Санкт-Петербургский политехнический университет Петра Великого
Институт прикладной математики и механики
Кафедра «Телематика (при ЦНИИ РТК)»

КУРСОВАЯ РАБОТА

по дисциплине «Семинар по роботизированным системам»
на тему «Муравьиный алгоритм и алгоритм коллективного распределения целей»

по направлению 02.04.01.02 «Организация и управление суперкомпьютерными системами»

Выполнил: Титов А.И.
Проверил: Глазунов В.В.

Санкт-Петербург
2019

Оглавление

ПОСТАНОВКА ЗАДАЧИ

Целью курсовой работы является реализация и исследование алгоритмов для построения оптимальных путей роботов до целей. Далее под роботом, для простоты, будет иметься в виду непосредственно начальная координата пути, а под целью, соответственно, конечная координата.

Таким образом, требуется создать карту местности, на которой помещается набор роботов и набор целей, после чего каждому роботу оптимально назначить цель и построить путь до нее.

Для этого требуется реализовать следующие алгоритмы:

- Алгоритм для процедурного построения реалистичной карты местности [1];
- Алгоритм коллективного распределения целей [2];
- Алгоритм поиска пути [3].

Для генерации реалистичной карты среды выбран алгоритм Diamond-Square [4]. Алгоритм коллективного распределения целей описан в [2] в главе «Алгоритм коллективного улучшения плана 3.7» на стр. 102. Для вычисления пути от робота до цели рассматривается муравьиный алгоритм [5].

Выполняются следующие задачи для достижения цели:

1. Реализация алгоритма Diamond-Square;
2. Реализация муравьиного алгоритма;
3. Реализация алгоритма коллективного распределения целей;
4. Исследования реализованного функционала.

Реализация осуществляется на языке Python. Исследование реализованного функционала заключается в следующем:

1. Генерация 10 различных карт для каждого размера из: 25x25, 50x50, 100x100, 250x250, 500x500, 1000x1000;
2. На каждой из карт сгенерированных генерируются наборы роботов и, соответственно, цели к ним в численности: 5, 10, 20, 50 (наборов каждого размера для каждой карты тоже должно быть по 10, но из соображений производительности этот пункт опущен);
3. Для заданных наборов распределяются цели по роботам;
4. Для каждого построенного набора строятся графики, отображающие зависимость времени выполнения программы от размеров карт и численности роботов. (В изначальном задании указано построить график содержащий все измеренные времена, но для наглядности графики строятся только средних элементов замеров, а полную картину отражают таблицы в ПРИЛОЖЕНИЕ Б);
5. Теоретическое исследование реализуемого функционала.

1 Описание алгоритмов

1.1 Алгоритм Diamond-Square

1.2 Муравьиный алгоритм

1.3 Алгоритм коллективного распределения целей

2 Программная реализация

3 Результаты

ЗАКЛЮЧЕНИЕ

ЛИТЕРАТУРА

[1] Miguel Monteiro de Sousa Frade. Genetic Terrain Programming // Universidad de Extremadura, 2008, pp. 103

[2] Каляев И.А. Модели и алгоритмы коллективного управления в группах роботов // Физматлит, 2009, 279с.

[3] Gregor Klančar. Path Planning // Wheeled Mobile Robotics, 2017, pp. 161-206

[4] Jacob Olsen. Realtime Procedural Terrain Generation // University of Southern Denmark, 2004, pp. 20

[5] M. Brand, M. Masuda, N. Wehner, X.-H. Yu. Ant colony optimization algorithm for robot path planning // Computer Design and Applications (ICCDA) 2010 International Conference on, vol. 3, 2010, pp. 436-440.

ПРИЛОЖЕНИЕ А. Исходный код

Ниже приведен исходный код на языке Python

main.py

```python
"""main file"""
import progressbar
from tools import plot_map
from tools import plot_paths
from tools import plot_heuristic_d
from tools import plot_pheromone
from tools import plot_time_correlation
from graph import Graph
from ant import EAlg
from planning import planning

EALG_OBJ = EAlg(
    50,
    10,
    1.0,
    1.0,
    0.9,
    50
)

def main_test():
    """Main function \n
        Result of this I will use for report"""
    sizes = (25, 50, 100, 250, 500, 1000)
    targets_numbers = (5, 10, 20, 50)
    prog_bar_it = [0]
    max_val = sum(targets_numbers) * len(sizes) * 10
    bar_ = [progressbar.ProgressBar(maxval=max_val).start()]
    for size in sizes:
        for targets_num in targets_numbers:
            time_file_path = "data/time/" + str(size) + "x" + str(size) + "/" + str(targets_num) + ".data"
            file = open(time_file_path, "w+")
            file.write("size: " + str(size) + "\n")
            file.write("targets_num: " + str(targets_num) + "\n")
            file.write("Map\tAntTime\tPlanTime\tFullTime" + "\n")
            ant_times = []
            plan_times = []
            full_times = []
            opt_paths = []
            graphs = []
            for map_it in range(10):
                graph = Graph(size, size, 0.3, 0.1)
                graph.generate()
                path, _, alg_time = planning(prog_bar_it, bar_, graph, EALG_OBJ, targets_num)
                opt_paths.append(path)
                graphs.append(graph)
                ant_times.append(alg_time["AntColony"])
                plan_times.append(alg_time["Planning"])
                full_times.append(alg_time["Whole"])
```

```
                file.write(str(map_it) + "\t" + str(alg_time["AntColony"]) + "\t" + str(alg_time["Planning"]) + "\t" + str(alg_time["Whole"]) + "\n")
            ant_times_sort = ant_times
            ant_times_sort.sort()
            ant_idx = ant_times.index(ant_times_sort[4])

            plan_times_sort = plan_times
            plan_times_sort.sort()
            plan_idx = plan_times.index(plan_times_sort[4])

            full_times_sort = full_times
            full_times_sort.sort()
            full_idx = full_times.index(full_times_sort[4])

            file.write("mean map by ant:" + "\t" + str(ant_times[ant_idx]) + "\t" + str(plan_times[ant_idx]) + "\t" + str(full_times[ant_idx]) + "\n")
            file.write("mean map by plan:" + "\t" + str(ant_times[plan_idx]) + "\t" + str(plan_times[plan_idx]) + "\t" + str(full_times[plan_idx]) + "\n")
            file.write("mean map by full:" + "\t" + str(ant_times[full_idx]) + "\t" + str(plan_times[full_idx]) + "\t" + str(full_times[full_idx]) + "\n")
            file.close()

            plot_file_name = "data/mean_paths/" + str(size) + "x" + str(size) + "/" + str(targets_num) + ".png"
            plot_paths(graphs[full_idx], opt_paths[full_idx], plot_file_name)

def examples_of_data():
    """Necessary for report"""
    size = 250
    graph = Graph(size, size, 0.3, 0.1)
    graph.generate()
    graph.init_pheromone_n_heuristics([50, 80])
    plot_heuristic_d(graph, "data/heuristics/heuristic_d.png")
    plot_pheromone(graph, "data/heuristics/pheromone.png")
    plot_map(graph, "data/maps/map_250.png")

def dev_test():
    """Function for development \n
        I use it for testing components"""
    size = 1000
    graph = Graph(size, size, 0.3, 0.2)
    graph.generate()

    prog_bar_it = [0]
    max_val = 50
    bar_ = [progressbar.ProgressBar(maxval=max_val).start()]
    opt_paths, _, alg_time = planning(prog_bar_it, bar_, graph, EALG_OBJ, 50)
    print(alg_time)
    plot_paths(graph, opt_paths, "data/mean_paths/test.png")

