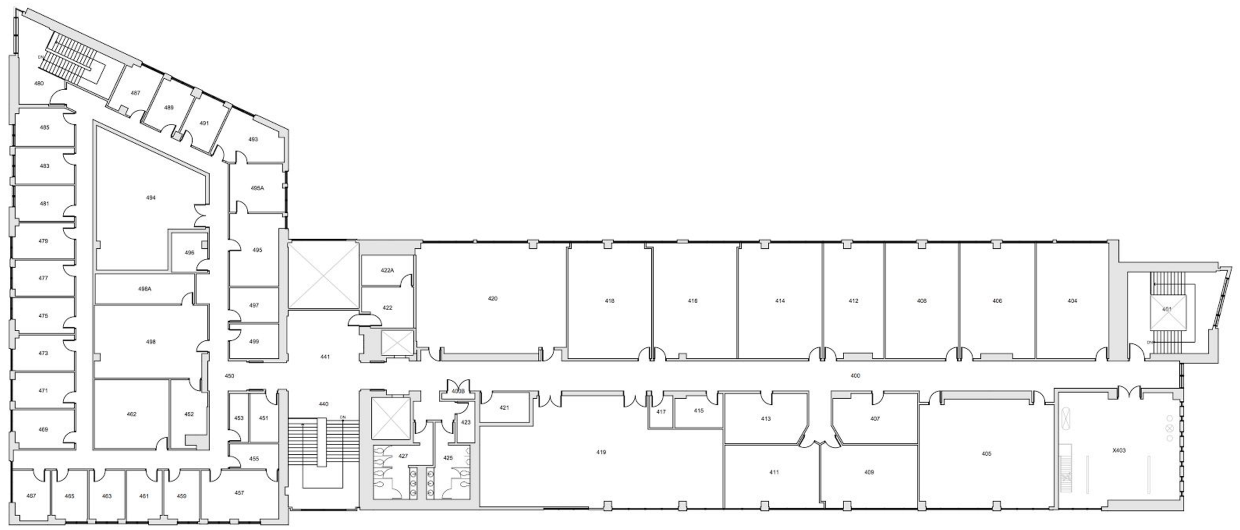
CSCI 402 — Final Project

Sam Shinn

Description

The goal of this project is to design a program which will allow a virtual robot to traverse a virtual representation of the 4th floor of WWU's Communications Facility (CF) building using a Q-Learning algorithm. By using the Q-Learning algorithm, there should be no need to program the various routes into the program, nor should the robot be required to recompute any route when requested to travel. Instead, after training the robot, the robot will have QTable (or multiple QTables) which it may reference 'at-a-glance' in order to gather instructions to reach the goal destination. For this particular program, our objective is to allow for more than one goal destination. The user will be able to, at runtime, order the robot to go to any room on the 4th floor of the CF building. Then, as another addition, the robot will return to an area indicated as 'home'.

