide an exponential improvement over the alternative of writing and developing our own robot/planner integration simulation which may require tedious coding and debugging of path planner specific functionality. Additionally, it is important that the algorithms are fully optimized (thanks to the open source community), so that they do not bottleneck the process unknowingly.

C. Tasks to Implement:

a) Path Generation: Inherent to the proposed simulation is the requirement for robust paths. Here, the comparison of path generation techniques, Dijkstra & A*, is the backbone of the simulation. Both methodologies require independent implementation in our simulation package to create a fair trial between the approaches.

b) Path Tracking: For the robot to make use of the generated paths, from the previous task, requires a techniques by which the robot can track the proposed paths to complete the desired objective. While important, the requirements of the task leave a lot of flexibility to choose the exact mechanisms through which path following is completed, so long as it is compatible with the primary path replanning objective. Built-in dynamics of the robot package will be used to ensure a simulation that is closer to reality.

c) Obstacle Detection and Avoidance: This task is the primary mechanism with which the robot informs itself that

the existing path it is following is no longer valid. The ability to sense an obstruction is pivotal to the completion of the simulation. Later, the proposed sensor suite will further cover the fidelity and limitations we intend to impose on the simulation.

d) Simultaneous Localization and Mapping (SLAM): For the robot to acquire a sense of situatedness, it must first determine its 'state' inside the environment. This task implements techniques which locate the robot in its current surrounding while simultaneously mapping this new environment for possible future use or external analysis.

D. Environment:

The environment of the simulation is a significant feature that we have complete control over. As such, there are a number of design factors within the simulation environment which directly correlate fundamental assumptions to the quality of the simulated results. However, since no simulation can perfectly model a realistic system or handle all assumptions, the following topics indicate the predominant philosophies and underlying assumptions that we desire to build into our simulation environment.

a) Environmental Knowledge: For this simulation, the intent is to investigate the mobile robot in unknown or partially known environments. The reason for this being is that partially known maps are far more likely to be encountered in actual physical testing. By placing this constraint on the robot simulation, the algorithms being tested provide far more utility and realism. These in turn provide a better correlation to the robots expected performance, if actually tested for physical deployment in the real world, which is not uncommon for many projects to be co-developed from both simulated and real world testing.

b) Map Design & Complexity: For testing a mobile robot that builds and navigates paths in a partially unknown environment, the philosophies motivating the design, layout, complexity, and implementation of the simulation map are of key importance. The primary design philosophy proposed by the team is to develop a single map with incredible detail and complexity seen in figure 5. The reasoning behind this decision states that if the work space of the mobile robot is large enough and varied enough, by merely selecting different starting and ending points for different trials, the simulations will include a component of variability and randomness. Things like rooms, doors, dead ends, and hallways, create uniqueness among tests. This variance between testing trials should help to explore possible edge cases, which demonstrate unexpected failure modes in the model or performance of the algorithm as well as generalize the simulation results and provide a measure of its robustness within which the results can be reasonably interpreted. In the case of the mobile robot replanning, depending on the mechanism of the algorithm driving the replanning, a particular set of start/end points may naively force the robot into a local minima, or infinite loop from which the robot might not escape. Conversely, given another set of start/end points, the robot may *never* become locked into an inescapable situation, even though the that failure mode still exists.

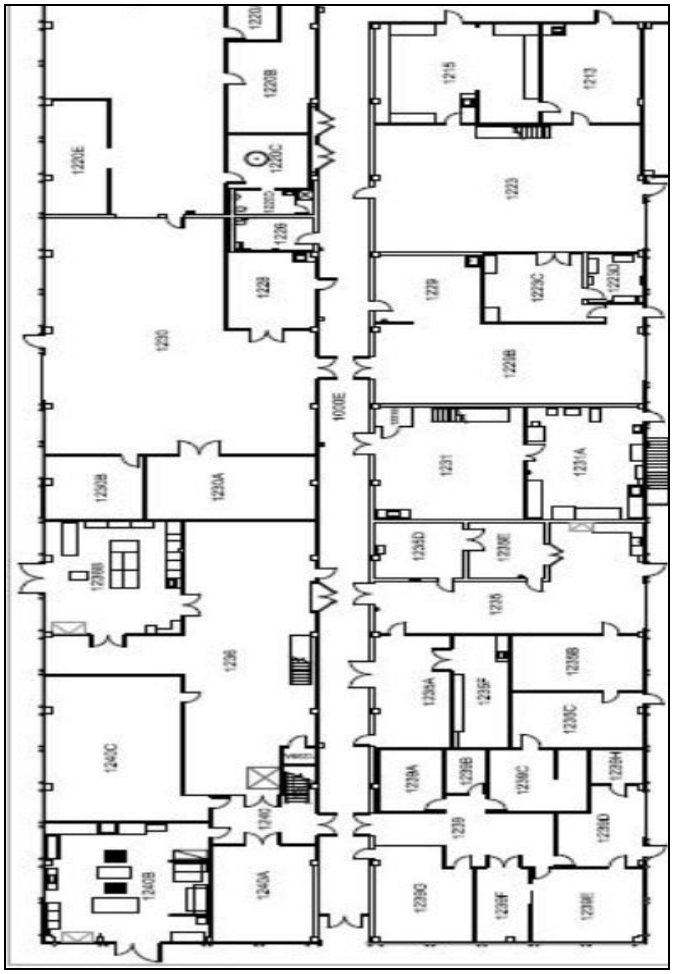

Figure 5. Proposed floor map of Bainer Hall at UC Davis. The dark zones will be populated with rooms and corridors with typical office furniture. Dynamic obstacles include doors at the specified locations and random obstacles in hallways.

c) Static Maps & Obstacles: The ability for the environment to change (room remodelled, door closed, moving obstacle...etc) is the primary means by which the algorithm determines it must replan the existing path. Therefore, introducing basic static changes into the environment is paramount to the successful design of an accurate simulation (figure 6). We will randomly close a percentage of doors to deny the robot's initial path and force it to replan.