```

```python
def time_correlation():
    """plot surface with mean times by ready data"""
    sizes = (25, 50, 100, 250, 500, 1000)
    targets_numbers = (5, 10, 20, 50)
    plot_time_correlation(sizes, targets_numbers)

# examples_of_data()
# dev_test()
main_test()
# time_correlation()
```

graph.py

```python
"""Graph class"""
import math
import numpy as np
from tools import get_conj
from tools import get_distance_proj
from tools import get_distance
from tools import get_mean

class Graph:
    """class for diamond square algorithm"""
    def __init__(self, n, m, R, pheromone_eur_par):
        self.n = n
        self.m = m
        self.max_element = pow(2, math.ceil(math.log(max(n, m) - 1, 2)))
        self.matrix = np.zeros((self.max_element + 1, self.max_element + 1))
        self.norm_matrix = np.zeros((self.max_element + 1, self.max_element + 1))
        self.height = (n + m) / 2
        self.pheromone_eur_par = pheromone_eur_par
        self.max_dist_z = 0.0
        self.max_dist_x_y = get_distance_proj([0, 0], [self.n - 1, self.m - 1])
        self.available_moves = [[float(0) for x in range(n)] for y in range(m)]
        self.heuristic_h = [[float(0) for x in range(n)] for y in range(m)]
        self.costs = [[float(0) for x in range(n)] for y in range(m)]
        self.heuristic_d = np.zeros((n, m))
        self.pheromone = np.ones((n, m))
        self.R = R
        self.first_call = True

    def generate(self):
        """main method"""
        self.matrix[0][0] = np.random.uniform(low=0, high=self.height)
        self.matrix[self.max_element][self.max_element] = np.random.uniform(low=0, high=self.height)
        self.matrix[self.max_element][0] = np.random.uniform(low=0, high=self.height)
        self.matrix[0][self.max_element] = np.random.uniform(low=0, high=self.height)

```

```python
        side_length = self.max_element
        while side_length != 1:
            x_1 = 0
            y_1 = 0
            x_2 = side_length
            y_2 = side_length
            while True:
                self.square(x_1, y_1, x_2, y_2)
                self.diamond(x_1, y_1, x_1, y_2)
                self.diamond(x_1, y_2, x_2, y_2)
                self.diamond(x_2, y_2, x_2, y_1)
                self.diamond(x_2, y_1, x_1, y_1)
                if y_2 == self.max_element:
                    if x_2 == self.max_element:
                        break
                    else:
                        x_1 += side_length
                        x_2 += side_length
                        y_1 = 0
                        y_2 = side_length
                else:
                    y_1 += side_length
                    y_2 += side_length
            side_length = int(side_length / 2)
        self.matrix = self.matrix[0:self.n, 0:self.m]
        self.matrix = np.around(self.matrix, decimals=3)
        self.max_dist_z = np.amax(self.matrix) - np.amin(self.matrix)
        self.norm_matrix = self.matrix - np.amin(self.matrix)
        self.norm_matrix = self.norm_matrix / self.max_dist_z

    def square(self, x_1, y_1, x_2, y_2):
        """square step of algorithm"""
        rad = (x_2 - x_1) / 2
        center_x = int(x_1 + rad)
        center_y = int(y_1 + rad)
        vertexes = [self.matrix[x_1][y_1],
                    self.matrix[x_2][y_2],
                    self.matrix[x_1][y_2],
                    self.matrix[x_2][y_1]]
        self.matrix[center_x][center_y] = (get_mean(vertexes)) + np.random.uniform(low=(- self.R * rad * 2), high=(self.R * rad * 2))

    def diamond(self, x_1, y_1, x_2, y_2):
        """diamond step of algorithm"""
        vertexes = []
        x = 0
        y = 0
        rad = 0.0
        if x_1 == x_2:
            center_y = int((y_1 + y_2) / 2)
            rad = abs(y_2 - center_y)
            if x_1 not in (0, self.max_element):
                vertexes += [self.matrix[x_1][y_1],
                             self.matrix[x_2][y_2],
                             self.matrix[x_1 - rad][center_y],
                             self.matrix[x_1 + rad][center_y]]
```

```
                else:
                    if x_1 == 0:
                        vertexes += [self.matrix[x_1][y_1],
                                     self.matrix[x_2][y_2],
                                     self.matrix[x_1 + rad][center_y]]
                    if x_1 == self.max_element:
                        vertexes += [self.matrix[x_1][y_1],
                                     self.matrix[x_2][y_2],
                                     self.matrix[x_1 - rad][center_y]]
                x = x_1
                y = center_y
            else:
                center_x = int((x_1 + x_2) / 2)
                rad = abs(x_2 - center_x)
                if y_1 not in (0, self.max_element):
                    vertexes += [self.matrix[x_1][y_1],
                                 self.matrix[x_2][y_2],
                                 self.matrix[center_x][y_1 - rad],
                                 self.matrix[center_x][y_1 + rad]]
                else:
                    if y_1 == 0:
                        vertexes += [self.matrix[x_1][y_1],
                                     self.matrix[x_2][y_2],
                                     self.matrix[center_x][y_1 + rad]]
                    if y_1 == self.max_element:
                        vertexes += [self.matrix[x_1][y_1],
                                     self.matrix[x_2][y_2],
                                     self.matrix[center_x][y_1 - rad]]
                x = center_x
                y = y_1
            self.matrix[x][y] = get_mean(vertexes) + np.random.uniform(low=(-self.R * rad * 2), high=(self.R * rad * 2))

    def get_size(self):
        """Get size of graph in format (,)"""
        return (self.n, self.m)

    def init_pheromone_n_heuristics(self, end_point):
        """Init pheromone matrix, heuristic_d matrix, heuristic_h matrix of distances to available \n
            moves by z and available_moves matrix of lists"""
        if self.first_call:
            for it in range(self.n):
                for jt in range(self.m):
                    dist_to_conj = []
                    conj_points = get_conj(self.norm_matrix, [it, jt])
                    for point in conj_points:
                        dist_to_conj.append(get_distance([it, jt], point, self.matrix))
                    self.costs[it][jt] = dist_to_conj
                    self.available_moves[it][jt] = conj_points
            self.first_call = False
        else:
            self.heuristic_h = [[float(0) for x in range(self.n)] for y in range(self.m)]
            self.heuristic_d = np.zeros((self.n, self.m))
```

```
            self.pheromone = np.ones((self.n, self.m))

        end_point_val = max(self.n, self.m)
        for it in range(self.n):
            for jt in range(self.m):
                if not (it == end_point[0] and jt == end_point[1]):
                    pheromone = 1 - (get_distance_proj([it, jt], end_point) / self.max_dist_x_y + np.random.uniform(- self.pheromone_eur_par, self.pheromone_eur_par))
                    if pheromone < 0:
                        self.pheromone[it][jt] = 0.0
                    else:
                        self.pheromone[it][jt] = pheromone
                z_dist_to_conj = []
                conj_points = get_conj(self.norm_matrix, [it, jt])
                for point in conj_points:
                    z_dist_to_conj.append(1 - abs(self.norm_matrix[point[0]][point[1]] - self.norm_matrix[it][jt]))
                self.heuristic_h[it][jt] = z_dist_to_conj
                self.heuristic_d[it][jt] = end_point_val - max(abs(it - end_point[0]), abs(jt - end_point[1]))

    def update_pheromone(self, pheromone_increment: np.array, evaporation_coef):
        """Updates pheromone values"""
        self.pheromone *= (1 - evaporation_coef)
        self.pheromone += pheromone_increment

    def get_pheromone(self, position):
        """Returns pheromone value for position"""
        return self.pheromone[position[0]][position[1]]

    def get_heuristic_h(self, position):
        """Returns list of distance by z to possible moves for ant"""
        return self.heuristic_h[position[0]][position[1]]

    def get_available_moves(self, position):
        """Returns list of possible moves for ant"""
        return self.available_moves[position[0]][position[1]]

    def get_heuristic_d(self, position):
        """Returns heuristic_d value for position"""
        return self.heuristic_d[position[0]][position[1]]

    def get_cost(self, position):
        """Returns list costs for conjugate positions"""
        return self.costs[position[0]][position[1]]

    def get_pos_parameters(self, position):
        """Returns parameters for possibility calculating \n
        return available_moves, pheromones, heuristic_d, self.get_heuristic_h(position)"""
        available_moves = self.get_available_moves(position)
        heuristic_d = []
        pheromones = []
        for move in available_moves:
```

```python
            heuristic_d.append(self.get_heuristic_d(move))
            pheromones.append(self.get_pheromone(move))
        min_h_d = min(heuristic_d)
        heuristic_d = [val - min_h_d for val in heuristic_d]
        sum_d = sum([math.exp(w_d) for w_d in heuristic_d])
        heuristic_d = [math.exp(w_d) / sum_d for w_d in heuristic_d]
        return available_moves, pheromones, heuristic_d, self.get_heuristic_h(position), self.get_cost(position)

    def get_matrix(self):
        """Returns surface in matrix formats"""
        return self.matrix
```

