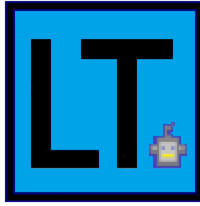

Lev Tours

Dr. Michael Leverington - Sponsor
David Failing - Mentor

Members
Erik Clark
Kyle Savery
Alexis Smith
David Robb
Ariana Clark-Futrell

Final Report

4/26/2021

Table of Contents

1. Introduction…………………………………………………………………………...…2
2. Process Overview……………………………………………………………………….....3
3. Requirements……………………………………………………………………………....3
3.1 Functional Requirements……………………………………………………….....3
3.2 Non-Functional Requirements…………………………………………………..5
4. Architecture and Implementation………………………………………………………....6
4.1 Backend……………………………………………………………………………...6
4.2 Frontend……………………………………………………………………………..9
5. Testing……………………………………………………………………………………...10
5.1 Unit Testing………………………………………………………………………..10
5.2 Integration Testing………………………………………………………………..11
5.3 Usability Testing…………………………………………………………………..12
6. Project Timeline…………………………………………………………………………..13
7. Future Work………………………………………………………………………………16
8. Conclusion……………………………………………………………………………...….16
9. Glossary……………………………………………………………………………..……..17
10. Appendix A: Development Environment and Toolchain…………………………..……..18
11. References…………………………………………………………………………..……...20

1. Introduction

In the field of computer science, mobile robotics are capable of increasingly complex tasks, and the necessary hardware for these tasks has become far more accessible in recent years. As a result, the costs associated with these materials have also decreased, allowing for a greater number of individuals and organizations to take part in the research and development of mobile robotic platforms. Most significantly, educational organizations can give students the opportunity to interact with robotics either from a mechanical or programmable standpoint. To take full advantage of mobile robotics, one of the main goals of our project, “Thirty Gallon Robot Part III, The Smiling Tour Guide”, designed by our sponsor Dr. Michael Leverington, is to demonstrate that an institution's budget for robotics can be further reduced, and used more effectively, by utilizing inexpensive materials. For Dr. Leverington’s robot, one of the most inexpensive components is the thirty gallon barrel that all of the other components are built around. When this project is complete, the robot will be an example of how instructors can get students involved in the world of robotics, which could lead to an expansion of the field.

Beyond computer science the industry of robotics is involved with numerous fields such as mechanical and electrical engineering, medicine, agriculture, and manufacturing. In 2019, the global robotics industry was valued at $62.45 billion [1]. A subset of this industry that is of particular relevance to our project is known as mobile robotics. This field is responsible for creating robots that can move in 3 dimensional space without the need for human assistance. The market size for mobile robotics was approximately $9.34 billion in 2018 [2].

The Thirty Gallon Robot project is in its third year of development. The first team created the foundation for Robot Assisted Tours (R.A.T) by programming the robot to respond to commands from an Xbox controller. The second team built upon R.A.T by creating a way for the robot to generate its own maps by navigating around the building. A stretch goal for the second team was to develop a framework for Wi-Fi localization capable of directing the robot in a building, but due to the COVID-19 pandemic they were unable to test their design within the engineering building and so that development was not completed at the time. Therefore, this task now falls upon our team. Wi-Fi localization will allow for the robot to navigate on its own throughout a building without the need for an Xbox controller. Requiring user control defeats the robot’s purpose of being an automated tour guide.

To complete this project’s third stage of development, our team consists of 5 members, as well as a mentor and a sponsor:

- Dr. Michael Leverington: Team Sponsor
- David Failing: Team Mentor

- Erik Clark: Team Lead, Communications
- Kyle Savery: Architect, Developer
- Ariana Clark-Futrell: Recorder
- David Robb: Release Manager, Developer
- Alexis Smith: Developer

2. Process Overview

The development process our project went through began with a research period that explored how our software could use Wi-Fi information to navigate a building. The most accessible way our team found of doing this was by using Relative Signal Strength Indicator (RSSI) values. This method was accessible in the sense that no additional hardware was needed (besides the provided laptop), which was ideal for this project since a goal was to keep the overall product as affordable as possible. RSSI values indicate how strong a signal is between an Access Point (AP) (typically, these are routers in the building) and the user's device. These values could be retrieved via the command line call, *iwlist*. However, this was the only way our team figured out how to get RSSI values for each detected AP and the most significant shortcoming for this procedure was that an individual Wi-Fi scan takes, on average, between 3 and 4 seconds.

For this reason, along with time constraints preventing us from researching this aspect of the project further, our team needed to pivot and place the emphasis of the remaining time on creating a Graphical User Interface (GUI). This GUI would rely on orientation and distances to direct the user around the building, instead of Wi-Fi. Once we established our plan for the second half of the project, our team used a couple of tools that supported us during the main development of the software. The first of which was basic version control on each file that our team was writing and the second were weekly task reports that described who in the team was responsible for which task, as well as how much of that task they needed to complete by the assigned date. Task assignment was based on the roles that each team member chose, as well as each member's strengths.

3. Requirements

Our main requirements have not significantly changed throughout the course of this project, but how the software achieves them has been modified. The following sections will cover our project's functional and non-functional requirements.