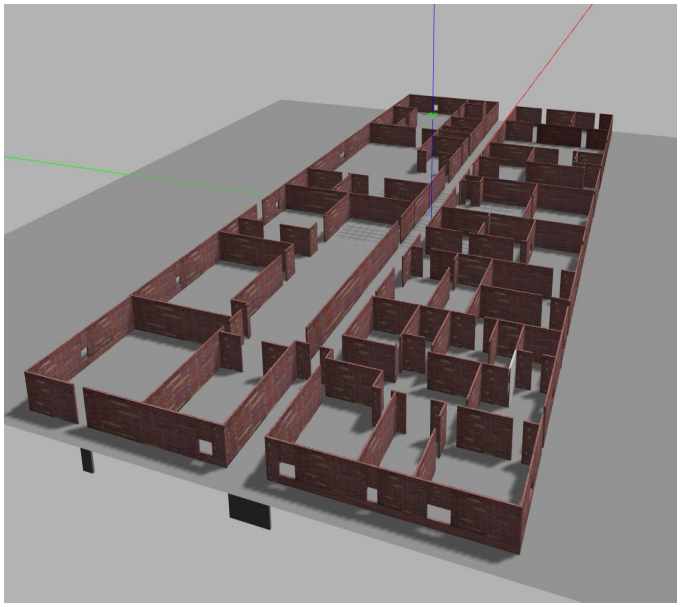

Figure 6. 3D Gazebo scenario for simulated models of the mobile robot and Bainer Hall. From these requirements we can extract two factors that need to be addressed: how to effectively locate the robot in an indoor environment, and how to navigate safely while avoiding collision with other entities in its path.

E. *Sensor Suite:*

One of the most important aspects of the simulation is the addition and modeling of realistic sensors that integrate with the robotic system. These sensors must provide the robot with its *only* source of external information and should reflect the realistic limitations of actual sensors such as limited effective range, finite sample rates, physical accuracy, and latency. In addition to these requirements, these sensors must be implemented in such a way as to obey the physical laws and principles governing the operating sensor. Below is a list of the proposed sensors which will be integrated with the robot for this simulation.

a) LIDAR: This sensor works by emitting laser pulses and measuring the time elapsed before its reflection from the laser off an object returns to the sensing unit. Given the speed of light, and the time of flight, the distance to the reflecting object can be computed. In this simulation, it is proposed that specifications and performance our robot's Lidar system, conform to existing technology such as the Velodyne VLP-16 [17] listed below.

- Effective Range 100 meters
- Accuracy +-3cm typical
- Horizontal Field of View 360 degrees
- Vertical Field of view 30 degrees
- Rotational Rate 5-20Hz
- Single Return Mode ~300,000 points/sec & Double Return Mode ~ 600,000 points/sec

Essentially, the laser scanner is capable of sensing 360 degrees that collects a set of data around the robot to use for SLAM (Simultaneous Localization and Mapping) and Navigation.

b) Inertial Measurement Unit (IMU): This sensor estimates translational speed so robot can calculate and estimate its current pose (orientation). On the map, the robot updates its odometry information with the encoder and the IMU sensor, and measures the distance from the pose of the sensor to the obstacle (wall, object, furniture, etc.). In this simulation, it is proposed that specifications and performance our robot's IMU system, conform to existing technology such as the IMU-MPU 9150 listed below.

- 3 Axis Accelerometer: range of ±2g, ±4g, ±8g and ±16g
- 3 Axis Gyroscope: range of ±250, ±500, ±1000, and ±2000°/sec
- 3 Axis Magnetometer: 3-axis Hall-effect magnetic sensor
- I2C Interface (Max Freq 400kHz)

c) Camera: This is an Intel® Realsense™ R200 technology, which benefits from a long range HD colour camera with an infrared sensor for movement detection. We used this default simulation camera for video rendering purposes only.

F. Proposed Simulation Procedure:

a) Simulation Setup

- Programmatically select start & end points.
- Run Dijkstra & A* algorithms to acquire 'initial' map.
- Randomize simulation environment (open or close doors..etc).

b) Initial Map Baseline (non-altered map)

- Simulate each algorithm on a non-altered map to characterize the initial performance of the path without map changes.

c) Replan Only Baseline (Altered map)

- Simulate replanned configuration of map, for each algorithm, as a control/baseline for scenario where robot must recalculate its initial path.

G. Performance & Analytical Metrics:

Given that the purpose of our investigation will be to compare methodologies, we will need an objective, quantifiable means of assessing the performance of each strategy. Our proposed success criteria are as follows:

a) Obstacle Avoidance Characteristic Metric: While the objective of the simulation is to replan paths to avoid hitting obstacles, the theoretical possibility of collisions is still non-zero. As such, the average velocity will be recorded

which takes into account the amount of time the robot is moving slowly or not at all

b) Control - Run Time vs Replan -Run Time: One of the most interesting metrics to consider is the total runtime of a forced replan simulation to the total runtime of a non-altered control for each trial with respect to each planning algorithm used.

