Charles Calder and Elias Posen
May 20, 2016

Corridor Graph: A Project in Intelligent Robotics and Perception

What and Why: An Introduction

We chose to tackle the problem of visual programming. Our goal was to create an interactive environment to show how different structures and syntaxes work in coding languages. The hardware we decided to use included an overhead camera and a Scribbler robot. Processing (Java) and Calico (Python) fulfilled our software needs. We also used wooden blocks and brightly colored duct tape to create an environment for our robot to perceive and navigate. The overhead camera encapsulated the "intelligence" of our robot, which was, to use the lingo of CMSC 360, a "bat." The camera enabled our robot to "know" the map of its entire environment. We used pixel filters and graphs to process the image of the robot's "**playspace**"[1] and plan a path for the robot to traverse. The interactive component was that the path the robot takes changes based on the placement of orange blocks, a blue "start" square, a green "source" square, and a yellow "end" square.

As programming becomes more ubiquitous in our digital world, more and more people are using it and trying to understand it. Visual programming is an accessible way to teach those who may not be so mathematically or technologically inclined about programming. One idea we had was to have a specific color or obstacle for the robot to sense which would represent a conditional. We envisioned this obstacle at a "fork" in the robot's path. As it approached this obstacle, it would have to decide which direction to take based on other circumstances, such as the amount of light in the room detected by its light sensors. The result of this would be a visual representation of the "if ____ then ____" construct, in logic and in programming.

Ultimately, these goals were too ambitious for the timeline that we had. We have a program which, given a photo of an orange **corridor**[2] and the three colored squares, plans a path from each square to the next while avoiding the orange walls. The program which passes commands to our Scribbler, however, is fickle. After a series of tests, we determined that the problem was not with our own math but with the inconsistency of the Scribbler's odometry. This means that we failed to create an intelligent robot that can traverse any path assembled by our

[1] the robot's world, the extent of our xy coordinate plane
[2] a hallway, with one end open and the other closed, assembled with orange building blocks or "obstacles"

peers, as planned. The absence of other structures, like if-conditionals and for loops, is due solely to the limited time we had.

Implementation

Image Processing & Graphs

Our first step in our program is to convert a picture of some corridor into a graph, for path planning purposes. The Processing IDE makes sense for this task, since it has built in functions which make image manipulation and processing more convenient. We decided to group the pixels into **clusters**[3]. In order to make sure that the entire corridor is detected, each cluster becomes a node in our graph. The program searches each pixel in the cluster and determines if the RGB value of that pixel denotes an orange wall, the blue start square, the green source square, or the yellow end square, according to the values we give and and some threshold that is predetermined. If any pixel in the cluster is determined to be a color of significance, then the entire cluster is considered to be that color and the node associated with that cluster becomes colored as well.

Once the graph is colored, the next step is to connect edges between neighbouring nodes that are not orange (the color of our obstacles). Each node knows of the node values of its neighbors using the Graph.makeNeighbor() and Graph.getNeighbor() functions that we added to our graph class. The majority of nodes have four neighbors (above, below, right and left), but there are some special cases that deal with nodes on the perimeter of the picture. In the connection process, a node is connected to all of its neighbors if that node is any color but orange. At this point, we have a fully connected graph where the orange nodes are unreachable. In relation to the playspace, this means that the graph is constructed in such a way that the robot will never attempt to go through a wall.

We also added x and y coordinates to each node that corresponds with the global reference frame. We measured how large the frame width was in the playspace to determine the number of millimeters per pixel in our photo. This enabled us to calculate the physical distance between each node (according to the cluster size) and thus set the x and y coordinates of each node accordingly. Node 0 is at position (0,0) in the top-left corner of the photo.

Each path has two segments; the path from the start square to the source square, and the path from the source square to the end square. To discover the shortest path for each of these, we use a breadth-first search algorithm.

