School of Data and Computer Science, Sun Yat-sen University

Homework4

RRT&Bi-RRT Implementation (Python)

18340013 陈琮昊

Dec. 8, 2020

Contents

1 实验任务

实现 `RRT` 算法和 `Bi-RRT`(双向 `RRT`) 算法, 在具有障碍物的地图内使用上述两种算法规划机器人的避障路径. 并进行性能对比与分析.

2 算法介绍

2.1 RRT 算法

`RRT`(快速扩展随机树) 算法是一种多维空间中有效率的规划方法. 它以一个初始点作为根节点, 通过随机采样增加叶子节点的方式, 生成一个随机扩展树, 当随机树中的叶子节点包含了目标点或进入了目标区域, 便可以在随机树中找到一条由从初始点到目标点的路径.

`RRT` 的一个弱点是在遇到有狭窄通道的环境时, 算法的收敛速度较慢. 因为狭窄通道面积小, 被碰到的概率低. 比如下图展示的例子这样, 要使用 `RRT` 通过一个很狭窄的通道, 有时 `RRT` 很快就找到了出路, 有时则一直出不去：

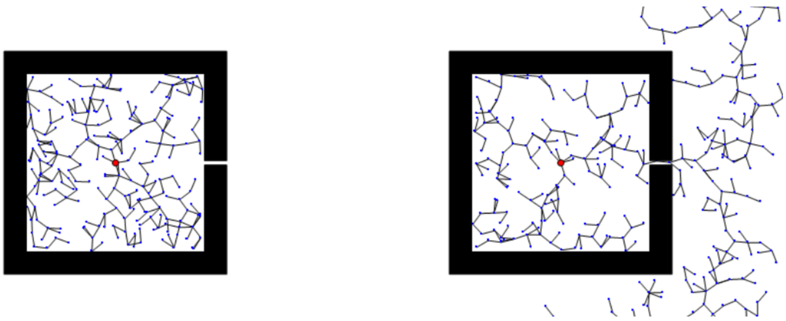

Figure 1. An example

2.2 Bi-RRT 算法

顾名思义,`Bi-RRT` 算法就是双向的 `RRT`. 分别从起始位置和目标位置扩展树, 直到两棵树的某个节点相遇为止. 很明显, 双向扩展的方式要比单向随机扩展的方式要好, 因为这样使得树的扩展方向更加明确, 因此性能上也会有所提高.

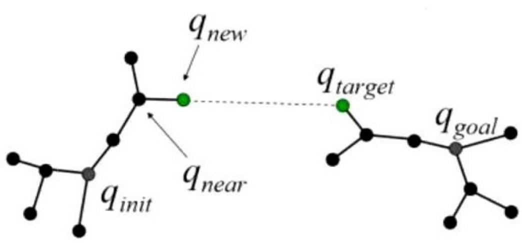

Figure 2. Bi-RRT

很明显, 由于其随机性, 找到的路径通常不是最优的; 但是不管什么机器人类型, 不管自由度是多少, 不管约束有多复杂都能使用该算法, 可以说普适性很强.

3 准备工作

在网上找了三张图, 这三张图中黑色的部分是机器人可以行走的地方, 白色的部分是不可行走的地方. 为了方便, 文件命名我直接命名为其难度. 很明显可以看到,`easy.png` 最容易, 因为黑色区域较开阔;`hard.png` 最难, 因为它中间只有很窄的一条黑色区域可以穿过.

