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: Моторин Д.Е.

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 сгенерированных карт создаются 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 для средних элементов измерений времени, а полную картину отражают таблицы в ПРИЛОЖЕНИЕ Б);
5. Теоретическое исследование реализуемого функционала.

1 Описание алгоритмов

В данной главе представлен разбор теоретических принципов работы реализуемых алгоритмов.

1.1 Алгоритм Diamond-Square

В данной работе для процедурной генерации реалистичной карты местности используется алгоритм Diamond-Square[4]. Прочие применяемые алгоритмы описаны в [1].

Алгоритм имеет ограничения на размеры получаемой карты высот - $(2^n+1)\times(2^n+1)$.

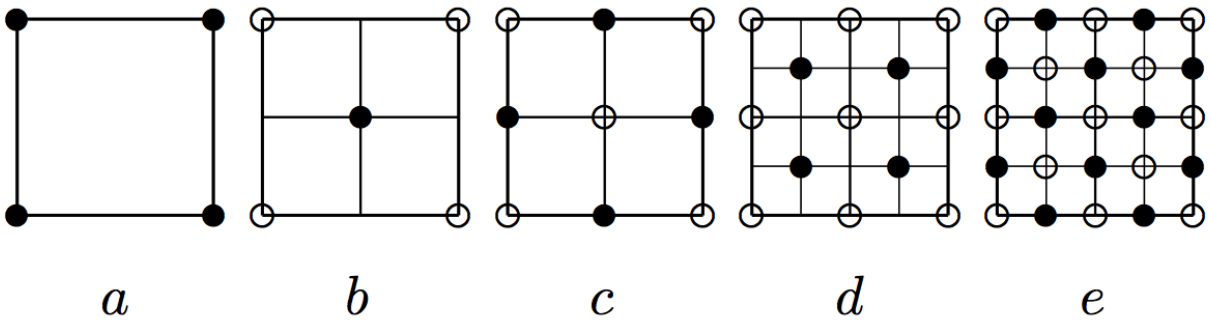


Рис. 1: Этапы работы алгоритма Diamond-Square

Алгоритм включает в себя следующие этапы (отображены на Рис. 1):

a) **Инициализация** Пусть n и m - заданные ширина и высота требуемой карты высот. Чтобы использовать алгоритм для заданного размера требуется создать карту с шириной и высотой равными:

$$n' = 2^{\log_2(max(n,m)-1)} + 1$$

Для каждого углового элемента полученной карты генерируем случайное значение. Диапазон случайных значений в общем случае может быть любым, но в данной работе возьмем его равным $[0, \frac{n+m}{2}]$. В целом данный диапазон не влияет на производительность алгоритма или на возможность его реализации, он может быть и отрицательным, как, например в случае, когда 0 - это уровень моря.

b) **Этап Diamond** На этом этапе находится центральная точка рассматриваемого квадрата, как среднее значение от его углов:

$$m[s/2][s/2] = \frac{m[0][0] + m[0][s] + m[s][0] + m[s][s]}{4} + rand(-R * s, R * s)$$

где m - это рассматриваемый квадрат, s - количество узлов на ребре квадрата, R - задаваемый коэффициент.

c) **Этап Square** Здесь рассматривается ромб с вершинами, вычисленными на этапе Diamond. Подход схожий с этапом Diamond, только теперь высчитываются центральные элементы ребер квадрата по двум углам квадрата и двум центральным точкам,

одна из которых - центр рассматриваемого квадрата, а вторая центр соседствующего квадрата. Существует проблема на краях генерируемой карты, ведь в таком случае нет центра соседствующего квадрата. В данной работе было принято следующее решение: при отсутствии соседствующего квадрата брать среднее от 3-ех точек - двух вершин и центра рассматриваемого квадрата.

d-e) **Итерации по рассматриваемым квадратам** Итерации проходят по строкам и столбцам с шагом s:
В каждой итерации по столбцам происходят этапы Square и Diamond, если итерация упирается в край карты, то происходит переход на следующую строку. При достижении правого нижнего угла матрицы значение s обновляется $s = s/2$ и итерации по строкам и столбцам происходят заново.
Критерий останова: $s = 2$.