This metric would allow comparable baselines between the same algorithms to be made by comparing the total run time of a pair of simulation trials. If simulation parameters are kept standardized, between control and trial runs and the two algorithms, the overall runtimes (separated per algorithm) can be aggregated into an average approximation of how much longer the forced replanning simulations took when compared to their control runs.

e) Failure to Complete: One of the final metrics to test is whether the simulation failed to complete in a reasonable time. While the cause for this failure may be the result of a variety of reasons, logging these failures and notating which algorithm that the simulation was running can provide information about the robustness of the planning methodologies used, as well as insight for future recommendations for each algorithm to increase utility and robustness.

Other measures categorizing why possible failure to complete errors took place may be developed (time permitting) as dictated by the observed performance of the simulation.

f) Sensitivity to Sensor Variability: As mentioned in the objective section of the proposed simulation, the secondary goal of this simulation will be to investigate how changing parameters related to the embodiment and situatedness of the robot improve or degrade the overall performance.

It is worth noting that sensors, as a predominant feature of the robots ability to interact with the world its in, should be examined to determine how differing resolutions or sensory horizons impact the ability of one algorithm over the other.

The signals produced by the modeled sensors generally do not reflect the real world as they are calculated and synthetically created. Real world signals usually contain variances from the ideal signal, which we will refer to as noise. Noise may be caused by a wide range of sources, e.g. variations of detector sensitivity, environmental variations, transmission errors or quantization errors. The overall performance of a sensor is ultimately limited by the noise that is added to the signal. Therefore, introducing noise to the simulation system is crucial to test, evaluate and improve the robustness of the tested algorithms and procedures and to increase their performance.

H. Simulation Packages:

The framework of ROS seamlessly allows large distributed problems commonly seen in complex robotics to broken down into small manageable pieces of code (called nodes). Since ROS espouses a language-agnostic platform, ROS nodes can be written in either C++ or Python. This provides a variety of programming and development options and easily allows blocks of functional code from external sources to be implemented thus aiding to encourage a quick development time.

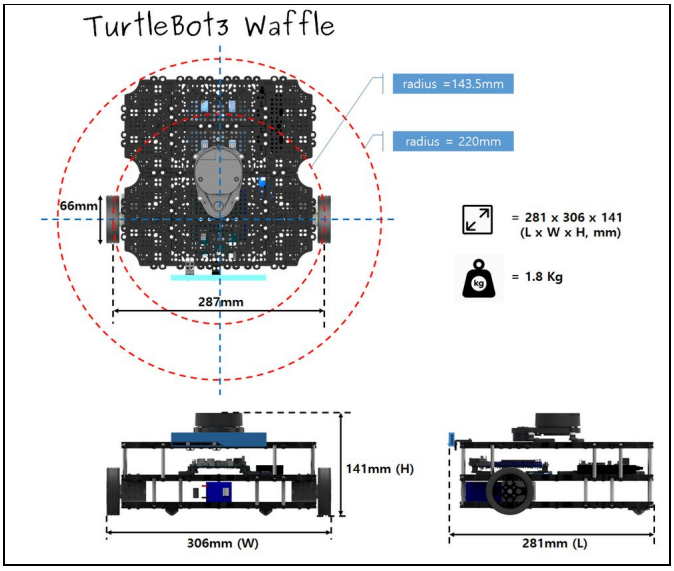

Figure 7. Turtlebot 3 size and weight specifications.

Table 1

Turtlebot3 Waffle specifications. It has an extensive resource base for use with ROS [19].

Items	**Waffle**
Max translational velocity	0.26 m/s
Max rotational velocity	1.82 rad/s (104.27 deg/s)
Max payload	30kg
Size (L x W x H)	281mm x 306mm x 141mm
Weight (+SBC+Battery+Sensors)	1.8kg
Expected operating time	2h
Actuators	Dynamixel XM430-W210

To further help develop inside of the ROS framework, we proposed to use the popular physical robot package “TurtleBot 3” shown in figure 6. This external package provides a far more gentle entry point for newcomers to ROS thus enabling minimal effort to be spent on developing standard robotic hardware and functionality from scratch.

In addition to this, one of the many benefits of developing inside of ROS is the ability to couple the simulation of a real time rigid body model of a robotic system inside of a 3D development environment known as Gazebo, which is an open-source software well integrated with ROS. Gazebo is a part of the Player Project [20] and allows simulation of robotic and sensors applications in three-dimensional indoor and outdoor environments. It has a Client/Server architecture and has a topic-based Publish/Subscribe model of interprocess communication. Gazebo will be used directly to simulate an academic building floor plan containing desks as obstacles and doors and walls as passageway interferences while RViz will be used to visualize the state of the robot, performance of the algorithms, and to debug faulty behaviours, and to record sensor data online.

H. Simulation Assumptions:

In order to zero in on the required functionality within the simulation, it is helpful to codify the assumptions about what is and is not included in the simulation. The following list of assumptions set the stage for the environment, capabilities, and limitations which have been baked into the code for this simulation.

1. GPS-denied Indoor Environment:

Mobile robot must rely on on-board sensors to localize itself, map its environment, and to estimate the robots current position and orientation

2. Realistic Error Simulation:

Physics engine has been selected on the basis of its ability simulate realistic sensor data, latency, odometer drift, bias accumulation, and slippage errors

3. No constant unicolor nor flat surfaces are present:

If the surroundings are poor in features (unicolor surface for cameras, or flat surface for the LiDAR), the matching algorithms can fail in detecting a match (in cases where no/too few features are captured).

