Final Report: DCCP Applied in Obstacle Avoidence

Wu Lan

2019/8/15

1 Introduction

In the previous six weeks, I focused on the implementation of DCCP package in different obstacle settings. Firstly, the simulation of the former testbed is set up with several examples of route planning simulation. Secondly, the obstacles are turned into circles and the DCCP algorithm is implemented in the simplest scenarios. Thirdly, various obstacle settings are designed with random starting point and ending point in order to examine different shapes of route planning. Fourthly, I set several complex mazes to test the ability of DCCP route planning and implement it in the map of Pembroke College.

- Week 1: Simulation of testbed & Previous Examples
- Week 2: Simple Implementation of DCCP
- Week 3: Examination of Planned Route in Different Settings
- Week 4: Maze Settings
- Week 5: Map Implementation

2 Previous Work and Week1 Simulation

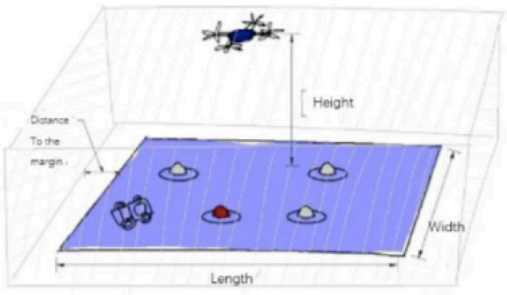


Figure 1: Illustration of the Miniaturized UAV-assisted UGV Testbed

In Figure 1, the miniaturized UAV-assisted UGV testbed is the prototype of the simulation system. The testbed is composed of one UAV flying at the height of about 3 meters, one UGV running on

the floor in an area of about 6x6 meters and a few beacon lights. For testing purposes, the testbed will only activate one beacon light at random each time. Only after the UGV passes through the activated beacon light, will the testbed turn on the next beacon light randomly. The test is designed to investigate how effective UAV can guide UGV towards its targets. And in each simulation, the value of starting point, car direction, and destination will be randomly assigned. Based on this model, three route planning algorithms without the function of obstacle avoidence are shown in the following.

- First-order Route Planning
  In Figure 2, the approximation method is to gradually approach the target point by rotating a certain proportion of the angle between the car-point (C0) and the target-point (T0) each time.

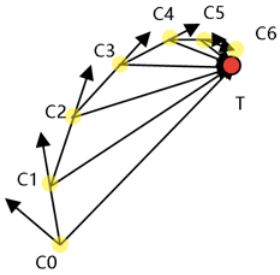


Figure 2: Approximation Method Illustration

This method does not produce a clear path, but adjust the direction of the car by capturing the position usign the camera in real time. Since the car is a differential steering system, the car direction can be adjusted by controling the speed of two motors, let they be Vin (inside speed,the smaller one) and Vout (outside speed, the bigger one). According to a paper about the differential steering system written by G.W.Lucas, there is a formula, let Vt be the direction from C_0 to T_0 :

$$\alpha = (V_0 - V_i)t/d \tag{1}$$

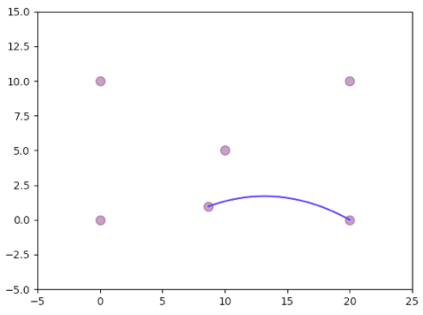


Figure 3: Simulation 1

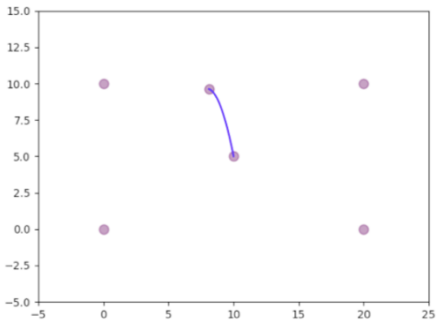


Figure 4: Simulation 2

- Quadratic Route Planning
  In Mathematics perspective, quadratic route appears to be much smoother due to the changing curvature compared to the first-order route planning. And under various scenarios, we can set different restricted conditions. Pictures in the following are outcome of the shortest path restriction in quadratic planning:

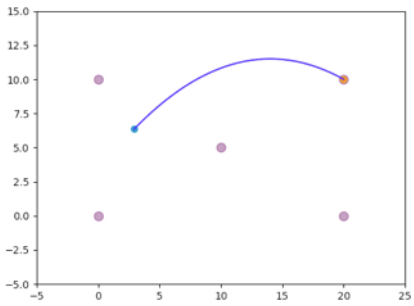


Figure 5: Simulation 1

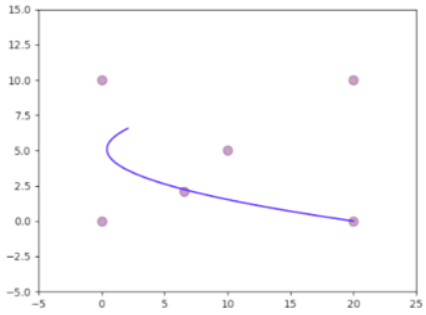


Figure 6: Simulation 2

- CVX Planning

```
constraint = []
## Hit 3 chosen way points
constraint += [x[:, fix_t0] == p0]
constraint += [x[:, fix_t1] == p1]
constraint += [x[:, fix_t2] == p2]
## Starting pos + vel
constraint += [x[:,0] == x_0.flatten()]
constraint += [u[:,0] == u_0.flatten()]
## Forward velocity (vectorized)
constraint += [x[:,1:] == x[:,:-1] + dt*u[:, :-1]]
diff_u = u[:, 1:] - u[:, :-1]
cost = cvx.sum_squares(diff_u)
problem = cvx.Problem(cvx.Minimize(cost), constraint)
problem.solve(solver=cvx.ECOS,verbose=False)
```

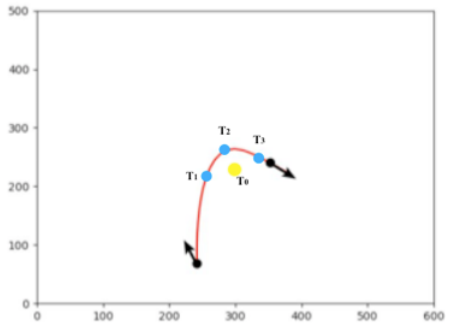


Figure 7: CVX Simulation

3 Simple Implementation of DCCP

In this section, I focus on the simple implementation of the disciplined convex- concave programming (DCCP), which combines the ideas of disciplined convex programming (DCP) with convex-concave programming (CCP). The Convex-concave programming is an organized heuristic for solving non-convex problems that involve objective and constraint functions that are a sum of a convex and a

concave term (Gu & Boyd, 2016). Generally speaking, previously we can only solve problems with convex object function and convex domain. Thus when applying it to real applications like route-planning by the cvxpy package in python, it largely restricts the range of limiting conditions. Since obstacles are ubiquitous in real cases, the route-planning system should not depend on an algorithm which could not solve problems with restrictions. Fortunately, based on the Hartman theorem, the Convex Optimization Problem is proved to have more extensive implications and it is possible to use similar method solving DCCP problem. With the DCCP package in python, we are able to use several simplified functions calculating an optimal route object to verified limiting conditions.

3.1 Method

- Turn Obstacles into Circles

Algorithm 1 Obstacle Analyzing

1. **Take** a picture of the road condition including car and obstacles
2. **Load** the color image and access the pixel value by its row and column coordinates
3. **Distinguish** obstacles and find the contour of it
4. **Find** the smallest circle to include each obstacle
5. **Output** the origin and radius of each circle

```
import cv2
import numpy as np
from picamera import PiCamera
from picamera.array import PiRGBArray
rawCapture = PiRGBArray(camera)
camera.capture(rawCapture,'bgr', use_video_port = True)
image = rawCapture.array
x_size = image.shape[1]
x_pic_length = rangefinder/range_pic_ratio
output_gray = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
ret, thresh = cv2.threshold(output_gray,threshold,255,cv2.THRESH_BINARY)
useless, contours, hierarchy = cv2.findContours(thresh,cv2.RETR_EXTERNAL,
                                        cv2.CHAIN_APPROX_SIMPLE)
(x1,y1),radius1 = cv2.minEnclosingCircle(cnt1)
```

- Solve the Shortest Path

  The goal is to find the shortest path connecting points a and b in R^d that avoids m circles, centered at p_j with radius r^j, j = 1,...,m. After discretizing the arc length parametrized path into points $x_0, \ldots, x_n$, the problem is posed as

$$\begin{array}{ll} \text{minimize} & L \\ \text{subject to} & x_0 = a, \quad x_n = b \\ & \|x_i - x_{i-1}\|_2 \leq L/n, \quad i = 1, \ldots, n \\ & \|x_i - p_j\|_2 \geq r_j, \quad i = 1, \ldots, n, \quad j = 1, \ldots, m, \end{array}$$

```
import cvxpy as cvx
import numpy as np
import dccp
x = cvx.Variable(d,n+1)
L = cvx.Variable()
constr = [x[:,0] == a, x[:,n] == b]
for i in range(1,n+1):
constr += [norm(x[:,i] - x[:,i-1])<= L/n]
for j in range(m):
constr += [norm(x[:,i] - center[:,j]) >= r[j]]
prob = Problem(Minimize(L), constr)
prob.solve(method = 'dccp')
```