После выполнения приведенных выше этапов построенная карта обрезается до размера $n \times m$. Для удобства дальнейшей работы муравьиного алгоритма из построенной карты генерируется нормированная карта.

1.2 Муравьиный алгоритм

Для построения путей роботов до целей был выбран муравьиный алгоритм [5]. Алгоритм имеет хорошую практическую применимость в такого рода задачах [5][6][7]. Существуют и прочие алгоритмы, которые весьма подробно описаны в [3].

Основная идея алгоритма заключается в следующем:
Запустить на карту некоторую популяцию муравьев, которые должны дойти от начальной точки до конечной, при том каждую вершину пути муравей выбирает основываясь на концентрации феромона на вершине. После того, как все муравьи построили путь - они обновляют концентрацию феромона для каждой вершины, которая была задействована в построении пути. Далее запускается следующая популяция муравьев, которая основывается на уже обновленных значениях концентрации феромона.

Данный алгоритм имеет низкую производительность при использовании его в чистом виде, однако при применении эвристик можно добиться значительного улучшения производительности.

Основные этапы реализуемого муравьиного алгоритма:

1. Изначально генерируется матрица размера равного размеру карты, содержащая начальные значения концентрации феромонов τ_{ij}. В общем случае значения матрицы случайны и очень близки к нулю, однако в данной работе применена следующая эвристика:
$$\tau_{ij} = 1 - (\frac{\rho((i,j), target)}{\max_{i,j}(\rho((i,j), target)} + rand(-R, R))$$
где ρ - это двумерное Евклидово расстояние, $target$ - цель, $0 < R < 1$ - это задаваемое значение шума (вносится для того, чтобы приблизить эвристику к общему подходу). Если полученное значение < 0, то $\tau_{ij} = 0$.

2. Популяция муравьев помещается в начальную координату робота.

3. Следующая вершина карты выбирается случайно. Вероятность выбора вершины j в общем случае задается следующей формулой:

$$p_{ij}^k(t) = \frac{\tau_{ij}^\alpha(t)}{\sum\limits_{j \in N_i^k} \tau_{ij}^\alpha(t)}$$

Здесь i – вершина в настоящий момент, N_j^k – список доступных вершин, α – параметр определяющий влияние концентрации феромона на результат, t - текущая итерация алгоритма.
Для достижения лучшей производительности используются две эвристики:

- Эвристика по расстоянию
  Значение эвристики $\eta_{d\ ij}$ рассчитывается по следующей формуле:

$$\eta_{d\ ij} = exp((target - \max(|j[0] - target[0]|, |j[1] - target[1]|) - \\ - \min_{j \in N_i}(target - \max(|j[0] - target[0]|, |j[1] - target[1]|)))$$

  где $target[0]$ - это координата цели по оси X, а $target[0]$ - это координата цели по оси Y. так же и с точкой j. В общем случае можно было использовать любое число в основании степенной функции, потому что это требуется исключительно для того чтобы избежать нулевой вероятности.

- Эвристика по высоте
  Значение эвристики $\eta_{d\ ij}$ рассчитывается по следующей формуле:

$$\eta_{h\ ij} = 1 - |m[j[0]][j[1]] - m[i[0]][i[1]]|$$

  где m - это сгенерированная карта местности.

Таким образом вероятность выбора вершины j с учетом эвристик задается формулой:

$$p_{ij}^k(t) = \frac{\tau_{ij}^\alpha(t)\eta_{d\ ij}^\beta \eta_{h\ ij}}{\sum\limits_{j \in N_i^k} \tau_{ij}^\alpha(t)\eta_{d\ ij}^\beta \eta_{h\ ij}}$$

4. После прохода всего пути всеми муравьями из каждого пути удаляются петли, а концентрация феромона изменяется согласно следующей формуле:

$$\tau_{ij}^\alpha(t+1) = (1-p)\tau_{ij}^\alpha(t) + \sum_{k=1}^{n_k} \frac{q}{L_k(t)}$$

Где p определяет скорость испарения феромонов, n_k – это количество муравьев, $L_k(t)$ – длина пройденного муравьем пути, q - задаваемый параметр.