4. Simulation & Reality Equivalence (1:1)

Simulation time and computational expenditure scaled to match real-time robot operation on physical hardware (even though simulation may run faster due to available computational resources)

IV. Techniques & algorithms (map search, decision, return to goal, etc.)

One of the most important aspects about the ROS framework is that it supports many common robotics software packages, control and planning algorithms, and general simulation and deployment functionality while simultaneously retaining the ability for the end user to create their own environment, planner, or tools as desired. With the additional support of a very active online community, ROS enables even simple simulation to incorporate non-trivial and advances techniques, programs, and capabilities with very little upfront cost to the user.

This section will document the techniques, algorithms, and other details which lay the foundation for the simulations executed for this project.

A. Dijkstra's Algorithm: This algorithm implements a "best-first-search" procedure for exploring a graph and selecting the best/lowest cost next step for the path to take to reach the desired goal.

For the purposes of our simulation, Dijkstra's Algorithm is used to by the global path planner responsible for creating the most feasible path from point "A" to point "B" given the current state of the environment.

B. A* Search Algorithm: This algorithm is functionally identical to Dijkstra in terms of its role within the simulation (eg. global path planning). The primary difference of A* is the addition of a heuristic function whose purpose is to "inform" (aka guide) the planner in an attempt to get the direct the planner to select future steps in the direction of the goal. The performance of A* will vary with the formulation of the heuristic used as well as the size of the search space.

C. Dynamic Window Approach Local Planner: The DWA local planner is the piece of software responsible for tracking the path created by the global planner as well as providing obstacle avoidance planning. This objective is performed by creating a viable search window and finding an optimal solution such that the robot can reach an intermediate goal in a short amount of time while remaining collision free, and accounting for the limitations in velocity and acceleration of the robot, given its dynamics.

D. ROS Navigation Stack: For the sake of continuity, efficiency, and easy interchangeable navigation capability, the ROS framework has elected to create a "navigation stack" or structure through which all navigation task must adhere. The benefit of this becomes readily apparent with the utilization of global planner plugins. The set structure of the navigation stack means that path planning modules termed 'plugins' can be quickly implemented once the version of any desired path planner has been implemented to fit the structure of the navigation stack. Effectively this highly modular sub-framework enables a quick and robust means to develop and implement new plan

E. Gmapping & SLAM: Gmapping package in ROS provides the tools to create a 2D base map by using laser and odometry data. The method is called SLAM (Simultaneous Localization and Mapping). SLAM algorithm creates a map of an unknown environment by performing localization in the area. After the unknown area has been mapped and a robot knows its position related to the map it can perform route planning and navigation. As a result, SLAM algorithm is a vital component for performing auto-navigation for the robot.

The laser needs to be equipped with a stationary and horizontally-mounted laser range finder. SLAM is also important function for avoiding obstacles along robot's path.

F. AMCL: Adaptive Monte Carlo Localization (AMCL) is the methodology the simulation utilizes to localize itself once provided a map of the current soundings. This is an important task for the robot to accurately and robustly reach its end objective. AMCL is a stochastic approach to localization which endeavors to lower the uncertainty of a robots possible position by creating a search space of probable guesses, and then comparing these to measurements and adjusting its future guesses to be as close to the actual position of the robot as possible. Unlike EKF SLAM which implements an 'Extended Kalman filter,' AMCL uses a particle filter instead of Gaussian distribution as the basis for its estimation.

A natural consequence of the statistical nature of AMCL's estimation and access to finite computational resources mean that uncertainty in sensor readings possess the potential to significantly degrade the localization accuracy of the algorithm, as will be discussed later.

G. Costmap: This package helped us to provide a configurable structure that maintains information about where the robot should navigate in the form of an occupancy grid. The costmap uses sensor data and information from the static map to store and update information about obstacles in the world as seen in figure 8.

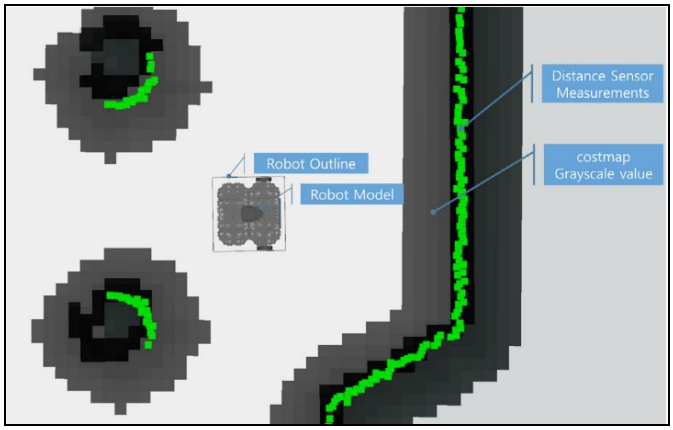

Figure 8. Costmap representation showing the sensor measurements, robot's model and outline, and costmap value.

V. SIMULATION RESULTS

The objective of this simulation is to mimic the behavior of the system (as much as is allowed by the available models) and to produce 'simulated' outputs which provide deeper experimental understanding of the problem under consideration as a proof of concept or refinement of existing knowledge, as means to offset physical prototyping costs as well as to speed up the production timeline.