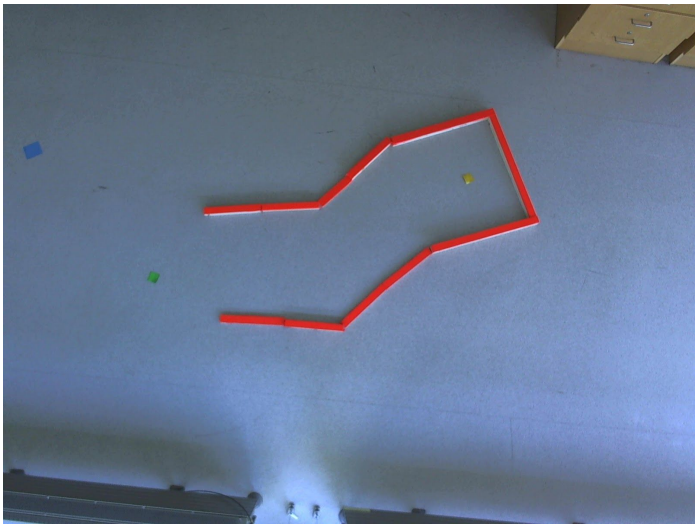
---

[3] p by p groups of pixels, treated as one object

Staircases

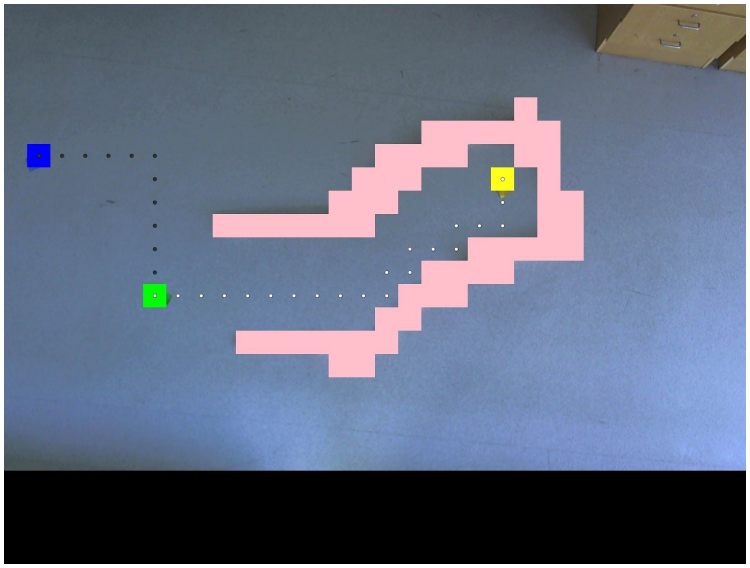
We discovered a conceptual issue with our graph after visualizing the path in Processing. Our graph produces an output file that contains the x,y coordinates of each node in the path. These coordinates are then be passed to a goto(x,y) function for our robot. By using the original path returned by BreadthFirstPaths, the robot would have had to, in certain cases, execute a series of 90 degree rotations followed by short forward movements in order to accomplish what is simply a diagonal line. This series of nodes that should produce a diagonal line is what we call a **staircase**. We projected that this would be an issue in terms of odometry; having more goto(x,y) statements left more room for distance errors and having so many 90 degree rotations would incur orientation issues. We therefore wrote a program that would recognize the first and last node of each staircase and create a path that would skip all the incremental steps mentioned earlier.

We implemented this by converting the initial path, an iterable, into a linked list. Using conditionals, we outlined procedures for all possible staircases: ascending, descending, horizontal, and vertical[4]. When we detect the beginning of a staircase, the program travels up to the top or down to the bottom of the staircase and notices when a staircase finished. At the end of the staircase, we add that final node to the simplified path and then the program checks the succeeding nodes to see if another staircase is occurring.

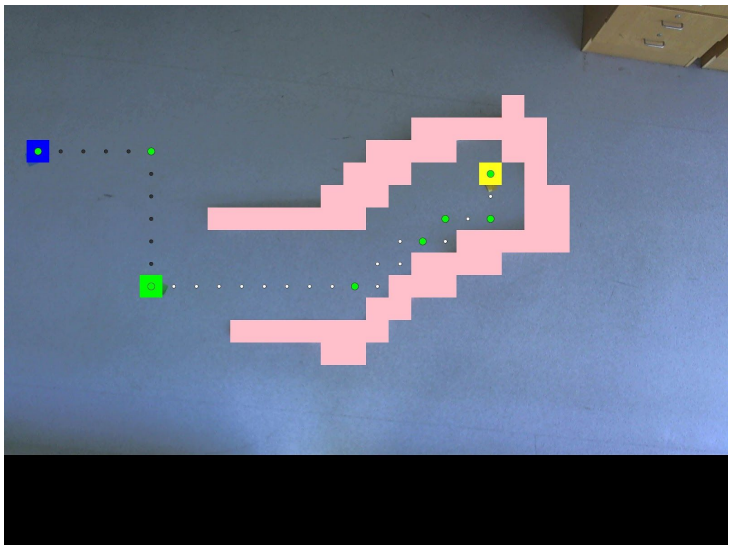


Original picture before any image processing or graph creation. The start square is blue, the source square is green, and the end square is yellow.

