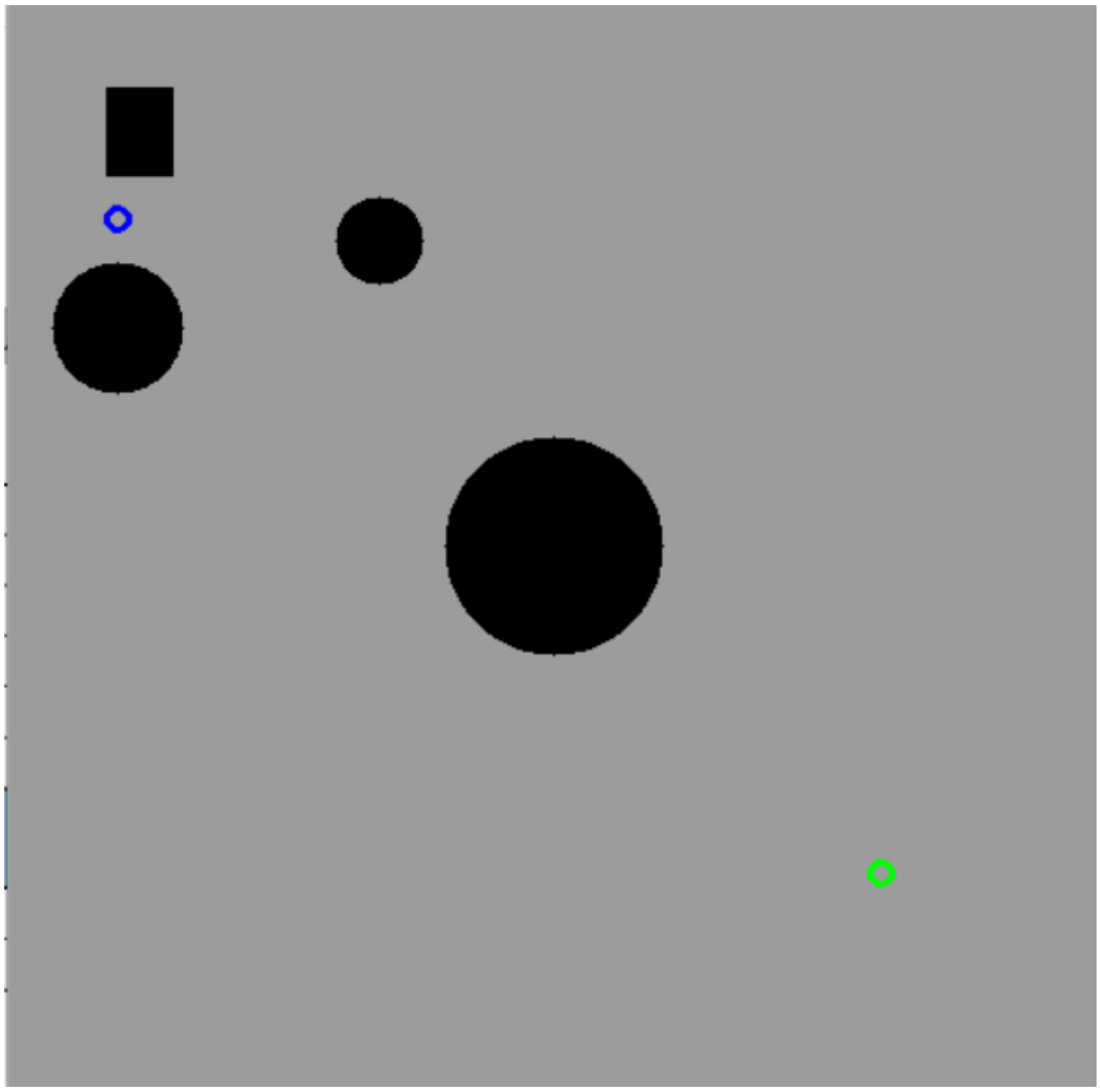
群体智能算法建模报告

1．问题提出

对于这次算法建模所求解的问题，我选择的是路径规划问题。

该问题的实际情况可以理解为：对于一个机器人，在一个具体空间环境下，需要如何避开环境中的障碍物，最终可以到达终点，并使得移动的距离最小。换言之，实际上就是给定环境信息，如果该环境内有障碍物，寻求起始点到目标点的最短路径，并且路径不能与障碍物相交。如图所示：



在本次建模实验中，我所建立的空间环境为一个`500*500`的正方形，其中有四个障碍物，包括三个圆形以及一个长方形。蓝色的点为起点，绿色的点为终点。

2．遗传算法

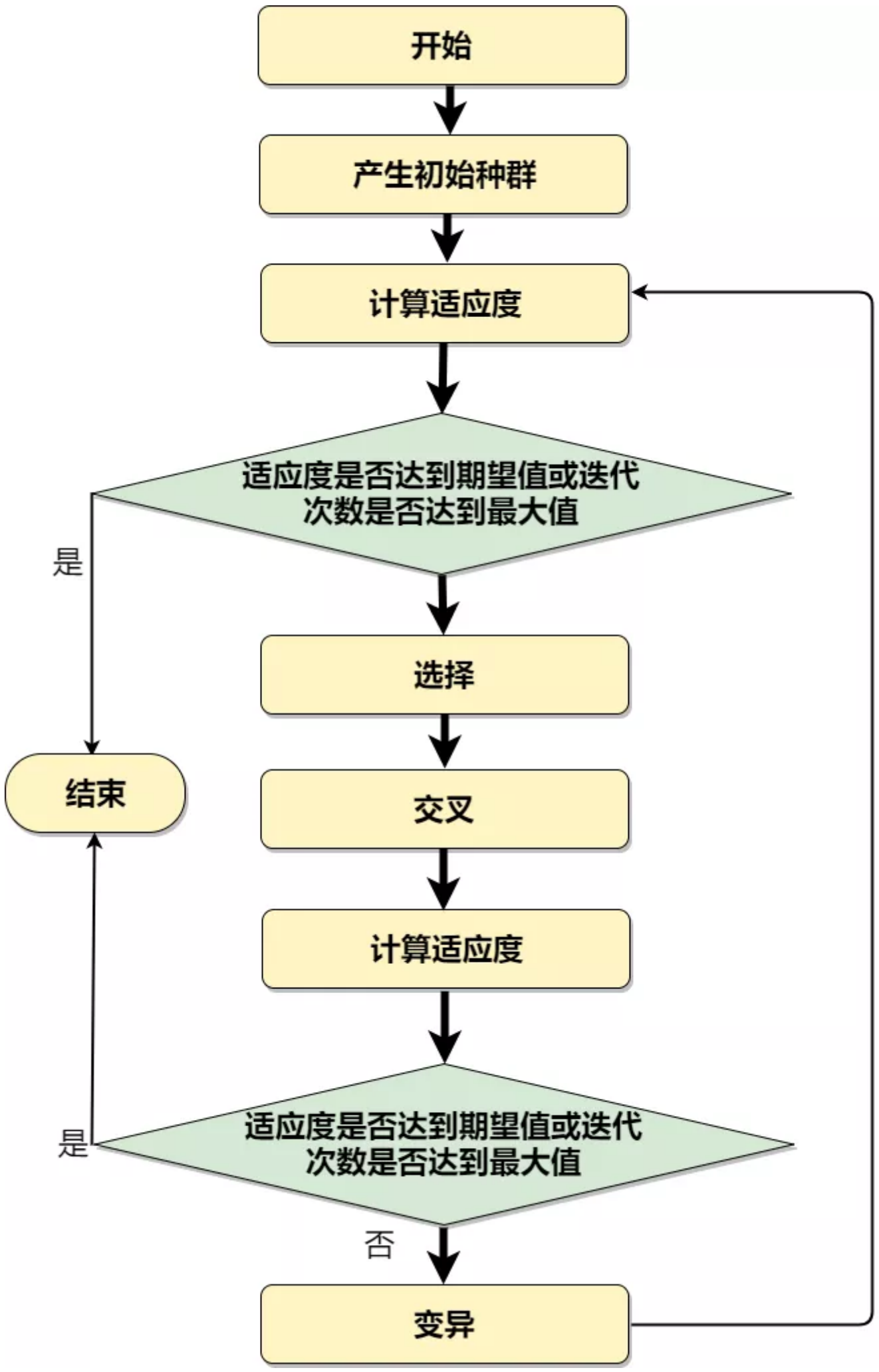
2.1 算法介绍

遗传算法（Genetic Algorithm，GA）起源于对生物系统所进行的计算机模拟研究。它是模仿自然界生物进化机制发展起来的随机全局搜索和优化方法，借鉴了达尔文的进化论和孟德尔的遗传学说。其本质是一种高效、并行、全局搜索的方法，能在搜索过程中自动获取和积累有关搜索空间的知识，并自适应地控制搜索过程以求得最佳解。可以把遗传算法的过程看作是一个在多元函数里面求最优解的过程。可以这样想象，这个多维曲面里面有数不清的“山峰”，而这些山峰所对应的就是局部最优解。而其中也会有一个“山峰”的海拔最高的，那么这个就是全局最优解。而遗传算法的任务就是尽量爬到最高峰，而不是陷落在一些小山峰。

遗传算法的有趣应用很多，诸如寻路问题，8数码问题，囚犯困境，动作控制，找圆心问题（在一个不规则的多边形中，寻找一个包含在该多边形内的最大圆圈的圆心），TSP问题，生产调度问题，人工生命模拟等。

