Solutions for HW1

Yunhai Han

January 14, 2020

1 A switching Bug 0 algorithm

1.1 i

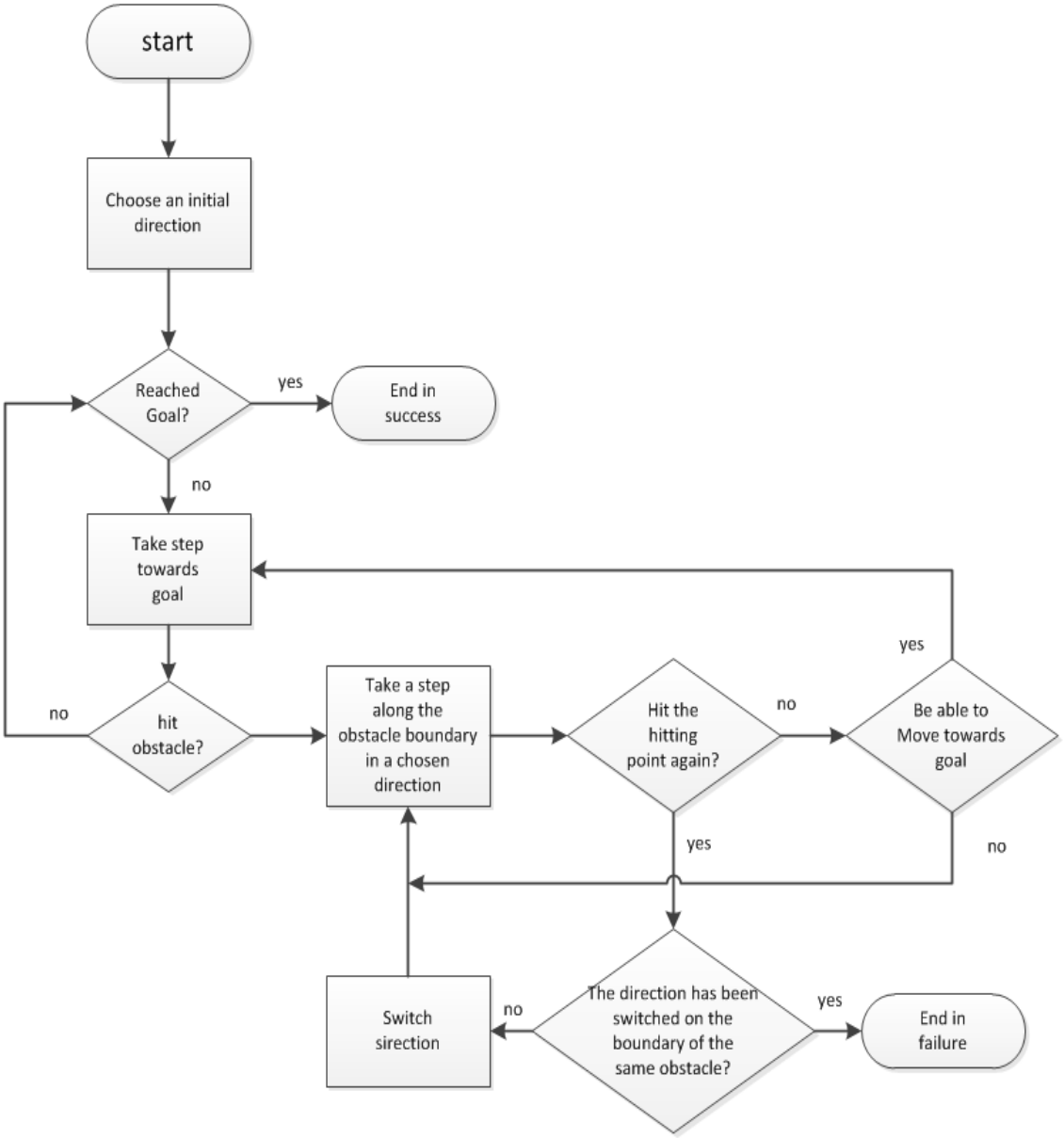

Figure 1: Flowchart

I draw the flowchart as shown in Figure 1.