5. Итерации продолжаются до достижения критерия остановки, который в данном случае задается указанием количества итераций.

Данный алгоритм имеет временную сложность $O(t_n p_n n^4)$, где t_n - количество итераций алгоритма, p_n - размер популяции, n - количество узлов на ребре карты высот.

1.3 Алгоритм коллективного распределения целей

Алгоритм коллективного распределения целей достаточно подробно описан в [2]. Идея заключается в том, чтобы назначить цель роботу только в том случае, если оценка эффективности для достижения роботом цели выше чем до других целей и выше, чем для других роботов и этой цели.

Путь имеется N роботов и соответственно такое же число целей. Задачу можно записать следующим образом:

$$Y_c = \sum_{j=1}^{N} \sum_{l=1}^{N} d_{j,l} n_{j,l} \to max$$

где $n_{j,l}$ определяется при ограничении:

$$\sum_{l=1}^{N} n_{j,l} \leq 1, j = 1, ..., N$$

В качестве оценки эффективности будем использовать Евклидово расстояние от начальной координаты робота до цели.
Алгоритм заключается в следующем:

1. Роботы делают попытку выбора целей в порядке возрастания номеров.

2. Каждый робот может выбирать не более, чем одну цель, после чего не участвует в выборе целей в данной реализации итерационной процедуры.

3. Цель, для которой выполняется условие:

$$\sum_{j=1}^{N} n_{j,l} < n_l^{max},\ l \in [1, M],\ n_l^{max} = 1, 2, 3, ...$$

   т. е. необеспеченная цель, выбирается роботом, еще не выбравшим какую-либо цель, для которого оценка эффективности dj,l этой цели имеет наибольшее значение по сравнению с другими роботами, не выбравшими свои цели.

4. Если на момент выбора цели j-м роботом группы имеется несколько необеспеченных целей, имеющих для данного робота одинаковые значения оценки эффективности, то робот выбирает только одну цель, в соответствии с заранее определенным правилом, одинаковым для всех роботов данной группы. Например, он может выбирать цель с наименьшим номером. Другими словами, если:

$$d_{j,l_1} = d_{j,l_2} = ... = d_{j,l_i}$$
$$l_1 < l_2 < ... < l_i$$

   И выполняется условие из предыдущего пункта, то j-м роботом выбирается цель с номером l_1, т. е. $i_j = l1$.

5. Если несколько роботов имеют одинаковые значения оценки эффективности для l-й цели ($l \in [1, M]$) при условии, что она не обеспечена, то выбор этой цели осуществляется роботом с наименьшим номером.

6. Если максимальное значение оценки эффективности $d_{k^*l^*} = \max_k d_{kl^*}$ некоторой цели T_{l^*} принадлежит роботу R_j, делающему выбор, т. е. $k^* = j$, то робот Rj выбирает данную цель, полагая $i_j = l^*$. В противном случае (когда $k^* \neq j$) робот R_j отказывается от выбора в данном итерационном цикле.

2 Программная реализация

Программ написана на языке Python3.7. Для реализации были использованы следующие библиотеки:

- *math*;
  Библиотека сожержит математические функции. В реализованной программе используется для таких функций, как exp() и ceil().

- *numpy*;
  Библиотека для работы с многомерными массивами. В программе используется как удобный контейнер для хранения матриц и массивов, а также для различного рода взаимодействия с ними.

- *bisect*;
  Библиотека обеспечивает поддержку списка в отсортированном порядке с помощью алгоритма деления пополам. В программе используется в реализации метода рулетки для случайного выбора с заданными весами.