2.2 算法思路

对于遗传算法中的每一个个体，其都用一定编码方式形成基因，借助遗传算子，选择、交叉、变异操作，对种群进行演化，选择出更适应环境的种群。算法流程图如下所示：



- 初始化

遗传算法中每一条染色体，对应着遗传算法的一个解决方案。而在本例中，一个解决方案实际上就是从起点到终点的一条路径，也就是是从起点到终点所途径的一系列的点对集合。为了方便，这里将路径拆分为x轴坐标和y轴坐标，每一对x轴路径和y轴路径组合后才是真正的路径。

```
    def init_generation(self):
        # init population
        gen_x = np.zeros((self.population_size, self.dim + 2))
        gen_y = np.zeros((self.population_size, self.dim + 2))
        for i in range(self.population_size):
            for j in range(self.dim + 2):
                cur_x, cur_y = self.generate_random_point()
                while self.pointCollison(cur_x, cur_y):
                    cur_x, cur_y = self.generate_random_point()
                gen_x[i, j] = cur_x
                gen_y[i, j] = cur_y
        gen_x[:, 0] = self.start_x
        gen_x[:, -1] = self.end_x
        gen_y[:, 0] = self.start_y
```

```
        gen_y[:, -1] = self.end_y
        return gen_x, gen_y
```

- 变异

复制时可能（很小的概率）产生某些复制差错，变异产生新的染色体，表现出新的性状。而在本例中，我们采用随机变异的方式，也就是对于基因中的每个位点都按照一定的几率进行变异。具体来讲，对于每个位点都产生一个随机数，如果随机数的值大于变异概率，则将其进行变异。

```
        for i in range(1, size_x):
            p = random.random()
            if p < self.pc:
                for j in range(1, size_y - 1):
                    mutation_x = self.x_min + (self.x_max - self.x_min) * random.random()
                    mutation_y = self.y_min + (self.y_max - self.y_min) * random.random()
                    while self.pointCollison(mutation_x, mutation_y):
                        mutation_x = self.x_min + (self.x_max - self.x_min) * random.random()
                        mutation_y = self.y_min + (self.y_max - self.y_min) * random.random()
                    new_gen_x[i, j] = mutation_x
                    new_gen_y[i, j] = mutation_y

            else:
                new_gen_x[i, :] = gen_x[i, :]
                new_gen_y[i, :] = gen_y[i, :]
```

- 交叉

这里的交叉方式采用算术交叉。着是指由两个个体的线性组合而产生出两个新的个体。算术交叉的操作对象一般是由浮点数编码所表示的个体.其定义为两个向量（染色体）的组合:$x1 = \lambda1 \times 1 + \lambda2 \times 2$; $x2 = \lambda1 \times 2 + \lambda2 \times 1$，其中λ1、λ2称为乘子。这里选择的$\lambda_2 = 1 - \lambda_1$，$\lambda_1$是0~1的随机数。不过交叉仅在随机数大于交叉概率时发生。

```
        for i in range(1, size_x - 1, 2):
            p = random.random()
            if p <= self.pa:
                cross_ratio = random.random()
                new_gen_x[i, :] = cross_ratio * gen_x[i, :] + (1 - cross_ratio) * gen_x[i + 1, :]
                new_gen_y[i, :] = cross_ratio * gen_y[i, :] + (1 - cross_ratio) * gen_y[i + 1, :]
                new_gen_x[i + 1, :] = cross_ratio * gen_x[i + 1, :] + (1 - cross_ratio) * gen_x[i, :]
                new_gen_y[i + 1, :] = cross_ratio * gen_y[i + 1, :] + (1 - cross_ratio) * gen_y[i, :]
            else:
                new_gen_x[i, :] = gen_x[i, :]
                new_gen_y[i, :] = gen_y[i, :]
                new_gen_x[i + 1, :] = gen_x[i + 1, :]
                new_gen_y[i + 1, :] = gen_y[i + 1, :]
```

- 选择

在进行选择操作时，首先需要定义一个解决方案的优劣。

我们可以用适应性函数来衡量这个解决方案的优劣。在本例中，适应性函数为:

$$fitness\_function(x) = \begin{cases} e^{150/distance}，未与障碍物相撞 \\ 0，与障碍物相撞 \end{cases}$$

而选择时，我们所采用的选择方法时锦标赛选择法。

遗传算法中的锦标赛选择策略每次从种群中取出一定数量个体（放回抽样），然后选择其中最好的一个进入子代种群。重复该操作，直到新的种群规模达到原来的种群规模。其具体操作为:

（1）确定每次选择的个体数量N

（2）从种群中随机选择N个个体(每个个体被选择的概率相同），根据每个个体的适应度值，选择其中适应度值最好的个体进入下一代种群

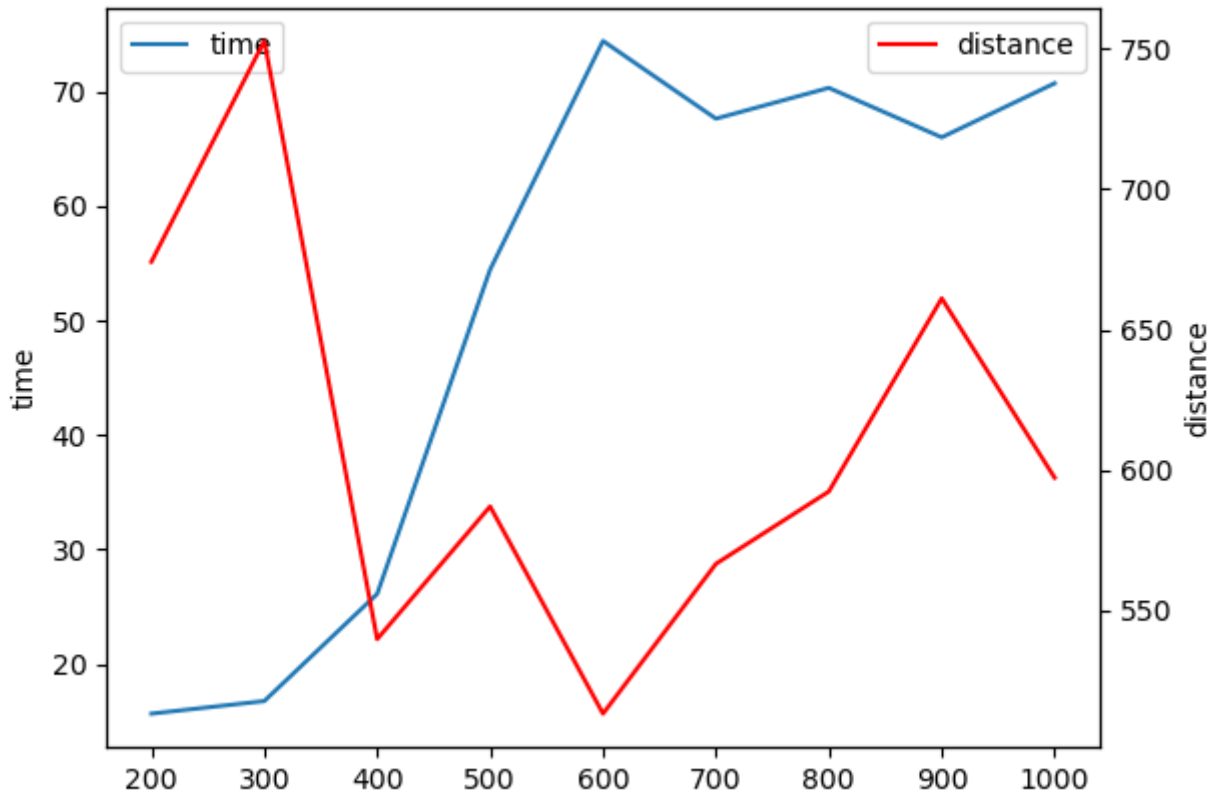
（3）重复步骤(2)多次（重复次数为种群的大小），直到新的种群规模达到原来的种群规模

```
    def select(self, gen_x, gen_y, fitness_value):
        size_x, size_y = gen_x.shape
        new_gen_x = np.zeros(gen_x.shape)
        new_gen_y = np.zeros(gen_y.shape)
        for i in range(size_x):
            mask = np.random.rand(size_x) >= 0.3
            best_index = np.argmax(fitness_value[mask])
            new_gen_x[i, :] = gen_x[best_index, :]
            new_gen_y[i, :] = gen_y[best_index, :]
        return new_gen_x, new_gen_y
```