1.2 ii

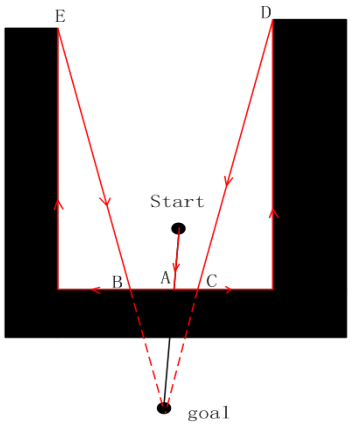

Figure 2: The diagram of such an environment

From the above figure, the red line shows the path the robot would move along when we implement switching bug 0 algorithm on it. It is clear that a path from start to goal exists, but the robot could never finds it. In this case, the switching algorithm could never reach the goal location.

1.3 iii

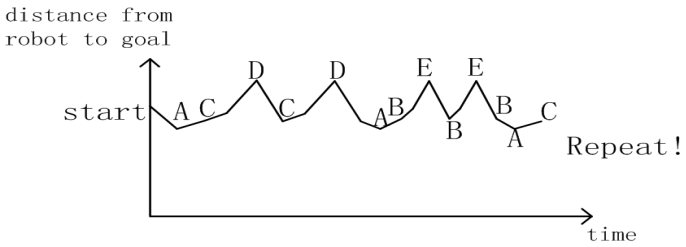

Figure 3: The distance between the robot and the goal location

From the above figure, I could tell the the symmetric property of this environment is the main reason for why the switching bug 0 algorithm would fail. In such an environment, the robot would not stop cycling until the battery goes dead.