For the simulations performed throughout this paper, we can classify two general bins in which our simulation results can be categorized. The first bin is the raw simulation data, from which post-processing and statistical analysis can distill relationships and behaviors which may not be readily seen from simple graph. The second category of simulation is related to visual observation made from RVIZ in which the operation and the real time state of the robot are visualized, within ROS. Both of the results help to test hypotheses and determine valid conclusions by allowing characterization of the behavior in question. While graphical and statistical data perform admirably to numerically quantify results, this analysis might miss more subtle issues which are far easier to identify via visualization of the simulated process. In the latter case, the innate human ability to sense and detect visual patterns proves of great service when visual analysis reveals inconsistency or exhibits behavior counter to the expected performance of the system.

For the analysis of this paper, both visual analysis of the simulation (for troubleshooting and behavioral diagnosis) and numerical analysis are implemented to draw final conclusions about the research question of interest.

A. Visual Observation:

1) A* Wall Hugging: After implementing turtlebot 3 in simulation using different path planners, it became clear that when compared to path planning via Dijkstra's Algorithm, A* has the propensity to drive the robot as close to the walls of the hallway and rooms as possible. When this behavior was noted, it also became clear that the proximity of the robot to wall was forcing the robot to collide with the wall which intern caused the robot initiate a reset procedure. It should be noted that A* creating paths very close to walls fits the profile of A*'s added heuristic which is weighting possible paths that are further from the objective more than possible paths which are closer to the objective. Because of this, if getting closer to the wall will 'generally' decrease the distance between the robot and the goal, A* will tend to take that path.

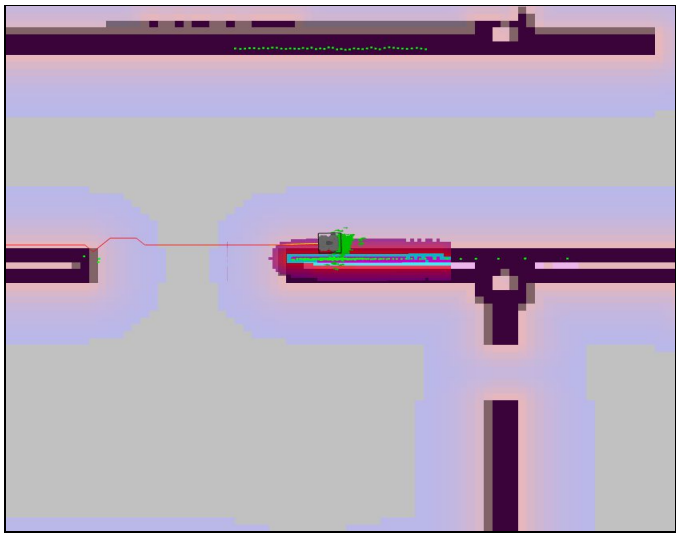

Figure 9. Robot's behavior as "wall-hugging" when being close to a wall and using A* algorithm.

2) Sensor Noise Correlation to Localization Failure: An interesting observation made during simulation runs was a notable failure in the ability of robot to localize itself during certain tests. Since the robot uses AMCL to localize itself inside of a map, the localization process is probabilistic in nature, as explained in a previous section. The failure mode of the AMCL was for the robot's estimated position to jump and oscillate between two or more points which were a significant distance from each other. Obviously this behavior is a consequence of the robots failure to localize. At first, this was assumed to be an isolated event, until subsequent testing with certain parameters proved to repeat the exact same behavior. While mystifying at first, a review of the mechanism behind AMCL quickly aided in identifying the cause of the failure.

As stated above, AMCL is probabilistic approach to localization by means of a particle filter, with the aim of predicting the highest probability distribution of where the robot will be during the next time step. In this sense, any behavior of the robot (extreme speed or acceleration) which renders those predictions invalid degrades the quality of the robots localization. While this failure mode is not uniquely caused by any single attribute or parameter, from running several testing across a spectrum of sensor parameters, we were able to correlate increased sensor noise standard deviation to the inability of the AMCL algorithm to reliably determine the robots actual location within the SLAMed map.

From a theoretical standpoint, this behavior fits the limitations of MCL methods in that, the increase of sensor noise and variability decrease the probability of the localizer correctly identifying the region of high probability for the next time step. Thus as the simulation progresses, error accumulation would create high uncertainty of the robots current position creating a negative feedback loop where in the robot is attempting to predict the region of highest probability while the sensor quality is simultaneously being degraded by sensitivity to noise to which they were previous immune in previous simulations, with lower noise standard deviations.

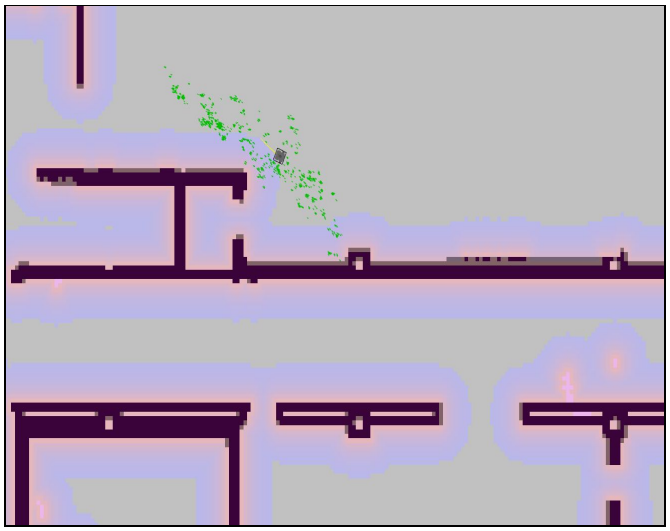

Figure 10. Added sensor noise correlates to localization failure (disperse green dots = localization failure mode).