ant.py

```python
"""Evolution algorithm implementation"""
import numpy as np
from graph import Graph
from tools import choice

class Ant:
    """Single ant behavior"""
    def __init__(self, graph: Graph, start, alpha, beta, q):
        self.graph = graph
        self.position = start
        self.alpha = alpha
        self.beta = beta
        self.path = [start]
        self.path_length = 0.0
        self.last_cost = 0.0
        self.q = q
        self.increase = [0.0]
        self.iteration = 0
        self.fail = False

    def get_pos(self):
        """Get ant's position"""
        return self.position

    def move(self):
        """Move ant in next graph's point"""
        available_moves, moves_pheromones, moves_heuristic_d, moves_heuristic_h, moves_costs = self.graph.get_pos_parameters(self.position)
        weights = []
        sum_w = 0.0
        for it in range(len(available_moves)):
            weight = moves_pheromones[it] ** self.alpha * moves_heuristic_d[it] ** self.beta * moves_heuristic_h[it]
            sum_w += weight
            weights.append(weight)
        weights = [w / sum_w for w in weights]
        choosen_idx = choice(weights)
        self.position = available_moves[choosen_idx]
        self.path.append(self.position)
```

```
        # if moves_costs[choosen_idx] == 0.0: # was Loch Ness bug and this is for safety
        #     moves_costs[choosen_idx] += 0.01
        self.increase.append(moves_costs[choosen_idx])
        self.path_length += moves_costs[choosen_idx]
        self.iteration += 1

    def get_position(self):
        """Returns position of ant"""
        return self.position

    def get_pheromone_increase(self, idx):
        """Return pheromone increase for idx move"""
        return self.q / self.increase[idx]

    def get_path_length(self):
        """Returns path length"""
        return self.path_length

    def get_path(self):
        """Returns path"""
        return self.path

    def delete_loops(self):
        """Delete loops from path"""
        for it in self.path:
            if self.path.count(it) > 1:
                idx = self.path.index(it)
                for jt in range(idx, len(self.path) - 1 - self.path[::-1].index(it)): #last idx
                    self.path.pop(idx)
                    self.path_length -= self.increase[idx]
                    self.increase.pop(idx)

class EAlg:
    """Evolution algorithm"""
    def __init__(self, pop_size, iter_size, alpha, beta, rho, q):
        self.pop_size = pop_size
        self.iter_size = iter_size
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.q = q

    def get_path(self, graph: Graph, start: [], end_point: []):
        """Main method of algorithm, which find best path\n
        It returns cost and path
        """
        path = []
        path_length = float('inf')
        graph.init_pheromone_n_heuristics(end_point)
        lim = 0
        if (graph.get_size()[0] + graph.get_size()[1]) / 2 < 100:
            lim = 10000
        else: lim = graph.get_size()[0] * graph.get_size()[1] / 10
        for it in range(self.iter_size):
```

```python
            pheromone_increment = np.zeros(graph.get_size())
            for ant_it in range(self.pop_size):
                ant = Ant(graph, start, self.alpha, self.beta, self.q)
                while ant.get_pos() != end_point:
                    ant.move()
                    if ant.iteration == lim:
                        ant.fail = True
                        break
                    pos = ant.get_pos()
                    pheromone_increment[pos[0]][pos[1]] += ant.get_pheromone_increase(len(ant.get_path()) - 1)
                if not ant.fail:
                    ant.delete_loops()
                    if ant.get_path_length() < path_length:
                        path = ant.get_path()
                        path_length = ant.get_path_length()
            if not ant.fail:
                graph.update_pheromone(pheromone_increment, self.rho)

        return path, path_length
```

