Министерство образования и науки Российской Федерации

Новосибирский государственный технический   
университет

Методы построения и анализа алгоритмов

Расчётно-графическое задание:

Построение пути для мобильного робота  
 в двумерном пространстве

Вариант №4

Факультет: ФПМИ

Группа: ПМИ – 12

Студент: Швадченко А. В.

Преподаватель: Щукин Г. А.

Новосибирск, 2022

Мобильный робот движется по плоскости. Робот - материальная точка с координатой; робот может двигаться в любом направлении без ограничений. Также на плоскости находятся объекты-препятствия. Заданы начальная и конечная координаты робота

Задача: найти кратчайший путь для робота из начальной координаты в конечную, не приводящий к столкновению робота с препятствиями.

Если путь не возможен - сообщить об этом.

Вариант №4: RRT, прямоугольники, без промежуточных координат.

Для решения задач используется взвешенный неориентированный граф. Препятствия (прямоугольники) задаются координатами двух противостоящих точек на диагонали.

Было реализовано внешнее GUI-приложение со следующими возможностями:

* Сохранение сцены в файл .bmp;
* Интерактивное задание сцены: ввод начала / конечной сборки робота, добавление препятствий.

**Код программы:**

**Файл GUI.py**

import os

import time

import wx

import RRT

ADD\_OBSTACLE = 1

ADD\_START\_POINT = 2

ADD\_END\_POINT = 3

LAUNCH\_RRT = 4

LAUNCH\_BEST\_PATH = 5

class MyFrame(wx.Frame):

def \_\_init\_\_(self, parent, title):

# Инициализация пользовательского окна

super().\_\_init\_\_(parent=parent, size=(1200, 800), pos=(100, 100), title=title,

style=wx.MINIMIZE\_BOX | wx.SYSTEM\_MENU | wx.CAPTION | wx.CLOSE\_BOX | wx.CLIP\_CHILDREN)

# Инициализация меню

menu\_bar = wx.MenuBar()

# Вкладка меню "Файл" и её компоненты

self.file\_menu = wx.Menu()

file\_item\_save = self.file\_menu.Append(wx.ID\_SAVE, '&Сохранить\tCtrl+S', 'Сохранить сцену как .png')

file\_item\_exit = self.file\_menu.Append(wx.ID\_EXIT, '&Выход\tCtrl+Q', 'Выход из приложения')

# Вкладка меню "Добавить" и её компоненты

self.add\_menu = wx.Menu()

add\_item\_start\_point = self.add\_menu.Append(ADD\_START\_POINT, '&Начальная точка\tCtrl+1', 'Добавить начальную точку пути')

add\_item\_end\_point = self.add\_menu.Append(ADD\_END\_POINT, '&Конечная точка\tCtrl+2', 'Добавить конечную точку пути')

add\_item\_obstacle = self.add\_menu.Append(ADD\_OBSTACLE, '&Препятствие\tCtrl+3', 'Добавить препятствие')

# Вкладка меню "Запустить" и её компоненты

self.launch = wx.Menu()

launch\_RRT = self.launch.Append(LAUNCH\_RRT, '&Построить все пути\tCtrl+P', 'Построить все возможные пути')

launch\_best\_path = self.launch.Append(LAUNCH\_BEST\_PATH, '&Построить кратчайший путь\tCtrl+B', 'Построить кратчайший путь из начальной в конечную точку')

# Добавление всех вкладок и их компонентов в меню

menu\_bar.Append(self.file\_menu, 'Файл')

menu\_bar.Append(self.add\_menu, 'Добавить')

menu\_bar.Append(self.launch, 'Запустить')

self.SetMenuBar(menu\_bar)

# События в меню

self.Bind(wx.EVT\_MENU, self.onQuit, file\_item\_exit) # Выход

self.Bind(wx.EVT\_MENU, self.onSave, file\_item\_save) # Сохранение

self.Bind(wx.EVT\_MENU, self.onAddStartPoint, add\_item\_start\_point) # Добавление нач. точки

self.Bind(wx.EVT\_MENU, self.onAddEndPoint, add\_item\_end\_point) # Добавление кон. точки

self.Bind(wx.EVT\_MENU, self.onAddObstacle, add\_item\_obstacle) # Добавление препятствия

self.Bind(wx.EVT\_MENU, self.onLaunchRRT, launch\_RRT) # Построение вершин дерева

self.Bind(wx.EVT\_MENU, self.onLaunchBestPath, launch\_best\_path) # Поиск и построение кратчайшего пути

# Переменные класса

self.start\_point\_pos = (-1, -1) # Координаты нач. точки

self.end\_point\_pos = (-1, -1) # Кординаты кон. точки

self.obstacle = [] # Координаты одного препятствия

self.obstacles = [] # Координаты препятствий (одно препятствие = две точки; из них прямоугольник)

self.iterations\_num = 0 # Количество итераций получения случайных точек на поле

self.step\_size = 0 # Шаг (размер) между вершинами графа

self.best\_path\_flag = False # Флаг на то, что получилось построить кратчайший путь

self.rrt = None # Объект класса RRTAlgorithm

# Инициализация "битовой карты" (для сохранения сцены как .bmp)

self.btm = wx.Bitmap((1184, 742))

self.btm\_dc = wx.MemoryDC()

self.btm\_dc.SelectObject(self.btm)

self.Bind(wx.EVT\_PAINT, self.onPaintBorder)

def onQuit(self, event):

# Событие на выход из приложения

dlg = wx.MessageDialog(self, 'Вы дейстительно хотите выйти из программы?', 'Выход', wx.YES\_NO | wx.NO\_DEFAULT | wx.ICON\_QUESTION)

res = dlg.ShowModal()

if res == wx.ID\_YES:

self.Close()

def onSave(self, event):