3.1 Functional Requirements

The functional requirements of a project are related to fundamental behaviors of the system. These behaviors define what a system does or does not do. Our two main functional requirements are that our software must navigate itself through a building and communicate with a user via a GUI.

As mentioned previously, instead of using Wi-Fi to navigate, our software takes into account the speed of the robot (which will be that of an average walking pace) and the distance/direction it must travel to reach its destination. The essential functionality of our navigation system is broken down as follows.

- <u>Shortest Path:</u> While guiding users throughout the building, it is important that the system does not waste time by visiting unnecessary areas. As a result, our software will always calculate the shortest path between any two points. This is done by employing Dijkstra's Shortest Path Algorithm. When a user needs to navigate to a different floor, two paths are calculated. The first directs them to the nearest elevator (as the robot will

not be able to traverse stairs) and the second completes their route by directing them from the elevator on the correct floor to their destination.

- Point to Point Navigation: A frontend user must be able to input a single destination within the building and subsequently be directed to that location. Available destinations will be displayed both on the building map and on a key that indicates which rooms/offices are around those destinations. Currently, the user must also input their starting destination, but once the software is integrated with the robot then the starting destination can simply be kept track of internally and the user only has to input their destination.

- Touring: This is an extension of point to point navigation, and really the most important requirement of our software. A frontend user has the ability to select any number of available tours that have been set up by a backend user. For example, an Engineering building on a college campus could contain tours titled “Electrical Engineering”, “Computer Science”, or “Mechanical Engineering”. Each of these tours would direct the user to any number of relevant locations within the building that pertain to that discipline. Furthermore, the software can easily be set up to display information about visited locations either through a text description or brief video.

With the fulfilment of our project’s navigation component, the next functional requirement concerns our software’s GUI. The purpose of this interface is to streamline the software’s interactions with both the back and frontend users. There are two main components of our GUI: image manipulation and the ability to communicate with a user which are outlined below.

- Image Manipulation
    - A tour guide would not be very useful without a map, so backend users must be able to upload maps for any floor within a building. These maps need to be displayed to the user along with all the nodes that have been set up on that floor. Our system does not actually modify these images, but overlays a floor’s nodes onto the screen whenever the floor is loaded. This display is used during the touring portion as well as during building setup.

- Communicate with the User (Back and Frontend)
    - Backend Communication
        - Backend users have the ability to add new admin users, edit existing building data, or add an entirely new building. For editing, a user is able to add, move, and connect/disconnect nodes. These nodes can then be added to any number of tours in an order as described by the user.

    - Frontend Communication
        - Frontend users, or the ones who have no knowledge of the inner workings of our software and simply want to use the service, must be able to request destinations within the building. These destinations can be

either a single destination the user wants to be directed to or the request can be made for the software to take the user to all points within a tour.

3.2 Non-Functional Requirements
The non-functional requirements of a project are requirements that specify a tangible set of criteria which performance can be objectively tested against. Similarly to functional requirements our software's non-functional requirements can be split up into two categories, navigation and the GUI.

- Navigation
    - Speed
        - An important aspect of Dr. Leverington's robot is that it is able to move quickly enough to match the average walking speed of a human. Clearly, the robot would not be an effective tour guide if it moved too quickly or slowly. In relation to our software, this means it must be functional while moving at the constant speed of the robot which can be set up to 3.125 mph. This requirement was taken into greater consideration in the beginning when our team planned on solely using Wi-Fi to navigate. Nevertheless, our software has been constructed so that any consistent speed can be used.
    - Definition of Success
        - When a destination point is requested the software will attempt to navigate the building and reach this point. We consider it a success if the software can get within 2 meters of the target. For example, if the user requests to be taken to room 102, then this task will be a success if the software is no more than 2 meters from the doorway after it traverses the building. With consistent speed and direction, our software can direct a user to their destination within this margin. Again, in regards to only using Wi-Fi the goal was to have the software achieve this margin with at least a 95% success rate, however more work is needed to complete this requirement (discussed further in section 7: Future Work).
    - Shortest Path Between Points
        - As mentioned in 3.1, Dijkstra's Shortest Path Algorithm is used to calculate the shortest path between two points in the building.
- Graphical User Interface (GUI)
    - When a user is interacting with the software via the GUI, we require that a destination request can be made within 2 clicks and any necessary typing to input the titles of the start and destination. Now complete, the user need only type in the titles and then click 'Start Route'.

4. Architecture and Implementation

The architecture of our system has two components: the back and frontend. The frontend consists of the GUI, while the backend contains the structure for our building data and a way to scan nearby Wi-Fi signals. Below is a description of these components as they are built at the completion of this project.