planning.py

```python
"""Planning algorithm"""
from time import time
import numpy as np
from graph import Graph
from ant import EAlg
from tools import get_distance_proj

def planning(prog_bar_it, bar_, graph: Graph, model: EAlg, number_of_targets):
    """Returns list with robot's paths to targets and list of lenghts this paths"""
    robots = []
    targets = []
    x_max, y_max = graph.get_size()

    for it in range(number_of_targets):
        while True:
            robot = [np.random.randint(low=0, high=x_max),
                     np.random.randint(low=0, high=y_max)]
            if robot not in robots:
                if robot not in targets:
                    robots.append(robot)
                    break
        while True:
            target = [np.random.randint(low=0, high=x_max),
                      np.random.randint(low=0, high=y_max)]
            if target not in robots:
                if target not in targets:
                    targets.append(target)
                    break

    costs = [[0.0 for x in range(number_of_targets)] for y in range(number_of_targets)]
```

```python
    for it in range(number_of_targets):
        for jt in range(number_of_targets):
            costs[it][jt] = get_distance_proj(robots[it], targets[jt])

    time_dic = {}

    plan_start = time()

    opt_paths = []
    opt_costs = []
    opt_pairs = []

    have_pair = np.zeros(number_of_targets)
    while not all(have_pair):
        it = 0
        while True:
            if not have_pair[it]:
                idx = costs[it].index(min(costs[it]))
                cost = costs[it][idx]
                if cost == min([row[idx] for row in costs]):
                    opt_pairs.append([it, idx])
                    have_pair[it] = True
                    costs[it] = [float('inf') for it in range(number_of_targets)]
                    for jt in range(number_of_targets):
                        costs[jt][idx] = float('inf')
                    break
                else: it += 1
            else: it += 1

    ant_start = time()
    time_dic["Planning"] = round(ant_start - plan_start, 3)

    for pair in opt_pairs:
        prog_bar_it[0] += 1
        bar_[0].update(prog_bar_it[0])
        path, cost = model.get_path(graph, robots[pair[0]], targets[pair[1]])
        opt_paths.append(path)
        opt_costs.append(cost)



    time_dic["AntColony"] = time() - ant_start
    time_dic["Whole"] = round(time_dic["AntColony"] + time_dic["Planning"], 3)

    return opt_paths, opt_costs, time_dic
```

tools.py

```python
"""usefull functions"""
import math
import bisect
import numpy as np
import matplotlib.pyplot as plt
```

```python
from matplotlib import rcParams
from mpl_toolkits.mplot3d import Axes3D

def plot_surface(matrix, sizes, targets_numbers, file_name):
    """Plot 3d surface"""
    rcParams.update({'font.size': 16})
    (x, y) = np.meshgrid(np.arange(matrix.shape[1]), np.arange(matrix.shape[0]))
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    surf = ax.plot_surface(x, y, np.log(matrix), cmap=plt.get_cmap("viridis"))
    ax.set_xlabel('Targets', labelpad=20)
    ax.set_ylabel('Map size', labelpad=20)
    ax.set_zlabel('ln(t)', labelpad=10)
    plt.xticks(range(len(targets_numbers)), targets_numbers)
    plt.yticks(range(len(sizes)), sizes)
    fig.colorbar(surf)
    fig.set_size_inches(12.5, 8.5)
    fig.savefig(file_name, dpi=100)
    plt.close(fig)

def plot_time_correlation(sizes, targets_numbers):
    """tool for plot surface from time data"""
    matrix = np.zeros((len(sizes), len(targets_numbers)))
    for it, _ in enumerate(sizes):
        root = "data/time/" + str(sizes[it]) + "x" + str(sizes[it]) + "/"
        for jt, _ in enumerate(targets_numbers):
            file_path = root + str(targets_numbers[jt]) + ".data"
            file = open(file_path)
            mean_time = float(file.readlines()[15].rstrip().rsplit("\t")[3])
            matrix[it][jt] = round(mean_time, 3)
    plot_surface(matrix, sizes, targets_numbers, "data/time/mean_surface.png")

def plot_map(graph, file_name):
    """Plot map in heatmap format"""
    plot_heatmap(graph.get_matrix(), file_name)

def plot_paths(graph, paths, file_name):
    """Plot path on map"""
    fig = plt.figure()
    ax = fig.add_subplot(111)
    pl = ax.imshow(graph.get_matrix(), cmap=plt.get_cmap("gist_earth"))
    fig.colorbar(pl)
    fig.set_size_inches(8.5, 8.5)
    for path in paths:
        ax.plot([x for x, y in path], [y for x, y in path], linewidth=2.0, c="orange")
        ax.plot(path[0][0], path[0][1], "ro", c="black")
        ax.plot(path[len(path) - 1][0], path[len(path) - 1][1], "ro", c="red")
    fig.savefig(file_name, dpi=100)
    plt.close(fig)

```

```
def plot_heuristic_d(graph, file_name):
    """Plot heuristic by distance in heatmap format"""
    plot_heatmap(graph.heuristic_d, file_name)

def plot_pheromone(graph, file_name):
    """Plot pheromone heatmap"""
    plot_heatmap(graph.pheromone, file_name)

def plot_heatmap(matrix, file_name):
    """Plot 2d heat map"""
    fig = plt.figure()
    ax = plt.imshow(matrix, cmap=plt.get_cmap("gist_earth"))
    fig.colorbar(ax)
    fig.set_size_inches(8.5, 8.5)
    fig.savefig(file_name, dpi=100)
    plt.close(fig)

def cdf(weights):
    """generate weights"""
    total = sum(weights)
    result = []
    cumsum = 0
    for w in weights:
        cumsum += w
        result.append(cumsum / total)
    return result

def choice(weights):
    """choice with prob"""
    cdf_vals = cdf(weights)
    x = np.random.uniform(low=0.0, high=1.0)
    idx = bisect.bisect(cdf_vals, x)
    return idx

def get_conj(matrix, point):
    """returns conjugate points for point in matrix"""
    conj_points = []
    top_left = True
    top_right = True
    bottom_left = True
    bottom_right = True
    if point[0] > 0:
        conj_points.append([point[0] - 1, point[1]])
    else:
        top_left = False
        top_right = False
    if point[0] < matrix.shape[0] - 1:
        conj_points.append([point[0] + 1, point[1]])
    else:
        bottom_left = False
        bottom_right = False
    if point[1] > 0:
        conj_points.append([point[0], point[1] - 1])
    else:
        bottom_left = False
        top_left = False
```

```
    if point[1] < matrix.shape[1] - 1:
        conj_points.append([point[0], point[1] + 1])
    else:
        top_right = False
        bottom_right = False

    if top_left:
        conj_points.append([point[0] - 1, point[1] - 1])
    if top_right:
        conj_points.append([point[0] - 1, point[1] + 1])
    if bottom_left:
        conj_points.append([point[0] + 1, point[1] - 1])
    if bottom_right:
        conj_points.append([point[0] + 1, point[1] + 1])

    return conj_points

def get_distance_proj(x, y):
    """distance between two point by x and y using Euclid metric"""
    return math.sqrt((x[0] - y[0]) ** 2 + (x[1] - y[1]) ** 2)

def get_distance(x, y, matrix):
    """distance between two point by x, y and z using Euclid metric"""
    return math.sqrt((x[0] - y[0]) ** 2 + (x[1] - y[1]) ** 2 + (matrix[x
    ↪ [0]][x[1]] - matrix[y[0]][y[1]]) ** 2)

def get_mean(some_list):
    """Returns mean value of list"""
    return sum(some_list) / len(some_list)
```