# Событие на сохранение файла

path = os.path.abspath(\_\_file\_\_)

file\_name = os.path.basename(\_\_file\_\_)

path = path[0:len(path)-len(file\_name)]

dlg = wx.MessageDialog(self, f'Файл сохранен в {path}img.bmp', 'Сохранить', wx.OK\_DEFAULT)

dlg.ShowModal()

self.btm.SaveFile('img.bmp', wx.BITMAP\_TYPE\_BMP)

def onAddStartPoint(self, event):

# Событие на добавление стартовой точки

#dlg = wx.MessageDialog(self, 'Кликните в любом месте поля для добавления начальной точки', 'Добавить начальную точку', wx.OK\_DEFAULT)

#dlg.ShowModal()

self.add\_menu.Delete(ADD\_START\_POINT)

self.Bind(wx.EVT\_LEFT\_DOWN, self.onMouseClick)

def onAddEndPoint(self, event):

# Событие на добавление конечной точки

#dlg = wx.MessageDialog(self, 'Кликните в любом месте поля для добавления конечной точки', 'Добавить конечную точку', wx.OK\_DEFAULT)

#dlg.ShowModal()

self.add\_menu.Delete(ADD\_END\_POINT)

self.Bind(wx.EVT\_LEFT\_DOWN, self.onMouseClick)

def onAddObstacle(self, event):

# Событие на добавление препятствий

#if len(self.obstacles) == 0:

#dlg = wx.MessageDialog(self, 'Кликните в двух любых местах поля для добавления препятствия в виде прямоугольника (эти две точки - диагональ прямоугольника)', 'Добавить препятствие', wx.OK\_DEFAULT)

#dlg.ShowModal()

self.Bind(wx.EVT\_LEFT\_DOWN, self.onMouseClick)

def onMouseClick(self, event):

# Событие на получение координат мыши при клике по сцене

pos = event.GetPosition()

if pos.x < 20:

pos.x = 20

if pos.x > 20 + 1144:

pos.x = 20 + 1144

if pos.y < 20:

pos.y = 20

if pos.y > 20 + 702:

pos.y = 20 + 702

if self.add\_menu.FindItemById(ADD\_START\_POINT) == None and self.start\_point\_pos == (-1, -1):

self.start\_point\_pos = (pos.x, pos.y)

wx.CallLater(10, self.onPaintDot)

self.Unbind(event=wx.EVT\_LEFT\_DOWN)

elif self.add\_menu.FindItemById(ADD\_END\_POINT) == None and self.end\_point\_pos == (-1, -1):

self.end\_point\_pos = (pos.x, pos.y)

wx.CallLater(10, self.onPaintDot)

self.Unbind(event=wx.EVT\_LEFT\_DOWN)

else:

self.obstacle.append((pos.x, pos.y))

if len(self.obstacle) == 2:

self.obstacles.append(self.obstacle)

wx.CallLater(10, self.onPaintRect)

self.obstacle = []

self.Unbind(event=wx.EVT\_LEFT\_DOWN)

def onPaintBorder(self, event):

# Событие на отрисовку границы поля, в пределах которого будут все объекты

dc = wx.PaintDC(self)

self.btm\_dc.SetPen(wx.Pen(wx.BLACK, 2))

self.btm\_dc.SetBrush(wx.Brush(wx.GREY\_BRUSH))

self.btm\_dc.DrawRectangle(-1, -1, 1187, 745)

self.btm\_dc.DrawRectangle(18, 18, 1147, 705)

dc.Blit(0, 0, 1200, 800, self.btm\_dc, 0, 0)

def onPaintDot(self):

# Событие на отрисовку начальной/конечной точки

dc = wx.ClientDC(self)

self.btm\_dc.SetPen(wx.Pen(wx.RED, 15))

if self.start\_point\_pos != (-1, -1):

if self.end\_point\_pos == (-1, -1):

self.btm\_dc.DrawLine(self.start\_point\_pos[0], self.start\_point\_pos[1], self.start\_point\_pos[0], self.start\_point\_pos[1])

else:

self.btm\_dc.DrawLine(self.start\_point\_pos[0], self.start\_point\_pos[1], self.start\_point\_pos[0], self.start\_point\_pos[1])

self.btm\_dc.DrawLine(self.end\_point\_pos[0], self.end\_point\_pos[1], self.end\_point\_pos[0], self.end\_point\_pos[1])

elif self.end\_point\_pos != (-1, -1):

self.btm\_dc.DrawLine(self.end\_point\_pos[0], self.end\_point\_pos[1], self.end\_point\_pos[0], self.end\_point\_pos[1])

dc.Blit(0, 0, 1200, 800, self.btm\_dc, 0, 0)

def onPaintRect(self):

# Событие на отрисовку препятствия в виде прямоугольника

dc = wx.ClientDC(self)

self.btm\_dc.SetPen(wx.Pen(wx.BLUE, 1))

self.btm\_dc.SetBrush(wx.Brush(wx.BLUE))

obstacle = self.obstacles[len(self.obstacles) - 1]

x = min(obstacle[0][0], obstacle[1][0]) + 1

y = min(obstacle[0][1], obstacle[1][1]) + 1

len\_x = max(obstacle[0][0], obstacle[1][0]) - x - 1

len\_y = max(obstacle[0][1], obstacle[1][1]) - y - 1

self.btm\_dc.DrawRectangle(x, y, len\_x, len\_y)

wx.CallLater(10, self.onPaintDot)

dc.Blit(0, 0, 1200, 800, self.btm\_dc, 0, 0)

def onLaunchRRT(self, event):

# Построение вершин дерева, если заданы начальная и конечная точки

if self.start\_point\_pos == (-1, -1) or self.end\_point\_pos == (-1, -1):

dlg = wx.MessageDialog(self, 'Сначала нужно добавить начальную и конечную точки', 'Ошибка', wx.OK\_DEFAULT)

dlg.ShowModal()

else:

# Ввод количества итераций цикла фукнции RRT

dlg = wx.TextEntryDialog(self, 'Введите количество итераций', 'Ввод данных', '1000')

res = dlg.ShowModal()

if res == wx.ID\_OK:

try:

iterations\_num = int(dlg.GetValue())

if iterations\_num <= 0:

dlg\_err = wx.MessageDialog(self, 'Неверное количество итераций', 'Ошибка', wx.OK\_DEFAULT)

dlg\_err.ShowModal()

return

self.iterations\_num = iterations\_num

except Exception as e:

dlg\_err = wx.MessageDialog(self, 'Неверное количество итераций', 'Ошибка', wx.OK\_DEFAULT)

dlg\_err.ShowModal()

return

else:

return

# Ввод размера шага (размера) ветви

dlg = wx.TextEntryDialog(self, 'Введите размер шага\n(Размер поля 1164×702)', 'Ввод данных', '30')

res = dlg.ShowModal()

if res == wx.ID\_OK:

try:

step\_size = int(dlg.GetValue())

if step\_size <= 0:

dlg\_err = wx.MessageDialog(self, 'Неверный размер шага', 'Ошибка', wx.OK\_DEFAULT)

dlg\_err.ShowModal()

return

self.step\_size = step\_size

except Exception as e:

dlg\_err = wx.MessageDialog(self, 'Неверный размер шага', 'Ошибка', wx.OK\_DEFAULT)

dlg\_err.ShowModal()

return

else:

return

# Отрисовка построенного дерева дерева

dc = wx.ClientDC(self)

self.btm\_dc.SetBrush(wx.Brush(wx.TRANSPARENT\_BRUSH))

self.btm\_dc.SetPen(wx.Pen('#fdc073', 3, wx.LONG\_DASH))

if self.step\_size < 15:

step = 15

elif self.step\_size > 30:

step = 30

else:

step = self.step\_size

self.btm\_dc.DrawRectangle(self.end\_point\_pos[0] - step - 1, self.end\_point\_pos[1] - step - 1, step\*2 + 3, step\*2 + 3)

self.btm\_dc.SetPen(wx.Pen(wx.RED, 15))

self.btm\_dc.DrawLine(self.end\_point\_pos[0], self.end\_point\_pos[1], self.end\_point\_pos[0], self.end\_point\_pos[1])

dc.Blit(0, 0, 1200, 800, self.btm\_dc, 0, 0)

self.rrt = RRT.RRTAlgorithm(self.start\_point\_pos, self.end\_point\_pos, self.iterations\_num, self.step\_size, self.obstacles)

self.rrt.RRT()

for p1 in self.rrt.tree\_nodes:

for p2 in p1.children:

self.btm\_dc.SetPen(wx.Pen('#ffffff', 3))

self.btm\_dc.DrawLine(p1.x, p1.y, p2.x, p2.y)

self.btm\_dc.SetPen(wx.Pen(wx.RED, 15))

self.btm\_dc.DrawLine(self.start\_point\_pos[0], self.start\_point\_pos[1], self.start\_point\_pos[0], self.start\_point\_pos[1])

self.btm\_dc.DrawLine(self.end\_point\_pos[0], self.end\_point\_pos[1], self.end\_point\_pos[0], self.end\_point\_pos[1])

dc.Blit(0, 0, 1200, 800, self.btm\_dc, 0, 0)

time\_sleep = 0.1 / self.iterations\_num

#time.sleep(time\_sleep)

self.btm\_dc.SetPen(wx.Pen(wx.BLACK, 5))

dots = self.rrt.definePointsInGoalRadius()

if len(dots) != 0:

self.best\_path\_flag = True

dlg = wx.MessageDialog(self, 'Все пути построены', 'Готово', wx.OK\_DEFAULT)

dlg.ShowModal()

def onLaunchBestPath(self, event):

# Построение и отрисовка кратчайшего пути по вершинам дерева, если те уже построены

if self.rrt != None and self.best\_path\_flag == True:

dc = wx.ClientDC(self)

best\_path = self.rrt.buildBestPath()

for i in range(len(best\_path) - 1):

self.btm\_dc.SetPen(wx.Pen(wx.GREEN, 5))

self.btm\_dc.DrawLine(best\_path[i][0], best\_path[i][1], best\_path[i+1][0], best\_path[i+1][1])

self.btm\_dc.SetPen(wx.Pen(wx.RED, 15))

self.btm\_dc.DrawLine(self.start\_point\_pos[0], self.start\_point\_pos[1], self.start\_point\_pos[0], self.start\_point\_pos[1])

dc.Blit(0, 0, 1200, 800, self.btm\_dc, 0, 0)

time.sleep(0.1)

self.btm\_dc.SetPen(wx.Pen(wx.GREEN, 5))

self.btm\_dc.DrawLine(best\_path[len(best\_path) - 1][0], best\_path[len(best\_path) - 1][1], self.end\_point\_pos[0], self.end\_point\_pos[1])

self.btm\_dc.SetPen(wx.Pen(wx.RED, 15))

self.btm\_dc.DrawLine(self.end\_point\_pos[0], self.end\_point\_pos[1], self.end\_point\_pos[0], self.end\_point\_pos[1])

dc.Blit(0, 0, 1200, 800, self.btm\_dc, 0, 0)

dlg = wx.MessageDialog(self, 'Кратчайший путь построен', 'Готово', wx.OK\_DEFAULT)

dlg.ShowModal()

elif self.best\_path\_flag == False:

dlg = wx.MessageDialog(self, 'Не удалось построить кратчайший путь', 'Ошибка', wx.OK\_DEFAULT)

dlg.ShowModal()

else:

dlg = wx.MessageDialog(self, 'Сначала запустите построение всех путей', 'Ошибка', wx.OK\_DEFAULT)

dlg.ShowModal()

app = wx.App()

frame = MyFrame(None, 'RRT algorithm')

frame.Show()

app.MainLoop()