3.2 Simulation Result

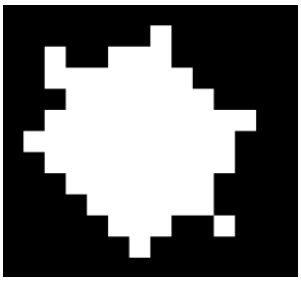


Figure 8: Obstacle Example

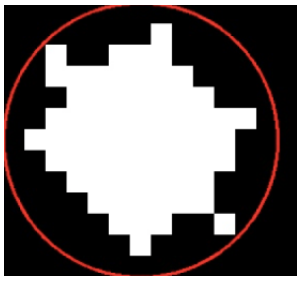


Figure 9: Finding the Smallest Circle

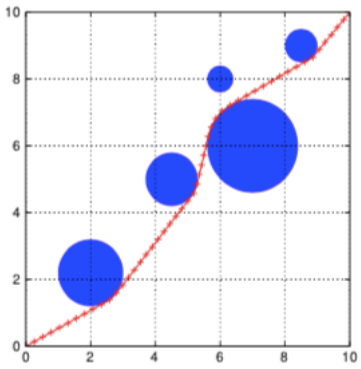


Figure 10: d=2, n=50

4 Examination of Planned Route in Different Settings

This section will briefly introduce the different outcomes as well as simulation settings with DCCP implementation. On one hand, I recognize obstacles and turn them into different shapes like circle and rectangle in order to implement the shortest path algorithm. On the other hand, I also discuss different scenarios if we randomly set the starting point and determination or change the properties of obstacles in terms of scale and location.

4.1 Method

In order to turn obstacles into different shapes, I simply take two distance calculation method when setting the constraints in DCCP program. As shown in the following figures, the Euclidian distance in 2-dimension space will naturally form a circle while the Manhattan distance can form a square, which largely influence the route-planning outcome.

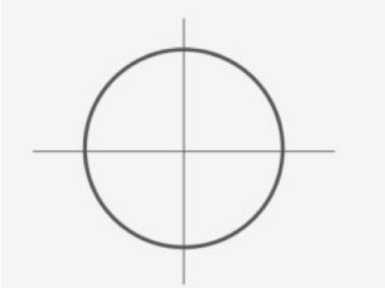


Figure 11: Euclidean Distance (d_1)

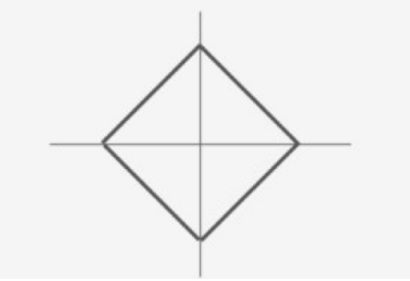


Figure 12: Manhattan Distance (d_2)

$$d_1(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2} \tag{2}$$

$$d_2(I_1, I_2) = \sum_p |I_1^p - I_2^p| \tag{3}$$

- Constraints of Circle Obstacles:

```
x = []
for i in range(n+1):
x += [Variable((d, 1))]
L = Variable(1)
constr = [x[0] == a, x[n] == b]
cost = L
for i in range(n):
constr += [norm(x[i]-x[i+1]) <= L/n]
for j in range(m):
constr += [norm(x[i]-p[:,j]) >= r[j]]
prob = Problem(Minimize(cost), constr)
```