2 The Bug 2 algorithm

2.1 i

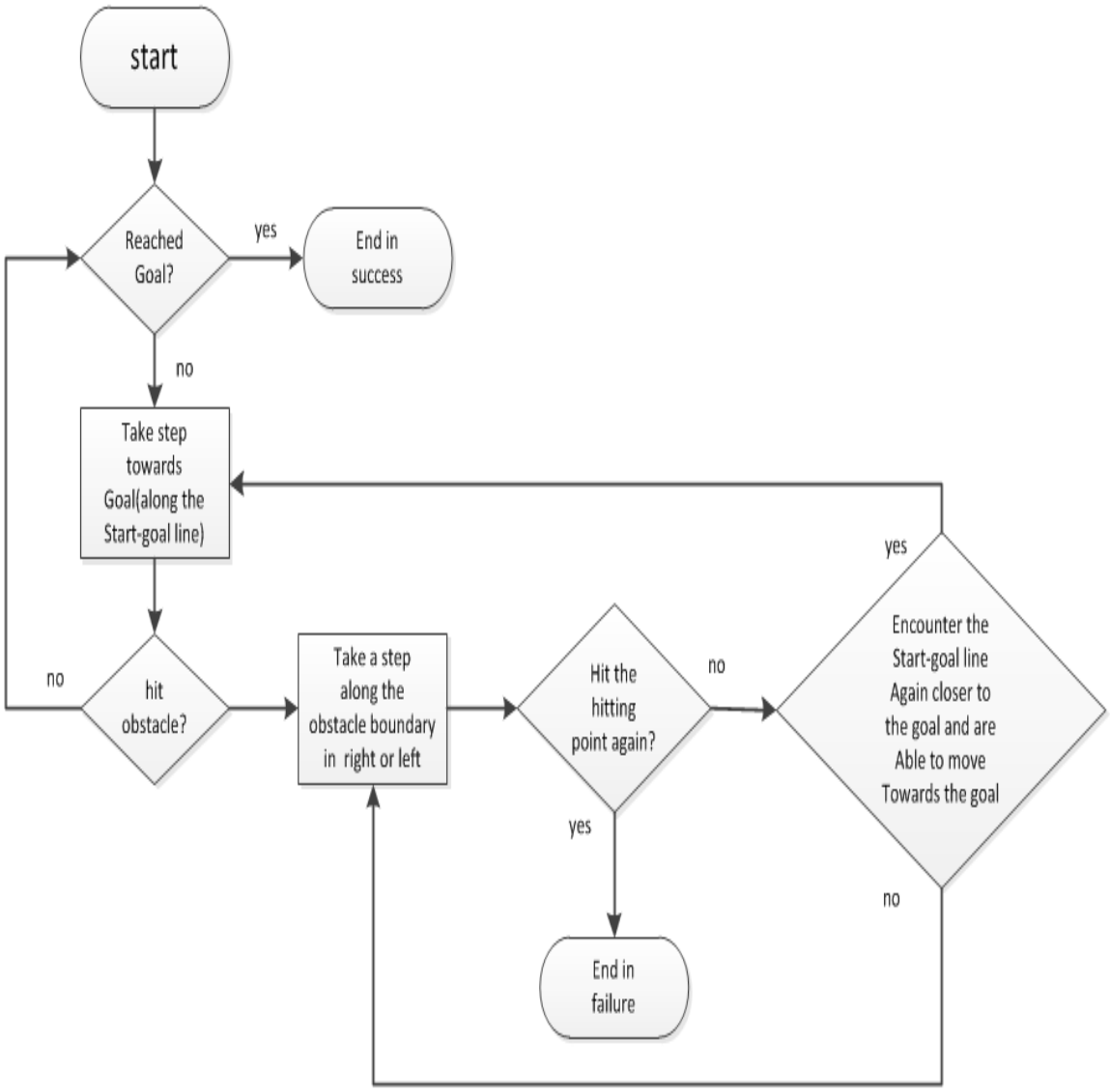

Figure 4: Flowchart

I draw the flowchart as shown in Figure 4.

2.2 ii

For the second part, I think if the robot chooses to turn into different directions when it hits the obstacle, the results could be totally different. I will show all these circumstances.

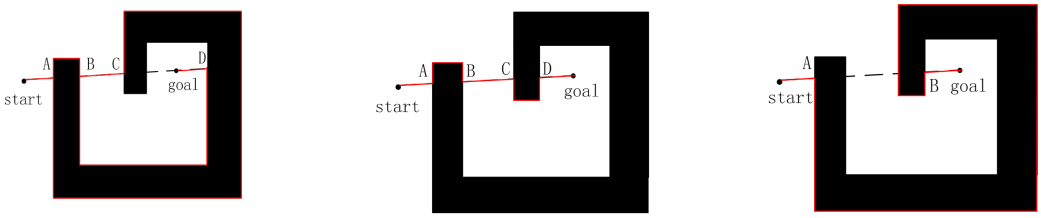

Figure 5: Robot's path using Bug 2 algorithm

From the above figures, the left one shows the result when the robot would turn left each time it encounters an obstacle; the middle one shows the result when the robot would first turn left and then turn right at different hitting points on the obstacle; the right one shows the result when the robot would turn right and in this case, it would only encounter the obstacle once. By comparison among these three figures, it seems that although Bug 2 algorithm could successfully find a path towards the goal for any turning directions, they greatly pose an effect on the amount of time for the robot to reach the goal. This means the different choices could bring in different results and only one of them is the best one for a certain workspace configuration.

2.3 iii

In this part, I am required to sketch a graph of the distance between the robot and goal location along the path. In order for the convenience, I just take the left figure as an example to show how I handle this problem and the solutions of the other figures could be obtained in the same way.