3) Failure to reach goal: another metric to measure the performance of Dijkstra and A* Search algorithms is the robot's failure to reach its waypoint goals. It was noted from figure 11 that Dijkstra algorithm has a higher failure-to-goal rate, for most waypoints, when compared to A*. While the cause for this failure may be the result of a variety of reasons, it shows the robustness of the planning methodologies used by A* Search algorithm is higher than Dijkstra's when dealing with re-planning scenarios due its implementation of a heuristic function that guides the robot to proceed to a better node as mentioned previously.

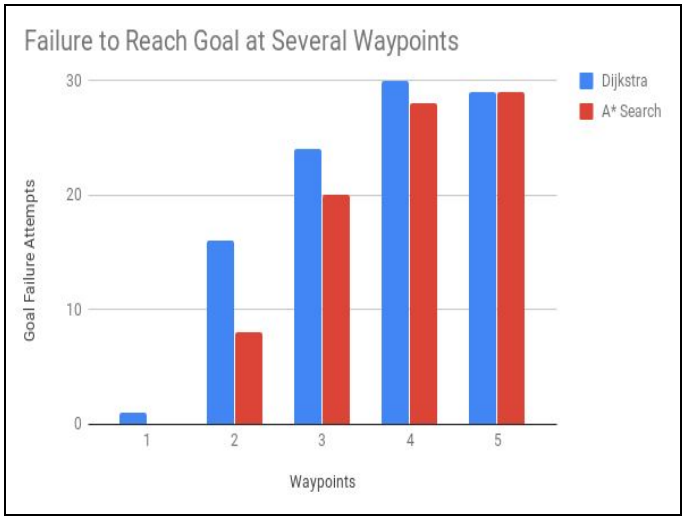

Figure 11. Comparison of failure attempts to reach waypoint-goals between Dijkstra and A* algorithms.

A. Collected Data:

There were two primary metrics that were collected at the end of each run: standardized waypoint deviation and average velocity. These were chosen because they encapsulated the movement behavior of the robot on a by waypoint basis.

Standardized waypoint deviation is the distance travelled by the robot (measured by dead reckoning) compared to the through-wall distance from the start to the goal (Figure 13). We expect that this will always be greater than 1 because the robot will have to travel around obstacles and cannot optimize a straight line path. This means that a greater deviation would signal a harsh course correction, especially if one parameter treatment fairs worse.

Average velocity is the distance travelled by the robot compared to the time it took per waypoint. A higher average velocity means the robot was travelling on an optimized path that did not encounter any walls or near-misses. As described above, DWA ensures the robot avoids obstacles in its local domain and does this by slowing down if too close. Therefore, we expect the average velocity to be skewed left with the mean near the max velocity of turtlebot, 0.26 m/s.

From figure 12, it was surprising to see that Dijkstra's total average distance travelled per waypoint was lower than A* Search since A* implements a heuristic function which is weighting possible paths that are further from the objective more than the ones closer to the objective thus it would make sense it would travel less to get to the goal. However, due to the "wall hugging" behavior as explained previously, the robot sometimes would get too close to the wall, stop, and back up for a few seconds to get unstuck and then keep going to its destination thus adding more to the total distance travelled. Another possible reason to this behavior might be due to the several configuration parameters for the costmap which dictate how much the robot would get close to a wall (wall boundaries).

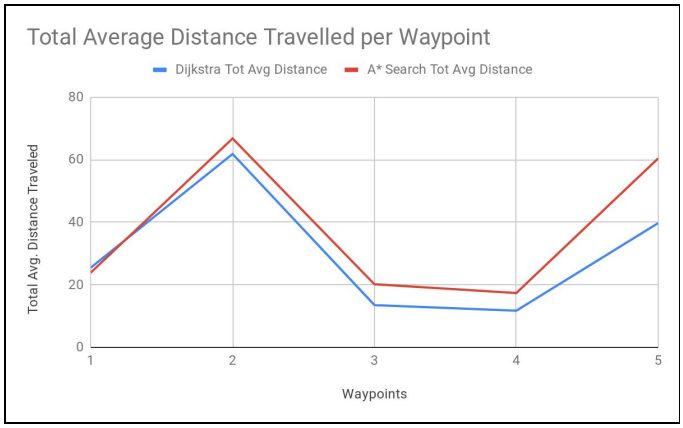

Figure 12. Total average distance travelled by robot for 5 different waypoint goals.

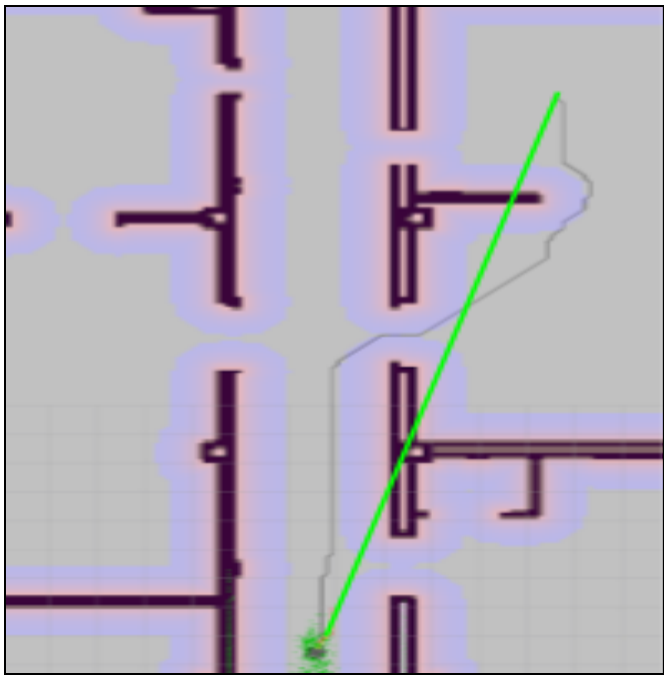

Figure 13. The waypoint deviation is computed by dividing the length of the black line by the length of the green line.