4.1 Backend Architecture

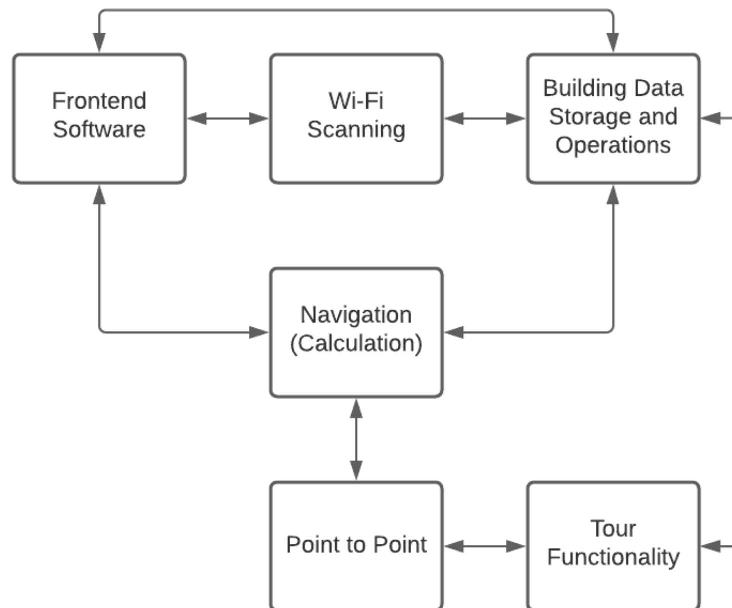

Figure 4.1: Overview of the Backend

4.11 Wi-Fi Scanning Module

The Wi-Fi scanning module is responsible for setting up a node's RSSI data set and estimating the device's location based on the nodes that have been created. Setting up a node's RSSI values, or *fingerprinting*, involves taking a sufficient number of Wi-Fi scans at that location. These values can then be reduced down to a range of RSSI values for each detected AP. Specifically, the minimum, maximum, and average values. Before performing any of these calculations it is important to remove any outliers (conventionally, any numbers greater or smaller than 1.5 times the interquartile range are removed). To estimate the device's position, a single scan is taken and each RSSI value for a detected AP is compared against the corresponding ranges for that AP stored with every node. The general idea being that the closer a sample value is to a node's average, the more points that are awarded to that location's *rating*. After all of these comparisons, the location with the highest rating is selected as the current location estimate. As shown in Figure 4.1, the scanning module communicates with the building data structure to store each node's RSSI data set, as well as access these sets during runtime. While the frontend software communicates with this scanning module to set up the data sets and to start/stop scanning procedures. The scanning procedure is a repeating scan while the device is traversing a building to estimate its position. This estimation is currently only sent to a

log file under a prefix “POSITION ESTIMATE”, as our system does not rely on Wi-Fi information to navigate (explored further in section 5.3).

4.12 Building Data Structure

The building data structure consists of 3 objects: Nodes, Floors, and Buildings, as well as the supporting functionality for each of these.

- Nodes: The simplest, yet most fundamental part of the data structure, a node object contains many pieces of information. The most important of which is the name of the node, whether or not this node is a transition (i.e. an elevator), the RSSI data set, and the physical *xy*-coordinates of the node in relation to the device’s screen. The node object comes with functionality to set up the RSSI data set via a call on the node object from the GUI. Within the editing portion of the GUI, a user has the option to press a button titled “Scan Wi-Fi” which will initiate the fingerprinting operation as described above. After which a node’s Wi-Fi information will be set up.

- Floors: A floor object consists of a list of nodes, those nodes’ names, and an adjacency matrix. This component comes with functionality to add new nodes, connect/disconnect existing ones, and calculate paths within a building. The navigation portion of the backend comes down to the latter of these functionalities (with the building object taking it one step further of navigating between floors). Paths calculated for point to point and tour navigation are sent to the GUI for display. A floor is also considered connected if and only if any node can be reached from any other node. As a result, if a backend user leaves at least one floor in a building disconnected then this building is not displayed to a frontend user. This choice is based on feedback received from our usability testing which will be discussed in section 5.3.

- Buildings: The building object is comprised of a list of floors, each of which is given a name (e.g. “1”, “A”, or “Ground”). This object is responsible for storage, directions, and formatting output for the GUI. Firstly, all buildings setup with our software are stored in a “buildings” directory. Inside of which are further directories named for each building. Each of these directories contains the information for any created tours as well as the map and node data for each floor. A building can be loaded by calling a building object’s “load” function along with that building's name and a building can be saved by using the “save” function. Secondly, the building object generates orientation dependent directions based on the locations of a floor’s nodes. For example, in Figure 4.2, the user’s destination is node *D*. In scenario 1, the user is approaching *B* via *A* so the next direction they receive will be to turn left toward *D*. In scenario 2, the user is approaching *B* via *C* and so the next direction will be to turn right toward *D*. This functionality is dependent upon the requirement that the nodes set up in buildings are always at right angles to one another.