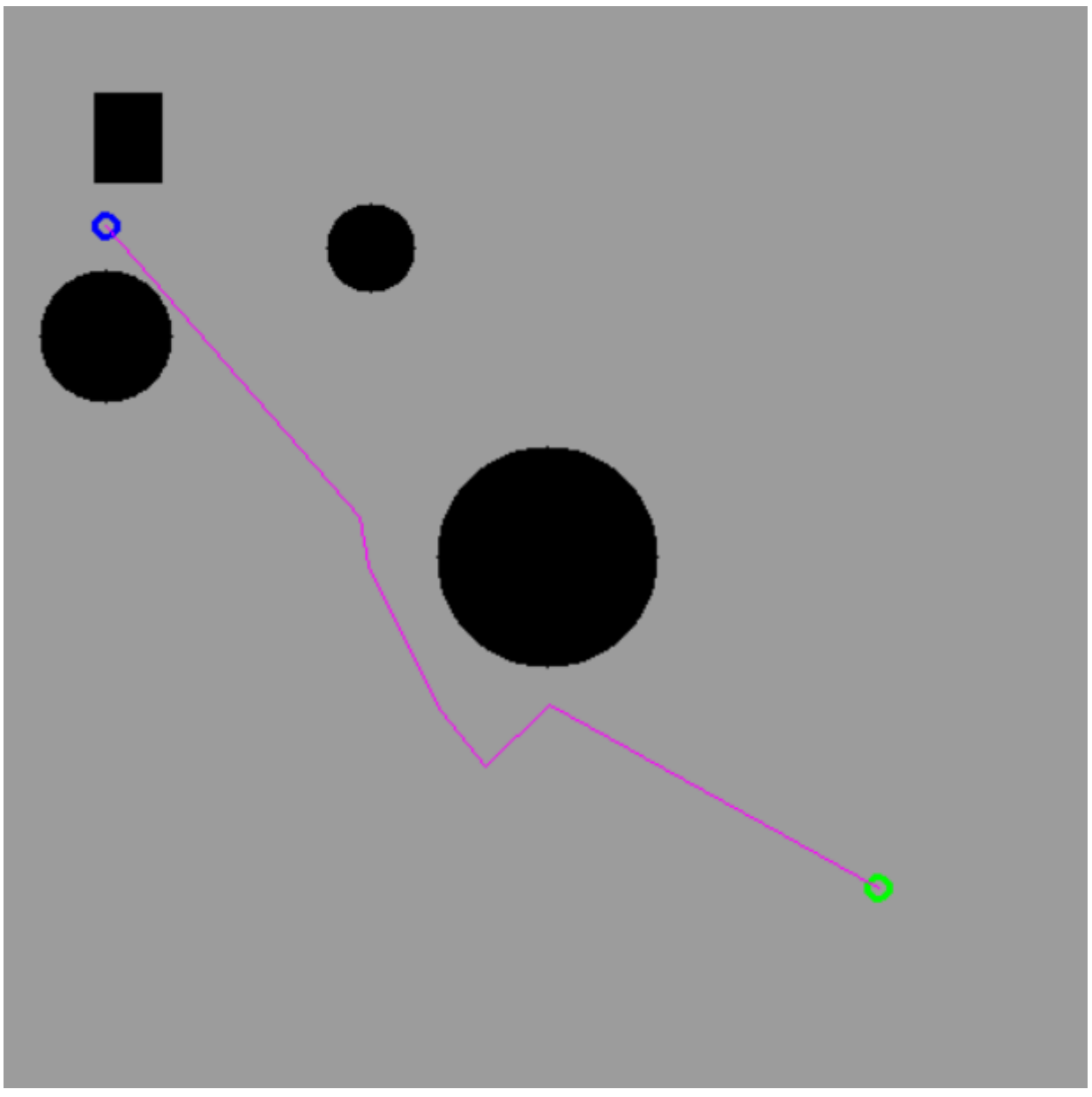
2.3 算法结果分析

迭代次数	种群数量	适应度	距离	路径	时间/s
200	50	0.25	674.16	=> (50, 100) => (200, 147) => (210, 164) => (352, 244) => (384, 191) => (306, 297) => (400, 400)	15.67
300	50	0.22	752.86	=> (50, 100) => (228, 315) => (173, 258) => (277, 331) => (321, 247) => (375, 335) => (400, 400)	16.77
400	50	0.32	539.68	=> (50, 100) => (184, 182) => (198, 344) => (262, 383) => (292, 370) => (316, 380) => (400, 400)	26.13
500	50	0.29	586.97	=> (50, 100) => (222, 192) => (169, 295) => (239, 309) => (319, 399) => (400, 397) => (400, 400)	54.46
600	50	0.33	513.12	=> (50, 100) => (165, 232) => (169, 254) => (202, 320) => (222, 345) => (251, 317) => (400, 400)	74.43
700	50	0.30	566.51	=> (50, 100) => (200, 163) => (221, 148) => (287, 134) => (306, 268) => (309, 259) => (400, 400)	67.64
800	50	0.29	592.29	=> (50, 100) => (289, 211) => (294, 179) => (280, 209) => (321, 212) => (311, 299) => (400, 400)	70.33
900	50	0.25	661.11	=> (50, 100) => (49, 114) => (141, 140) => (149, 357) => (158, 331) => (143, 377) => (400, 400)	66.01
1000	50	0.28	597.19	=> (50, 100) => (117, 117) => (249, 170) => (343, 244) => (277, 295) => (400, 351) => (400, 400)	70.73

下图是我所绘制的时间和距离随迭代次数增加的变化图，红色代表距离，蓝色代表时间。



下图为最优解，也就是迭代600次的结果。



上表为我所做的实验，可以看到，随着迭代次数的增加，并没有出现适应度减小的情况，而是先减小后增大。我认为这个原因是迭代次数增多后，交叉变异可能性也会变大，并不一定都是朝向好的方向发展的。所以迭代次数增加后，可能会导致整个种群朝着坏的方向进化。

此外，时间上也没有完全随着迭代次数的增长而增长。我觉得原因可能时计算机算力在不同时间段并没有完全相同所导致的。

3．差分进化算法

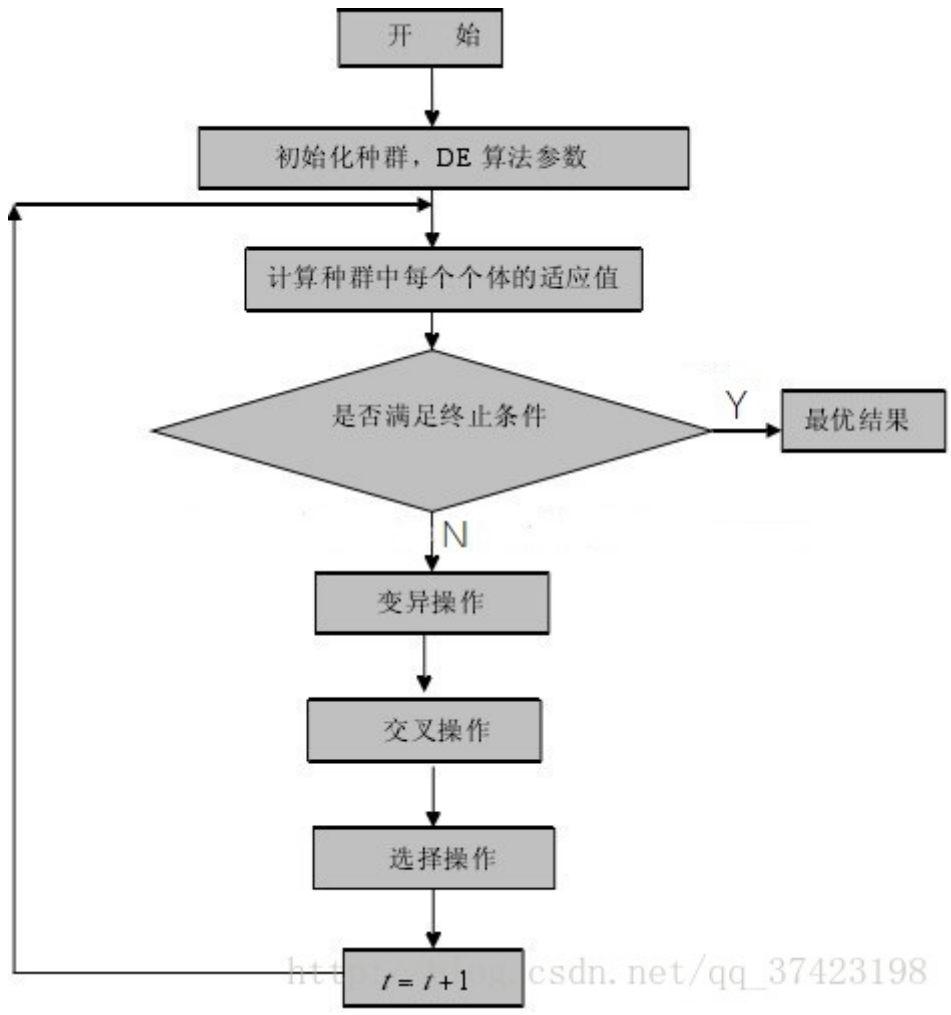
3.1 算法介绍

差分进化算法（differential evolution），又称微分进化算法，它起源于遗传退火算法，是一种求解最佳化问题的进化算法。它包含变异，交叉操作，淘汰机制。 本质上说，它是一种基于实数编码的具有保优思想的贪婪遗传算法。而差分进化算法与遗传算法不同之处，在于变异的部分是随机两个解成员变数的差异，经过伸缩后加入当前解成员的变数上，因此差分进化算法无须使用几率分布产生下一代解成员。

差分进化是一个非常简单但非常强大的随机全局优化器。由于其优异的性能，它在声学，空气动力学、航空航天、汽车及汽车工业和生物化学等科学和工程领域中得到了非常广泛的应用。

3.2 算法思路

差分算法和遗传算法类似，同样包括初始化、变异、交叉和选择四步,流程图如下所示：



- 初始化

初始化过程，也就是产生初代种群P^0的过程。实际上和遗传算法类似。在种群P^0中，包含N_p个个体，对于第i代第j个个体，可以将其记录为p_j^i。对于每个个体都表示N个优化参数x。通常在初始化时，会为每个个体设置两个上下边界值，其中b_j^U用于表示第j个个体的上边界，b_j^L用于表示第j个个体的下边界。

$$x_j^{0,i} = b_j^L + \alpha_j^i(b_j^U - b_j^L), \quad 1 \le j \le N,$$

对于本例来说，上下界即为空间环境的长宽，也就是0~500。

```
        cur_x = self.x_min + (self.x_max - self.x_min) * random.random()
        cur_y = self.y_min + (self.y_max - self.y_min) * random.random()
```