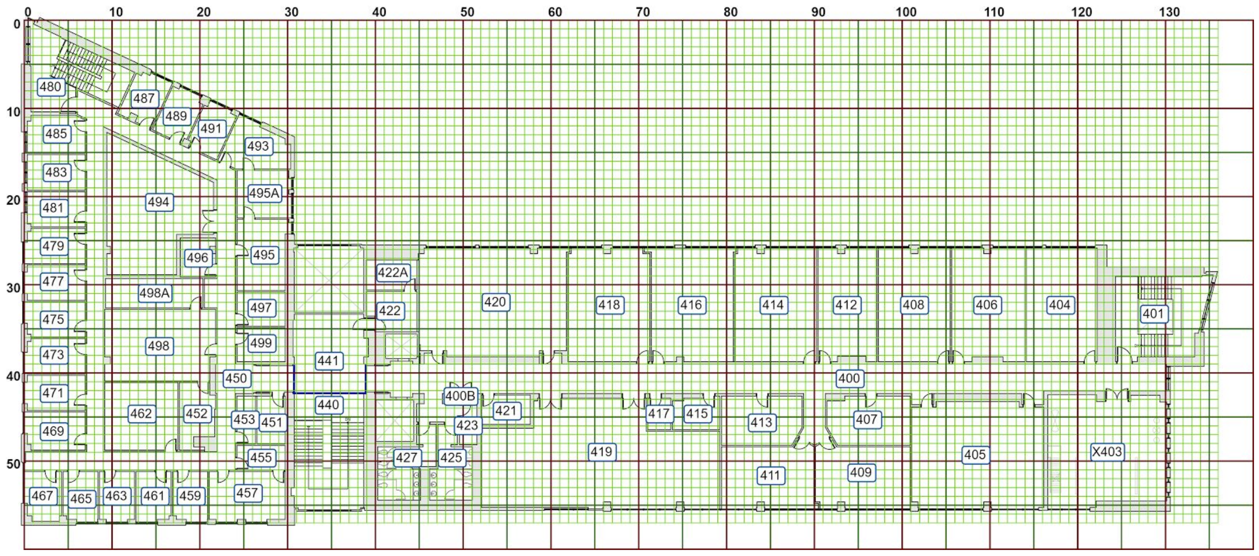


Design

Q-Learning is generally used when the goal state is fixed, and the Q-Table (which directs the robot's movements) is populated with calculated values which will allow the robot to find the goal state in the shortest route from any starting position. In such a case it is possible to use one single Q-Table for this purpose because the pathways converge to a single goal state. But this application is different in that the robot will be required to travel from some position to any arbitrary position, meaning that the pathways are now divergent. This means that it's not possible for a single Q-Table to show pathways to all arbitrary states. What must be done instead is have multiple Q-Tables which can be selected based on the goal destination.

However, two significant problems with creating unique Q-tables for each potential goal state is that doing so introduces wasteful redundancies in memory usage and training time. Take for example two rooms which are right next to each other; most of the path to these rooms are the same path, only branching slightly at the end. With entirely separate Q-Tables for each destination, the pathways have to be stored and trained independently, even though a large portion of the pathway are identical. Surely a better model (which would incidentally better model human intelligence) would create, record, and use one single pathway to reach whatever state is just before the two rooms, and then use more detailed pathway information to find the specific room which is the robot's destination.

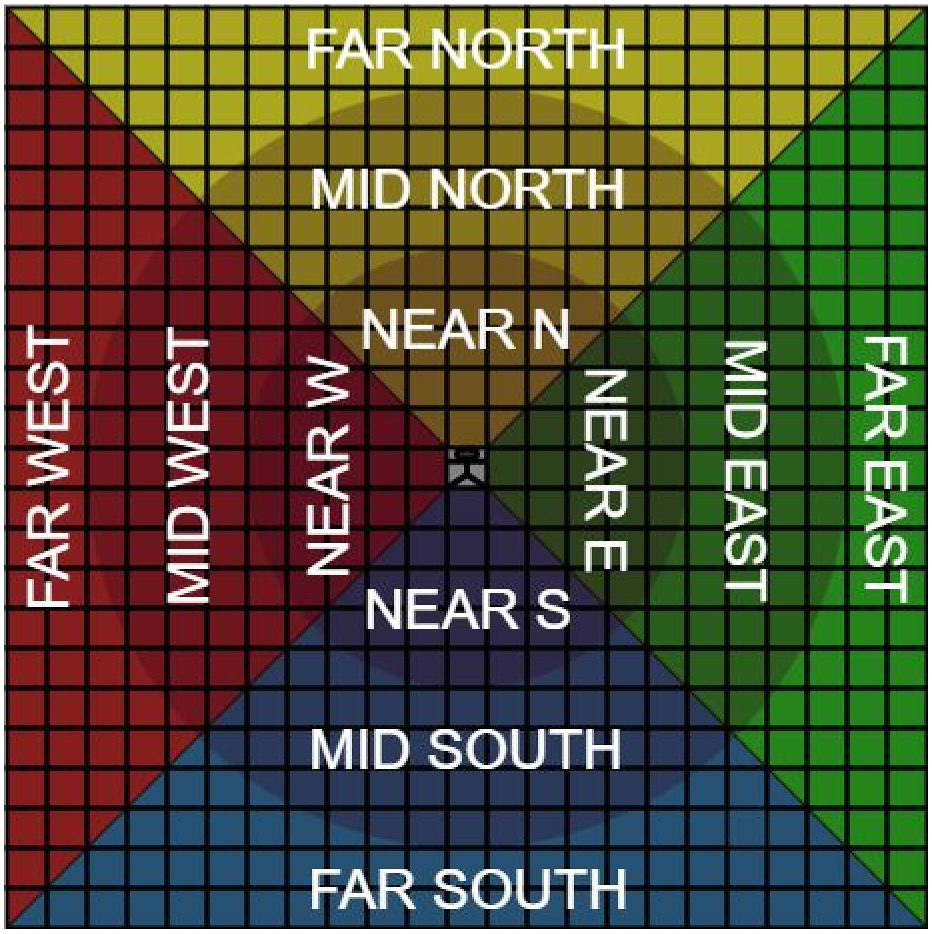
They key to reducing the number of QTables is to somehow find a way to generalize some number of various goal destinations into some kind of "generalized goal destination," which can be treated as a singular entity until it is no longer appropriate to lump together those goal destinations in a generalized form. To do this, we will overlay a grid on top of the blueprint of the 4th floor of the CF building. The purpose of this grid is that it provides a non-arbitrary structure to all the locations of the CF building. That is, if we similarly laid this grid overtop any other blueprint, the grid representation of space offers an ordered structure which would not inherently exist if we were to, for example, consider rooms as arbitrarily numbered states in an abstract diagram.