- *time*;
  Библиотека для работы со временем. Используется в реализации измерений времени работы алгоритмов.

- *matplotlib*;
  Графическая библиотека. Используется для построения графиков.

- *progressbar*.
  Библиотека для вывода в консоль состояния выполнения программы в процентном соотношении. Используется для мониторинга состояния программы во время выполнения, так как программа запускалась для длительных расчетов данный инструмент показался необходимым.

В качестве среды разработки использовалась Visual Studio Code. Длительные тесты проводились на виртуальной машине предоставляемой компанией Amazon в качестве одной из услуг AWS (Amazon Web Sevices) - Amazon Elastic Compute Cloud (EC2). Была использована виртуальная машина Amazon Linux AMI (1 vCPUs, 2.5 GHz, Intel Xeon Family).

Все функции и модули программы содержат комментарии.
Главный модуль программы - **main.py**. В нем содержатся исключительно функции для тестирования программ и параметры для запуска алгоритмов. Также при реализации программы были подобраны оптимальные параметры для инициализации алгоритмов.

Для инициализации объекта графа, используются следующие значения параметров:

Коэффициент гладкости карты: 0.3
Коэффициент зашумленности начального феромона: 0.1

Для муравьиного алгоритма используются следующие значения параметров:

Размер популяции:	80
Количество итераций:	10
Коэффициент влияния концентрации феромона:	1.0
Коэффициент влияния эвристики по расстоянию:	1.0
Коэффициент испарения феромона:	0.9
Параметр влияния длины пути на значение феромона:	100

Модуль **graph.py** содержит класс *Graph*, описывающий карту высот, а также все взаимодействующие с ней элементы, такие как матрицы эвристик, матрица концентраций феромонов, нормированная карта высот и пр. . Параметры, определяющие размер карты высот, гладкость карты и зашумленность начальной концентрации феромонов подаются в конструктор класса. Карта высот генерируется в методе класса *generate()*. Взаимодействие с феромонами и эвристиками также определено в этом классе.

Модуль **ant.py** содержит два класса: *Ant* и *EAlg*. Параметры, влияющие на поведение муравьиного алгоритма подаются в конструктор класса *EAlg*. Метод этого класса, принимающий в качестве аргументов объект класса *Graph*, начальные координаты пути и конечные координаты пути, выполняет поиск оптимального пути. В этом методе инициализируется объекты класса *Ant*, который непосредственно строит путь на каждой итерации, обновляет феромоны, и т.д. .

Модуль **planning.py** содержит функцию *planning*, которая в качестве аргументов принимает объекты классов *Graph* и *EAlg*, размер наборов роботов (целей), а также объекты инструментального характера для работы библиотеки *progressbar*. В данной функции выполняется алгоритм коллективного распределения целей.

Все модули взаимодействуют с модулем **tools.py**. В этом модуле функции инструментального характера необходимые как для функционирования основных алгоритмов (например, поиск соседних точек в графе), так и функции для построения графиков и вывода результатов.

Полный листинг программы можно посмотреть в ПРИЛОЖЕНИЕ А.

3 Результаты

В данном разделе представлены непосредственно результаты работы реализованной программы, а также их исследование.

3.1 Diamond-Square

Было необходимо сгенерировать карты размером 25×25, 50×50, 100×100, 500×500, 1000×1000. Результаты работы алгоритма Diamond-Square представлены на Рис. 2-7.