Figure 3. easy.png

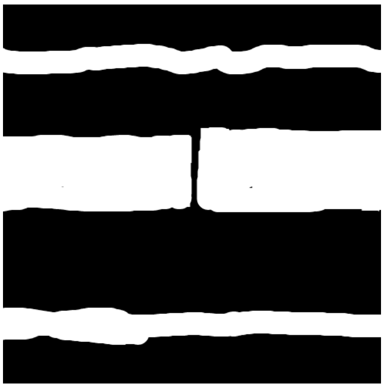

Figure 4. medium.png

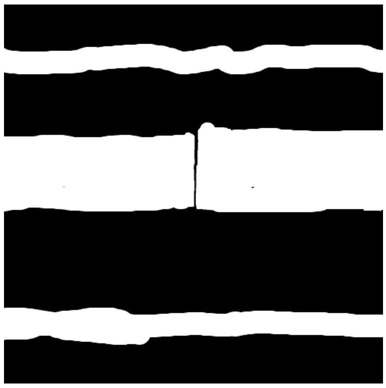

Figure 5. hard.png

至于显示图形化界面, 则是调用了 `Python` 的 `pygame` 库, 这是我之前无聊时发现的一个库, 用它写过一个很小的游戏. 只需 `import pygame` 即可.

4 代码实现

有 5 个.py 文件, 看似很多但每个文件思路很清晰, 代码量也不是很多:`maths.py` 是有关于数学上的一些公式 (函数) 的实现;`rrt_birrt.py` 则是实现两个类，分别代表 `RRT` 算法和 `Bi-RRT` 算法;`img_env.py` 则是图片的相关处理;`rrt_example.py` 则是用 `RRT` 算法寻找路径;`birrt_example.py` 同理. 代码部分占据较多空间, 实验结果可直接跳至倒数第二页.

maths.py

```
# -*- coding: UTF-8 -*-
#定义一些数学上的函数运算
import numpy as np
# 弧度转角度
def radtodeg(rad):
    return rad*(180.0/np.pi)
# 角度转弧度
def degtorad(deg):
    return deg*(np.pi/180.0)
# 比较两个向量
def combinevector(a, op, b):
    if len(a) != len(b):
```

```python
            raise ValueError("维度不匹配!")
        if op == '+':
            acc = [i+j for i, j in zip(a, b)]
        elif op == '-':
            acc = [i-j for i, j in zip(a, b)]
        elif op == '*':
            acc = [i*j for i, j in zip(a, b)]
        elif op == '/':
            acc = [i/float(j) for i, j in zip(a, b)]
        else:
            raise ValueError("不支持该运算!")
        if a.__class__ == tuple:
            return tuple(acc)
        else:
            return acc
# 有关向量的运算
def vectoroperator(a, op, b=1, param=1):
    if op == '+':
        acc = [i+b for i in a]
    elif op == '-':
        acc = [i-b for i in a]
    elif op == '*':
        acc = [i*b for i in a]
    elif op == '/':
        acc = [i/float(b) for i in a]
    elif op == 'int':
        acc = [int(i) for i in a]
    elif op == 'round':
        acc = [np.around(i, param) for i in a]
    else:
        raise ValueError("不支持的运算类型!")
    if a.__class__ == tuple:
        return tuple(acc)
    else:
        return acc
# 欧氏距离
def vectoreuclidean(a, b):
    if len(a) != len(b):
        raise ValueError("维度不匹配!")
    e = 0
    for i in range(len(a)):
        e += pow(a[i] - b[i], 2)
    return np.sqrt(e)
```