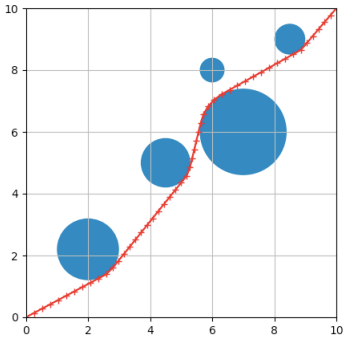
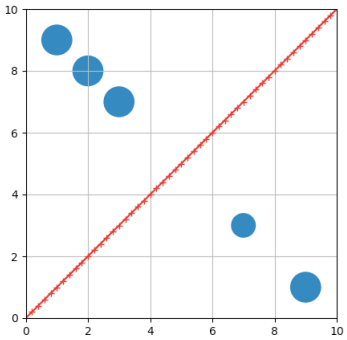
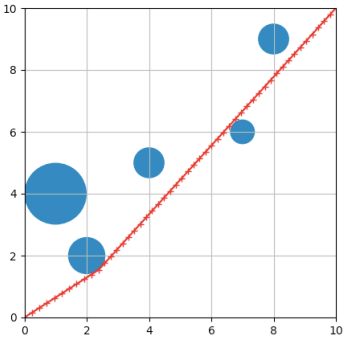
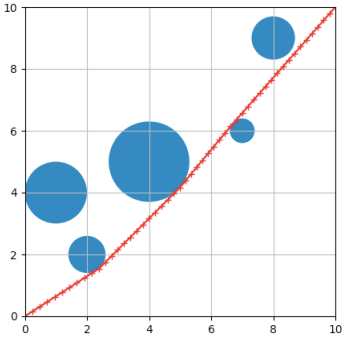
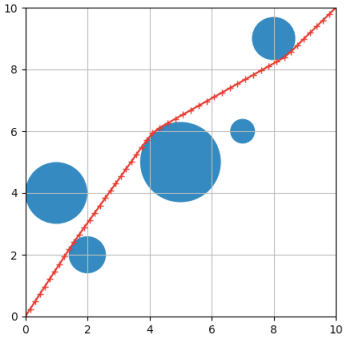
- Constraints of Rectangular Obstacles:

```
x = []
for i in range(n+1):
x += [Variable((d, 1))]
L = Variable(1)
constr = [x[0] == a, x[n] == b]
cost = L
for i in range(n):
constr += [norm(x[i]-x[i+1]) <= L/n]
for j in range(m):
```

```
constr += [abs(x[i][0]-p[:,j][0]) + abs(x[i][1]-p[:,j][1]) >= r[j]]
prob = Problem(Minimize(cost), constr)
print "begin to solve"
result = prob.solve(method='dccp')
print "end"
```