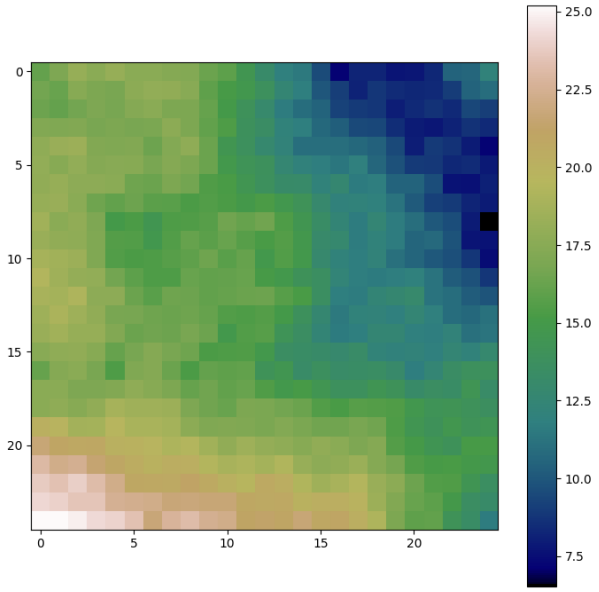


Рис. 2: Карта размера 25x25

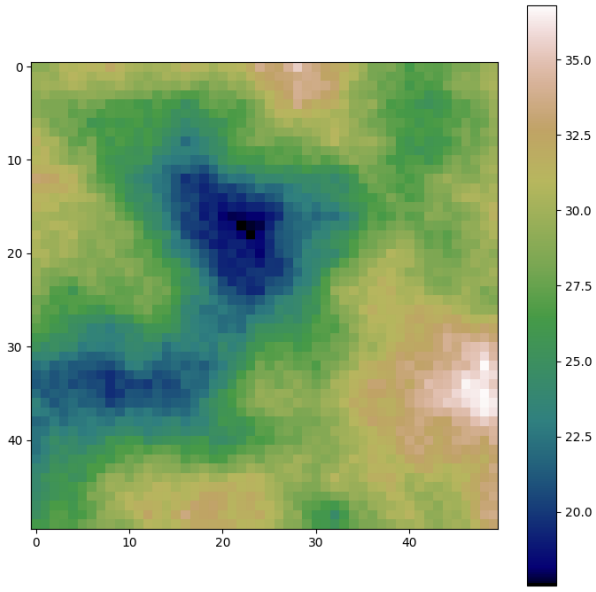


Рис. 3: Карта размера 50x50

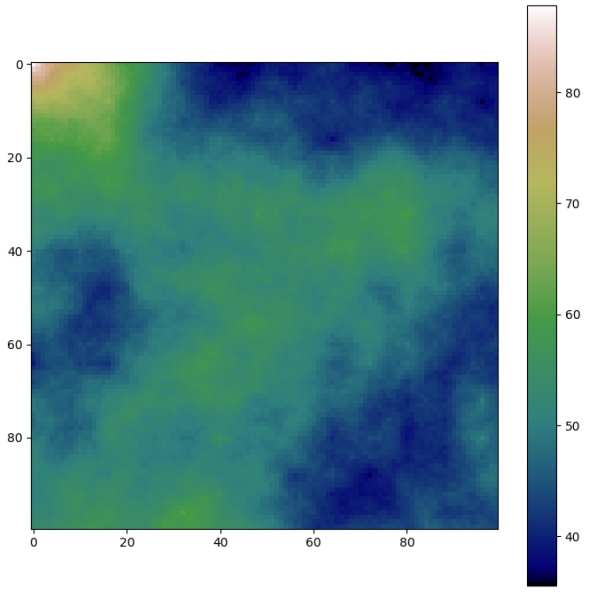


Рис. 4: Карта размера 100x100