rrt_birrt.py

```python
# -*- coding: UTF-8 -*-
# rrt和B-rrt算法实现文件
import random
import numpy as np
class Node:
    def __init__(self, state):
        self.state = state
        self.parent = None
        self.cost = 0
def distance(s1, s2, metric='euclidean'):
    if len(s1) != len(s2):
        raise ValueError("维度不一致!")
    else:
        if metric == 'euclidean':
            acc = 0
            for i in range(len(s1)):
                acc += pow(s1[i] - s2[i], 2)
            return np.sqrt(acc)
```

```python
        elif metric == 'manhattan':
            acc = 0
            for i in range(len(s1)):
                acc += abs(s1[i] - s2[i])
            return acc
        else:
            raise ValueError("不支持的距离类型!")
def randomSampler(a, b):
    return random.uniform(a, b)
# 规划器
class MotionPlanner:
    def __init__(self, env):
        self.env = None
        self.start = None
        self.goal = None
        if self.is_env_valid(env):
            self.env = env
        else:
            raise ValueError("环境有问题!")
    def is_env_valid(self, env):
        if len(env.state_limits) != 2:
            return False
        else:
            for lim in env.state_limits:
                if len(lim) != env.state_dim:
                    return False
            return True
    def set_start(self, start):
        if len(start) == self.env.state_dim:
            self.start = start
        else:
            raise ValueError("起点有问题!")
    def set_goal(self, goal):
        if len(goal) == self.env.state_dim:
            self.goal = goal
        else:
            raise ValueError("终点有问题!")
    def plan(self, start, goal):
        self.set_start(start)
        self.set_goal(goal)
# RRT实现
class RRT(MotionPlanner):
    def __init__(self, env, step_size, growth='connect'):
        MotionPlanner.__init__(self, env)
        self.step_size = step_size
        self.growth = growth
        self.tree = []
    def start_tree(self, start, goal):
        MotionPlanner.plan(self, start, goal)
        self.tree = [Node(self.start)]
        if self.env.collision(start):
            raise ValueError("起点遇到冲突!")
    def clear_tree(self):
        self.tree = []
    def extend_tree(self, des_state=None, stopAtGoal=False):
        old_len = len(self.tree)
        reachedGoal = False
        if des_state == None:
            des_state = self.random_free_state()
        near_node = self.find_nearest_neighbor(des_state)
        success, next_state, cost = self.env.paraofforward(des_state,
            near_node.state, self.step_size)
        if success:
            next_node = Node(next_state)
```

```
            next_node.parent = near_node
            next_node.cost = near_node.cost + cost
            self.tree.append(next_node)
            reachedRand = self.env.distance(des_state, next_state) <
                self.step_size
            reachedGoal = self.env.distance(self.goal, next_state) <
                self.step_size
        if self.growth == 'connect':
            while success and not reachedRand and not (stopAtGoal and
                reachedGoal):
                near_node = next_node
                success, next_state, cost =
                    self.env.paraofforward(des_state, near_node.state,
                    self.step_size)
                if success:
                    next_node = Node(next_state)
                    next_node.parent = near_node
                    next_node.cost = near_node.cost + cost
                    self.tree.append(next_node)
                    reachedRand = self.env.distance(des_state,
                        next_state) < self.step_size
                    reachedGoal = self.env.distance(self.goal,
                        next_state) < self.step_size
        elif self.growth == 'extend':
            pass
        else:
            pass
        return (stopAtGoal and reachedGoal),
            self.tree[old_len:len(self.tree)]
    def grow_tree(self, start, mode='samples', goal=None, samples=0):
        self.start_tree(start)
        new_nodes = [start]
        goalFound = False
        if mode == 'samples':
            for i in range(samples):
                self.extend_tree()
        elif mode == 'goal' and goal != None:
            while not goalFound:
                goalFound, new_nodes = self.extend_tree(stopAtGoal=True)
        else:
            raise ValueError("无效生成!")
    def search_for_goal(self, goal, nodes_list):
        for node in nodes_list:
            if self.env.distance(node.state, goal) < self.step_size:
                return True, node
        return False, None
    def find_path(self, goal):
        MotionPlanner.set_goal(self, goal)
        if len(self.tree) == 0:
            raise ValueError("树空!")
        best_node = self.find_nearest_neighbor(self.goal)
        found_path = self.path_to_root(best_node)
        path_cost = best_node.cost
        return found_path, path_cost
    def plan(self, start, goal):
        MotionPlanner.plan(self, start, goal)
        self.grow_tree(start)
        path, cost = self.find_path(goal)
        return path, cost
    def random_free_state(self):
        while True:
            rand_state = []
            lim = self.env.state_limits
            for i in range(self.env.state_dim):
                rand_state += [randomSampler(lim[0][i], lim[1][i])]
```

```python
            rand_state = tuple(rand_state)
            if not self.env.collision(rand_state):
                break
        return rand_state
    def find_nearest_neighbor(self, state):
        min_dist = float('inf')
        min_node = None
        for node in self.tree:
            node_dist = distance(state, node.state)
            if node_dist < min_dist:
                min_dist = node_dist
                min_node = node
        return min_node
    def next_state_action(self, des_state, node):
        min_dist = distance(node.state, des_state)
        sel_action = None
        next_state = None
        for action in self.env.action_list:
            new_state = self.env.next_state(node.state, action)
            if self.env.collision(new_state):
                continue
            new_dist = distance(new_state, des_state)
            if new_dist < min_dist:
                min_dist = new_dist
                sel_action = action
                next_state = new_state
        return next_state, sel_action
    def path_to_root(self, node):
        path_to_root = []
        curr_node = node
        while curr_node != None:
            path_to_root += [curr_node]
            curr_node = curr_node.parent
        return path_to_root
# Bi-RRT实现
class BiRRT(RRT, MotionPlanner):
    def __init__(self, env, step_size, growth='ce'):
        MotionPlanner.__init__(self, env)
        self.step_size = step_size
        self.meet_thr = 0.1
        if growth == 'ce':
            self.start_growth = 'connect'
            self.goal_growth = 'extend'
        elif growth == 'cc':
            self.start_growth = 'connect'
            self.goal_growth = 'connect'
        elif growth == 'ec':
            self.start_growth = 'extend'
            self.goal_growth = 'connect'
        elif growth == 'ee':
            self.start_growth = 'extend'
            self.goal_growth = 'extend'
        else:
            raise ValueError("双向生成树类型错误!")
        self.rrt_start = RRT(env, self.step_size, self.start_growth)
        self.rrt_goal = RRT(env, self.step_size, self.goal_growth)
    def start_tree(self, start, goal):
        MotionPlanner.plan(self, start, goal)
        self.rrt_start.start_tree(start, goal)
        self.rrt_goal.start_tree(goal, start)
    def clear_tree(self):
        self.rrt_start.clear_tree()
        self.rrt_goal.clear_tree()
    def extend_tree(self):
        goalFound_g, new_goal_nodes =
```

```
            self.rrt_goal.extend_tree(stopAtGoal=True)
        if len(new_goal_nodes) != 0:
            target = new_goal_nodes[-1].state
        else:
            target = None
        goalFound_s, new_start_nodes =
            self.rrt_start.extend_tree(des_state=target, stopAtGoal=True)
        treesMet, meet_point = self.tree_meet(new_start_nodes,
            new_goal_nodes)
        return treesMet, meet_point, new_start_nodes, new_goal_nodes
    def grow_tree(self, start, goal):
        self.start_tree(start, goal)
        treesMet = False
        while not treesMet:
            treesMet, meet_point, start_nodes, goal_nodes =
                self.extend_tree()
        return meet_point
    def plan(self, start, goal):
        meet_point = self.grow_tree(start, goal)
        found_path, path_cost = self.find_path(meet_point)
        return found_path, path_cost
    def tree_meet(self, new_start_nodes=None, new_goal_nodes=None):
        for s in self.rrt_start.tree:
            for g in new_goal_nodes:
                if self.env.distance(s.state, g.state) < self.step_size:
                    return True, (s, g)
        for g in self.rrt_goal.tree:
            for s in new_start_nodes:
                if self.env.distance(s.state, g.state) < self.step_size:
                    return True, (s, g)
        return False, (None, None)
    def find_path(self, meet_point):
        start_meet = meet_point[0]
        goal_meet = meet_point[1]
        start_path = self.rrt_start.path_to_root(start_meet)
        goal_path = self.rrt_goal.path_to_root(goal_meet)
        path_cost = meet_point[0].cost + meet_point[1].cost +
            self.env.distance(meet_point[0].state,meet_point[1].state)
        goal_path.reverse()
        return goal_path + start_path, path_cost
```