**Файл RRT.py**

import random

import math

import sys

class TreeNode:

# Вершина дерева

def \_\_init\_\_(self, x, y, parent):

self.x = x

self.y = y

self.children = []

self.parent = parent

class RRTAlgorithm:

def \_\_init\_\_(self, start, goal, iterations\_num, step\_size, obstacles):

# Инициализация дерева

self.start = start

self.goal = goal

self.step\_size = step\_size

self.iterations\_num = iterations\_num

self.obstacles = obstacles

self.tree\_nodes = []

self.tree\_nodes.append(TreeNode(self.start[0], self.start[1], None))

def randPoint(self):

# Генерация случайной точки

flag = True

while flag:

point = (random.randint(20, 1164), random.randint(20, 722))

flag = False

for node in self.tree\_nodes:

if node.x == point[0] and node.y == point[1]:

flag = True

return point

def ifInObstacle(self, p):

# Проверка: находится ли точка внутри препятствия

for obstacle in self.obstacles:

left\_border = min(obstacle[0][0], obstacle[1][0])

up\_border = max(obstacle[0][1], obstacle[1][1])

right\_border = max(obstacle[0][0], obstacle[1][0])

bottom\_border = min(obstacle[0][1], obstacle[1][1])

if p[0] > left\_border and p[0] < right\_border and p[1] < up\_border and p[1] > bottom\_border:

return True

return False

def ifCollision(self, p1, p2):

# Проверка: есть ли коллизия между отрезком и заданными препятствиями (если да, вернет еще препятствие)

for obstacle in self.obstacles:

p\_border1 = (min(obstacle[0][0], obstacle[1][0]), min(obstacle[0][1], obstacle[1][1]))

p\_border2 = (min(obstacle[0][0], obstacle[1][0]), max(obstacle[0][1], obstacle[1][1]))

p\_border3 = (max(obstacle[0][0], obstacle[1][0]), max(obstacle[0][1], obstacle[1][1]))

p\_border4 = (max(obstacle[0][0], obstacle[1][0]), min(obstacle[0][1], obstacle[1][1]))

x1 = p1[0]

y1 = p1[1]

x2 = p2[0]

y2 = p2[1]

if self.linesCollisionPoint(x1, y1, x2, y2, p\_border1[0], p\_border1[1], p\_border2[0], p\_border2[1]) != (-1, -1) or \

self.linesCollisionPoint(x1, y1, x2, y2, p\_border2[0], p\_border2[1], p\_border3[0], p\_border3[1]) != (-1, -1) or \

self.linesCollisionPoint(x1, y1, x2, y2, p\_border3[0], p\_border3[1], p\_border4[0], p\_border4[1]) != (-1, -1) or \

self.linesCollisionPoint(x1, y1, x2, y2, p\_border4[0], p\_border4[1], p\_border1[0], p\_border1[1]) != (-1, -1):

return [True, obstacle]

return [False, '']

def linesCollisionPoint(self, x1, y1, x2, y2, x3, y3, x4, y4):

# Проверка: пересечение двух отрезков (если да, вернет точку пересечения, если нет - (-1, -1))

if y2 - y1 != 0:

q = (x2 - x1) / (y1 - y2)

s = (x3 - x4) + (y3 - y4)\*q

if s == 0:

return (-1, -1)

f = (x3 - x1) + (y3 - y1)\*q

k = f / s

else:

if y3 - y4 == 0:

return (-1, -1)

k = (y3 - y1) / (y3 - y4)

p\_x = x3 + (x4 - x3)\*k

p\_y = y3 + (y4 - y3)\*k

if p\_x <= max(x1, x2) and p\_x >= min(x1, x2) and p\_y <= max(y1, y2) and p\_y >= min(y1, y2) \

and p\_x <= max(x3, x4) and p\_x >= min(x3, x4) and p\_y <= max(y3, y4) and p\_y >= min(y3, y4):

return (p\_x, p\_y)

else:

return (-1, -1)

def nearestPointFromTree(self, p):

# Вовзращает ближайшую вершину из дерева к точке (либо вершина, либо точка на ветви)

# Поиск ближайшей вершины

min\_length\_to\_node = 1000000000

min\_node = None

if len(self.tree\_nodes) == 1:

return [(self.tree\_nodes[0].x, self.tree\_nodes[0].y), 'first\_point']

for node in self.tree\_nodes:

length = math.sqrt(abs(node.x - p[0])\*\*2 + abs(node.y - p[1])\*\*2)

if length < min\_length\_to\_node:

min\_length\_to\_node = length

min\_node = node

# Поиск ближайшей точки на ветви между потомками ближайшей вершины и этой вершины

coef = 1000

min\_length\_to\_child\_edge = 1000000000

min\_child = None

min\_point\_on\_child\_edge = None

if len(min\_node.children) != 0:

for child in min\_node.children:

vector = (child.x - min\_node.x, child.y - min\_node.y)

perpend\_vector = (vector[1]\*coef, -vector[0]\*coef)

additional\_p = (p[0] + perpend\_vector[0], p[1] + perpend\_vector[1])

if self.linesCollisionPoint(p[0], p[1], additional\_p[0], additional\_p[1], min\_node.x, min\_node.y, child.x, child.y) == (-1, -1):

additional\_p = (p[0] - perpend\_vector[0], p[1] - perpend\_vector[1])

intersection\_p = self.linesCollisionPoint(p[0], p[1], additional\_p[0], additional\_p[1], min\_node.x, min\_node.y, child.x, child.y)

length = math.sqrt(abs(intersection\_p[0] - p[0])\*\*2 + abs(intersection\_p[1] - p[1])\*\*2)