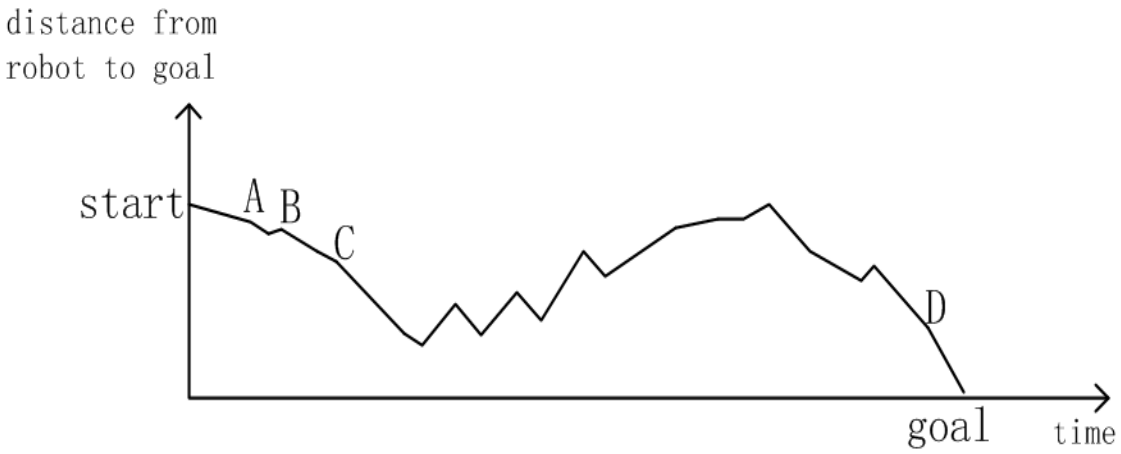

Figure 6: The graph of distance

From the figure 6, you can see the continuous function itself(with search phase) is not a monotonic function, but if we select the values at hit points and leave points and together form a new discrete function, it is always a monotonically decreasing function.

2.4 iv

The reason is really simple. Each time the robot encounter the obstacle, it is only allowed to leave the obstacle at the leave point which makes it closer to the goal. If it can not find such a point, it means there does not exist such a possibly path between the start point and the goal point, which contradicts to what the question assumes. For the same reason, the robot could not hit the same obstacle twice because the distance function at each hit points and leave points is a monotonically decreasing function. Besides, there are finite number of obstacles in the environment, so the distance could finally decease to a certain value at the leave point of the obstacle which is closest to the goal point. if the path really exist, there must be some free space around the goal point. In other words, the robot could always reach the goal point along the start-goal line starting from the leave points of the closest obstacle. Hence, it could finally reach the goal point.

3 Programming:Lines and segments

3.1 computeLineThroughTwoPoints

It is easy to derive the analytic formulas as I show below:

$$\begin{cases} x_1a + y_1b + c = 0 \\ x_2a + y_2b + c = 0 \\ a^2 + b^2 = 1 \end{cases} \tag{1}$$

$$\begin{cases} (x_1 - x_2)a + (y_1 - y_2)b = 0 \\ a^2 + b^2 = 1 \end{cases} \tag{2}$$

From the above equations, we can easy obtain the answers.

$$\begin{cases} a = \frac{1}{\sqrt{1+(\frac{x1-x2}{y1-y2})^2}} \\ b = -\frac{x1-x2}{y1-y2} * a \\ c = -x1 * a - y1 * b \end{cases} \quad \begin{cases} a = -\frac{1}{\sqrt{1+(\frac{x1-x2}{y1-y2})^2}} \\ b = -\frac{x1-x2}{y1-y2} * a \\ c = -x1 * a - y1 * b \end{cases} \tag{3}$$

Althought there are two solutions, they represent the same line equation(For the second solution, I could multiple -1 on both sides of the equation). The correctness of the answers could be esaily done by tests using Python.

Pay attention, this method could not work when the two points are on a horizonal line:$y_1 - y_2 = 0$. In this case, a does not exist. However, since there is only one special case, I could test whether the input two points fall into this case, and if so, I would quickly obtain the formula; if not, do the above calculations.

3.2 computeDistancePointToLine

From the above part, we could obtain the analytic formula of a line by point p_1 and point p_2. As we learnt from high school, the distance(D) from a point($p = (x1, y1)$) to a line($ax + by + c = 0$) could be obtained through a simple formula:

$$D = \frac{|ax_1 + by_1 + c|}{\sqrt{a^2 + b^2}} \tag{4}$$

From this definition, I could easily calculate the distance.

There is no special case here if I could obtain the line formula from the previous part.

3.3 computeDistancePointToSegment

For the distance from q to the segment with extreme points (p_1, p_2), I could use the function shown above. Also, by comparing the distances between point q and points $p1, p2$, I could know which one is closer to q. In order to conclude whether the closest point to q is strictly inside the segment or not, I need to find the intersection point of the given line segment and the line which passes through q and is orthorgoal to the line segment. If the intersection point is strictly inside the segment, w should be 0. Otherwise, w should be given with the value corrsponding to one of the extreme points.

$$\begin{cases} a_1x + b_1y + c_1 = 0 \text{(known using two extreme points)} \\ a_2x_q + b_2y_q + c_2 = 0 \\ a_1a_2 + b_1b_2 = 0 \\ a_2^2 + b_2^2 = 1 \end{cases} \tag{5}$$

From the above equations, I could obtain the formula for the line which passes through q and is orthorgoal to the line segment.

$$\begin{cases} a_1x + b_1y + c_1 = 0 \\ a_2x + b_2y + c_2 = 0 \end{cases} \tag{6}$$

After I obtain the intersection point, It is very easy to see whether it is strictly inside the line segment.