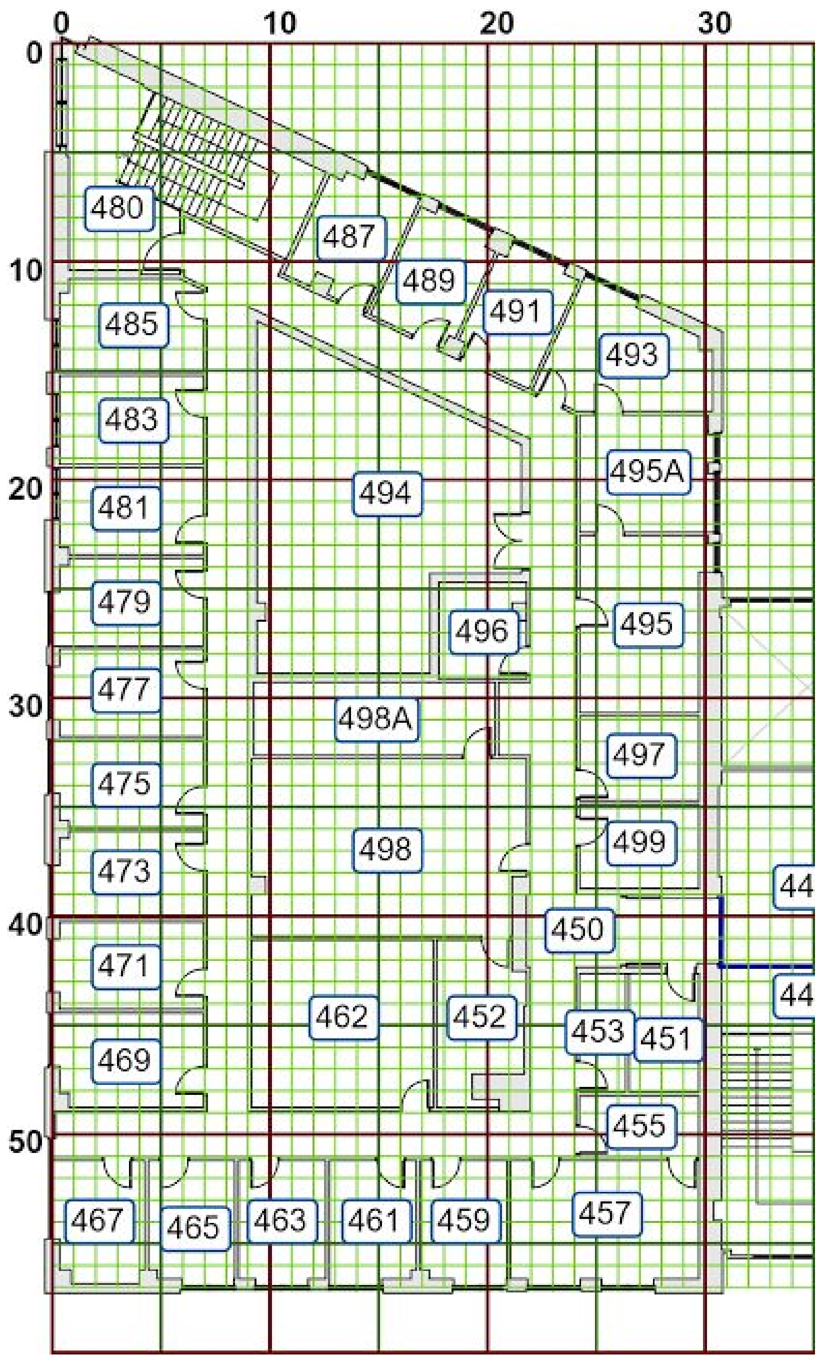
Now that we have an ordered structure, however, we still must find a way to generalize the various goal states. To generalize goal states, we have to divide the map into sections. On this challenge I came up with three main ideas:

- Divide the map into static sections
  - This means cutting up the blueprint into quarters, ninths, etc.
  - The robot uses a QTable which leads it to the section containing its destination
  - The robot uses another QTable which leads it to the specific destination
- Detect a structural hierarchy and divide sections based on that (find hallway, then room)
  - This idea is to, essentially, identify rooms and their connections as a state space
  - Use QTables of destinations in state space until close enough,
  - Then use QTables of grid to reach specific coordinates when close enough
- Divide the area around the robot into sections
  - This means using the angle and distance to the goal to make generalizations
  - The robot has a different QTable for various ranges of distances and angles
  - It checks position and angle to goal destination to determine which QTable to use

Option 1 seemed as though it would still use too many QTables, resulting in high redundancy of training and memory usage. Option 2 seemed daunting, and I think would deviate too far from the core exercise of implementing Q-Learning. If we used the hierarchical structure I described, it's at that point that it is not much different from actually calculating the optimal path for each goal request. Q-Learning is characterized by a certain amount of 'dumbness' allowed in the robot. That is, the robot shouldn't need to do much thinking, it should be able to more or less look at a set of instructions and follow this biggest arrow. For those two reasons, I decided on option 3.



Option three creates categories divided by direction and distance between the robot and the goal. These directions and distances are measured in absolute terms, and not measured along whatever path leads to the goal (that would be too 'smart' and again defeat the purpose of Q-Learning). The diagram above shows an example of dividing the area surrounding the robot into 12 sections, though we could reasonably change the number of distances and angles used. Which of these sections contains the goal destination is how the robot determines which QTable to use in order to navigate the area. The reason we still need QTables for navigation (as opposed to simply following the calculated direction) is because we don't want the robot to get stuck in dead-ends or on corners.



For example, consider that the robot is issued a goal destination of 477, a room far on the west side of the building, and the robot has just entered the leading hallway from the central stairs (the hallway is labeled 450). If we were to simply head west because 477 is west, we would run right into the wall! Therefore, the necessity of the QTable is still present, and so the exercise is still a valid one.

Furthermore, it's not enough to categorize destinations simply by direction: We must also specify how far away our goal is, because other potential goal destinations are between our robot and our requested goal destination of room 477 –namely, room 498. Take note that the robot will have been trained so that all rooms are potential goals, so if it had one QTable for all destinations which are west, 498 and 477 would be generalized to the same QTable. The QTable would likely lead the robot to one of the rooms, but not both. So by including distance in the generalization, the robot can use a QTable which leads it to faraway, western destinations (namely 477), which should follow the hallway around around the central rooms.

Implementation

This program was written in C. The executable which manages I/O, initialization, and training and running of the QTables is “test.exe”. One personal goal of this project was to write a program which was not limited to navigating the 4th floor of the CF building, but could navigate any area when provided with an appropriately formatted input file.

In order to come up with an input file which could represent the 4th floor of the CF building, I created a higher resolution grid to lay over the blueprint, —the same one which is featured in this document— which also clearly displays the room names for each room. Using this GridMap as a reference, I used Google Sheets to create an xlsx spreadsheet to represent the map. In this spreadsheet, each cell represents a cell on the grid, and cells which are part of a room are labeled with their room name. Cells which connect two rooms are labeled with a string containing the names of the connecting rooms. By formatting the spreadsheet this way, the program can detect if cells are connected by whether adjacent cells contain each other’s name (this allows for cells which are adjacent to be disconnected, as if by a very thin wall). From Google sheets, a csv version of the file may be downloaded, which serves as the input for the program.