ПРИЛОЖЕНИЕ Б. Таблицы замеров времени

Ниже приведены замеры времени (с.) муравьиного алгоритма для каждой сгенерированной карты и каждого количества роботов:

№ карты\Кол-во роботов	**5**	**10**	**20**	**50**
1	1.5568	1.69588	2.75487	4.2076
2	1.69226	1.80428	3.21331	4.2441
3	1.09952	1.34256	2.66641	6.43867
4	0.96775	3.23578	2.67067	4.53471
5	0.80875	1.89084	3.0447	5.53788
6	1.1738	2.62321	2.6756	5.01334
7	1.40806	1.95556	2.0411	6.20485
8	1.08748	2.11052	2.38643	5.60215
9	1.26882	1.84033	2.1164	5.56547
10	0.73943	1.60901	3.37735	5.15904
Средний элемент	1.09952	1.84033	2.67067	5.15904

Размер карты: 25x25

№ карты\Кол-во роботов	**5**	**10**	**20**	**50**
1	2.15365	3.59408	6.54677	10.73975
2	1.71077	3.3017	6.41902	11.11771
3	3.33052	2.96686	7.60223	12.24219
4	2.48163	4.87254	9.19686	12.41132
5	3.9287	3.95599	6.41092	11.49145
6	1.6465	4.43041	7.68628	13.13047
7	2.31836	4.70432	5.39536	13.06448
8	1.93215	2.44238	5.48949	12.79632
9	2.99847	2.71091	7.70761	15.13225
10	2.68102	4.48696	6.36677	15.12783
Средний элемент	2.31836	3.59408	6.41902	12.41132

Размер карты: 50x50

№ карты\Кол-во роботов	**5**	**10**	**20**	**50**
1	6.32895	8.73452	14.94095	24.71504
2	5.26532	9.12678	13.68367	30.10719
3	4.87571	12.12137	14.66411	26.09427
4	5.98595	7.40209	12.80597	23.22383
5	3.94488	10.1101	17.56255	28.4082
6	7.12389	8.61739	13.43984	25.22285
7	5.36162	8.51896	13.89583	25.27979
8	5.02578	9.56457	11.74729	23.15173
9	5.34449	13.91983	14.71402	22.08825
10	6.81967	9.45882	17.0239	31.06226
Средний элемент	5.34449	9.12678	13.89583	25.22285

Размер карты: 100x100

№ карты\Кол-во роботов	**5**	**10**	**20**	**50**
1	42.28026	32.36046	52.14858	107.19252
2	19.01642	27.00166	69.75315	103.9241
3	20.36057	30.91856	45.16012	103.12069
4	17.64865	38.52182	52.3477	115.31906
5	19.1544	39.6922	60.39992	123.05207
6	25.56097	25.46197	49.70063	121.08852
7	18.23112	34.0119	58.34534	93.98898
8	13.38648	34.57604	51.70425	105.00388
9	17.92189	35.83925	48.98953	98.69694
10	23.87057	26.68619	54.96598	103.89223
Средний элемент	19.01642	32.36046	52.14858	103.9241

Размер карты: 250x250

№ карты\Кол-во роботов	**5**	**10**	**20**	**50**
1	47.88517	73.60561	137.27325	308.64456
2	48.11748	105.09704	167.23236	321.9515
3	64.0862	109.30441	143.13767	335.52867
4	59.32068	187.01288	168.12613	364.50322
5	75.32691	124.74065	178.59087	319.94925
6	61.19254	79.99268	128.15281	503.20078
7	51.79195	87.23905	198.67009	341.6798
8	58.80499	87.76271	282.89796	311.80583
9	56.37451	80.41181	277.18092	283.16109
10	56.46719	103.25168	243.64185	409.125
Средний элемент	56.46719	87.76271	168.12613	321.9515

Размер карты: 500x500

№ карты\Кол-во роботов	**5**	**10**	**20**	**50**
1	985.46655	292.70027	847.49595	2510.47386
2	289.54574	551.28631	879.1982	1478.33745
3	775.55588	516.55046	493.11726	2206.79318
4	612.46158	1196.72224	596.36857	2167.35525
5	356.07551	626.02555	757.65001	2049.00236
6	258.13347	1190.34369	1356.15285	2417.78241
7	1100.10644	554.87698	1412.94535	2335.37329
8	715.70122	372.30212	1571.87906	4075.23897
9	905.14539	777.73305	2050.20857	4158.50932
10	170.51577	1581.04043	2182.6365	4512.15843
Средний элемент	612.46158	554.87698	879.1982	2335.37329

Размер карты: 1000x1000

Ниже приведены замеры времени (с.) алгоритма планирования для каждой сгенерированной карты и каждого количества роботов:

№ карты\Кол-во роботов	**5**	**10**	**20**	**50**
1	0.0	0.0	0.0	0.004
2	0.0	0.0	0.001	0.004
3	0.0	0.0	0.0	0.005
4	0.0	0.0	0.001	0.003
5	0.0	0.0	0.0	0.004
6	0.0	0.0	0.0	0.004
7	0.0	0.0	0.0	0.004
8	0.0	0.0	0.0	0.004
9	0.0	0.0	0.0	0.004
10	0.0	0.0	0.0	0.005
Средний элемент	0.0	0.0	0.0	0.004