if length < min\_length\_to\_child\_edge:

min\_length\_to\_child\_edge = length

min\_child = child

min\_point\_on\_child\_edge = intersection\_p

# Поиск ближайшей точки на ветви между "отцом" ближайшей вершины и этой вершины

min\_length\_to\_parent\_edge = 1000000000

intersection\_p = None

if min\_node.parent != None:

vector = (min\_node.x - min\_node.parent[0], min\_node.y - min\_node.parent[1])

perpend\_vector = (vector[1]\*coef, -vector[0]\*coef)

additional\_p = (p[0] + perpend\_vector[0], p[1] + perpend\_vector[1])

if self.linesCollisionPoint(p[0], p[1], additional\_p[0], additional\_p[1], min\_node.x, min\_node.y, min\_node.parent[0], min\_node.parent[1]) == (-1, -1):

additional\_p = (p[0] - perpend\_vector[0], p[1] - perpend\_vector[1])

intersection\_p = self.linesCollisionPoint(p[0], p[1], additional\_p[0], additional\_p[1], min\_node.x, min\_node.y, min\_node.parent[0], min\_node.parent[1])

length = math.sqrt(abs(intersection\_p[0] - p[0]) \*\* 2 + abs(intersection\_p[1] - p[1]) \*\* 2)

if length < min\_length\_to\_parent\_edge:

min\_length\_to\_parent\_edge = length

# Возвращаем ближайшую точку дерева (либо точка вершины, либо точка на ветви между отцом и вершиной, либо точка на ветви между вершиной и потомком)

if min\_length\_to\_node <= min\_length\_to\_child\_edge and min\_length\_to\_node <= min\_length\_to\_parent\_edge:

return [(round(min\_node.x), round(min\_node.y)), 'node\_point']

elif min\_length\_to\_child\_edge <= min\_length\_to\_parent\_edge:

return [(round(min\_point\_on\_child\_edge[0]), round(min\_point\_on\_child\_edge[1])), 'child\_edge', min\_child]

else:

return [(round(intersection\_p[0]), round(intersection\_p[1])), 'parent\_edge', min\_node]

def steerToPoint(self, p1, p2):

# Возвращает точку на луче [p1, p2) с некоторым заданным шагом такую, что не будет коллизии

vector = (p2[0] - p1[0], p2[1] - p1[1])

vector\_length = math.sqrt(vector[0]\*\*2 + vector[1]\*\*2)

if vector\_length <= 1:

return (-1, -1)

new\_vector = (vector[0] \* self.step\_size / vector\_length, vector[1] \* self.step\_size / vector\_length)

new\_point = (p1[0] + new\_vector[0], p1[1] + new\_vector[1])

if new\_point[0] < 20 or new\_point[0] > 1164 or new\_point[1] < 20 or new\_point[1] > 722:

return (-1, -1)

if self.ifCollision(p1, new\_point)[0] == False:

return (round(new\_point[0]), round(new\_point[1]))

else:

x = new\_point[0]

y = new\_point[1]

obstacle = self.ifCollision(p1, new\_point)[1]

left\_border1 = (min(obstacle[0][0], obstacle[1][0]), min(obstacle[0][1], obstacle[1][1]))

left\_border2 = (min(obstacle[0][0], obstacle[1][0]), max(obstacle[0][1], obstacle[1][1]))

up\_border1 = left\_border2

up\_border2 = (max(obstacle[0][0], obstacle[1][0]), max(obstacle[0][1], obstacle[1][1]))

right\_border1 = up\_border2

right\_border2 = (max(obstacle[0][0], obstacle[1][0]), min(obstacle[0][1], obstacle[1][1]))

bottom\_border1 = right\_border2

bottom\_border2 = left\_border1

border\_lines = [[left\_border1, left\_border2], [up\_border1, up\_border2], [right\_border1, right\_border2], [bottom\_border1, bottom\_border2]]

min\_length = 1000000

new\_x = x

new\_y = y

for border in border\_lines:

point = self.linesCollisionPoint(p1[0], p1[1], x, y, border[0][0], border[0][1], border[1][0], border[1][1])

if point != (-1, -1):

length = math.sqrt(abs(p1[0] - point[0])\*\*2 + abs(p1[1] - point[1])\*\*2)

if length < min\_length:

min\_length = length

new\_x = point[0]

new\_y = point[1]

return (round(new\_x), round(new\_y))

def RRT(self):

# Основная функция алгоритма, которая строит дерево

for k in range(self.iterations\_num):

rand\_p = self.randPoint()

nearest\_p = self.nearestPointFromTree(rand\_p)

steer = self.steerToPoint(nearest\_p[0], rand\_p)

while steer == (-1, -1) or steer[0] < 20 or steer[0] > 1164 or steer[1] < 20 or steer[1] > 722 \

or nearest\_p[0][0] < 20 or nearest\_p[0][0] > 1164 or nearest\_p[0][1] < 20 or nearest\_p[0][1] > 722 \

or self.ifCollision(nearest\_p[0], steer)[0] == True:

rand\_p = self.randPoint()

nearest\_p = self.nearestPointFromTree(rand\_p)

steer = self.steerToPoint(nearest\_p[0], rand\_p)

if self.ifInObstacle(nearest\_p[0]) == True or self.ifInObstacle(steer) == True:

k -= 1

continue

if nearest\_p[1] == 'first\_point':

new\_node = TreeNode(steer[0], steer[1], (self.tree\_nodes[0].x, self.tree\_nodes[0].y))

self.tree\_nodes[0].children.append(new\_node)

self.tree\_nodes.append(new\_node)

elif nearest\_p[1] == 'node\_point':

new\_node = TreeNode(steer[0], steer[1], (nearest\_p[0][0], nearest\_p[0][1]))

for i in range(len(self.tree\_nodes)):