[4] Note that a "horizontal staircase" is simply a series of nodes with the same y coordinate which form a line. The end-points of this line become the start and end nodes for the staircase. The "vertical staircases" are similarly defined.



Picture after the image processing and graph creation. The clusters that discover the different squares turn the color of those respective squares. Furthermore, the clusters that detect the orange wall turn pink. The path is represented by circles, each of which correspond to a single node in the graph. The path from the start position to the source position is colored black and the path from the source position to the end position is colored white.



Visualization of the path, following the use of the staircase function. The green nodes represent the path returned. In this case, the staircase function reduced the path from 32 nodes and 32 goto() calls to 8 nodes and 8 goto() calls. This reduces the odometry error that comes about through multiple forward commands and rotations.

Working with Scribblers

Because of our decision to work with a Scribbler, we had to use a Python IDE in addition to Processing. Calico was a clear choice since we all had experience using it. Our Processing program outputs the coordinates of the start and end points of each straight-line portion (post staircase) of the path. The first two integers become the start pose of the robot, i.e. the x and y coordinates. Our Python program reads in the remaining integers and passes them to our "goto()" function from Lab 1. This function commands the robot to travel from its current position to an inputted position.

As aforementioned, the odometry of the Scribbler is fickle and each executed command introduces more error to our coordinate readings. This is why the Staircase portion of this

program was so essential. Without a program to measure such portions of the path, our Scribbler would receive many more commands than necessary and the amount of error between its "known" position and its actual position would compound. Additionally, we measured the number of "clicks" per wheel rotation in the Scribbler's wheels through a series of tests which may or may not truly represent the actual value; this could also be adversely affecting the accuracy of our program and the Scribbler's ability to function.

Evaluation

Our final result is an intelligent robot which, devoid of hardware issues, can travel from point A to point B to point C without hitting any obstacles in its way. While the path-planning is not live, it is complete, meaning that in the case that a path exists for the robot to traverse, our program will find it. We did have some issues with RGB pixel filtering, but our use of brightly colored duct tape was a logical and effective way for the robot to distinguish between the different tasks which each color demanded. The method we used to combine Python and Processing, which was outputting numbers from Processing that our Python program consumed, was also successful.

We had some minor issues with coordination of schedules amongst team members. Our team works well when we are all together at once. We aimed to do that as much as possible and accomplished good work in those settings, but did not foresee every necessary task. The staircases, for example, were not something we planned for and they took much longer than expected to conquer. However, this meant that we did not outline a clear division of labor, which made scheduling conflicts, absences of group members, and keeping group members up to date more difficult to deal with. The final product contains more-or-less equal contributions from each member, but next time we will try to define a more clear distribution of work.

If we had had more time, we would have liked to incorporate live updating through the use of QR codes or April tags. At face value, the program would be more impressive since our audience could change the path while the robot is already moving, and the robot would be able to detect those changes and compensate for them. Unfortunately, we had issues even in attempting to incorporate data from the Scribbler's IR sensors into our pre-planned path. This highlights another issue we had, which was that the path that our program finds frequently takes our Scribbler dangerously close to our walls. Accumulated error in odometry often lead to collisions. Future work, in addition to live updating, could include adapting our breadth-first search algorithm to use more of the space and avoid obstacles more efficiently.

We faced challenges with our work environment as well which, given more time and resources, we could have tackled. The overhead camera in RCK 107 is tilted; even though the nodes in our photo were spaced equally apart in terms of pixels, this did not directly translate to

equality of distance between the node in the robot's playspace. Additionally, this lab has floor-to-ceiling windows. This made it a beautiful and uplifting space to work, but provided inconsistent amounts of light depending on the time of day and weather conditions when we were working. In a different room with sufficient, consistent exposure to light, we could have set hard RGB threshold values that need not change with each new photo. Ultimately, we decided to include unique pixel thresholding for each image taken to combat this uncertainty, but it made our program a little less user-friendly and less intelligent than we would have liked.

The organization and interface of our program needs improvement as well. We had planned to have one "start" button that executes each part of the program, but this was unattainable within the time we had. Instead we have to (1) run a program connected to the overhead camera, (2) manually import the photo into our processing program, (3) manually open Calico, connect to the Scribbler, and run our Python program. This is less than ideal.

Finally, we had planned to incorporate more components of "visual programming," as aforementioned. We were unable to create structures or define colors which commanded the robot to make a decision "conditional style," or execute one action a certain number of times, or take audience input. However, we have written the foundation for a program which has the potential to be all of this. We hope to enact these improvements and pursue this project further, perhaps with some robot other than a Scribbler.