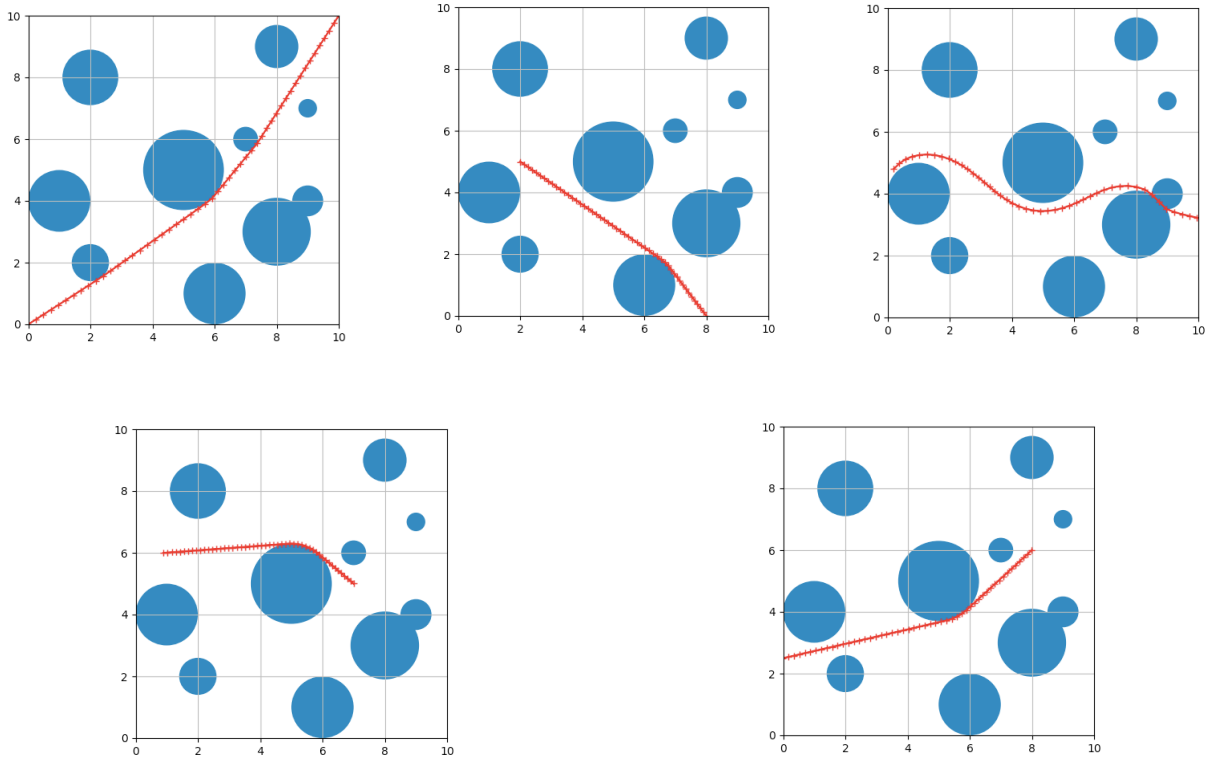
4.2 Simulation Result

- Circle & Settled Starting/Ending Points

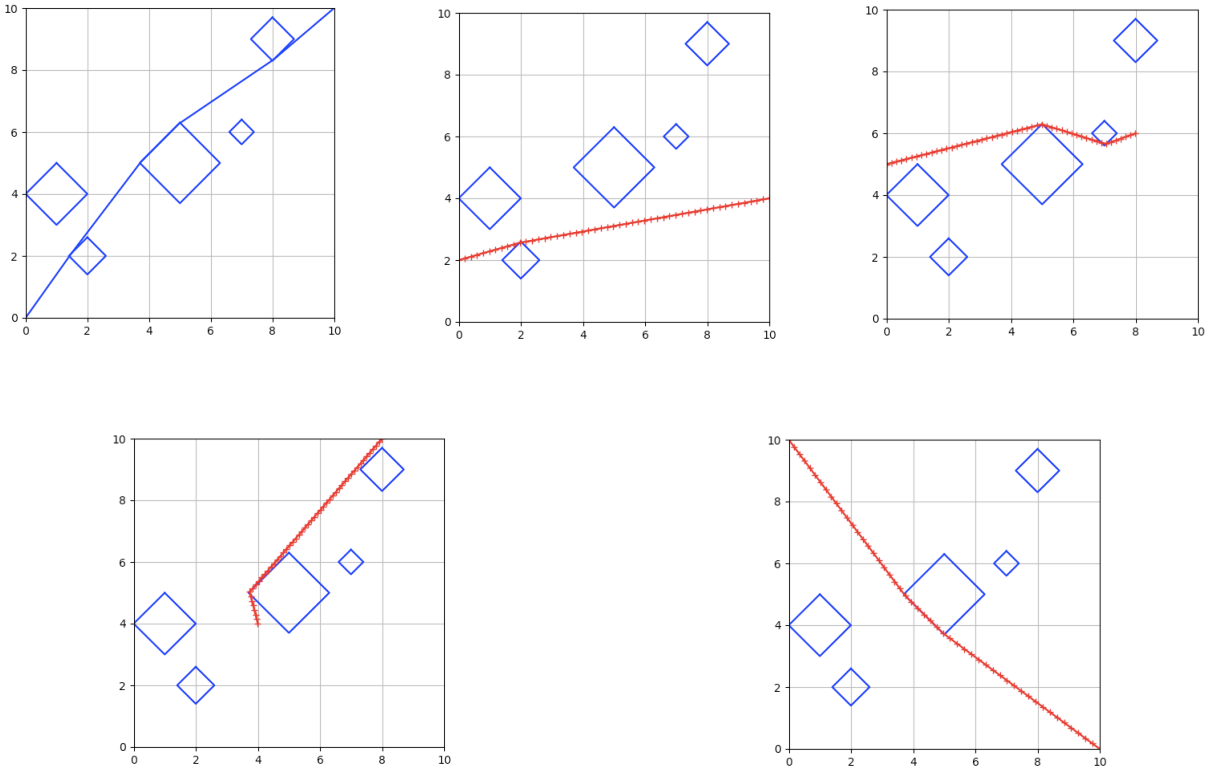
With settled starting point and ending point, we can find that the shape of the planned route is influenced by the location of the obstacles. In some cases, if the obstacles are placed away from the line connected two points, the shortest path is absolutely the straight line. And when the original path is influenced by obstacles to a small extent, the planned route appear to be conterminous segments tangent to those circles in the way. Also, in the cases that obstacles impact the previous straight line to a large extent, curves will be the main element of the final shortest path. Apparently, those curves are part of the periphery of obstacles in most situations.

- Circle & Random Starting/Ending Points



Outcomes in these figures are quite similar to settled S/E situations. However, one point should be highlighted is that in the third figure above the route goes through the narrow gap between two circles. On one hand, in real circumstances, we should avoid this kinds of planned route for safety concerns. On the other hand, given that car also has volume, we should better set the radius of circle much larger.

- Rectangle & Random Starting/Ending Points



For rectangle obstacles, the shape of route will be much easier since it consists of segments only. It also can be a straight line, but most time it appears to be conterminous segments containing the vertex and side of the rectangle.