if self.tree\_nodes[i].x == nearest\_p[0][0] and self.tree\_nodes[i].y == nearest\_p[0][1]:

self.tree\_nodes[i].children.append(new\_node)

break

self.tree\_nodes.append(new\_node)

else:

child = nearest\_p[2]

parent = None

for i in range(len(self.tree\_nodes)):

if self.tree\_nodes[i].x == child.x and self.tree\_nodes[i].y == child.y:

parent = self.tree\_nodes[i].parent

#self.tree\_nodes[i].parent = nearest\_p[0]

break

for i in range(len(self.tree\_nodes)):

if self.tree\_nodes[i].x == parent[0] and self.tree\_nodes[i].y == parent[1]:

new\_node\_for\_nearest = TreeNode(nearest\_p[0][0], nearest\_p[0][1], (parent[0], parent[1]))

new\_node\_for\_nearest.children.append(child)

new\_node\_for\_steer = TreeNode(steer[0], steer[1], (new\_node\_for\_nearest.x, new\_node\_for\_nearest.y))

new\_node\_for\_nearest.children.append(new\_node\_for\_steer)

self.tree\_nodes[i].children.append(new\_node\_for\_nearest)

self.tree\_nodes.append(new\_node\_for\_nearest)

self.tree\_nodes.append(new\_node\_for\_steer)

break

def definePointsInGoalRadius(self):

# Возвращает ближайшие точки к целевой, лежащие в некотором радиусе

nearest\_p = self.nearestPointFromTree(self.goal)

if self.step\_size < 15:

step = 15

elif self.step\_size > 30:

step = 30

else:

step = self.step\_size

left\_border = self.goal[0] - step

up\_border = self.goal[1] + step

right\_border = self.goal[0] + step

bottom\_border = self.goal[1] - step

points = []

for node in self.tree\_nodes:

if node.x >= left\_border and node.x <= right\_border and node.y <= up\_border and node.y >= bottom\_border:

if self.ifCollision((node.x, node.y), self.goal)[0] == False and self.ifInObstacle((node.x, node.y)) == False and self.ifInObstacle(self.goal) == False:

if (node.x, node.y) not in points:

points.append((node.x, node.y))

if self.ifCollision(nearest\_p[0], self.goal)[0] == False and self.ifInObstacle(nearest\_p[0]) == False and self.ifInObstacle(self.goal) == False:

if nearest\_p[0] not in points:

points.append(nearest\_p[0])

if nearest\_p[0] in points:

if nearest\_p[1] == 'node\_point':

return points

else:

for node in self.tree\_nodes:

if node.x == nearest\_p[0][0] and node.y == nearest\_p[0][1]:

return points

child = nearest\_p[2]

parent = None

for i in range(len(self.tree\_nodes)):

if self.tree\_nodes[i].x == child.x and self.tree\_nodes[i].y == child.y:

parent = self.tree\_nodes[i].parent

#self.tree\_nodes[i].parent = nearest\_p[0]

break

for i in range(len(self.tree\_nodes)):

if self.tree\_nodes[i].x == parent[0] and self.tree\_nodes[i].y == parent[1]:

new\_node = TreeNode(nearest\_p[0][0], nearest\_p[0][1], (parent[0], parent[1]))

new\_node.children.append(child)

for j in range(len(self.tree\_nodes[i].children)):

if self.tree\_nodes[i].children[j].x == child.x and self.tree\_nodes[i].children[j].y == child.y:

self.tree\_nodes[i].children[j] = new\_node

#self.tree\_nodes[i].children.append(new\_node)

break

self.tree\_nodes.append(new\_node)

break

return points

return points

def buildBestPath(self):

# Строит кратчайший путь из начальной к целевой точке

nearest\_points\_to\_goal = self.definePointsInGoalRadius()

if len(nearest\_points\_to\_goal) == 0:

return None

paths = []

paths\_weights = []

for point in nearest\_points\_to\_goal:

p = point

path = [p]

weight = 0

while p != self.start:

for node in self.tree\_nodes:

if node.x == p[0] and node.y == p[1]:

weight += math.sqrt(abs(node.x - node.parent[0])\*\*2 + abs(node.y - node.parent[1])\*\*2)

p = node.parent

path.insert(0, p)

paths.append(path)

paths\_weights.append(weight)

min\_weight = 1000000000

min\_weight\_ind = 0

for i in range(len(paths\_weights)):

if paths\_weights[i] < min\_weight:

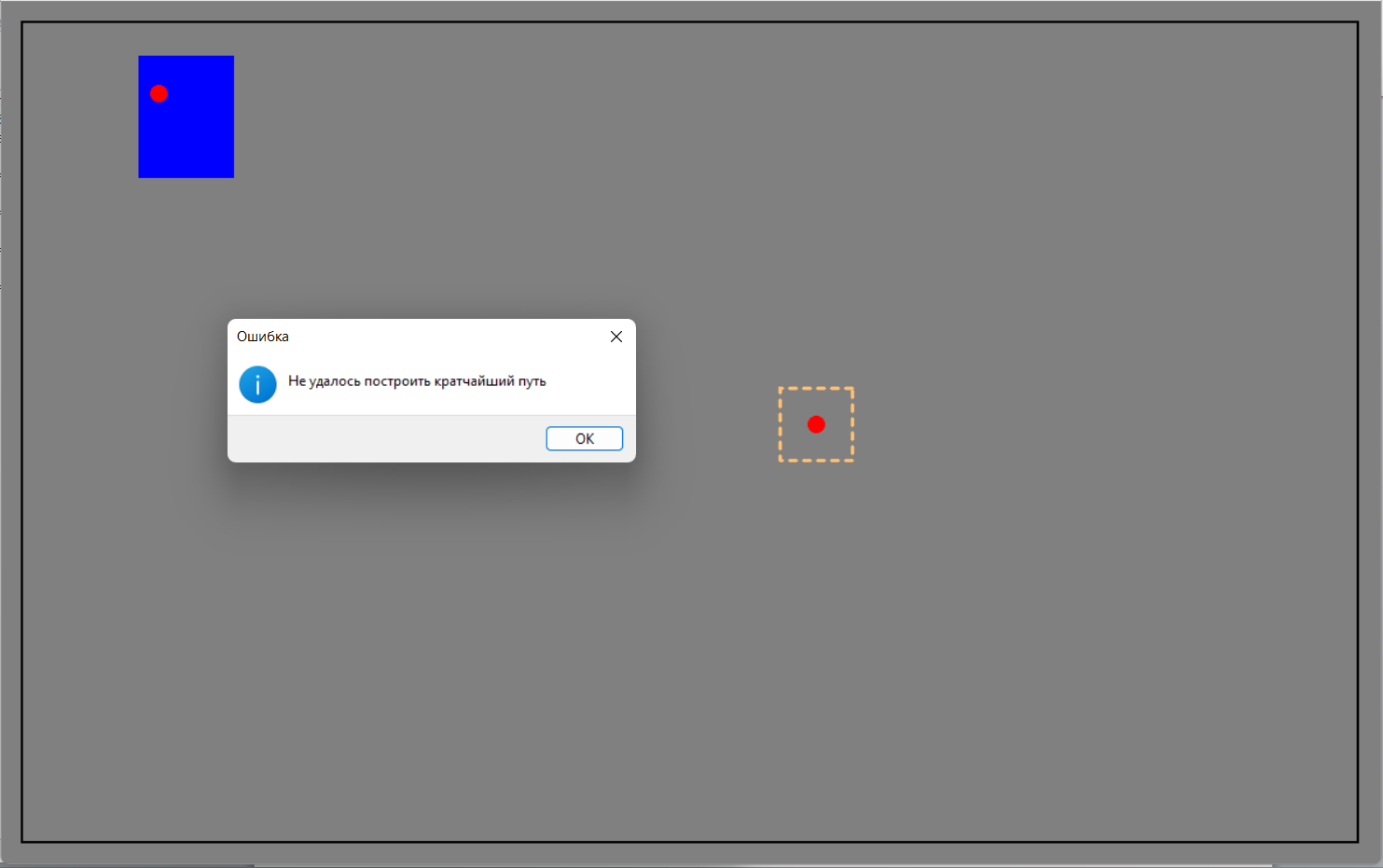
min\_weight = paths\_weights[i]

min\_weight\_ind = i

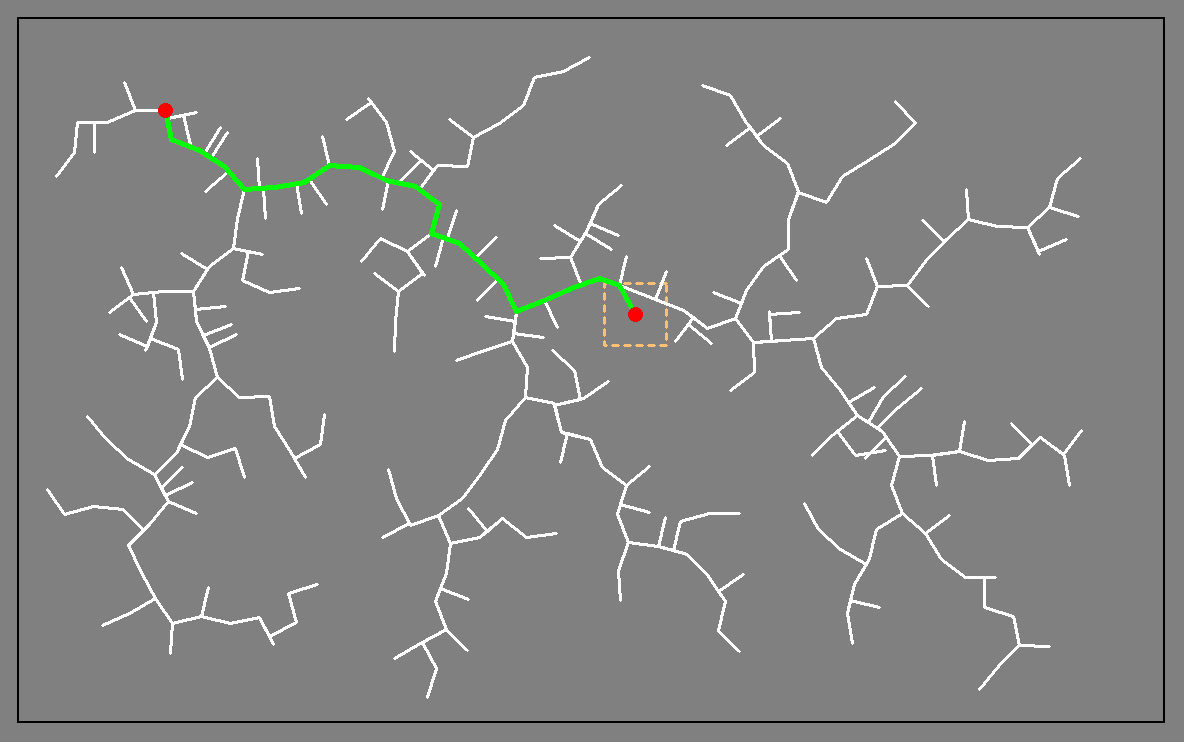
return paths[min\_weight\_ind]