Also, there are two special cases.The first one is when the two points are on a vertical line and the second line is horizontal. The second one is when the two points are on a horizontal line and the second line is vertical. In these two cases, I could not use the same way to calcalate the slopes of two lines. However, since they are really special and it's esay to tell whether the input points fall in one of them, I could handle them without any difficulty.

3.4 Verification

For the first part, I input four groups of points which correspond to four different cases respectively. The four cases are:

- The two points are on a general line
- The two points are on a horizontal line
- The two points are on a vertical line
- The two points are the same

```python
if __name__=='__main__':
    p1=[1, 1]
    p2=[2, 2]

    p3=[1, 1]
    p4=[2, 1]

    p5=[1, 1]
    p6=[1, 2]

    p7=[1, 1]
    p8=[1, 1]
    q=[3, 4]
    a, b, c=computeLineThroughTwoPoints(p1, p2)
    a1, b1, c1=computeLineThroughTwoPoints(p3, p4)
    a2, b2, c2=computeLineThroughTwoPoints(p5, p6)
    a3, b3, c3=computeLineThroughTwoPoints(p7, p8)
    if (a, b, c)==(0, 0, 0):
        print("p1 and p2 must be two different points!")
    else:
        print(a, b, c)
    if (a1, b1, c1)==(0, 0, 0):
        print("p1 and p2 must be two different points!")
    else:
        print(a1, b1, c1)
    if (a2, b2, c2)==(0, 0, 0):
        print("p1 and p2 must be two different points!")
    else:
        print(a2, b2, c2)
    if (a3, b3, c3)==(0, 0, 0):
        print("p1 and p2 must be two different points!")
    else:
        print(a3, b3, c3)
```

Figure 7: Verification codes

```
>>>
 RESTART: C:\Users\Administrator\Desktop\winter quarter\MAE145\Homework\Homework
1\python_program.py
0.7071067811865475 -0.7071067811865475 0.0
0 1 -1
1.0 0.0 -1.0
p1 and p2 must be two different points!
```

Figure 8: Results

From the above two figures, you could see the results are correct.

For the second part, I test the function in two different cases:the point in on the line or off the line.

```
if __name__=='__main__':
    p1=[1,1]
    p2=[2,2]
    q=[3,4]
    q1=[1.5,1.5]
    a,b,c=computeLineThroughTwoPoints(p1,p2)
    if (a,b,c)==(0,0,0):
        print("p1 and p2 must be two different points!")
    else:
        D1=computeDistancePointToLine(q,p1,p2)
        D2=computeDistancePointToLine(q1,p1,p2)
        print(D1)
        print(D2)
```

Figure 9: Verification codes

```
>>>
 RESTART: C:\Users\Administrator\Desktop\winter quarter\MAE145\Homework\Homework
1\python_program.py
0.7071067811865475
0.0
```

Figure 10: Results

From the above two figures, you could see the results are correct.

For the third part, I test the function in different three cases when w should be assigned different values.

```
if __name__=='__main__':
    p1=[1,1]
    p2=[2,2]
    q=[3,4]
    q1=[1,2]
    q2=[10,8]
    q3=[-10,8]
    a,b,c=computeLineThroughTwoPoints(p1,p2)
    if (a,b,c)==(0,0,0):
        print("p1 and p2 must be two different points!")
    else:
        D,w=computeDistancePointToSegment(q1,p1,p2)
        print(D,w)
        D,w=computeDistancePointToSegment(q2,p1,p2)
        print(D,w)
        D,w=computeDistancePointToSegment(q3,p1,p2)
        print(D,w)
```

Figure 11: Verification codes

```
 RESTART: C:\Users\Administrator\Desktop\winter quarter\MAE145\Homework\Homework
1\python_program.py
0.7071067811865476 0
1.4142135623730951 2
12.727922061357855 1
```

Figure 12: Results

From the above two figures, you could see the results are correct.