Robotics Lab 7: Simultaneous Localization And Mapping (SLAM)

Authors: Stuart Harley, Nathan Chapman

Date: 10/21/2021

Abstract

When controlling a robot, you have the choice to use many different methods and sensors. In this lab, we explore using an overhead camera as a mapping tool to get a robot from one space to another while avoiding obstacles. However, due to issues in getting the overhead camera functioning correctly, we were instead provided with images containing start and goal locations, and obstacles. We developed an A* algorithm to perform an informed path search to get from the start point to the end point in the most efficient way possible. Using our implementation, we were able to successfully navigate the provided black and white images. After that, our program can process and navigate color images. Because these images contained significantly more pixels, they took longer to complete the A* algorithm. We did notice that our pathways aren't always the most efficient. It was unclear why, but the paths tended to hug the obstacles at the end of the path instead of heading directly for the goal. This is due to some issue in our A* algorithm, but we couldn't seem to find the error in our program.

Methods

In this lab, we are searching black and white images containing obstacles for the optimal path from the start point to the end point. To do this we implemented the A* search algorithm, shown below in Image 1.

```python
def getNeighbors(img,p):
    neighbors = []
    for i in range(-1, 2, 2):
        if p[0]+i < img.shape[0] and p[0]+i >= 0 and img[p[0]+i, p[1]] == 0:
            neighbors.append([p[0]+i, p[1]])
        if p[1]+i < img.shape[1] and p[1]+i >= 0 and img[p[0], p[1]+i] == 0:
            neighbors.append([p[0], p[1]+i])
    return neighbors

def dist(p1,p2):
    return np.linalg.norm(np.array(p1) - np.array(p2))

#assume start and goals are tuples of (y,x)
def astar(img, start, goal):
    visited = []
    p = dist(start, goal)
    F = [(start, p)]
    while F != []:
        testPath, testPrice = F.pop(0)
        nk = [testPath[-2], testPath[-1]]
        if nk not in visited:
            visited.append(nk)
            if goal == nk:
                finalPath = []
                for i in range(0, len(testPath), 2):
                    finalPath.append(testPath[i:i+2])
                return finalPath
            newPaths = []
            for n in getNeighbors(img, nk):
                if n not in testPath:
                    p = dist(start, nk) + dist(nk, goal)
                    newPaths.append((testPath + n, p))
            F = F + newPaths
        F.sort(key = lambda x: x[1])
    return []
```

Image 1: A Search Algorithm Code*

The A* algorithm is based on the Breadth-first search algorithm. In Breadth-first search, we search by expanding the shallowest path. We do this using a FIFO queue. Because we are expanding the shallowest path first, this means that we are guaranteed to find the optimal solution if there is one. However, this process has large time and space requirements to complete compared to the A* algorithm. Below in Image 2 is a diagram that shows how Breadth-First search functions.

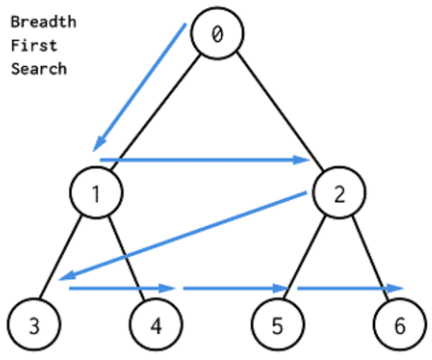

Image 2: Breadth-First Search Diagram

The A* algorithm shown above in Image 1 is built upon the Breadth-First search algorithm, but it also implements a cost value to each possible move. Therefore, instead of expanding the shallowest path first, you expand the least costly path first. This means that instead of using a FIFO queue like Breadth-First Search, we are using a priority queue based on this calculated cost. For our purposes, the cost is the distance the path has gone plus the estimated distance to the goal. Distance is calculated using Euclidian distance. What this means is that our algorithm expands the shortest path that is estimated to be closest to the goal first.

In this lab we also use a variety of OpenCV operations. OpenCV is an open-source computer vision library. We use this library for some of the later images that we want to navigate. Because our algorithm is designed to work on black and white images, but we are given color images, we need to perform image segregation to change them to black and white where white represents obstacles. To do this we use the inRange Image thresholding method to identify areas of specific colors. We also use this method is identify the green start locations and the blue end locations.

Task 1

In task 1 we are navigating the following images shown below in Images 3 through 5. Because these images are in black and white, we did not need to perform any image thresholding. The start and end points are not visible in these images but were given with pixel coordinates. However, in general the start location was in the upper left and the end goal was in the bottom right of the image.