**Тесты**

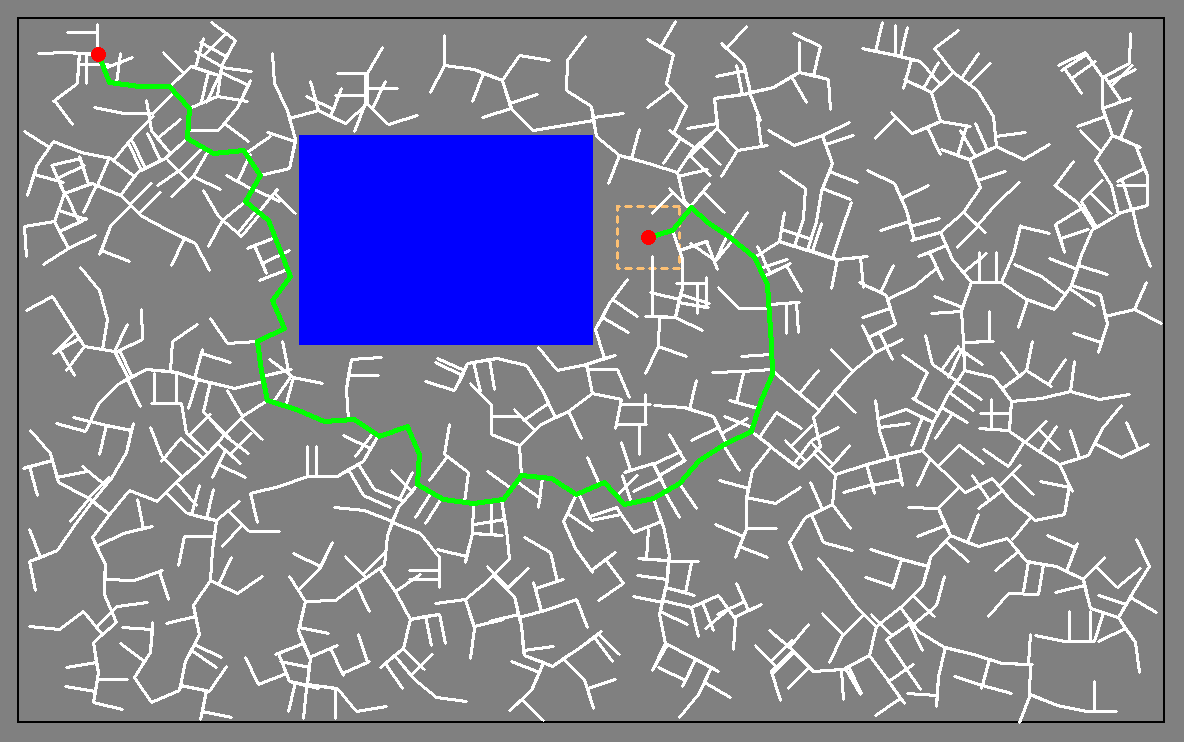
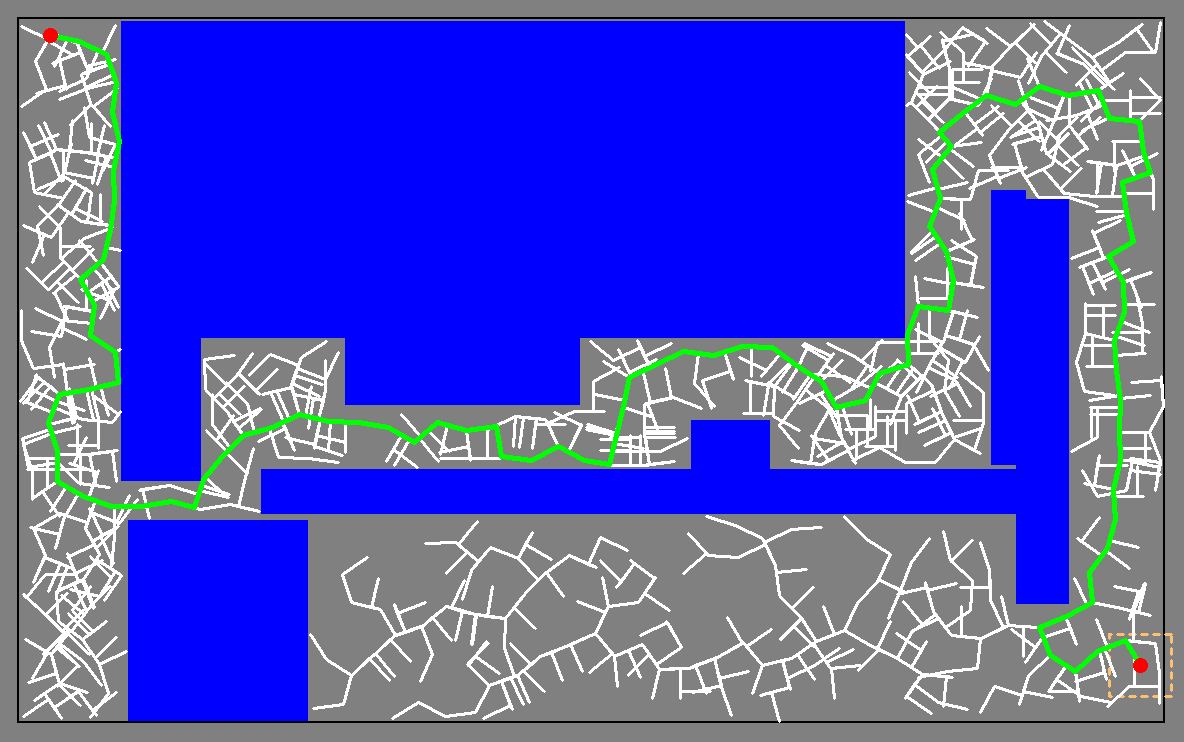
1. Тест при отсутствии пути из начальной точки в конечную.

****

1. Тест без препятствий.



1. Тест с одним препятствием.



1. Тест с большим количеством препятствий.

Программа на всех тестах работает корректно и выводит ожидаемый результат.