The first phase of the program’s operation is to gather arguments from the user. Critically, these arguments include the name of the input file and the name of the home room. The home room being, of course, where the robot should return to once it has reached a destination specified by the user. This flexibility of changing the homeroom was an essential part to my personal goal of making the program applicable to other map environments. Unfortunately, these critical features were the extent of what was implemented. I would have liked to have implemented more customizability, allowing the user to set the learning rate, discount, or the number of QTables used by changing how many categories of direction and distance to use for goal generalization. But these features were considered non-essential and were not reached.

The second phase is to read the input file and generate an internal representation of the data. There are two primary data structures, one which is a state space arranging the cells of rooms, and the respective actions which may be taken from each cell. These cells are arranged in a 2-dimensional matrix, mimicking the grid overlaying the blueprint, and the list of actions contained in each cell is represented by an array of 4 integers which, in order, indicate actions to move east, north, west, south (this order is to mimic the measurement for a unit circle). This state space is used in multiple ways: Sometimes simply as a map, sometimes as a reward table (by setting the ENWS values moving into goal states very high), and they are also used as the QTables.

The other primary data structure is a linked list in order to track the names of each room. The purpose of this is twofold: One, it reduces memory use by tracking the memory addresses of the various room names, which the cells in the state space can use to avoid redundantly storing room name data. And two, during the training process, the program needs a list of all the potential goal destinations. By iterating through this linked list, the program has the room names it needs.

The third phase is training. Training takes place for each potential goal destination, and QTables are updated in a systematic manner by updating cells in a sequence managed by a queue. The first coordinates thrown into the queue are those of the goal. The program uses the

Q-Learning algorithm to update the edge weights of this cell, and then throws the neighbors of this cell into the queue. Cells already trained this way are tracked in a 2-dimensional matrix called ‘registry’, preventing an infinite loop of neighbors throwing each other into the queue forever. This method allows the program to train very quickly and completely in one fell swoop.

How the Q-Learning algorithm interacts with the goal generalization is similar if we consider the how the goal is categorized to be just another layer of states for the robot to be in. Suppose that we are updating the East edge of cell 24,58, and our goal is categorized in MID-EAST. Then we calculate the Q value for the E edge of cell 24,58 in the MID-E QTable by using the maximum possible Q value of the expected state in our update calculation. To do that, we look at cell 24,59 (because we moved along edge E), and see what category the goal destination is in now. Imagine, since we are closer to the goal destination, that it is now categorized as NEAR-E. So our maximum Q Value is one of the edges from cell 24,59 in the NEAR-E QTable. This Q-Value is used in a calculation possibly involving reward values, the Q-Value the edge being updates already has, the learning rate and the discount.

The fourth and final phase is testing and interaction. It is during this phase that the user should be able to issue a command to the robot by naming a room for it to go to. Some kind of display, either a list of coordinates or a textual representation of the map, would show the robot make its way from ‘home’ to the specified destination and back to ‘home’ again. Unfortunately, phase four is not implemented due to hangups on completing phase three.

Discussion & Future Work

As the project is incomplete, there is much future work to be done:

- Get phase three working satisfactorily.
- Implement phase four entirely.
- Enhance phase one by adding customization options.

Although the project is incomplete, I was able to produce some output files so that various aspects of the program may be inspected. For example, I produced a file which displays the various QTables after training, as well as a file which catalogs the calculations made to relate the the goal position to the robot’s, informing the choice of which QTable to use.

The current state of the Q-Learning algorithm is not the standard as I had been taught it; I have adjusted it many times over trying to get better results and its current state should be understood as a ‘work-in-progress’. Part of the trouble in making this project go smoothly is the sheer scale and depth of it. At many times did I mismanage my data structures or iterations, having normal troubleshooting issues which were hard to diagnose. In particular, keeping straight east, north, west, south, was an exhausting challenge. And so with so many moving parts, it was hard to tell if failure was due to my Q-Learning algorithm or some other peripheral element.

As important as phase four is, I would have sooner prefered to enhance phase one. If I had successfully implemented easily customizable parameters, I could have tried many more settings at a much faster rate in order to experiment and produce more satisfying results. As is, I was reduced to simply changing values which were embedded in the code, or not changing the values at all. In particular, I think the goal generalization categories need to be of higher granularity, so as to better differentiate between goal destinations.