- 变异

差分进化算法的变异又被称之为差分变异，在当前种群中随机选择三个个体，将其中两个相减得到差分向量(Difference Vector)并叠加在第三个个体（基个体，Base Vector)上得到变异个体。

对于基个体的选择，一般有current，rand，best和better四种方式。current即是选择当前基个体，best则是选择种群中的最优个体，better则选择优于当前个体的个体，通常情况这是随机选择的。rand则是从种群中随机挑选个体。

而在这里则选用rand的方式进行种群中的个体挑选。

```
        for i in range(gen_x.shape[0]):
            r1 = random.randint(0, size_x)
            r2 = random.randint(0, size_x)
            r3 = random.randint(0, size_x)
            while r1 == i or r2 == i or r3 == i or r1 == r2 or r2 == r3 or r1 == r3:
                r1 = random.randint(0, size_x)
                r2 = random.randint(0, size_x)
                r3 = random.randint(0, size_x)
            gen_x_mutation[i, :] = gen_x[r1, :] + self.F * (gen_x[r2, :] - gen_x[r3, :])
            gen_y_mutation[i, :] = gen_y[r1, :] + self.F * (gen_y[r2, :] - gen_y[r3, :])
```

- 交叉

和遗传算法类似，差分进化算法同样具有交叉过程。这里我所选择的交叉方式为二项式交叉。二项交叉指的是针对每个分量产生一个0到1的随机小数，若该随机数小于交叉算子CR则进行交换。 交叉算子系数越大，交叉个体从变异个体获得的信息越大，全局搜索能力越强，收敛慢；反之，交叉算子越小，其局域搜索能力就越强，易早熟。

```
    def cross_over(self, gen_x, gen_y, gen_x_mutation, gen_y_mutation):
        gen_x_crossover = np.zeros(gen_x.shape)
        gen_y_crossover = np.zeros(gen_y.shape)
        size_x, size_y = gen_x.shape
        for i in range(size_x):
            for j in range(size_y):
                p = random.random()
                if p <= self.cr:
                    gen_x_crossover[i, j] = gen_x_mutation[i, j]
                    gen_y_crossover[i, j] = gen_y_mutation[i, j]
                else:
                    gen_x_crossover[i, j] = gen_x[i, j]
                    gen_y_crossover[i, j] = gen_y[i, j]
```

- 选择

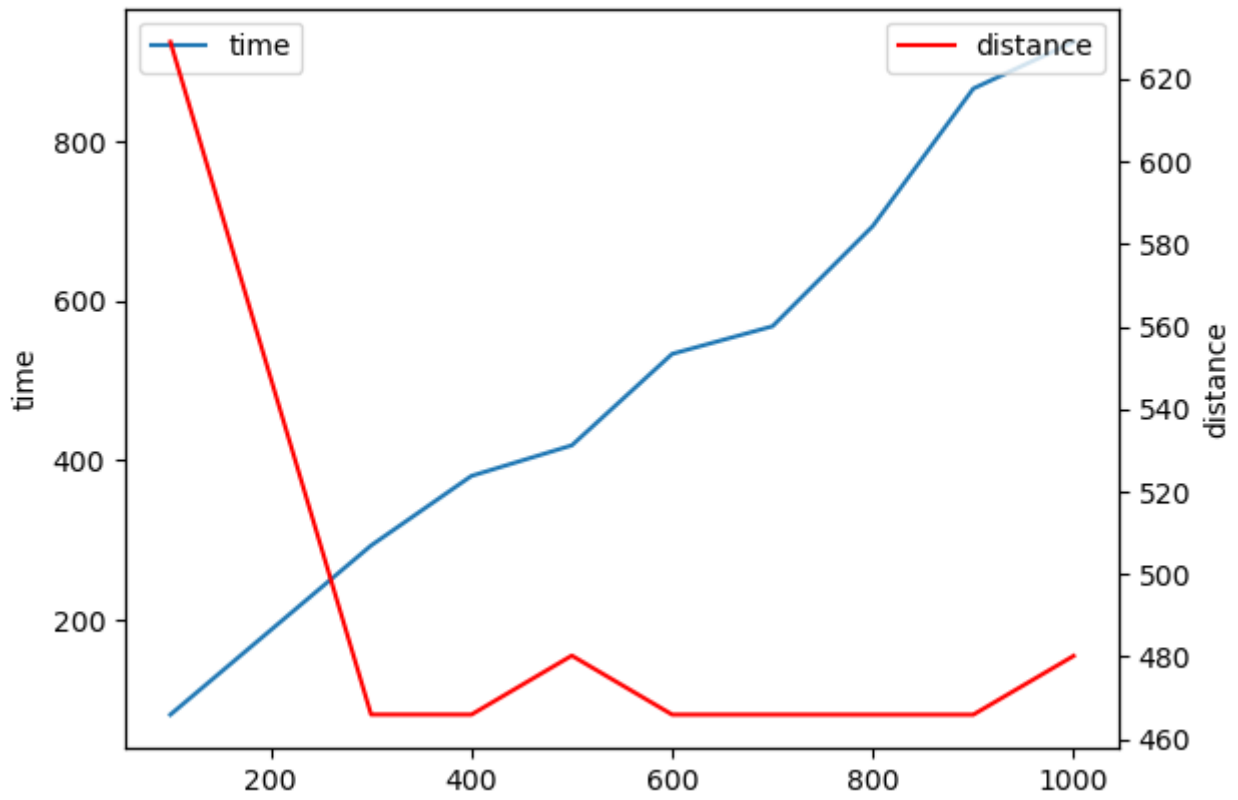
选择，也被称为母子竞争。顾名思义，实际上是母亲和孩子互相竞争存活至下一代。

```
    def select(self, gen_x, gen_y, gen_x_crossover, gen_y_crossover):
        crossover_fitness = self.CalculateFitness(gen_x_crossover, gen_y_crossover)
        fitness = self.CalculateFitness(gen_x, gen_y)
        new_gen_x = np.zeros(gen_x_crossover)
        new_gen_y = np.zeros(gen_y_crossover)
        for i in range(self.population_size):
            if fitness[i] > crossover_fitness[i]:
                new_gen_x[i] = gen_x[i]
                new_gen_y[i] = gen_y[i]
            else:
                new_gen_x[i] = gen_x_crossover[i]
                new_gen_y[i] = gen_y_crossover[i]
        return new_gen_x, new_gen_y
```