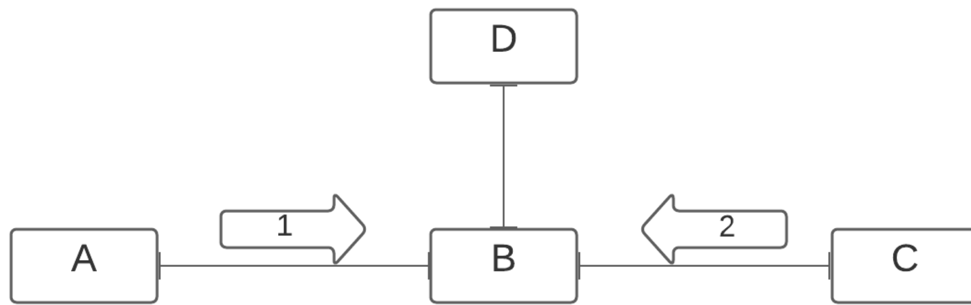

Figure 4.2: Process of outputting directions

However, this is really to maintain modularity between our software and the robot. Once integrated, these directions would likely be given to the robot in terms of degrees and so nodes can effectively be set up in any configuration. Lastly, the building object formats the output of node names to be presentable to the user via the GUI, i.e., alphabetizing and organizing a node's nearby locations as shown in Figure 4.3.

Legend

A - Main Entrance
B - Room 101
- Room 102
C - Lab 106
D - Bathroom
E - Advising Offices
F - Elevator
G - Room 103
H - Computer Lab
I - Fire Exit

Figure 4.3: User view of location names within a building

4.2 Frontend Architecture

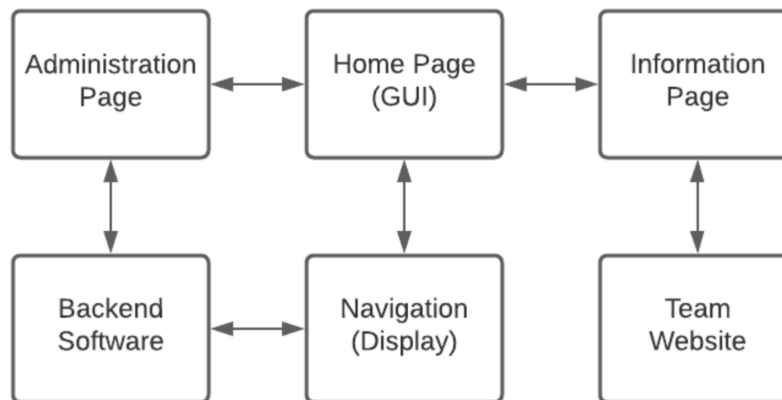

Figure 4.1: Overview of Frontend

4.21 Graphical User Interface Components

The frontend's interface utilizes a homepage which branches off into three other pages: Administration, Navigation, and Information. Referring to the high-level overview of the frontend system (Figure 4.1), the purposes of these pages are described as follows:

- Administration: The administration page is used by backend users to not only set up the software in a new building, but also allows the existing information about the current building to be edited. This editing includes adding/removing a location in the building's set of registered nodes. For each of these nodes the user is able to "Scan Wi-Fi" as referenced in section 4.12. When a user selects this option our software will remind them they need to be in the physical location within the building they are trying to set up and once they are, then the system can begin scanning nearby Wi-Fi signals for a default 25 iterations.

- Navigation: The navigational (or *Tour)* page is where the software will typically be interacting with a frontend user. This page will display a map of the building, and allow the user to make destination requests to any of the available locations. The user can also select a "tour" mode which will take them to important locations throughout the building. When a route is selected, then the user will be directed to each necessary node in the shortest path between them and their destination. Simultaneously, the software will begin performing Wi-Fi scans to repeatedly estimate the device's position in the building. During a route, the user is able to hit pause which will inform the device to wait at the next established node. Our system must wait until the next node to pause due to the structure of Tkinter (the Python module our software uses to create the GUI) that prevents us from being able to pause the route in between two nodes. Once at the next node, a pop-up box will appear and wait for the user to hit 'OK' before continuing the route.

- Information: The information page will supply the user with basic information such as frequently asked questions. This page also links to our team's website, which will show the user all of our detailed documentation about this project.

4.22 Graphical User Interface Behavior

The general behavior behind our software's GUI relies on *switching frames*. For example, whenever a user selects the up arrow to look at the next floor the interface does not just reload a new map image in place of the previous floor's, but actually reloads the entire frame using the incremented floor number and currently selected building. For simplicity, all created buildings are loaded on startup and grabbed by the program whenever a user chooses a new building. As mentioned in section 4.12 detailing the Wi-Fi Scanning Module, our software outputs a log file which contains estimations of the device's current locations. However, the majority of the entries in this log file originate from the GUI. Significant events such as frame switching, backend users logging in, building data being loaded/saved, and updates regarding currently running tours will all be sent to a single log file for a given execution of our software (as in each run of our software gets its own log file).

5. Testing

As with any software product, it is important to maximize the amount of time that the product is tested to ensure the system is as bug free as possible. To test our product, our team conducted unit, integration, and usability tests. For all of these methods, their planning and boundary cases can be explored in greater detail within our "Software Testing Plan" document located on our team website. Below will focus on a brief summary of how we planned to test our product with each method and the improvements/fixes we made based on our findings.

5.1 Unit Testing

Unit testing is a vital component to the development of any software system. A unit test is designed to isolate an individual unit of the software and test its robustness. These tests administer carefully designed inputs that cover the edge cases (or equivalence classes) for the values that a specific unit is expected to handle without relying on other components of the software.