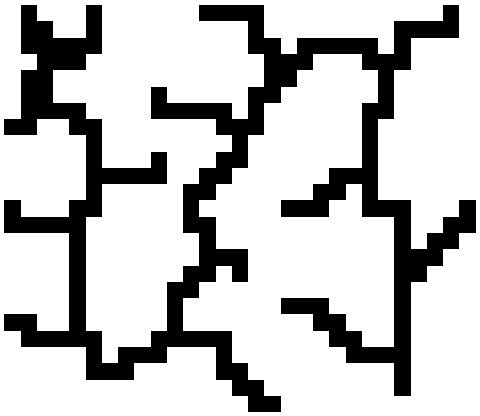

Image 3: Task 1 Image 1

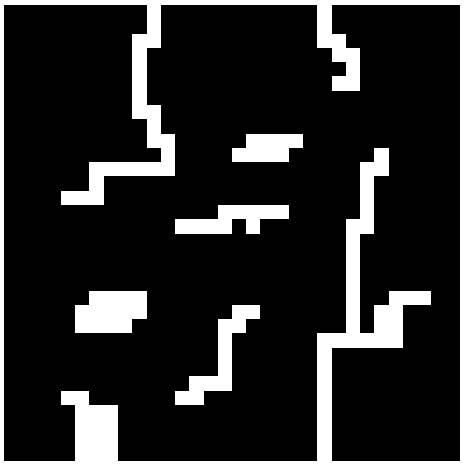

Image 4: Task 1 Image 2

Image 5: Task 1 Image 3

Task 2

In task 2 we perform image thresholding on the following Image shown in Image 6 to change the image to black and white and identify the start and end points before using the A* algorithm to find an optimal path. To do so, we first find the start point as the centroid of the blue circle by thresholding the image based on that color. Then we do the same for the end point, or green circle. Finally, we threshold the image to make anything that isn't red, black. This allows the algorithm to visualize the path it needs to take.

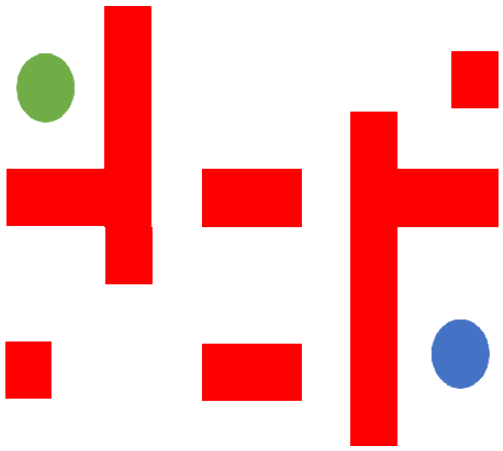

Image 6: Task 2 Image

Task 3

In task 2 we again perform image thresholding on the following Image shown in Image 6 to change the image to black and white and identify the start and end points before using the A* algorithm to find an optimal path. As done before, we first threshold the green and blue circles to get centroids of the start and stop points. Then, we make the orange path white, as that's the path we were to follow. We wrap it up by passing that processed image with starting point and end points to the A* algorithm.

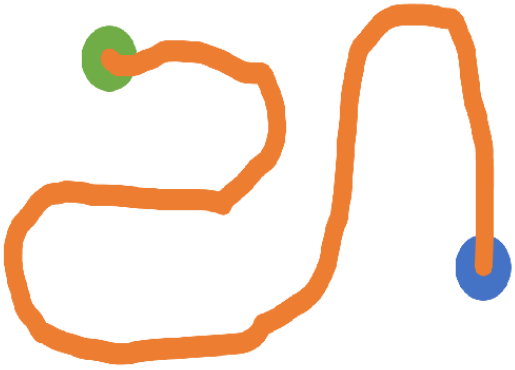

Image 7: Task 3 Image

Results

Task 1

Shown below in images 8 through 10 are the paths identified by the A* algorithm for each of the task 1 images.