5 Maze Setting

This section focuses on a more complex setting of mazes in order to prove DCCP's ability in shortest path planning under different scenarios. So firstly, the circle packing algorithm is implemented still with the DCCP package, showing a list of circles externally tangent with each other packing in the smallest square. Therefore, we can get a suitable pattern setting avoiding possibility of overlap. Similarly, we can also change the element of maze into square or obtain a mix of square and circle seeing which pattern looks like the original obstacle to a larger extent. To set the overall pattern more likely to be a maze, I also adjust parameters and leave gap among circles. After that, we randomly choose several starting points and ending points to do the route-planning. The results will be shown in the following report.

- Circle Packing
  In order to package circles inside a minimum square, we need to set the place of their origin one by one. For example, assuming that there are totally m circles (c_1, c_2,$\cdots c_m$) with corresponding radius (namely r_i) and origin (x_i and y_i), we need to check the place from the first to the end. For circle c_j, it should be compared with circles whose number larger than i.

$$\sqrt{(x_j - x_q)^2 + (y_j - y_q)^2} \preceq r_j + r_q \tag{4}$$

  where q > j.
  Finally we minimize the area circles occupy with the following code:

```
constr = []
for i in range(n-1):
for j in range(i+1, n):
constr.append(cvx.norm(cvx.vec(c[i,:]-c[j,:]), 2) >= r[i]+r[j]+0.1)
prob1 = cvx.Problem(cvx.Minimize(cvx.max(cvx.max(cvx.abs(c), axis=1) +
                                         r)), constr)
prob1.solve(method = 'dccp', solver='ECOS', ep = 1e-2, max_slack = 1e-2
                                         )
```

  Here are figures of maze setting created by the circle packing algorithm:

```
#Parameter Setting of Figure 1
r = np.linspace(1,4,10)
#Parameter Setting of Figure 2
r = 1
#Parameter Setting of Figure 3
r = np.linspace(1,2,50)
```

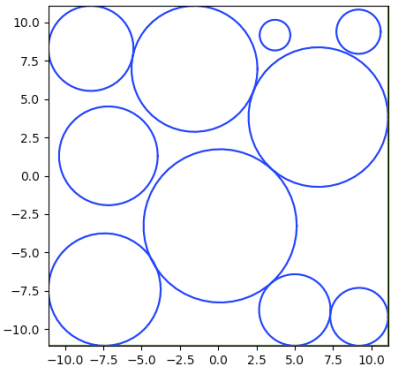


Figure 13: n = 10

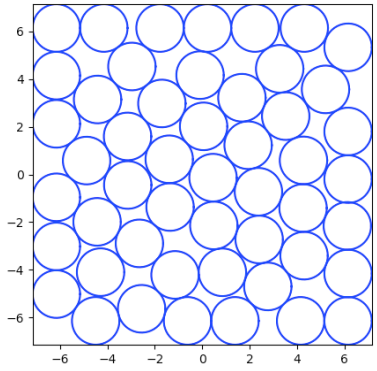


Figure 14: n = 50

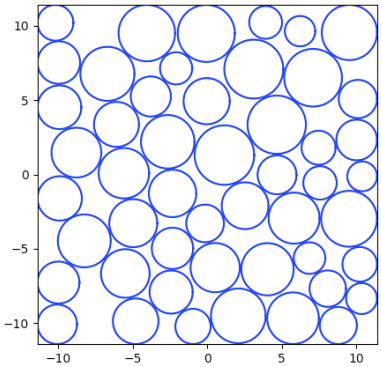


Figure 15: n = 50

In order to make those figures more like a maze with gap, the equation changes to

$$\sqrt{(x_j - x_q)^2 + (y_j - y_q)^2} \preceq r_j + r_q + 0.1 \tag{5}$$

and the result is as following with specific starting point and ending point.

```
x = []
for i in range(50+1):
x += [cvx.Variable((2, 1))]
L = cvx.Variable(1)
constr = [x[0] == a, x[50] == b]
cost = L
for i in range(50):
constr += [cvx.norm(x[i]-x[i+1]) <= L/50]
for j in range(50):
constr += [cvx.norm(x[i]-p[:,j]) >= r[j]]
prob2 = cvx.Problem(cvx.Minimize(cost), constr)
print "begin to solve"
result = prob2.solve(method='dccp')
print "end"
```

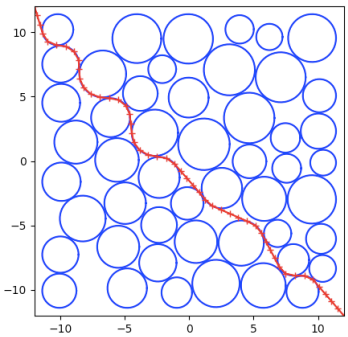


Figure 16: S: (-12, 12) E: (12, -12)