Testing our product with concrete unit tests differed significantly between the front and backend components. For the backend, testing involved creating a series of boundary cases for each function that our team decided should be unit tested. For instance, the pathing algorithm in our building data structure could be tested by seeing if the function behaves as expected when starting form a node with 1, 2, 3, or 4 connections, ending at a node with those differing number of connections, and checking to make the sure the function always chooses the shortest path by calculating the correct paths out beforehand. A few changes our team made based off of the unit tests administered to the backend were subtle, yet important to the proper execution of the software. The most noteworthy of which was a bug where the program would not correctly calculate a path if the start and destination was the same node, and instead the user would see a path containing every node on the map. Fortunately, much of the functionality of the backend

software is related to getting and setting information passed in from the GUI, so it was simple to test these functions as they were being written.

Testing the frontend of our system was much more revealing of the software's bigger issues. To test the GUI, we would run through the sections we were testing to ensure every button performed its expected function and no reasonable combination of buttons resulted in a bug. The most significant issues we discovered during this phase was that there were some buttons that did not have the proper logic surrounding them to prevent crashes. For instance, there are many buttons in our GUI that rely on the user selecting an option from a list before pressing the button. However, our team completely overlooked what happens when the user presses the button before selecting an element from the list. To fix this issue, anytime a user presses one of these buttons a pop-up will appear asking them to select something from a nearby list. Another issue that was encountered was that a backend user can create a building without adding any floors to it. This was a desired operation, except our team had not implemented the functionality for how to add floors to this building later on. As a result, selecting this building from the selection page would crash the program. The solution was simple and now a pop-up appears asking the user if they would like to create a floor. In fact, virtually all of the bugs regarding our GUI revolved around these issues where a specific button lacked the necessary checks before performing an unsafe operation. The remainder of the bugs found in our GUI were discovered during integration testing.

5.2 Integration Testing

The goal of integration testing is to bring together components of the overall system and test their interactions in order to ensure that the product as a whole operates as expected. Since most software is developed by multiple people, integration testing mitigates possible bugs and errors in the communication between the modules that have been developed independently of one another.

To test the integration of our software, our team needed to test the interaction between all three of the following main components.

- Building Data Structure and GUI: To function properly, our GUI needed to be able to load any stored building, save any changes to a building's data (including tours), and utilize the pathing algorithm to display the shortest path and corresponding directions for the user's tour. Testing each of these was straightforward; we would create a new building, set up numerous nodes on several floors, then test a variety of paths to see if the display shown on the tour's side was correct. This building was then saved only to be loaded again and the same paths were used to test the proper loading of the building. This process was repeated many times while covering all boundary cases our team considered. The interaction between these two components was critical to the success of our product and so our team spent the majority of the testing phase performing variations of the above process. While important, the bugs we found during this testing were generally mundane and related to how the GUI was accessing a building's data, all of which were simple fixes.

- GUI and Wi-Fi Scanning Module: The interaction between these two components is extremely brief, as there are only a few lines of code within the GUI that directly reference the Wi-Fi module. When a route is started, the system will begin scanning Wi-Fi and when the route completes the scanning is stopped. A simple enough operation, where our integration testing revealed a critical flaw: if the user exited the software in the middle of a route then the Wi-Fi scanning would not terminate, effectively never closing the log system's log file. Moreover, the scanning operation relies on the command line call *iwlist*, which is only available on Linux machines. As a result, if the software is run on the wrong machine (it should be noted that only the Wi-Fi functionality of our software is system dependent) then the log file would be overwhelmingly populated with thousands of lines indicating there was an error scanning. A successful scan takes a few seconds to complete, so the number of lines written to the log file can be easily examined as opposed to the split second it takes to return an error. To fix these issues, if the module encounters some number of consecutive errors (default is 5) then it will automatically stop scanning for the remainder of the route. The scanning will also stop if the GUI gets shutdown.

- Wi-Fi Scanning Module and Building Data Structure: The building data structure relies on receiving an RSSI data set from the scanning module for each of its created nodes. While the scanning module relies on comparing recent Wi-Fi scans to this stored data. To test this integration, several nodes in a building were set up with RSSI data and then this building was saved/loaded to ensure that the data was always equivalent. Then our team would run routes and view the log file output to determine if the position estimates were reasonable. There were no notable bugs found in the interaction between these components during our testing.

5.3 Usability Testing

Usability testing allows us to test the functionality of our system with end users. This will evaluate how well the user can interact with our product and what functional or aesthetic changes need to be made to make sure our software is as streamlined as possible. As the creators of this software we may believe our system is sufficiently intuitive, but there are likely many things that will show up during this testing phase that highlight any shortcomings. By letting people who have no prior knowledge of this software attempt to use it, they can give constructive feedback on how easy the system is to use. Below are the changes our team made to our software after receiving this feedback.

- Originally, our team had two current location indicators on the tour page. One was a node that would be highlighted green to indicate the "true" location based off of a starting location and distances/directions traveled. Another was a small image of a robot head that would be placed over nodes selected as the position estimate from our Wi-Fi module. Our team thought this was the best way to showcase the Wi-Fi functionality of our software, however, our test users indicated that they only found it confusing and were not sure which indicator was meant to be their actual location. This was due to the

fact that our position estimates are highly variable and so the robot head would frequently jump around the screen. Consequently, our team scrapped this concept and instead went with just outputting the position estimates to the log file.