VI. DISCUSSION OF RESULTS

To assess the relative importance of changes to the embodiment of the robot, we'll conduct a multivariate linear regression. This method describes the relationship between several explanatory variables to a response variable. The following summary tables show how the variation in the waypoint deviation and average velocity can be attributed to the embodiment changes that were made to turtlebot.

Table 2

Multiple regression summary table for the average velocity. Low p-values indicate variables that may describe embodiment changes that were critical to robot's performance.

Avg. Velocity	Coefficient	p-value
Intercept	0.177	<0.001
Lidar Range	0.025	<0.001
Open Door Pop.	.005	0.07
Laser Noise	-0.002	0.52
Gyro Noise	-0.003	0.08
Accel Noise	-0.003	0.08
Dijkstra/A*	-0.0042	0.14

Table 3

Multiple regression summary table for standard waypoint deviation.

Deviation	Coefficient	p-value
Intercept	2.33	<0.001
Lidar Range	-0.69	<0.001
Open Door Pop.	-0.03	0.07
Laser Noise	-0.09	0.52
Gyro Noise	0.015	0.08
Accel Noise	0.015	0.08
Dijkstra/A*	0.24	0.14

B. To address the main thrust of the paper: will the turtlebot implementation of A* perform better than the dijkstra implementation, we conduct a t-test over the difference in waypoint deviations and average velocities. Additionally, the regression suggests that there may be a difference in performance due to the range of the robot's lidar sensor so we will test that as well.

Table 4

Velocity and Deviation p-values comparison.

	Velocity p-value	Deviation p-value
A* vs. Dijkstra	0.084284	0.427804
Lidar Range	<0.001	<0.001

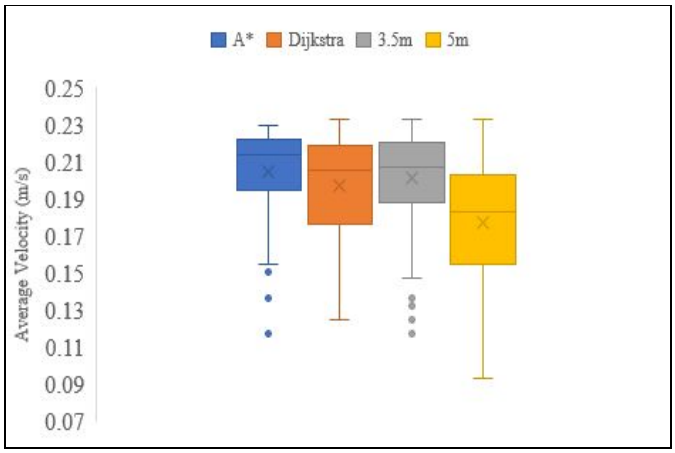

Figure 14. Boxplots of the average velocity. The 5m lidar range treatment shows a lower average velocity than the other 3.5m because it spends more time delocalized.

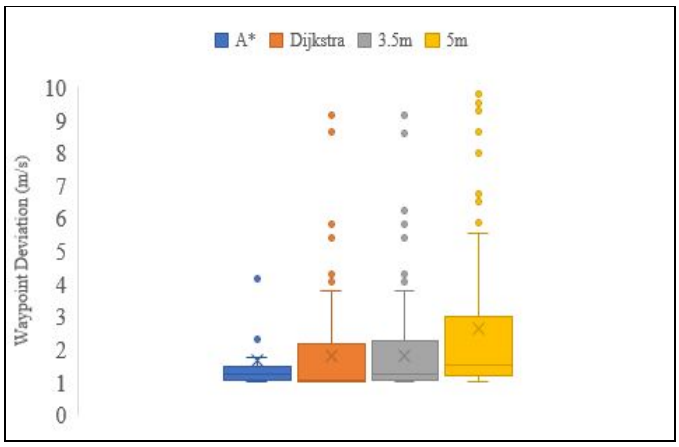

Figure 15. Boxplots of the waypoint deviation. A* shows the smallest deviation because the algorithm plans paths along the wall.

C. The statistics suggest that the performance of turtlebot in this scenario is heavily dependent on the range of the lidar sensor. Systems on the turtlebot rely on its lidar to localize itself in the map. AMCL fuses readings from other sensors to find out where turtlebot is in the global plan. The waypoints were chosen to be points in the middle of rooms, away from anything a 0.5m lidar sensor could reach. This means that turtlebot must rely on odometry and imu, both dead reckoning sensors. Evidently, these sensors are not as good at localising in the global plan as lidar. This is observed in Figure 10, where turtlebot is far enough away from a wall and the AMCL particles are dispersed.

We also see that the p-value for A* vs. Dijkstra velocity is almost significant. This can be explained by the behavior of turtlebot with A*'s heuristic. Because the planner pushes the

plan toward the wall, closer to the goal, turtlebot must slow down to avoid collision.

Finally, we see that the p-value for the changing door parameter is also low. We could have lowered this more by closing more doors in each trial. However, it still shows that changing static obstacles will cause the robot to degrade in performance.