img_env.py

```
# 环境搭建
import maths
import rrt_birrt
import numpy as np
# 前进
def forward(destination, curstate, stepsize):
    diffstate = maths.combinevector(destination, '-', curstate)
    unitvector = maths.vectoroperator(diffstate, '/',
        np.linalg.norm(diffstate))
    newstate = maths.combinevector(curstate, '+',
        maths.vectoroperator(unitvector, '*', stepsize))
    return newstate
class ImgEnv:
    # 初始化2维图像
    def __init__(self, img):
        self.state_dim = 2
        self.img = img
        self.FREE = 0
        self.state_limits = [(0, 0), (img.shape[0]-1, img.shape[1]-1)]
    # 距离用欧氏距离
    def distance(self, s1, s2, metric='euclidean'):
```

```
        return rrt_birrt.distance(s1, s2, metric)
    # 要在图的边界内
    def limits(self, point):
        limit = self.state_limits
        for i in range(self.state_dim):
            if point[i] < limit[0][i] or point[i] > limit[1][i]:
                return False
        return True
    # 检查该点可走还是不可走
    def collision(self, point):
        # print(type(point))
        pointlist = list(point)
        # print(point)
        # print(pointlist)
        '''由于角标限制,要将tuple内元素取整,方法是tuple->list,
        将list内每个元素强制转换为int后再将list->tuple.'''
        for i in range(self.state_dim):
            pointlist[i] = int(pointlist[i])
        point = tuple(pointlist)
        # print(type(pointlist[i]))
        # print(type(pointlist))
        # for i in range(self.state_dim):
        #     pointlist[i] = int(pointlist[i])
        if self.limits(point) and self.img[point] == self.FREE:
            return False
        else:
            return True
    # 前进过程中一些参量
    def paraofforward(self, destination, curstate, stepsize):
        newstate = forward(destination, curstate, stepsize)
        newstate = maths.vectoroperator(newstate, 'int')
        cost = self.distance(newstate, curstate)
        if self.collision(newstate):
            flag = False
        else:
            flag = True
        return flag, newstate, cost
```