- Users found our floor adding page to be overly confusing. After they inputted the floor number and uploaded a corresponding map image they would click the “Add” button. At which point there was no information about what to do next or whether or not the operation was successful. It was also unclear if they could keep adding more floors. To fix this issue we added the pop-up shown below.

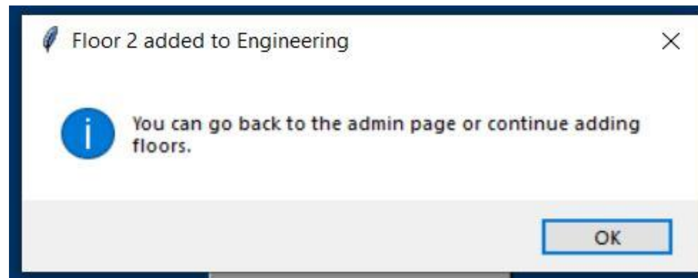

Figure 5.1: Confirmation pop-up indicating a floor has been added

- As mentioned in section 4.12, if a building has at least one floor that is disconnected, then this building is not displayed to the user. Originally, the disconnected nodes would not be displayed since the software would be unable to direct the user to them anyways. This only confused our test users though and they were wondering as to why only some of the building was accessible. To fix this issue we decided to only allow connected buildings to be displayed on our tour page.

- Lastly, our users consistently commented on the fact that they could only input a node’s label as the destination rather than an associated location. So if node *A* was next to Room 101 and Room 102, then the users wanted to be able to enter either of those rooms as a destination rather than just *A*. Once this functionality was added our team considered our software to be thoroughly tested and revised.

6. Project Timeline

To help describe this project’s timeline, Figure 6.1 shows a list of significant milestones during the first semester of this project between September and November 2020. While Figure 6.2 shows a Gannt chart of our team’s task schedule between January and April 2021. In regards to the second semester, tasks were split up based on whether they pertained to the front or backend of our software. The backend team consisted of Erik Clark and Kyle Savery, while the frontend team was led by David Robb. There was some overlap between these teams in order to streamline the integration process. Significant milestones for the frontend was getting to display an image of a map, manipulate nodes over it, and show the nodes involved in a route highlighted in the order they should be traversed. The biggest events for the backend development were correctly calculating the shortest path between multiple floors and completing

our software's tour functionality. At this moment, our team has completed all of the assigned tasks in regards to the Gannt chart.

Project Outline – Fall 2020 Semester		
Week	**Deliverable/Task**	**Description**
9/14 – 9/18	**Discuss Project Requirements with Dr. Leverington**	Determine what Dr. Leverington expects out of this project. We will need to take these initial requirements and begin planning our product.
	Team Standards Document	Details how the team will be run, including methods of communication, dealing with conflict, member roles, etc.
9/28 – 10/2	**Fully Operational Website**	Initialized website to contain areas for project descriptions, high level requirements, important documents, and team information. To be improved upon throughout entirety of project duration.
	Compare Viable Programming Languages	Our top considerations for which languages to use for this software were C/C++ and Python. As C did not have available libraries for capturing information about Wi-Fi, we decided to use Python.
10/23	**Technological Feasibility Report**	Report covering why we chose a particular solution to a given challenge.
10/26 – 11/20	**Prototyping**	During this period, we created and refined our initial prototype to display to the user their current location in the building.
11/13	**Design Review 1**	Our team created a video to discuss our project's problem, solution, requirements, and associated risks.
11/20	**Requirements Specification Document**	This report formally presents the functional and non-functional requirements for this project.
11/23 – 11/24	**Technical Prototype Demo**	During a presentation with our sponsor and mentor, we demonstrated our prototype to show that our product can be implemented.