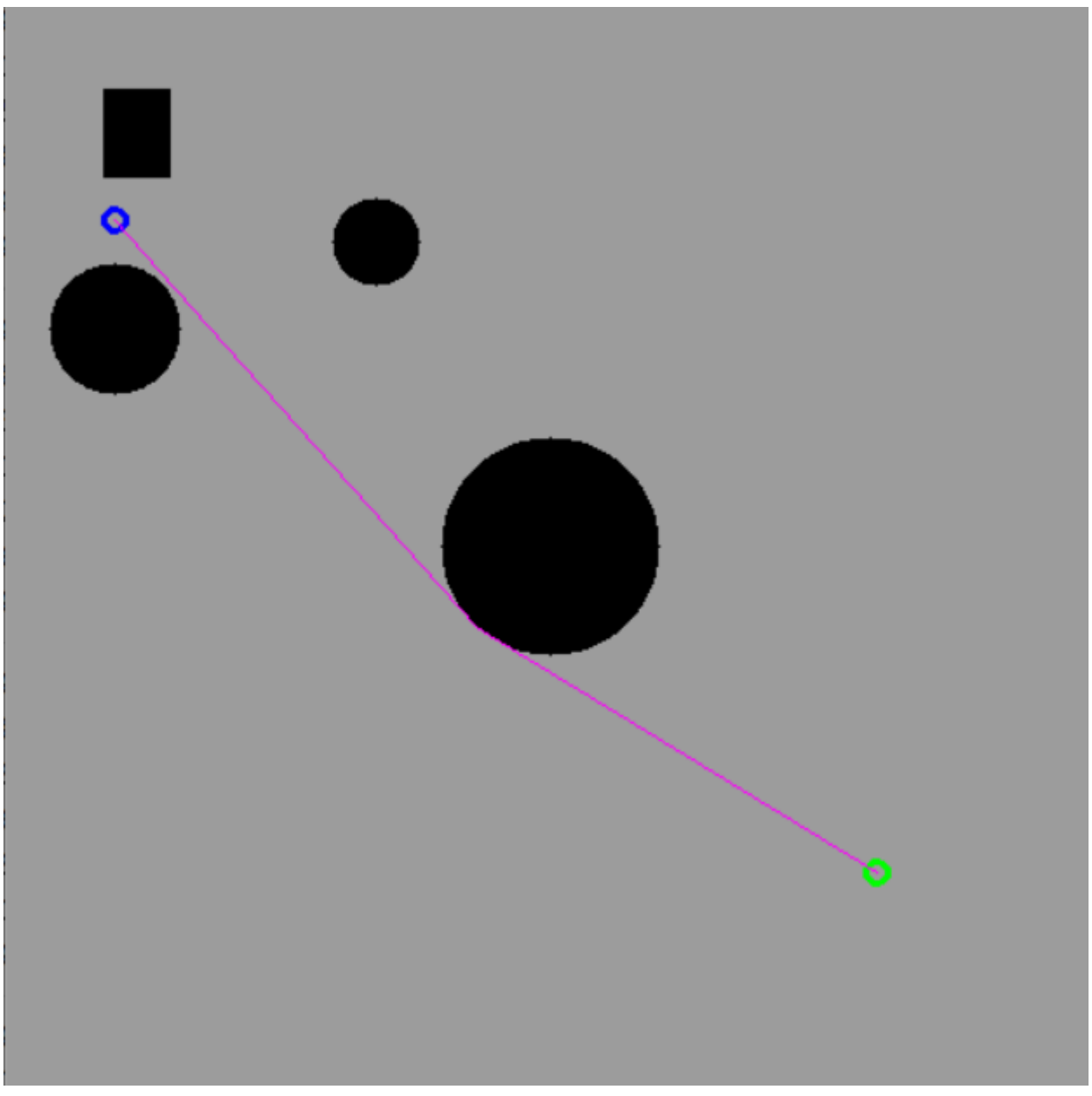
3.3 算法结果分析

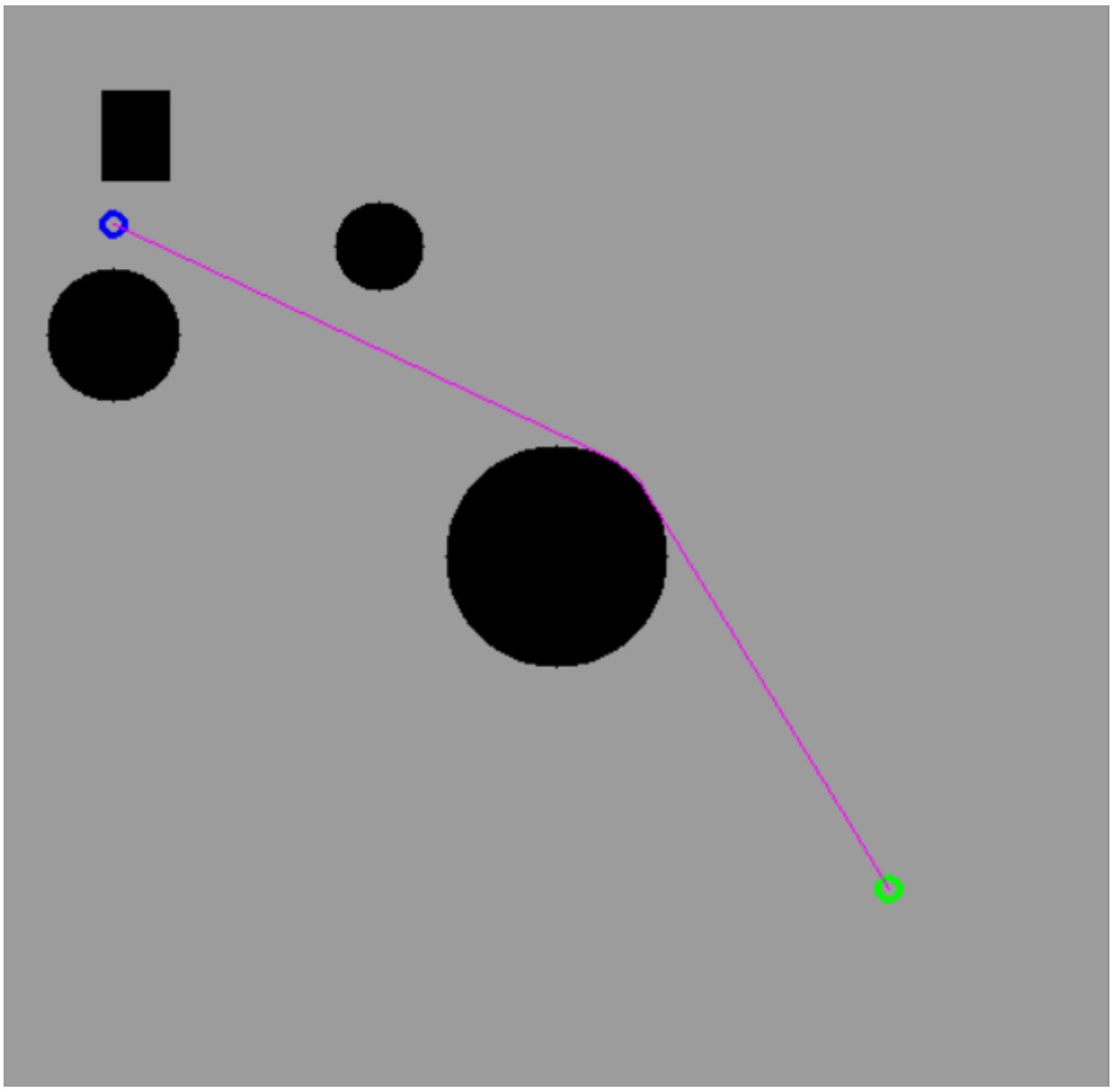
迭代次数	种群数量	适应度	距离	路径	时间/s
100	50	0.26	628.92	=> (50, 100) => (200, 147) => (210, 164) => (352, 244) => (384, 191) => (306, 297) => (400, 400)	81.09
300	50	0.37	466.01	=> (50, 100) => (78, 132) => (93, 149) => (216, 287) => (222, 291) => (263, 316) => (400, 400)	293.31
400	50	0.38	466.00	=> (50, 100) => (146, 208) => (212, 282) => (215, 286) => (221, 291) => (320, 351) => (400, 400)	380.59
500	50	0.36	480.28	=> (50, 100) => (271, 204) => (278, 208) => (287, 215) => (291, 222) => (382, 370) => (400, 400)	418.94
600	50	0.38	465.99	=> (50, 100) => (146, 209) => (214, 285) => (219, 289) => (222, 291) => (396, 398) => (400, 400)	533.68
700	50	0.38	465.99	=> (50, 100) => (86, 140) => (132, 193) => (215, 286) => (220, 290) => (326, 355) => (400, 400)	568.34
800	50	0.38	465.98	=> (50, 100) => (117, 176) => (157, 221) => (214, 285) => (218, 288) => (221, 291) => (400, 400)	694.19
900	50	0.38	465.98	=> (50, 100) => (50, 100) => (214, 285) => (217, 288) => (221, 291) => (296, 336) => (400, 400)	866.49
1000	50	0.37	480.21	=> (50, 100) => (274, 206) => (280, 210) => (285, 214) => (290, 220) => (358, 332) => (400, 400)	925.07

下图是我所绘制的时间和距离随迭代次数增加的变化图，红色代表距离，蓝色代表时间。可以看出，随着迭代次数增加，时间显著增长，而距离则是先下降再趋于平缓。



在DE算法中，可以发现，其寻找出的路径是非常贴合障碍物的，效果很好。





对于DE算法，可以发现时间上有了明显的增加，我认为这是因为这个算法中大量使用到了随机数的产生，因而导致算法时间大幅度加长。但是可以发现，DE算法尽管在时间上有所损耗，但是效果有了显著提高。从结果的可视化来开，DE算法能够很聪明的找到一条贴合障碍物的路线。

4. 粒子群算法

4.1 算法介绍

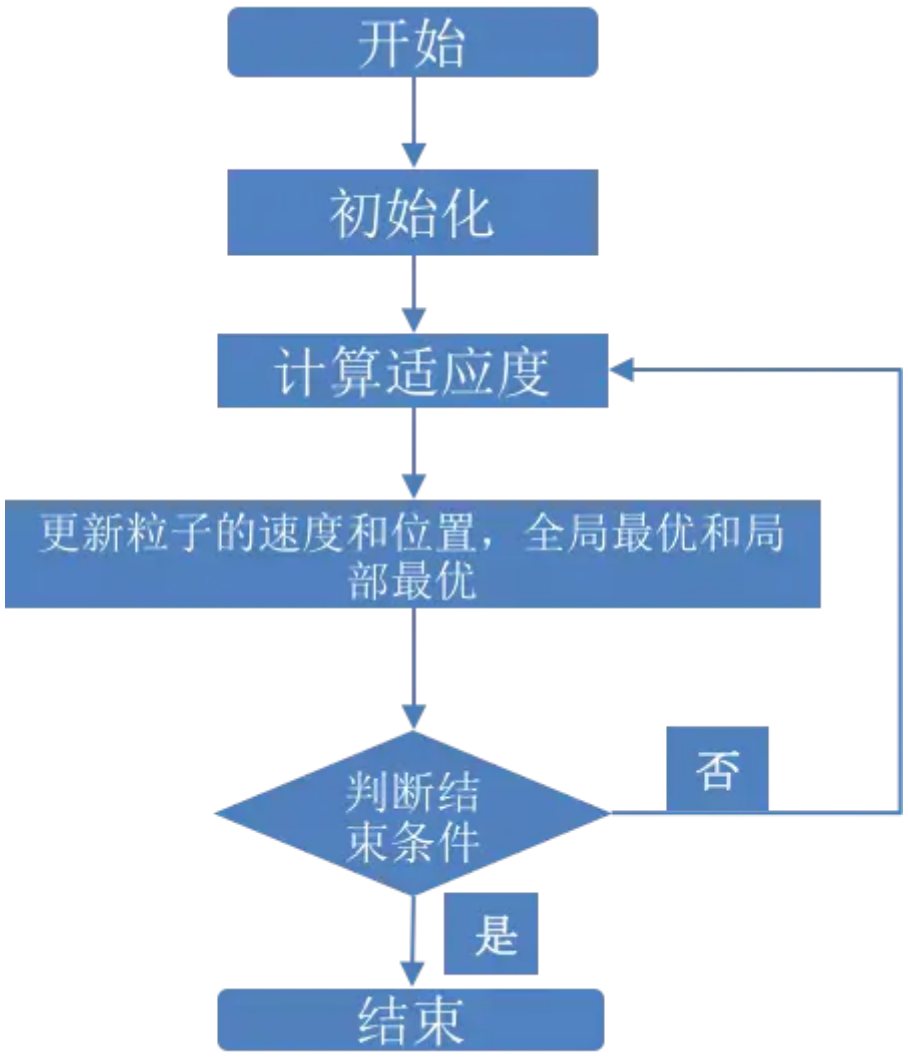
粒子群算法是计算数学中用于解决最优化的搜索算法，也是最为经典的智能算法之一。应用主要是在工程和计算机科学还有行为管理研究科学里面。