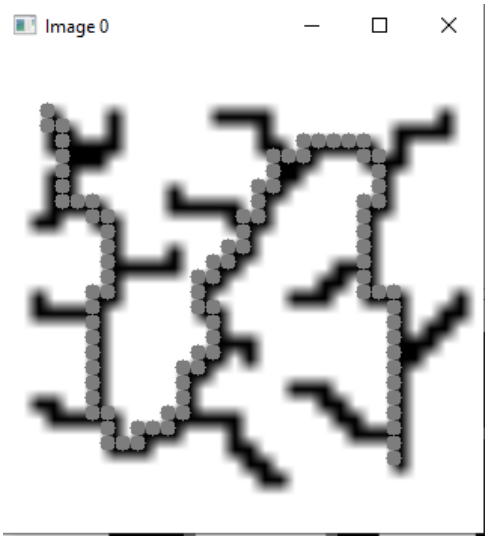

Image 8: Task 1 Image 1 Solution

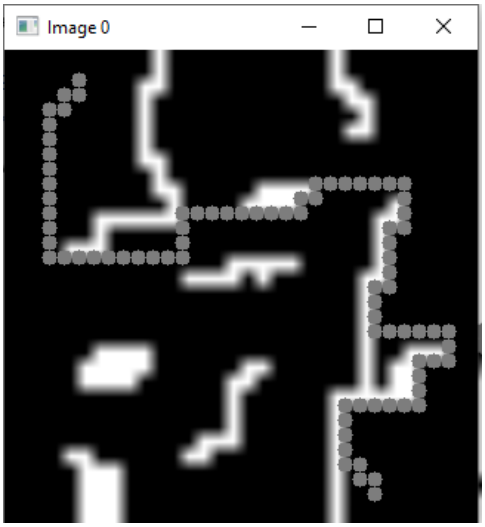

Image 9: Task 1 Image 2 Solution

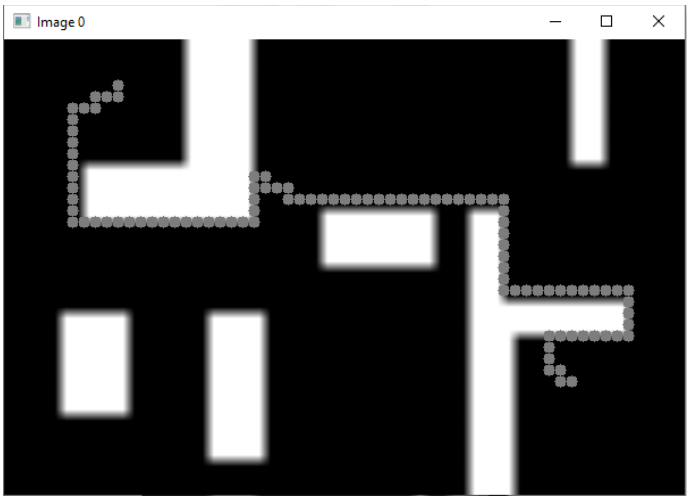

Image 10: Task 1 Image 3 Solution

Task 2

Shown below in Image 11 is the path identified by the A* algorithm.

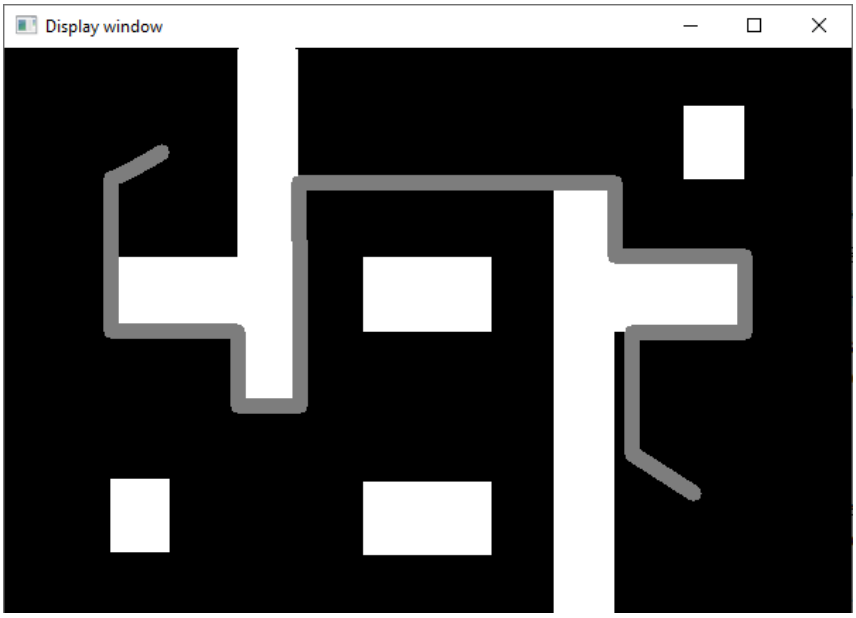

Image 11: Task 2 Image Solution

Task 3

Shown below in Image 12 is the path identified by the A* algorithm.