Figure 6.1: Schedule for September to November 2020

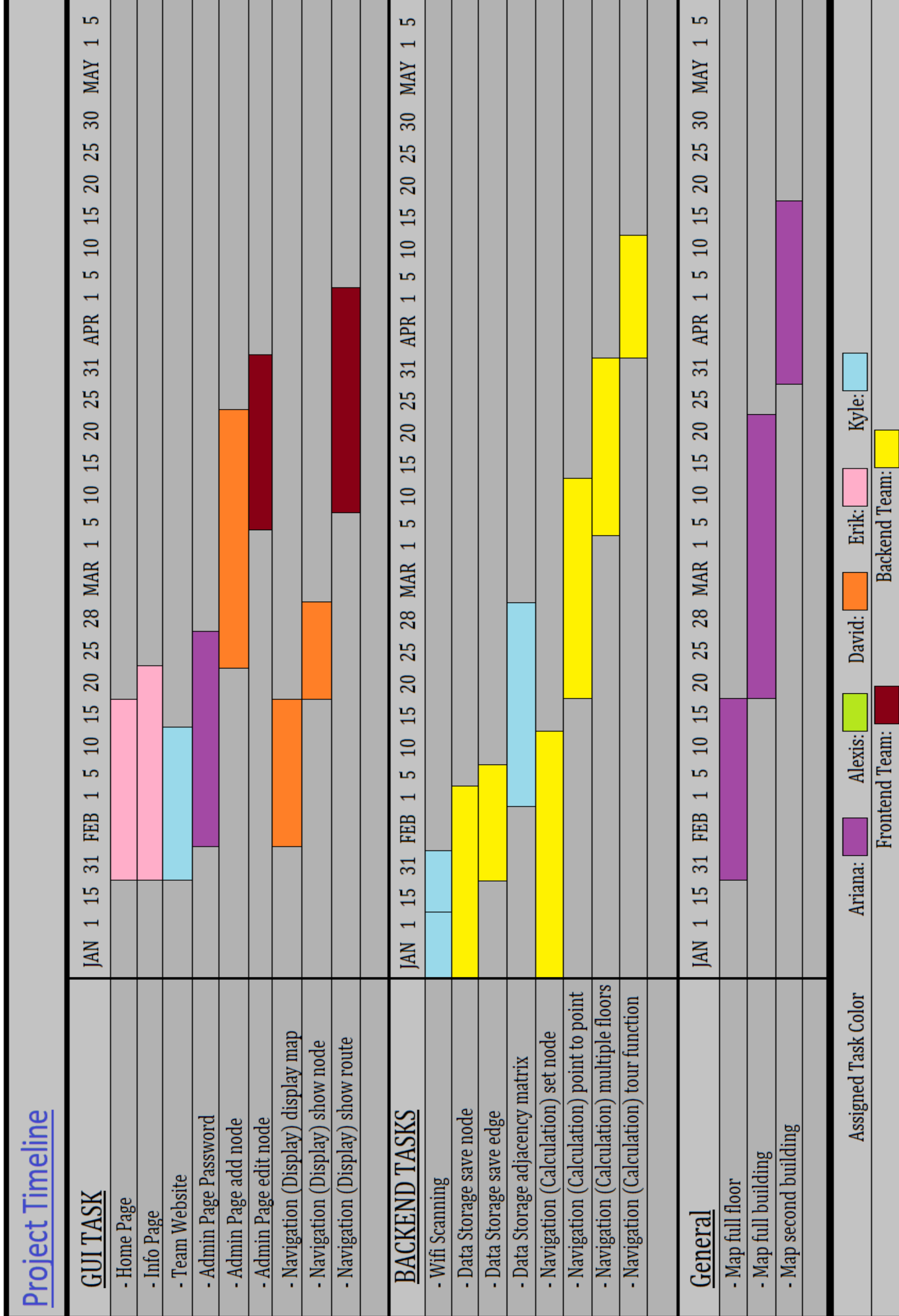

Figure 6.2: Schedule for January to April 2021

7. Future Work

Future work involving our software would be the process of integrating with the robot, incorporating more navigation systems, as well as implementing more complex solutions to Wi-Fi localization than first anticipated. Integration should be the simplest of these three tasks as our team's design process has been completely modular from the robot. Our software was written in Python as this language is compatible with the Robot Operating System (ROS) and is built such that the robot can simply act as the frontend user. Additional navigation systems such as Lidar, which is a method for measuring distances by using light, will likely need to be a focus of future teams for the robot to become truly autonomous. As for using Wi-Fi, advanced approaches such as artificial intelligence will help the software better compare RSSI data sets and be more resistant to unexpected values.

There is plenty of work to be done in taking our product, that establishes buildings and gives a user tours around those buildings, to a complete robot that does not require any outside assistance to act as the tour guide.

8. Conclusion

The capabilities of mobile robotics continue to expand and the societal problems that can be solved by them are expectedly growing. As a result, mobile robotics is an increasingly valuable field and the more people who can get involved early on expands the subject area both further and faster. Our project, "The Thirty Gallon Robot", seeks to demonstrate affordable ways that educational institutions can get students involved with robotics. By contributing our software to a robot that is as inexpensive as possible, our team (and previous teams) believe that we can show other instructors that getting their students involved with robotics is not only possible, but truly rewarding.

The specific problem our team worked to solve over this project was developing a software product that would allow backend users to easily set up any building so frontend users could receive tours to significant locations. This problem exists as our team sponsor, Dr. Leverington, has a robot in its third year of development that does not currently have a way to autonomously navigate the engineering building at Northern Arizona University. Our product is a major step in completing this robot tour guide.

Originally, our team was going to solely use Wi-Fi to navigate a building (as a requirement), but this was determined not to be feasible in the given time frame and so future work is needed if this functionality is to ever be implemented. Our team needed to focus our efforts on creating the Graphical User Interface (GUI) that anyone with little to no prior knowledge of the software could easily use. This GUI needed to fulfill the requirement of guiding individuals around any building with sufficient accuracy (within 2 meters). The main functionalities that our GUI has implemented to accomplish this are as follows:

- Frontend Use
    - The user can input their starting location, a destination, and then receive directions along the shortest path to their destination.

  - The user can select a *tour* mode where any tours set up by a backend user can be selected which will take the user to a variety of important destinations in the building.
  - The user can access an information page that contains frequently asked questions, as well as be linked directly to our team website to explore more detailed information about this project.