Размер карты: 25x25

№ карты\Кол-во роботов	**5**	**10**	**20**	**50**
1	0.0	0.0	0.001	0.005
2	0.0	0.0	0.0	0.003
3	0.0	0.0	0.0	0.004
4	0.0	0.0	0.0	0.004
5	0.0	0.0	0.001	0.003
6	0.0	0.0	0.001	0.004
7	0.0	0.0	0.0	0.004
8	0.0	0.0	0.0	0.003
9	0.0	0.0	0.0	0.004
10	0.0	0.0	0.0	0.004
Средний элемент	0.0	0.0	0.0	0.004

Размер карты: 50x50

№ карты\Кол-во роботов	**5**	**10**	**20**	**50**
1	0.0	0.0	0.001	0.003
2	0.0	0.0	0.0	0.003
3	0.0	0.0	0.0	0.003
4	0.0	0.0	0.0	0.003
5	0.0	0.0	0.0	0.004
6	0.0	0.0	0.0	0.004
7	0.0	0.0	0.0	0.003
8	0.0	0.0	0.0	0.003
9	0.0	0.0	0.0	0.004
10	0.0	0.0	0.0	0.003
Средний элемент	0.0	0.0	0.0	0.003

Размер карты: 100x100

№ карты\Кол-во роботов	**5**	**10**	**20**	**50**
1	0.0	0.0	0.0	0.003
2	0.0	0.0	0.0	0.004
3	0.0	0.0	0.0	0.004
4	0.0	0.0	0.0	0.004
5	0.0	0.0	0.0	0.004
6	0.0	0.0	0.0	0.004
7	0.0	0.0	0.0	0.003
8	0.0	0.0	0.0	0.003
9	0.0	0.0	0.0	0.003
10	0.0	0.0	0.0	0.004
Средний элемент	0.0	0.0	0.0	0.004

Размер карты: 250x250

№ карты\Кол-во роботов	**5**	**10**	**20**	**50**
1	0.0	0.0	0.0	0.003
2	0.0	0.0	0.0	0.004
3	0.0	0.0	0.0	0.004
4	0.0	0.0	0.001	0.003
5	0.0	0.0	0.001	0.003
6	0.0	0.0	0.0	0.003
7	0.0	0.0	0.0	0.004
8	0.0	0.0	0.0	0.003
9	0.0	0.0	0.001	0.003
10	0.0	0.0	0.0	0.003
Средний элемент	0.0	0.0	0.0	0.003

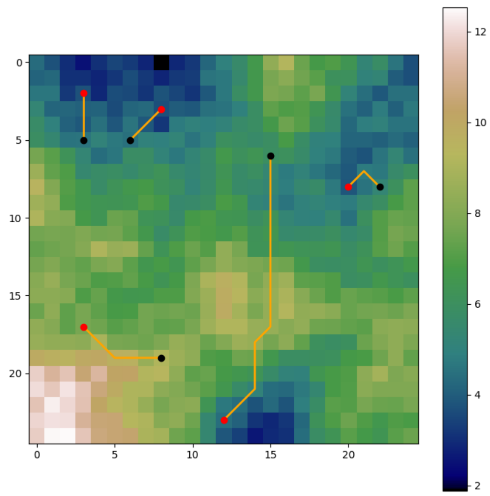
Размер карты: 500x500

№ карты\Кол-во роботов	**5**	**10**	**20**	**50**
1	0.0	0.0	0.003	0.005
2	0.0	0.0	0.0	0.004
3	0.0	0.0	0.001	0.004
4	0.0	0.0	0.0	0.003
5	0.0	0.0	0.001	0.004
6	0.0	0.0	0.001	0.004
7	0.0	0.0	0.0	0.004
8	0.0	0.0	0.001	0.004
9	0.0	0.0	0.001	0.004
10	0.0	0.0	0.002	0.005
Средний элемент	0.0	0.0	0.001	0.004

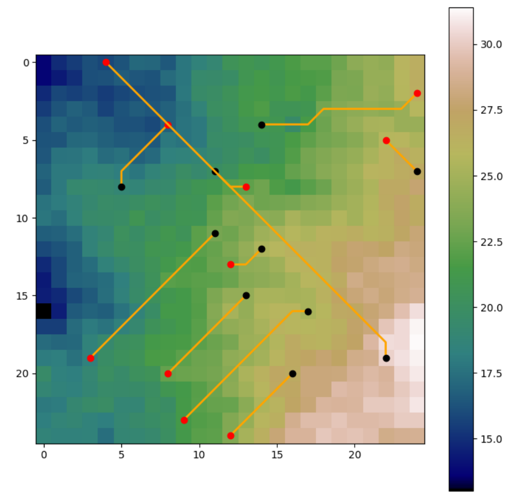
Размер карты: 1000x1000

ПРИЛОЖЕНИЕ В. Графики решений

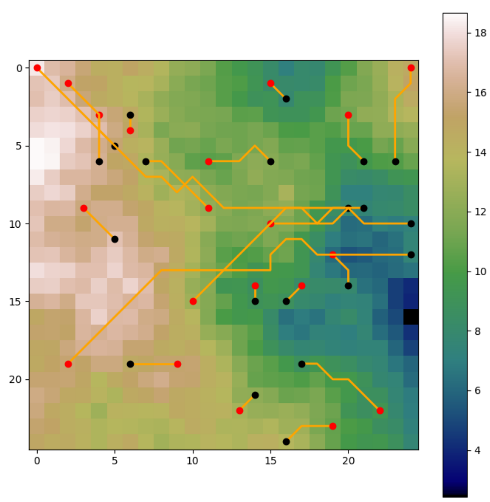
Ниже представлены построенные пути с средним временем выполнения для разного числа роботов и разных размеров матриц (черным отмечены роботы, красным - цели):