粒子群算法(Particle Swarm Optimization, PSO)是1995年Eberhart博士和Kennedy博士一起提出的。粒子群算法是通过模拟鸟群捕食行为设计的一种群智能算法。区域内有大大小小不同的食物源，鸟群的任务是找到最大的食物源（全局最优解），鸟群的任务是找到这个食物源。鸟群在整个搜寻的过程中，通过相互传递各自位置的信息，让其他的鸟知道食物源的位置最终，整个鸟群都能聚集在食物源周围，即我们所说的找到了最优解，问题收敛。

粒子群算法是一个简洁且强大的优化算法。它可以嵌套非凸模型、逻辑模型、复杂模拟模型、黑箱模型、甚至实际实验来进行优化计算。虽然粒子群算法在诸多领域均有不错表现，如电力系统，生物信息，物流规划等，但对于一些特定问题的求解，粒子群并不能保证解的质量。因此，粒子群算法的发展仍在继续，其研究仍有许多未知领域，如粒子群理论的数学验证等等。

4.2 算法思路

粒子群算法的目标是使所有粒子在多维超体中找到最优解，其流程如下图所示：



- 初始化

首先给空间中的所有粒子分配初始随机位置和初始随机速度。

在本例中，我们将每一条路径视为为一个粒子。粒子的个数为n，也即是有n条路径，同时，每个粒子又有m个染色体，也就是中间过渡点的个数。同时，还需要对x方向和y方向的速度进行初始化。

```
    def init_generation(self):
        # init population
        gen_x = np.zeros((self.population_size, self.dim + 2))
        gen_y = np.zeros((self.population_size, self.dim + 2))
        for i in range(self.population_size):
            for j in range(self.dim + 2):
                cur_x, cur_y = self.generate_random_point()
                while self.pointCollison(cur_x, cur_y):
                    cur_x, cur_y = self.generate_random_point()
                gen_x[i, j] = cur_x
                gen_y[i, j] = cur_y
        gen_x[:, 0] = self.start_x
        gen_x[:, -1] = self.end_x
        gen_y[:, 0] = self.start_y
        gen_y[:, -1] = self.end_y
        return gen_x, gen_y

    def init_velocity(self):
        vx = self.vmax * np.random.rand(self.population_size, self.dim)
        vy = self.vmax * np.random.rand(self.population_size, self.dim)
        return vx, vy
```

- 适应度计算

该例中，我们定义的适应度函数为：

$$fitness\_function(x) = \begin{cases} 1/distance，未与障碍物相撞 \\ 0，与障碍物相撞 \end{cases}$$

在计算完适应度后，保留个体最优和全局最优。

```
    def fit_cmp(self, gen_x, gen_y, fitness_values, best_fit, best_pos_x, best_pos_y):
        best_index = np.argmax(fitness_values)
        best_fitness = fitness_values[best_index]
        best_x = gen_x[best_index, :]
        best_y = gen_y[best_index, :]
        for i in range(gen_x.shape[0]):
            if fitness_values[i] > best_fit[i]:
                best_pos_x[i, :] = gen_x[i, :]
                best_pos_y[i, :] = gen_y[i, :]
                best_fit[i] = fitness_values[i]
        return best_fit, best_pos_x, best_pos_y, best_fitness, best_x, best_y
```

- 速度和位置更新

根据每个粒子的速度、问题空间中已知的最优全局位置和粒子已知的最优位置依次推进每个粒子的位置。随着计算的推移，通过探索和利用搜索空间中已知的有利位置，粒子围绕一个或多个最优点聚集或聚合。

每一个粒子是由一组粒子在搜索空间中运动，受其自身的最佳过去位置pbest和整个群或近邻的最佳过去位置gbest的影响。每次迭代粒子i的第d维速度更新公式为：

$$v_{td}^{k}=wv_{td}^{k-1}+c_1r_1\left(\text{pbest }t_{td}-x_{id}^{k-1}\right)+c_2r_2\left(\text{gbest }t_d-x_{td}^{k-1}\right)$$

粒子i第d维的位置更新公式为：

$$x_{id}^{k}=x_{id}^{k-1}+v_{id}^{k-1}$$

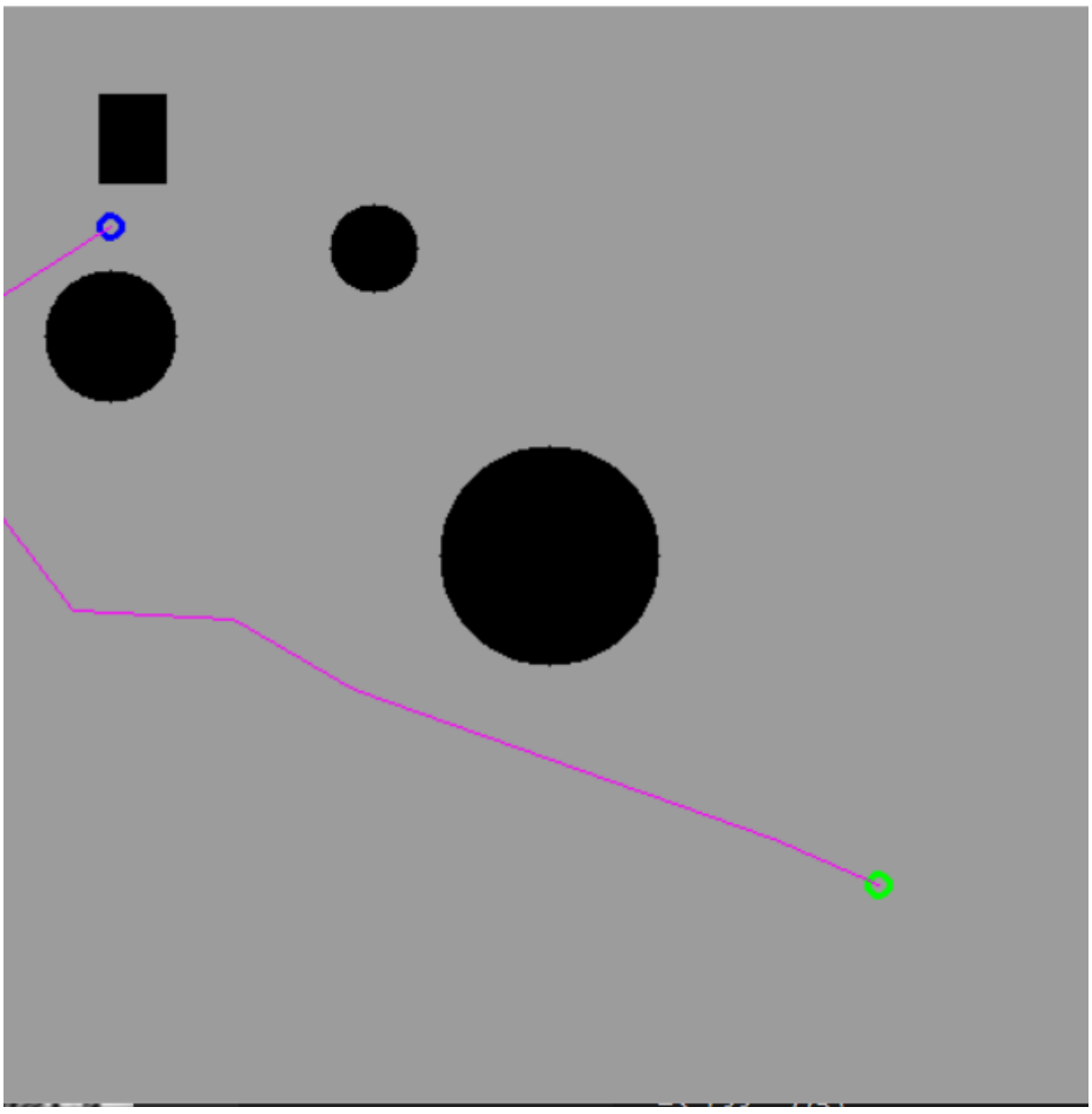
其中，v_{id}^{k}是第k次迭代粒子i飞行速度矢量的第d维分量，x_{id}^{k}是第k次迭代粒子i位置矢量的第d维分量。

```
    def update_velocity(self, vx, vy, gen_x, gen_y, best_pos_x, best_pos_y, best_x, best_y):
        new_vx = self.w * vx + (self.c1 * random.random() * (best_pos_x - gen_x) + self.c2 *
random.random() * (best_x - gen_x))[:, 1:-1]
        new_vy = self.w * vy + (self.c1 * random.random() * (best_pos_y - gen_y) + self.c2 *
random.random() * (best_y - gen_y))[:, 1:-1]
        return new_vx, new_vy


    def update_pos(self, gen_x, gen_y, vx, vy):
        gen_x[:, 1:-1] += vx
        gen_y[:, 1:-1] += vy
        return gen_x, gen_y
```