- Backend Use
  - An authorized user can access the administration section in our GUI, from which they are able to add new users, edit existing building data, or add a new building.
  - Within the editing section, the user is able to add new nodes, connect nodes, add nearby locations to any node, create an RSSI data set for each node, and create multiple tours within each building that directs the frontend user to any number of locations.

This software product has been thoroughly tested both during development and afterwards. As such, our team is confident that we have reduced the numbers of existing bugs to the best of our ability.

There are many ways that future capstone teams can expand upon our product once it is integrated with the robot. Additional navigation systems and accurate Wi-Fi localization is the first step to completing the autonomy aspect of the robot tour guide. In terms of our team's experience, however, we have been exposed to the real-world of software engineering and learned a great deal about both the technical and non-technical aspects. Our ability to thoughtfully document a product has improved and the members of our team can now go forward with the knowledge of how to successfully plan, research, develop, test, and deliver a product that can be utilized by real-world users.

9. Glossary

Access Point (AP). A hardware device that allows users to connect to a wired network, generally via Wi-Fi.

Adjacency Matrix. A 2-dimensional matrix used to represent a graph containing vertices and edges. In the context of our software, vertices are referred to as *nodes*. An adjacency matrix stores values between zero and infinity for each entry. The ordering of the rows and columns is dictated by a list of vertices. For example, one column may represent vertex *A* and a row may represent vertex *B.* The entry at their intersection describes how far apart *A* and *B* are in the graph, with infinity indicating that two vertices are not connected.

Backend User. A backend user is someone who has been given access to the admin section of our software. That is, these individuals are able to create and edit building information that can then be used during the tour functionality.

Fingerprinting. A technique in Wi-Fi localization that is useful for estimating position. A location within a building is “fingerprinted” when some number of Wi-Fi setup scans are taken at that location in order to build up a data set that describes the ranges of RSSI values for each detected AP. Once a building is fingerprinted, then a single scan can be compared to each fingerprinted location and the data set that it is most similar to is chosen as the closest location.

Frontend User. These are everyday users that will use our software to receive tours around a building that has been set up by a backend user. These users are not expected to have any prior knowledge of the software.

Graphical User Interface (GUI). A type of user interface that allows the user to interact with the software visually by using graphical icons instead of relying only on text based input.

Lidar. A method for determining distance to an object by targeting it with a laser and measuring the time that it takes for the reflected light to return to the device.

Location Rating. The rating of a location in the context of our software is how close of a match the location is to a recent Wi-Fi scan. The higher the rating of a location, the greater the probability of our system choosing this location as a position estimate.

Media Access Control Address (MAC). A unique identifier that is given to a network interface. This is used to keep track of every AP within a building.

Node. A node in our software is a location within a building that has been given a name and is connected to other nodes. These can be visited by a user during a tour. A node can have any number of *associated names*. These are locations that the node is meant to represent. For example, node *A* may be associated with rooms 101, 102, and 103. A user can input either *A* or one of the associated names in order to visit it.

Relative Signal Strength Indicator (RSSI). A measurement of the strength in a received radio signal. These values range from 0 to -100 dBm with 0 being the strongest signal.

Wi-Fi Scan. The process of getting the current RSSI values for each detected AP near the user’s device. A typical scan using our hardware takes between 3 and 4 seconds.

10. Appendix A: Development Environment and Toolchain

- **Hardware:** The hardware that was used for this project was a Linux laptop that was provided by the client. The only requirement to access some of the functionality of our software is that the laptop needs to run Linux and have Wi-Fi capabilities. Without a Linux machine, the software will be unable to scan nearby Wi-Fi information via the command line call *iwlist*. However, the software can function without Wi-Fi so the GUI can be used on any machine that has version 3.X of Python installed.

- **Toolchain:** For the development of this software, our team members each used their own preferred text editors (Atom, Sublime, VSCode). Code was executed on the provided laptop to ensure any and all changes were viable. Github was used to store our source code and constant communication was used to ensure no code would be merged when it should not have been.

- **Setup:** This project is already set up on the provided laptop, however, to set it up on a different machine be sure to follow these steps:
    - Make sure that Python is installed and updated to a 3.X version.
    - Install all Python dependencies such as Tkinter. For a full list of required modules reference our "Maintenance Manual".
    - After all dependencies are successfully installed, load the source code into a directory on the machine, then open a terminal and change into that directory.
    - Once there execute the command 'python3 Gui.py'. This will launch the software. If the software fails to launch, the most likely error is that a module has not been installed correctly. Note: Installing modules can vary from machine to machine so be sure the installation process follows the recommended method for your machine.

- **Production Cycle:** Once the setup is complete, refer to our "User Manual" on how to set up buildings with the product. This manual will cover day to day operations in more detail.

11. REFERENCES

[1] Pramod, B., & Shadaab, K. (2020, October). Robotics Technology Market Size, Share and Analysis: Forecast - 2027. Retrieved November 04, 2020, from https://www.alliedmarketresearch.com/robotics-technology-market

[2] Rahul, K. (2019, January). Mobile Robotics Market Size, Industry Analysis and Applications by 2026. Retrieved November 04, 2020, from https://www.alliedmarketresearch.com/mobile-robotics-market