Image 12: Task 3 Image Solution

Discussion

1. Greedy search is an algorithm where the cost of a path is just its distance to the goal. A greedy search algorithm would not have worked as well for this lab as A*. This is because there are obstacles in the way of the "robot", so it often has to move in directions that are not towards the goal. Greedy search would continue to expand the path that is physically closest to the goal, but not actually because of obstacles, so it would result in less efficient paths being created. These paths would get as close as they could to the goal but behind some obstacle and then backtrack along the obstacle until it got around it.

2. The images for tasks 2 and 3 were significantly larger (more pixels) than the images for task 1. Because of this, the A* algorithm took longer to run on these images because each pixel in the image corresponds to another possible move. The task 2 solution took between 5-10 minutes to run while the task 3 solution took about 20 seconds to run. In comparison, the images in task 1 finished almost instantly. The third image in task 1 was also larger but it was scaled down to achieve this runtime. In the same manner we could have chosen to scale down the images in tasks 2 and 3 so that there were fewer pixels (possible paths) to search. However, since there were no time constraints for performance, we just let the algorithm run on the original images.

3. A* produced a path, which is a list of points. If we were looking at a real top-down view of the robot, we could translate those points to actual robot movements. To do this, we would need to determine the real-life conversion between 1 pixel of the image and the

distance it represents in real life. We could then use this constant to calculate the correlated distances to make the robot to move along the path.

Supplementary Materials

Contained in this section are code screenshots for the computer vision functions, A* search methods, and tasks 1 through 3. The only python file submitted is main.py which contains all the functionality of this lab.