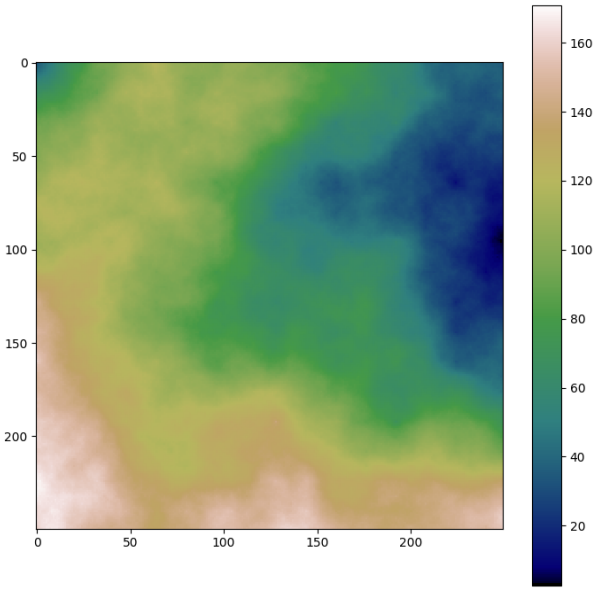


Рис. 5: Карта размера 250x250

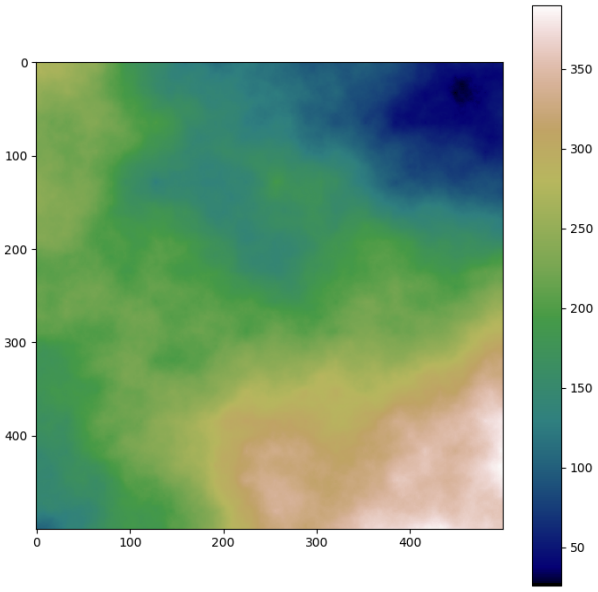


Рис. 6: Карта размера 500x500

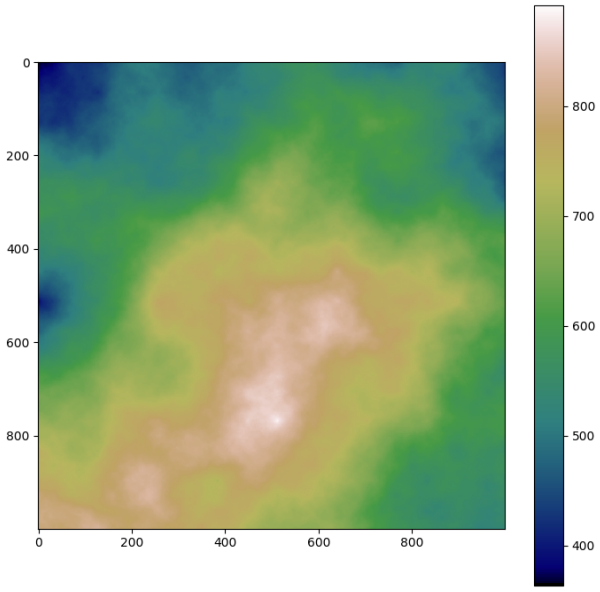
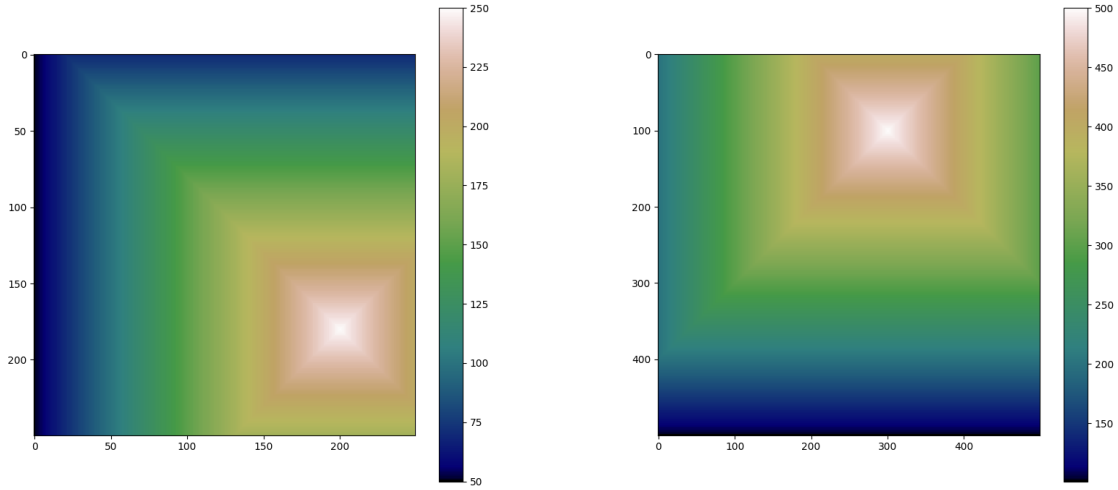