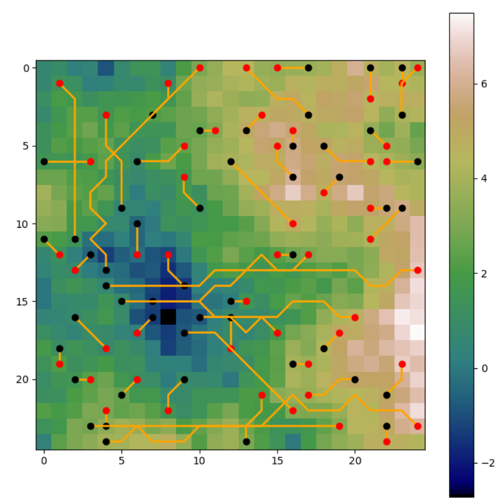
5 роботов



10 роботов

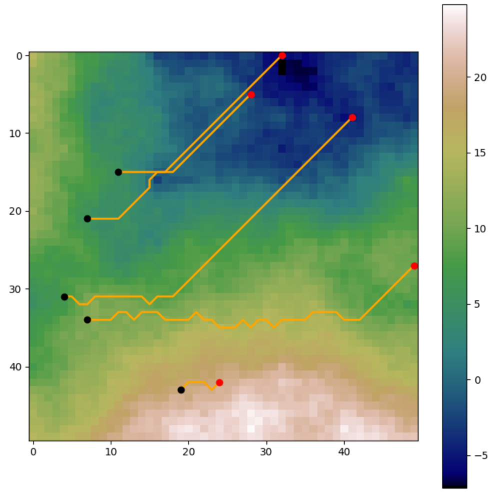


20 роботов

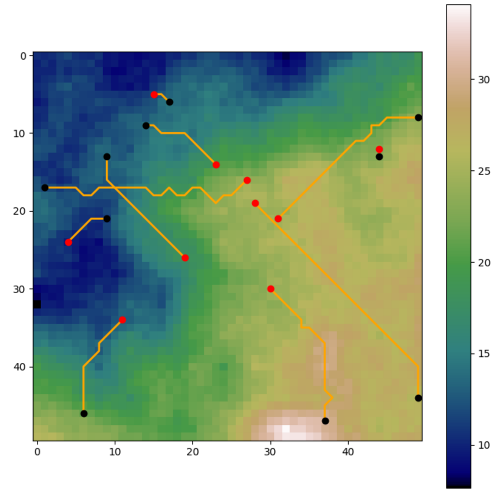


50 роботов

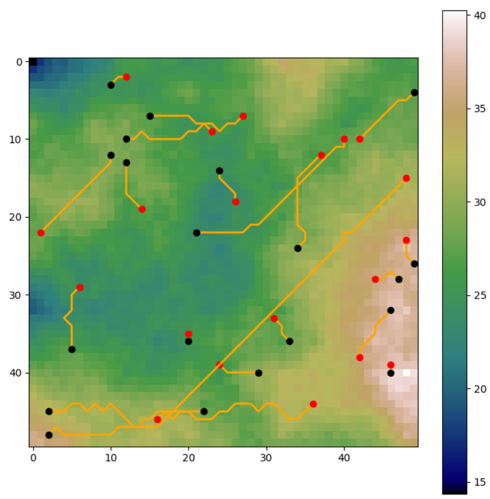
Размер карты: 25x25



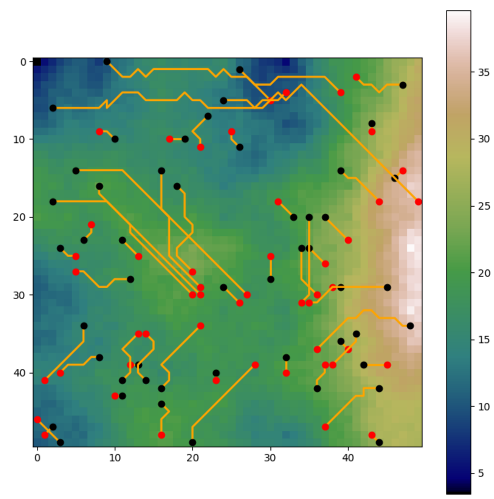
5 роботов



10 роботов

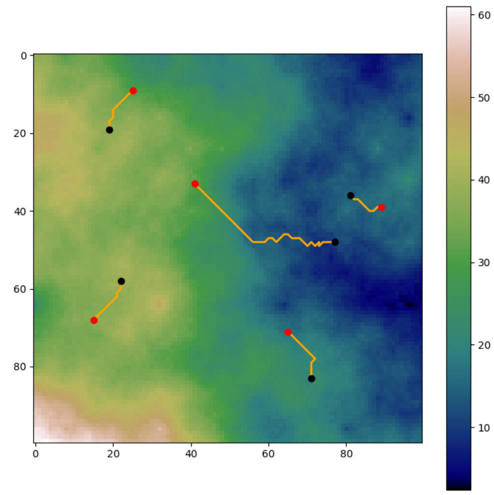


20 роботов

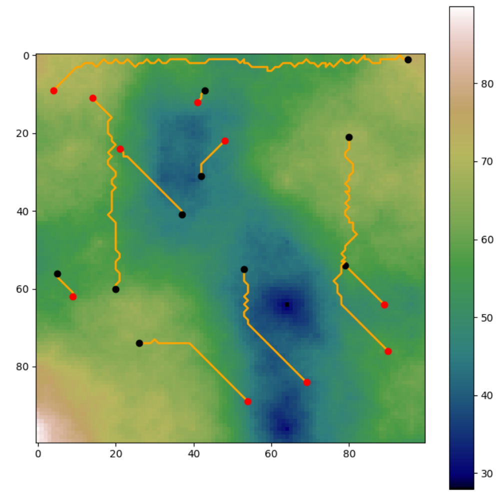


50 роботов

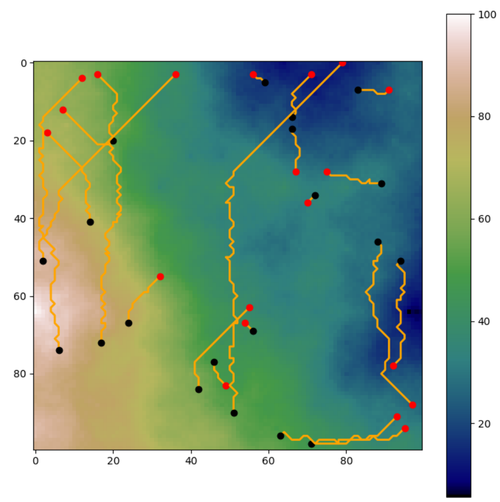
Размер карты: 50x50



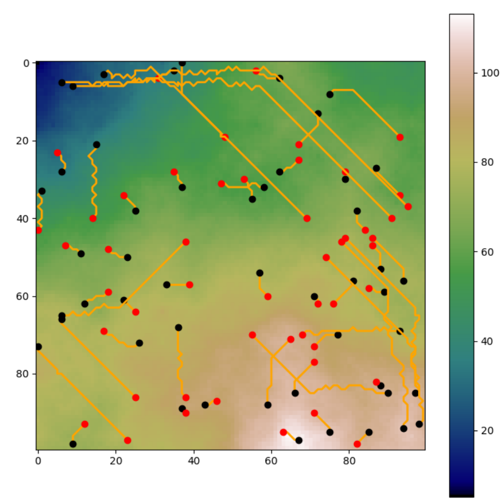
5 роботов



10 роботов