rrt_example.py

```
# -*- coding: UTF-8 -*-
# RRT测例
import time
import rrt_birrt
import pygame
from img_env import ImgEnv
# Parameter
RED = (255, 0, 0) # Red color
GREEN = (0, 255, 0) # Green color
BLUE = (0, 0, 255) # Blue color
START = (50, 150) # Start position
GOAL = (450, 350) # Goal position
step_size = 10 # RRT Step Size
# 载入的图像文件
IMG_NAME = "images/easy.png"
# IMG_NAME = "images/medium.png"
# IMG_NAME = "images/hard.png"
# 图形界面显示
pygame.init()
clock = pygame.time.Clock()
img = pygame.image.load(IMG_NAME)
pixacc = pygame.surfarray.array2d(img)
screen = pygame.display.set_mode(img.get_size())
```

```
screen.fill((0, 0, 0))
screen.blit(img, (0, 0))
pygame.draw.circle(screen, RED, START, 5)
pygame.draw.circle(screen, GREEN, GOAL, 5)
env = ImgEnv(pixacc)
start = time.time()
# RRT
rrt = rrt_birrt.RRT(env, step_size, 'extend')
rrt.start_tree(START, GOAL)
samples = 0
done = False
goalFound = False
drawPath = True
while not done:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True
    if not goalFound:
        goalFound, new_nodes = rrt.extend_tree(stopAtGoal=True)
        samples += 1
        # 将生成树的过程显示在图上
        for node in new_nodes:
            pygame.draw.line(screen, RED, node.state, node.parent.state)
    if goalFound and drawPath:
        time_taken = time.time() - start
        path, cost = rrt.find_path(GOAL)
        # 将路线标记在图上
        for node in path:
            if node.parent != None:
                pygame.draw.line(screen, GREEN, node.state,
                    node.parent.state, 3)
        drawPath = False
    pygame.display.update()
tree_size = len(rrt.tree)
pygame.quit()
# 打印相关量
print("抽得样本数 =", samples)
print("执行时间 =", time_taken, "s")
print("路径长度 =", len(path)-1)
print("开销 =", cost)
print("树的总规模 =", tree_size)
```