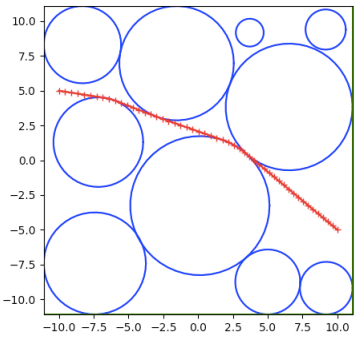


Figure 17: S: (-10, 5) E: (10, -5)

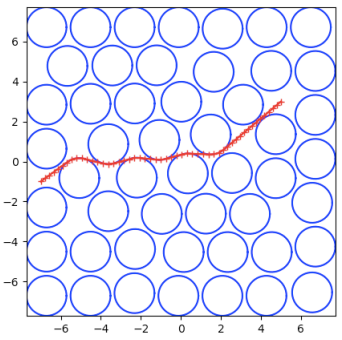


Figure 18: S: (-6.5, -1) E: (5, 3)

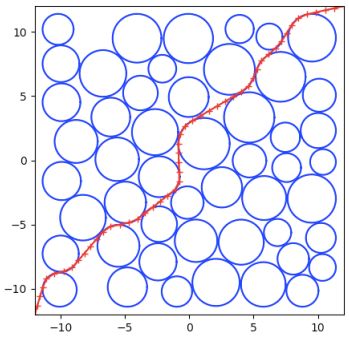


Figure 19: S: (-12, -12) E: (12, 12)

- Square Packing

  Similarly, we use square to replace circles in the code above following the same origin set method. Still by minimizing the total area, we get settings:

```
constr = [x[0] == a, x[n] == b]
cost = L
for i in range(n):
constr += [norm(x[i]-x[i+1]) <= L/n]
for j in range(50):
constr += [abs(x[i][0]-p[:,j][0]) + abs(x[i][1]-p[:,j][1]) >= r[j]]
prob = Problem(Minimize(cost), constr)
result = prob.solve(method='dccp')
```

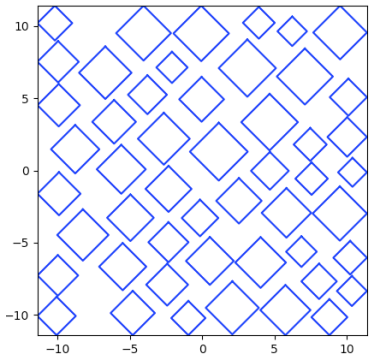


Figure 20: Setting

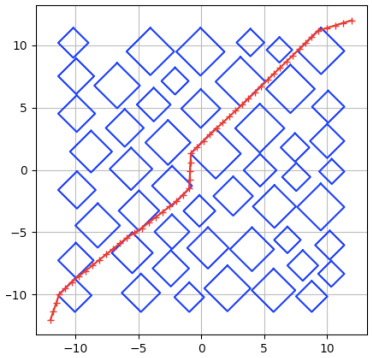


Figure 21: S: (-12, -12) E: (12, 12)

6 Map Simulation

Here is a map of the Pembroke College, and I will take a part of it to do the path planning.

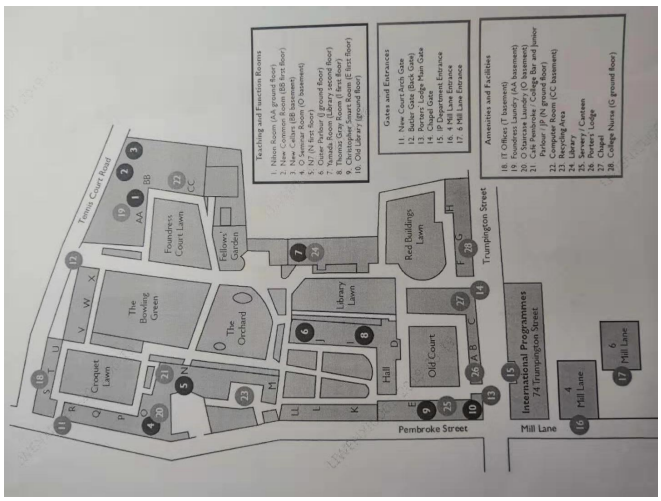


Figure 22: Pembroke College Map

Each of the building will be represented by a list of circles in order to simulate the shape of it. After that, the DCCP package will be applied to find the shortest path.