Python Code

```
import cv2 as cv
import sys
import numpy as np
from math import pow,sqrt

callBackImg = None

def printRGB(event, x, y, flags, params):
    (height,width,channels)=callBackImg.shape
    # EVENT_MOUSEMOVE
    if event == cv.EVENT_LBUTTONDOWN:
        if channels==3:
            print(str(x)+" "+str(y)+" "+str(callBackImg[y][x][2])+" "+str(callBackImg[y][x][1])+" "+str(callBackImg[y][x][0]))
        else:
            print(str(x) + " " + str(y) + " " + str(callBackImg[y][x][0]))


def showImage(images):
    global callBackImg
    for i in range(len(images)):
        if i == 0:
            callBackImg = images[i]
        cv.imshow("Image "+str(i),images[i])
        cv.setMouseCallback("Image "+str(i), printRGB)
    while True:
        k = cv.waitKey(0)
        if k == ord("q"):
            break
    cv.destroyAllWindows()

def getNeighbors(img,p):
    neighbors = []
    for i in range(-1, 2, 2):
        if p[0]+i < img.shape[0] and p[0]+i >= 0 and img[p[0]+i, p[1]] == 0:
            neighbors.append([p[0]+i, p[1]])
        if p[1]+i < img.shape[1] and p[1]+i >= 0 and img[p[0], p[1]+i] == 0:
            neighbors.append([p[0], p[1]+i])
    return neighbors

def dist(p1,p2):
    return np.linalg.norm(np.array(p1) - np.array(p2))
```

```
#assume start and goals are tuples of (y,x)
def astar(img, start, goal):
    visited = []
    p = dist(start, goal)
    F = [(start, p)]
    while F != []:
        testPath, testPrice = F.pop(0)
        nk = [testPath[-2], testPath[-1]]
        if nk not in visited:
            visited.append(nk)
            if goal == nk:
                finalPath = []
                for i in range(0, len(testPath), 2):
                    finalPath.append(testPath[i:i+2])
                return finalPath
            newPaths = []
            for n in getNeighbors(img, nk):
                if n not in testPath:
                    p = dist(start, nk) + dist(nk, goal)
                    newPaths.append((testPath + n, p))
            F = F + newPaths
        F.sort(key = lambda x: x[1])
    return []

def findBlob(orig):
    (height,width,channels)=orig.shape
    lower_range = (254, 254, 254)
    upper_range = (256, 256, 256)
    mask = cv.inRange(orig, lower_range, upper_range)
    mask = mask.reshape(height,width,1)
    return mask

def drawPath(img,path):
    for p in path:
        cv.circle(img, (p[1], p[0]), 5, (125), -1)
```

```
def get_center(img):
    (height, width, channels) = img.shape
    sumx = 0
    sumy = 0
    count = 0
    for i in range(height):
        for j in range(width):
            if img[i][j] > 100:
                sumx += j
                sumy += i
                count += 1
    return int(sumx / count), int(sumy / count)

def testImage1():
    orig = cv.imread(cv.samples.findFile("testImage1.png"))
    thres = findBlob(orig)
    # note for the start and goal,
    # the coordinates are backwards
    # y is [0] and x is [1]
    start = [27, 26]
    goal = [4, 3]
    path = astar(thres, start, goal)
    return [thres,path]

def testImage2():
    orig = cv.imread(cv.samples.findFile("testImage2.png"))
    thres = findBlob(orig)
    # note for the start and goal,
    # the coordinates are backwards
    # y is [0] and x is [1]
    start = [30, 25]
    goal = [2,5]
    path = astar(thres, start, goal)
    return [thres,path]
```

```python
def testImage2():
    orig = cv.imread(cv.samples.findFile("testImage2.png"))
    thres = findBlob(orig)
    # note for the start and goal,
    # the coordinates are backwards
    # y is [0] and x is [1]
    start = [30, 25]
    goal = [2,5]
    path = astar(thres, start, goal)
    return [thres,path]

#Example with a larger image
def testImage3():
    orig = cv.imread(cv.samples.findFile("testImage3.png"))
    (height, width, channels) = orig.shape

    #We have to resize here or else the astar will take forever
    scale = 10
    resize = cv.resize(orig, (int(width / scale), int(height / scale)))
    thres = findBlob(resize)
    # note for the start and goal,
    # the coordinates are backwards
    # y is [0] and x is [1]
    start = [30,50]
    goal = [4,10]

    path = astar(thres, start, goal)
    return [thres,path]
```

```python
def task2():
    global callBackImg
    orig = cv.imread(cv.samples.findFile("task2.png"))

    start = cv.inRange(orig, (195, 113, 67), (197, 115, 69))
    start = start.reshape(start.shape[0], start.shape[1], 1)
    startX, startY = get_center(start)

    end = cv.inRange(orig, (70, 172, 111), (72, 174, 113))
    end = end.reshape(end.shape[0], end.shape[1], 1)
    endX, endY = get_center(end)

    img = cv.inRange(orig, (0, 0, 254), (0, 0, 256))
    img = img.reshape(img.shape[0], img.shape[1], 1)

    path = astar(img, [startY, startX], [endY, endX])
    print(path)
    drawPath(img, path)

    cv.circle(img, (startX, startY), 5, (125), -1)
    cv.circle(img, (endX, endY), 5, (125), -1)
    cv.imshow("Display window", img)
    while True:
        k = cv.waitKey(1)
        if k == ord('s'):
            cv.imwrite('testSave.png', callBackImg)
        elif k == ord('q'):
            break
    return 0
```

```python
def task3():
    global callBackImg
    orig = cv.imread(cv.samples.findFile("task3.png"))

    start = cv.inRange(orig, (195, 113, 67), (197, 115, 69))
    start = start.reshape(start.shape[0], start.shape[1], 1)
    startX, startY = get_center(start)

    end = cv.inRange(orig, (70, 172, 111), (72, 174, 113))
    end = end.reshape(end.shape[0], end.shape[1], 1)
    endX, endY = get_center(end)

    img = cv.inRange(orig, (48, 124, 236), (50, 126, 238))
    img = img.reshape(img.shape[0], img.shape[1], 1)
    img = cv.bitwise_not(img)

    path = astar(img, [startY, startX], [endY, endX])
    print(path)
    drawPath(img, path)

    cv.circle(img, (startX, startY), 5, (125), -1)
    cv.circle(img, (endX, endY), 5, (125), -1)

    cv.imshow("Display window", img)
    while True:
        k = cv.waitKey(1)
        if k == ord('s'):
            cv.imwrite('testSave.png', callBackImg)
        elif k == ord('q'):
            break
    return 0
```

```python
def main():
    [img, path] = testImage3()
    # [img,path] = testImage2()
    # [img, path] = testImage1()


    #We are assuming we are doing astar on smaller
    #images. We then have to resize them and the
    #paths back up
    (height,width,channels) = img.shape
    scale = 10
    resize = cv.resize(img,(width*scale,height*scale))
    (height, width) = resize.shape
    resize = resize.reshape(height, width, 1)
    rPath = []
    for p in path:
        rPath.append([p[0]*scale,p[1]*scale])

    print(rPath)

    drawPath(resize,rPath)
    showImage([resize])

if __name__ == '__main__':
    # main()
    # task2()
    task3()
```