birrt_example.py

```
# -*- coding: UTF-8 -*-
# Bi-RRT测例文件
import time
import rrt_birrt
import pygame
from img_env import ImgEnv
# Parameter
RED = (255, 0, 0) # Red color
GREEN = (0, 255, 0) # Green color
BLUE = (0, 0, 255) # Blue color
START = (50, 150) # Start position
GOAL = (450, 350) # Goal position
step_size = 10 # RRT Step Size
# 载入的图像文件
IMG_NAME = "images/easy.png"
# IMG_NAME = "images/medium.png"
# IMG_NAME = "images/hard.png"
# 图形界面显示
pygame.init()
clock = pygame.time.Clock()
```

```python
img = pygame.image.load(IMG_NAME)
pixacc = pygame.surfarray.array2d(img)
screen = pygame.display.set_mode(img.get_size())
screen.fill((0, 0, 0))
screen.blit(img, (0, 0))
pygame.draw.circle(screen, RED, START, 5)
pygame.draw.circle(screen, GREEN, GOAL, 5)
env = ImgEnv(pixacc)
start = time.time()
# Bi-RRT
birrt = rrt_birrt.BiRRT(env, step_size, 'ce')
birrt.start_tree(START, GOAL)
samples = 0
done = False
goalFound = False
drawPath = True
while not done:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True
    if not goalFound:
        goalFound, meet_point, start_nodes, goal_nodes =
            birrt.extend_tree()
        samples += 1
        # 在图上标出生成树
        for node in start_nodes + goal_nodes:
            pygame.draw.line(screen, RED, node.state, node.parent.state)
    if goalFound and drawPath:
        time_taken = time.time() - start
        path, cost = birrt.find_path(meet_point)
        # 在图上标记出路径
        for node in path:
            if node.parent != None:
                pygame.draw.line(screen, GREEN, node.state,
                    node.parent.state, 3)
        drawPath = False
    pygame.display.update()
tree_size = len(birrt.rrt_goal.tree) + len(birrt.rrt_start.tree)
pygame.quit()
# 打印相关量
print("抽得样本数 =", samples)
print("执行时间 =", time_taken, "s")
print("路径长度 =", len(path)-1)
print("开销 =", cost)
print("树的总规模 =", tree_size)
```

5 实验结果

实验结果的衡量主要看如下参量: 执行时间, 抽样数, 路径长度, 开销, 树的规模 (Bi-RRT 为两棵树的总规模). 本表格只列出每种情况在测试 50 次的情况下最好和最坏的情况.

测例难度	方法	最好/坏	执行时间	抽样数	路径长度	开销	树的规模
easy	RRT	好	0.300s	470	80	758	293
easy	RRT	坏	6.779s	2413	95	901	2033
medium	RRT	好	0.253s	395	64	97	293
medium	RRT	坏	13.273s	5425	67	631	2176
hard	RRT	好	0.790s	995	68	638	487
hard	RRT	坏	16.031s	6732	75	709	1779
easy	Bi-RRT	好	0.047s	50	86	812	155
easy	Bi-RRT	坏	0.227s	191	87	827	322
medium	Bi-RRT	好	0.042s	61	67	654	113
medium	Bi-RRT	坏	11.633s	3739	69	672	2092
hard	Bi-RRT	好	0.126s	147	69	655	208
hard	Bi-RRT	坏	25.830s	4931	61	597	2843