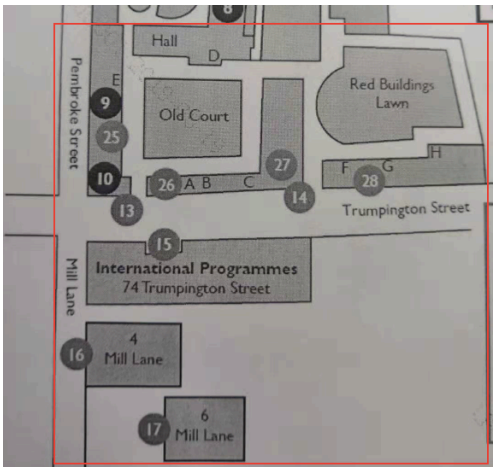


Figure 23: Simulation Part of the Map

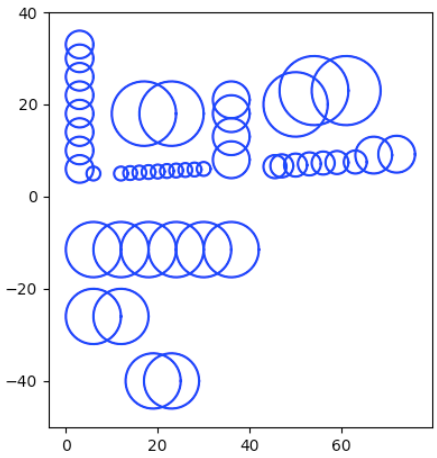


Figure 24: Simulation

Errors appear like what shown in the following picture. On one hand, the number of point we set in order to plan the route is small (here $n = 50$). When two points are far away with each other, they may turn out to be a line passing the circle of building. On the other hand, the error also indicates that the range of both x and y should be limited in order to get a optimized path.

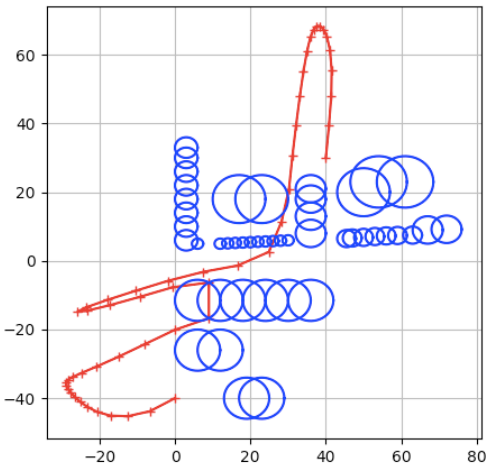


Figure 25: Route Planning with error

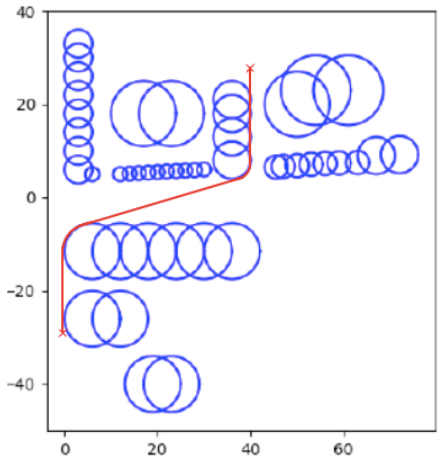


Figure 26: Correct Planning

Here, for the figure on the right, I increase n to 100 and set a range for both x and y with the following constraints:

```
constr = [x[0] == a, x[n] == b]
cost = L
n = 100
for i in range(n):
constr += [norm(x[i]-x[i+1]) <= L/n]
constr += [-10<=x[i].value[0]<=90, -50<=x[i].value[1]<=40]
for j in range(m):
constr += [norm(x[i]-p[:,j]) >= r[j]]
```

7 Conclusion

The six-week supervision provides me with fantastic research experience in various aspects. Firstly, in simulation level, I really appreciate the progress I make in DCCP applications under the guidance of my supervisor. From a simple setting optimization to the complex maze and the final map, the supervision opportunity endows me with a better skill handling obstacle avoidance problem as well as a deeper understanding of DCCP implementation. Secondly, my supervisor gives me many useful suggestions like powerful tools and platforms in the research process. For instance, the Github has a lot of shared code which makes sense when solving problems; the Jupyter notebook also owns strong advantages and clear interface. All these tools are meaningful for this supervision process and are sure to benefit more in my future learning. What's more, the requirement and advice from the supervisor greatly promotes my ability in independent thinking, which also shows me the implementation value of the research itself. I will continually focus on this interesting topic and try to do better in the near future.