Рис. 7: Карта размера 1000x1000

3.2 Муравьиный алгоритм

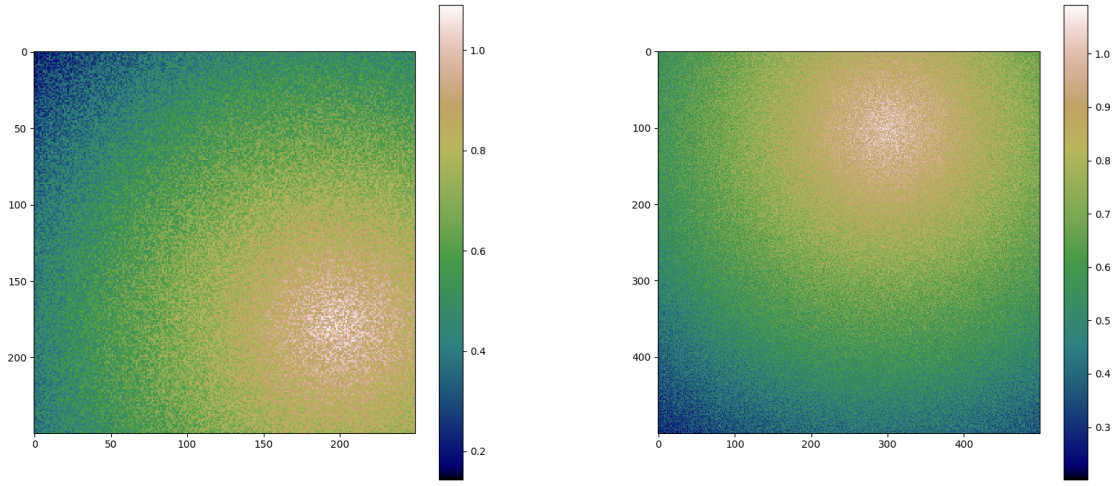
Так как реализованный алгоритм отличается от классического представления муравьиного алгоритма использованием эвристик, то имеет смысл продемонстрировать генерируемые в процессе работы программы карты эвристики. Так как эвристика по высоте определяется только в зависимости от координат соседних точек, то ее сложно представить в виде графика. На Рис. 8 представлены карты эвристики по расстоянию для разного размера карты высот и разных целевых точек.



Карта 250x250, Цель ([180, 200) Карта 500x500, Цель (100, 300)

Рис. 8: Эвристики по расстоянию

На Рис. 9 представлены карты начальной концентрации феромонов для разного размера карты высот и разных целевых точек. Карты зашумлены для приближения работы алгоритма в начальной итерации к общему случаю с случайными значениями концентрации феромонов.



Карта 250x250, Цель (50, 80) Карта 500x500, Цель (345, 345)

Рис. 9: Начальная концентрация феромонов

На Рис. 10-11 представлены результаты поиска пути для этих эвристик и начальных феромонов. Начальная координата - черная, конечная - красная.