可以看到, 在 easy 难度下,Bi-RRT 可以说完胜 RRT; 在 medium 难度下,Bi-RRT 和 RRT 在很多指标都很接近; 只看最好的情况下 Bi-RRT 比 RRT 要快很多. 但如果只看最坏的情况下, 在 medium 难度下与 RRT 差不多, 而在 hard 难度下要比 RRT 要慢. 个人认为在 hard 难度下, 图中可通过的缝隙过于狭窄, 即便 Bi-RRT 的两颗树虽然会互相提供给对方信息, 但两棵树在缝隙附近相遇就很难, 而 RRT 只需要起点的生成树穿过该缝隙即可, 因此 RRT 比 Bi-RRT 要快. 至于机器人的大小对于本问题的影响, 如果机器人的半径为 r, 我们可以将周边障碍物扩展 r, 机器人的大小不变, 起到的效果是一样的. 经过实验发现: 在 medium(hard) 难度下, 当机器人半径 >4(2) 时, 无论如何也不能通过障碍物 (easy 情况就不讨论了因为可行区域比较开阔). 当然, 由于实现过程中存在一些类型转换, 导致数据可能损失一定的精度, 这也是需要注意的问题.

6 附录

代码运行后可以观察到输出信息如下：

```
C:\Users\czh\.conda\envs\Pycharm\python.exe "D:/Pycharm/PyCharm 2020.2.2/imageRRT-master/rrt_example.py"
pygame 2.0.0 (SDL 2.0.12, python 3.6.12)
Hello from the pygame community. https://www.pygame.org/contribute.html
抽得样本数 = 1568
执行时间 = 1.3977222442626953 s
路径长度 = 72
开销 = 676.2123055599062
树的总规模 = 563
```

Figure 6

这里放几张效果图，红色线为随机生成树，绿色线为找到的路径，红色点为起点，绿色点为终点：

Figure 7

Figure 8

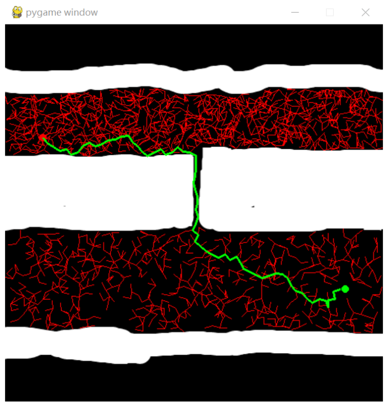

Figure 9

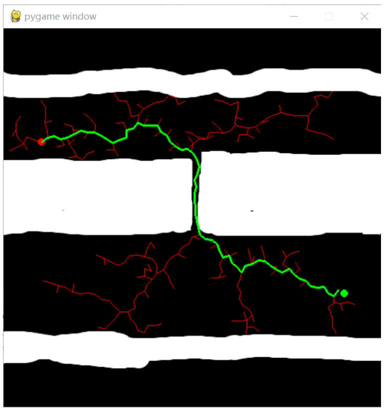

Figure 10

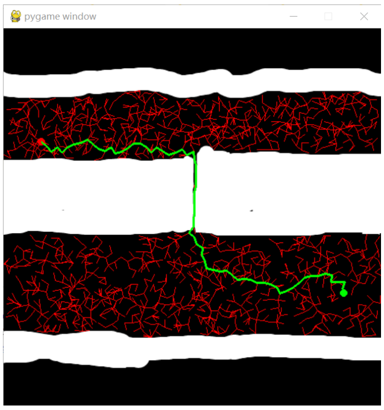

Figure 11

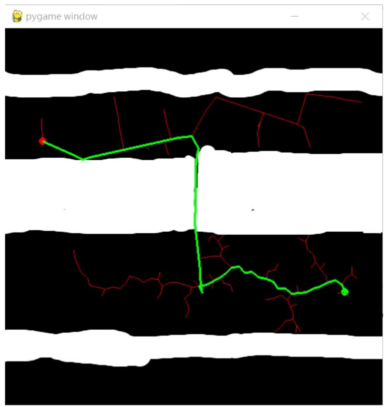

Figure 12