4.3 算法结果分析

迭代次数	种群数量	适应度	距离	路径	时间/s
200	50	0.00（碰撞了障碍物）	634.68	=> (50, 100) => (-21, 171) => (-36, 167) => (262, 332) => (271, 376) => (360, 400) => (400, 400)	42.40
300	50	0.00（碰撞了障碍物）	634.68	=> (50, 100) => (-21, 171) => (-36, 167) => (262, 332) => (271, 376) => (360, 400) => (400, 400)	42.40
400	10	0.00（碰撞了障碍物）	509.79	=> (50, 100) => (32, 131) => (36, 165) => (101, 240) => (138, 270) => (282, 331) => (400, 400)	32.51
500	10	0.00（碰撞了障碍物）	915.56	=> (50, 100) => (34, 125) => (99, 169) => (39, 105) => (-71, 305) => (103, 293) => (400, 400)	32.65
600	10	0.00（碰撞了障碍物）	1408.26	=> (50, 100) => (-174, -2) => (155, 222) => (193, 401) => (426, 233) => (531, 333) => (400, 400)	32.44
700	10	0.00（碰撞了障碍物）	828.94	=> (50, 100) => (234, 136) => (202, 243) => (203, 377) => (193, 331) => (187, 459) => (400, 400)	35.00
800	10	0.00（碰撞了障碍物）	2458.26	=> (50, 100) => (-196, -652) => (-123, -664) => (-44, 66) => (-17, 425) => (139, 284) => (400, 400)	32.34
900	10	0.00（碰撞了障碍物）	1505.84	=> (50, 100) => (138, -72) => (311, 282) => (374, 549) => (386, 316) => (304, 174) => (400, 400)	35.82
1000	10	0.002	653.29	=> (50, 100) => (-52, 165) => (33, 275) => (106, 279) => (161, 311) => (352, 379) => (400, 400)	32.37

在粒子群算法中，可以发现，其寻找出的路径会越过边界。虽然粒子群算法能够找到一条可行路径，但是这条路径并不能使人太过满意。

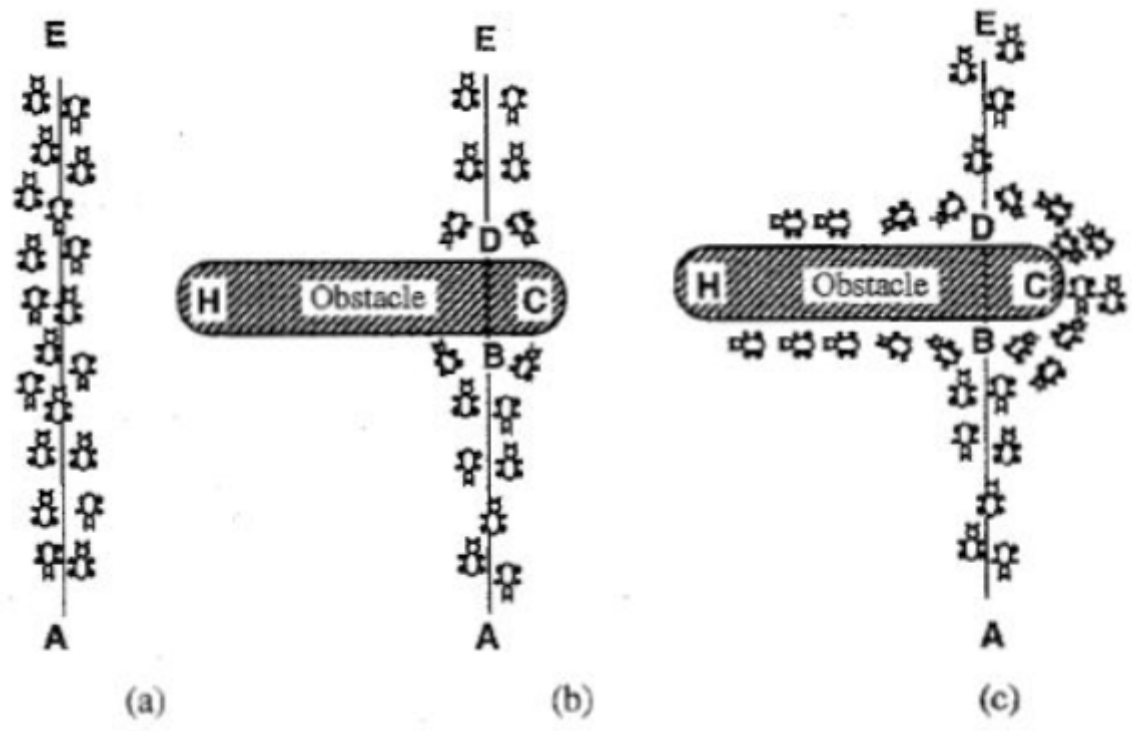


5. 蚁群算法

5.1 算法介绍

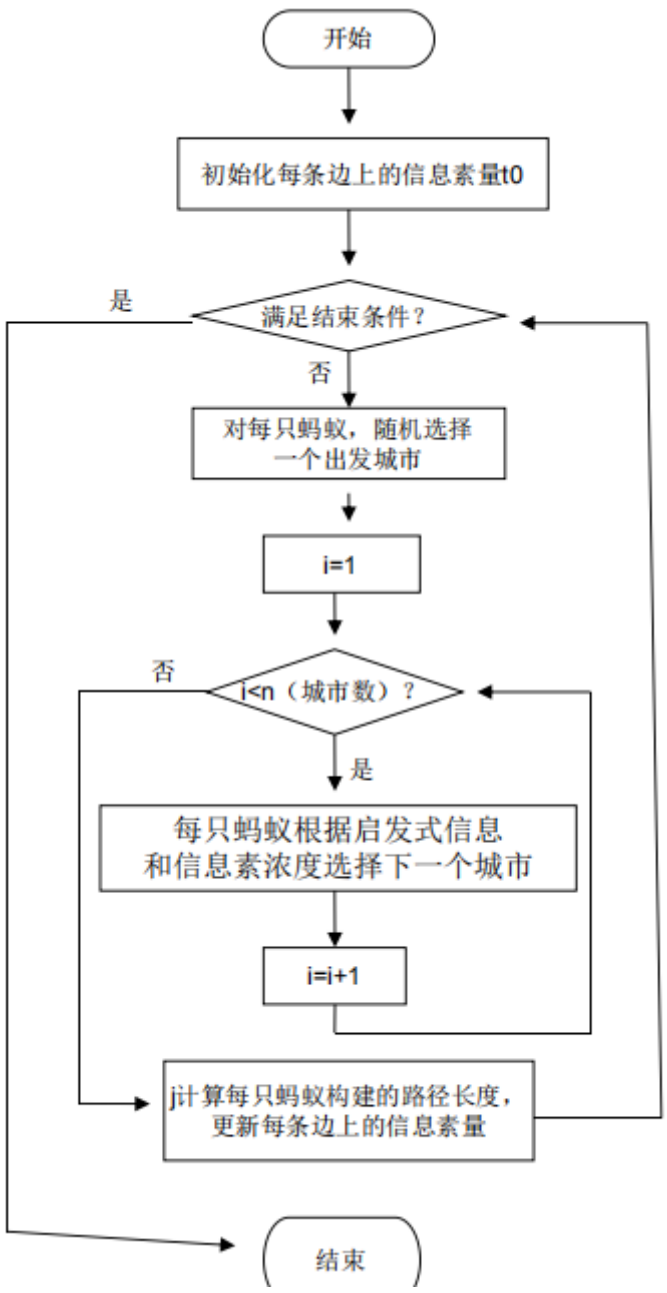
蚁群算法（Ant Clony Optimization， ACO）是一种群智能算法，它是由一群无智能或有轻微智能的个体（Agent）通过相互协作而表现出智能行为，从而为求解复杂问题提供了一个新的可能性。蚁群算法最早是由意大利学者Colorni A., Dorigo M. 等于1991年提出。经过20多年的发展，蚁群算法在理论以及应用研究上已经得到巨大的进步。

蚁群算法是一种仿生学算法，是由自然界中蚂蚁觅食的行为而启发的。在自然界中，蚂蚁觅食过程中，蚁群总能够按照寻找到一条从蚁巢和食物源的最优路径，如下图所示：



5.2 算法思路

算法流程图如下：



机器人从起始位置开始，根据当前位置的沿着八个方向（0，45° ，90°，135°，180°，225°，270° ，315°）的信息素和启发因素，利用轮盘赌的方式选择一个方向，移动到下一个位置。在编程的过程中，定义的精度为50，也就是每次移动的距离为50。

对于空间环境，将其划分为m*n个点，则可以存为矩阵，定启发因素矩阵维度为(m， n,8)，定义信息素矩阵维度为(m,n,8)，分别代表各个点的八个方向的启发因素。

对于每只蚂蚁来说，其主要有三个步骤：

- 找路径。

对于每只蚂蚁，都会根据信息素、启发因子、以及障碍物进行方向的选择。

$$p^{k}(p,\theta_i)=\begin{cases}\dfrac{\tau(p,\theta_i)^{a}\eta(\theta_i)^{b}}{\sum_{i=1}^{8}\tau(p,\theta_i)^{a}\eta(\theta_i)^{b}} & \text{如果pn} \notin obs \\ 0 & \text{其他}\end{cases}$$

- 信息素增量更新

信息素增量更新只在最优路径上对信息素进行更新，更新公式如下：

$$\tau_{ij}=(1-p)\ \tau_{ij}+\Delta\tau_{ij}^{best}$$

$$\Delta\tau^{best}_{ij}=\begin{cases}\dfrac{1}{L^{best}} & \text{如果}e^{ij}\text{包含在最优路径中} \\ 0 & \text{其他}\end{cases}$$

在实际算法中，可能会多次到达同一个点，因此，需要对重复点之间的路径进行裁剪，在编程的过程中，我本来选用的是在选取下一个点的时候，判断是否该点之前走过，如果走过，则重新选点，否则该点可以走。