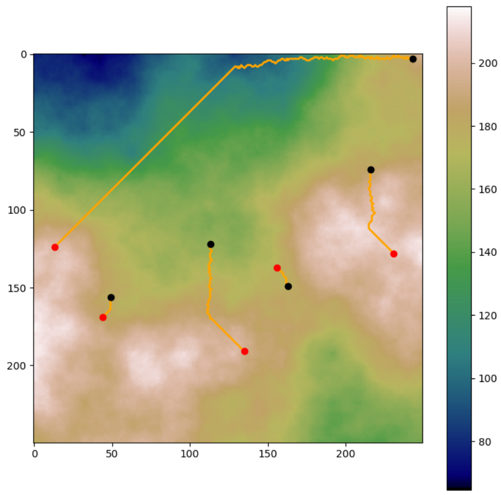


20 роботов

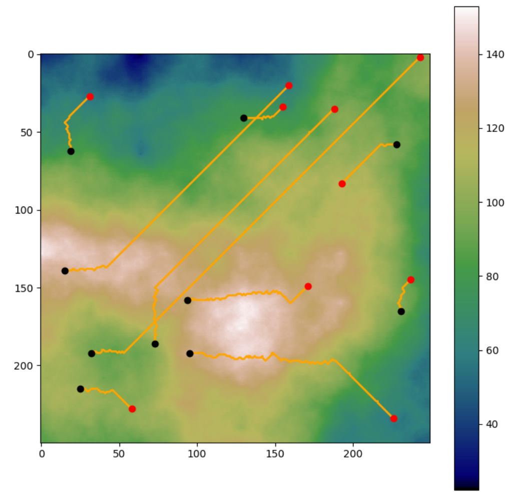


50 роботов

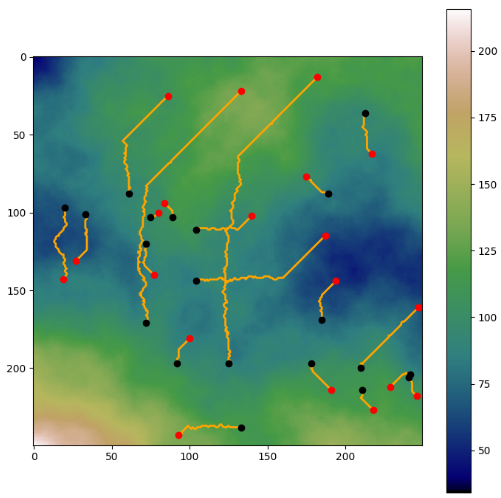
Размер карты: 100x100



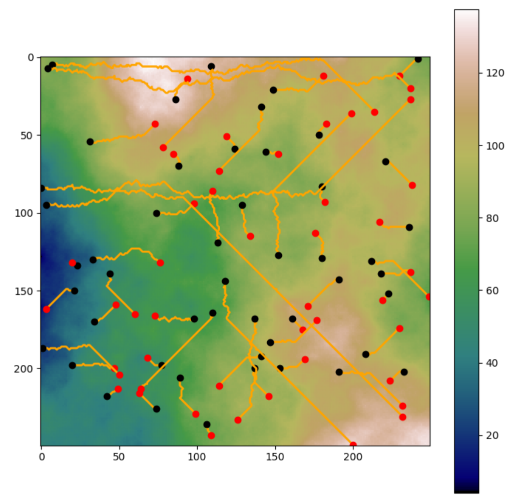
5 роботов



10 роботов

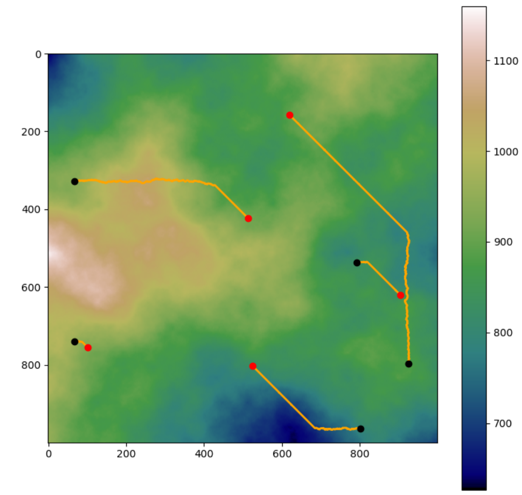


20 роботов

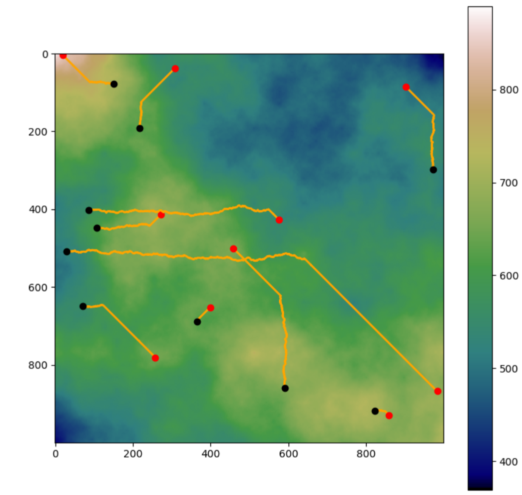


50 роботов

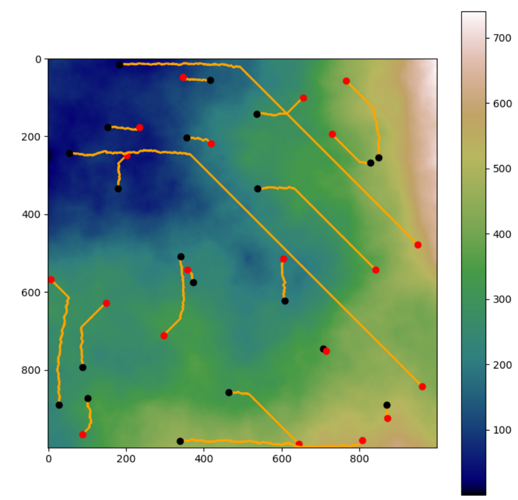
Размер карты: 250x250



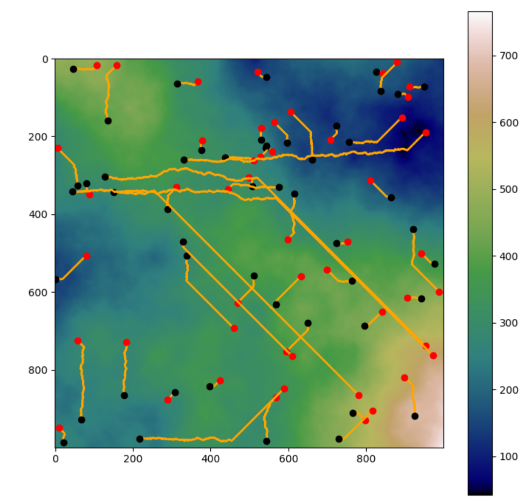
5 роботов



10 роботов



20 роботов



50 роботов

Размер карты: 1000x1000