VII. Future work

The primary feature of the simulation presented in this paper is limited scope which it seeks to primarily to investigate the performance and robustness of path planners in an indoor environment with static obstacles designed to force global replanning. While this simulation has provided enough detail for the questions we are seeking to answer, future work on the characterization of the planners would require a much more comprehensive and thorough break down of the global path planning performance and behavior.

One such area of interest which was unable to be incorporated into this simulation due to time limitations is the addition of dynamic obstacles and environments. In such a simulation, using the framework presented above, the navigational tradeoff between the local and global planners as well as the algorithms implemented by these planners becomes of significant importance when discussing the efficiency of a robot to complete an assigned task as well as the robustness of the overall system to unexpected perturbation, localization errors, sensor failures, and navigation of partially known environments.

Another series of future tests could involve the implementation of newer and/or more experimental path planners like D* Lite and Artificial Potential Fields (APF). Since the architecture of many newer path planners are drastically different, each providing its own set of theoretical guarantees and limitation, the ability to simulate scenarios under the direction of these different path planners is a valuable tool for determining the strengths and weaknesses of newer algorithms, especially when incorporating as realistic physic models as possible. This type of simulation would be particularly well suited for robotic systems whose designers anticipate a set of frequently encountered scenarios or objects from which different initial or environmental conditions might prompt undesired or unexpected behavior.

Finally, the most practical area for continued development of simulations, like the one presented in this paper, would be the actually design, fabrication, and deployment of simulated robotic system on to actual hardware for physical testing of a robot (Waffle could be an option). Since the development of most robots is generally application specific and highly expensive for actual commercial systems, the venue of simulation provides designers with insight into the systems behavior as well as providing the infrastructure for debugging or creating firmware which seek to solve an existing problem on currently deployed hardware.

References

[1] El Houssein Chouaib Harik, Combining Hector SLAM and Artificial Potential Field for Autonomous Navigation Inside a Greenhouse, MDPI Robotics Article May 2018

[2] P. Goebel, “Robot Cartography: ROS SLAM,” Pi Robot. [Online]. Available: https://www.pirobot.org/blog/0015/. [Accessed: 04-Feb-2019].

[3] Hani Safadi, Local Path Planning Using Virtual Potential Field, 2007 https://www.cs.mcgill.ca/~hsafad/robotics/

[4] P. Goebel, “Robot Cartography: ROS SLAM,” Pi Robot. [Online]. Available: https://www.pirobot.org/blog/0015/. [Accessed: 04-Feb-2019].

[5] Yang, L., Qi, J., Song, D., Xiao, J., Han, J. and Xia, Y. (2016). Survey of Robot 3D Path Planning Algorithms. Journal of Control Science and Engineering, 2016, pp.1-22.

[6] Dave Ferguson, Maxim Likhachev, Anthony Stentz “A Guide to Heuristic-based Path Planning.” American Association for Artificial Intelligence. [Online]. Available:https://www.cs.cmu.edu/~maxim/files/hsplanguide_icaps05ws.pdf

[7] N. Correll, Introduction to Autonomous Robots, 1st edition, ISBN-13:978-1493773077, 2014.

[9] Huang, Yizhen & Yi, Qingming & Shi, Min. (2013). An Improved Dijkstra Shortest Path Algorithm. 10.2991/iccsee.2013.59.

[10] Faisal Khamayseh, Nabil Arman. Improvement of Shortest-Path Algorithms Using Subgraphs Heuristics. Journal of Theoretical and Applied Information Technology 10th June 2015. Vol.76. No.1

[11] Sven Koeniga, Maxim Likhachevb, David Furcy, “Lifelong Planning A*.” Retrieved February 19 2019, from https://www.cs.cmu.edu/~maxim/files/aij04.pdf

[12] Soha S. Zaghloul, Hadeel Al-Jami, Maha Bakalla, Latifa Al-Jebreen, Mariam Arshad, Arwa Al-Issa. Parallelizing A* Path Finding Algorithm. International Journal Of Engineering And Computer Science ISSN:2319-7242 Volume 6 Issue 9 September 2017, Page No. 22469-22476

[13] Zhang, Liang, et al. ”Global path planning for mobile robot based on A* algorithm and genetic algorithm,” Robotics and Biomimetics (ROBIO), 2012 IEEE

International Conference on. IEEE, California, USA, pp. 1795-1799.

[14] N. H. Barnouti, S. S. M. Al-Dabbagh, and M. A. S. Naser, ”Pathfinding in Strategy Games and Maze Solving Using A Search Algorithm,” Journal of Computer and Communications, p. 15, 2016

[15] L. H. O. Rios and L. Chaimowicz,”PNBA*: A Parallel Bidirectional Heuristic Search Algorithm,” Proceedings of the XXXI Congressa da Sociedade Brasileira de Computaao (CSBC), Brazil, Brasilia, 2011.

[16] M. Phillips, M. Likhachev, and S. Koenig,”PA* SE: Parallel A* for Slow Expansions,” in ICAPS, New Hampshire, USA, pp. 208216, 2014.

[17] Autonomous Stuff.com 2015. [Online]. Available: https://www.autonomoustuff.com/wp-content/uploads/2016/08/VLP-16-Puck.pdf [Accessed Feb. 6th 2019].

[18] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., ... & Ng, A. Y. (2009, May). ROS: an open-source Robot Operating System. In ICRA workshop on open source software (Vol. 3, No. 3.2, p. 5).

[19] “What is a TurtleBot?,” TurtleBot2. [Online]. Available: https://www.turtlebot.com/. [Accessed: 07-Feb-2019].

[20] Player/Stage project. (2016) The Player Project. [Online]. Available: http://playerstage.sourceforge.net/