但是，在实际运行时发现，这样会大幅度增加算法的运行成本，导致算法运行不出来结果。因此最终还是舍弃了这个方案。

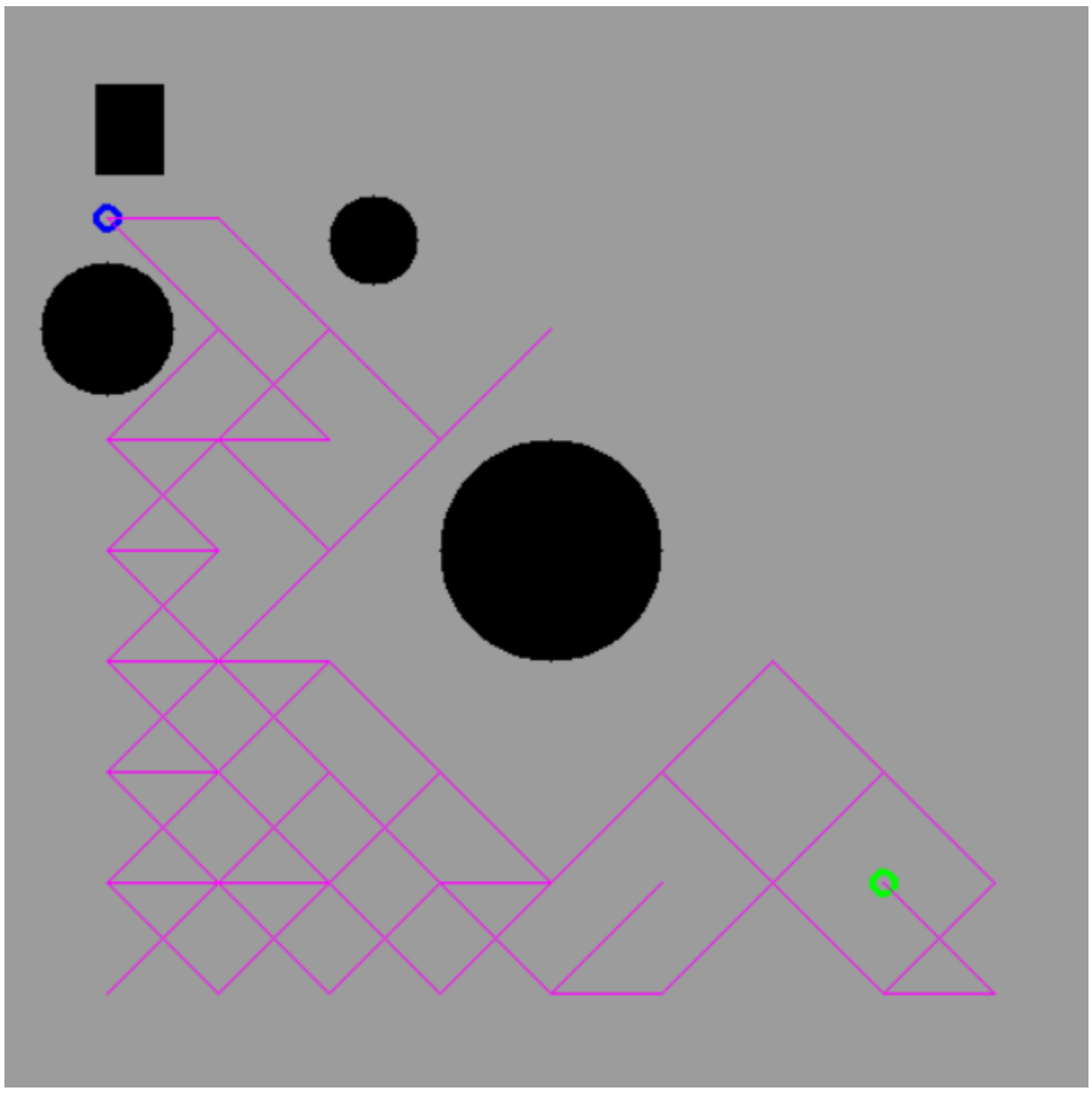
```
    def calculate(self):
```

```
        px = np.zeros((self.m, 1))
        py = np.zeros((self.m, 1))
        px[:, 0] = self.start_x
        py[:, 0] = self.start_y
        theta = np.zeros((self.m, 1))
        while True:
            gen_x = (px[:, -1] == self.end_x * np.ones(1))
            gen_y = (py[:, -1] == self.end_y * np.ones(1))

            if np.sum(gen_x) + np.sum(gen_y) == 2 * self.m:
                break
            theta = np.hstack((theta, np.zeros((self.m, 1))))
            px = np.hstack((px, np.zeros((self.m, 1))))
            py = np.hstack((py, np.zeros((self.m, 1))))



            for i in range(self.m):
                while True:
                    if px[i, -2] == self.end_x and py[i, -2] == self.end_y:
                        px[i, -1] = px[i, -2]
                        py[i, -1] = py[i, -2]
                        theta[i, -1] = theta[i, -2]
                        break

                lasttx = int(px[i, -2] / self.precision)
                lastty = int(py[i, -2] / self.precision)

                ttij = self.tij[lasttx, lastty, :].reshape((1, 8))
                nnij = self.nij[lasttx, lastty, :].reshape((1, 8))



                pij = ttij * nnij / np.sum(ttij * nnij)
                next_theta = int(self.select(pij))



                ppx = px[i, -2] + self.precision * self.IFunction(np.cos(self.sitaran[next_theta]))
                ppy = py[i, -2] + self.precision * self.IFunction(np.sin(self.sitaran[next_theta]))

                # for j in range(px[i, :].shape[0]):
                #     if px[i, j] == ppx and py[i, j] == ppy:
                #         flag1 = False
                #         break

                # ab = np.vstack((px[i, :], py[i, :]))
                # abc = np.vstack((ppx, ppy))

                # for mt in range(ab.shape[1]):
                #     flag = np.sum(ab[:, mt] - abc)
                #     if flag == 0:
                #         print(i)
                #         break
                flag = self.lineCollision((px[i, -2], py[i, -2]), (ppx, ppy))



                if ppx > self.x_min and ppy > self.y_min and ppx < self.x_max and ppy < self.y_max
and not flag:
                    tx = int(ppx / self.precision)
                    ty = int(ppy / self.precision)
                    if self.obs[tx, ty] == 0:
                        px[i, -1] = ppx
                        py[i, -1] = ppy
```

```
                    theta[i, -1] = next_theta
                    break


        return px, py, theta
```

5.3 算法结果分析

迭代次数	蚁群数量	精度	距离	路径	时间/s
10	100	50	28504.37	\	168.90
20	50	50	76196.46	\	145.98
20	10	50	85681.74	\	30.93
20	20	50	21016.14	\	56.40
10	50	50	42166.60	\	81.44
10	10	50	34647.01	\	21.24
10	20	50	39718.08	\	41.05
10	30	50	59405.80	\	421.76



这个结果的出现，原因我认为是这样的：

- 精度选择。我所选择的精度是50，相对于路径来说尺寸非常大，所以导致蚁群算法难以找到一个较优解。但是一旦把精度降低，就会使得整个算法运行迭代时间过长，难以收敛。
- 选点具有重复性。每次选择点可能会导致点选择具有重复性，这样就会导致多余的路径。因此，就导致了蚂蚁可能会来回往复的走。
- 运行时间过长蚁群基数不够大。由于蚁群算法时间复杂度较高，蚁群数量过大时同样会导致算法运行迭代时间过长，难以收敛。

6．算法对比

我对四种算法进行了对比，并列出表格如下：

算法名称	算法优势	算法劣势
遗传算法	1．时间优势 遗传算法是算法中运行速度很快的一个算法，同条件下是DE算法的1/5左右。它能够在较短时间内寻找一个较优的解，收敛速度较快。 2．实现简单 遗传算法的实现相对来说很简单，并没有复杂的操作。	1.效果没有十分理想。 与DE算法相比，遗传算法是的效果显然不如DE算法。 2.遗传算法不一定随着迭代次数增加而提高效果。 在这次实验中，可以看到迭代次数的增加并没有能够很好的提高遗传算法的效果。
差分进化算法	1.效果优势 DE算法是一个效果优秀的算法，从可视化的结果来看，可以看出DE算法可以找到一个贴合障碍物的路径。 2.实现简单 DE算法的实现来说也很简单，同样没有复杂的操作。 3.效果稳定 随着迭代次数的增加，差分进化算法保持了效果的稳定性。	1.时间劣势 差分进化算法的计算时间非常慢，是遗传算法的五倍左右。
粒子群算法	1.效果发散 可以看到，粒子群算法的路径非常发散。也就是说，在障碍物很多的情况下，粒子群算法或许能够找到一个较优的可行解。 2.时间优势 从时间上来说，粒子群算法收敛速度相对较快	1.效果劣势 可以看到，粒子群算法会寻找到超出边界的点，容易陷入局部极值。
蚁群算法	1.实现简单 蚁群算法实现简单，工程实现较为容易。	1.效果劣势 从结果来看，蚁群算法是四种算法中收敛效果最弱的一个。 2.时间劣势 蚁群算法运行时间长，将蚁群数量提高就导致整个算法运行非常缓慢。 3.容易陷入死胡同，导致程序一直在运行而无法停下。

7．如何运行

7.1 环境

- 系统：Windows10
- python: 3.8.5
- 所需要安装的包： numpy: 1.19.2 matplotlib 3.3.2 opencv-python 4.5.1.48

7.2 运行

- 运行main.py可以直接在命令行中输入`python main.py`。
- 为了更加可以方便可视化，可以在命令行中输入`python visualization.py`。此时可以观察每一代的寻路过程。