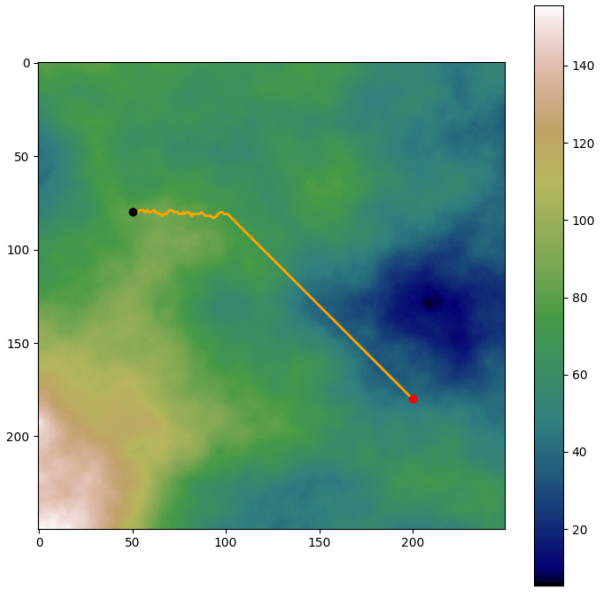


Рис. 10: Построенный путь для робота на карте размера 250x250

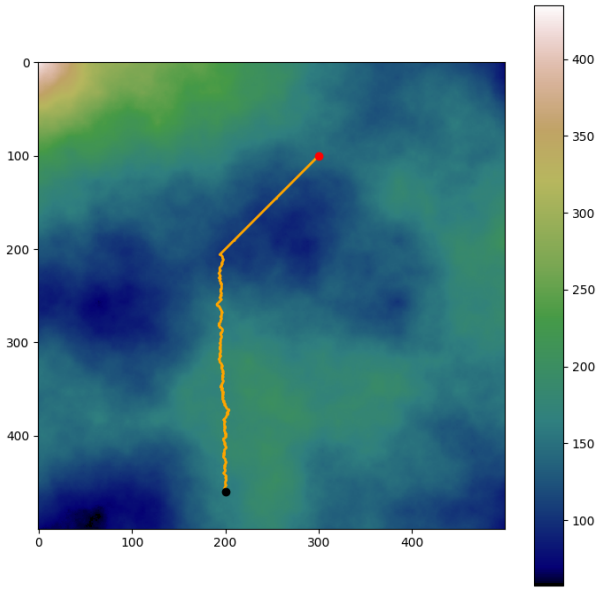


Рис. 11: Построенный путь для робота на карте размера 500x500

3.3 Коллективное распределение целей

Примеры результатов работы алгоритма коллективного распределения целей представлены на Рис. 12-13.

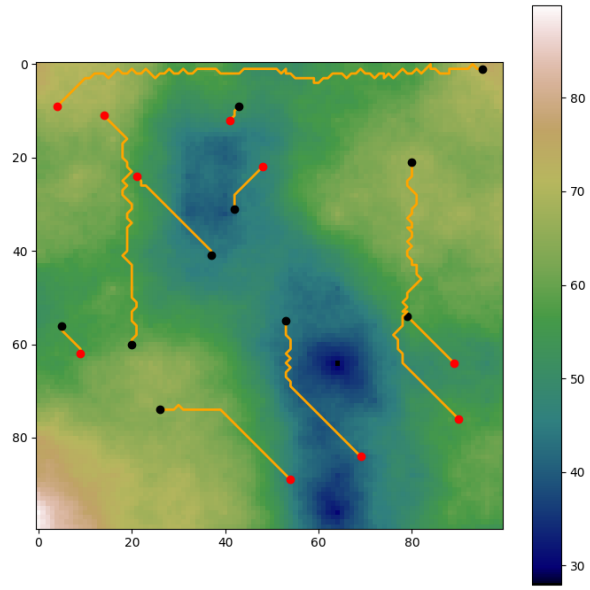


Рис. 12: Построенные пути для 10 роботов на карте размера 100x100

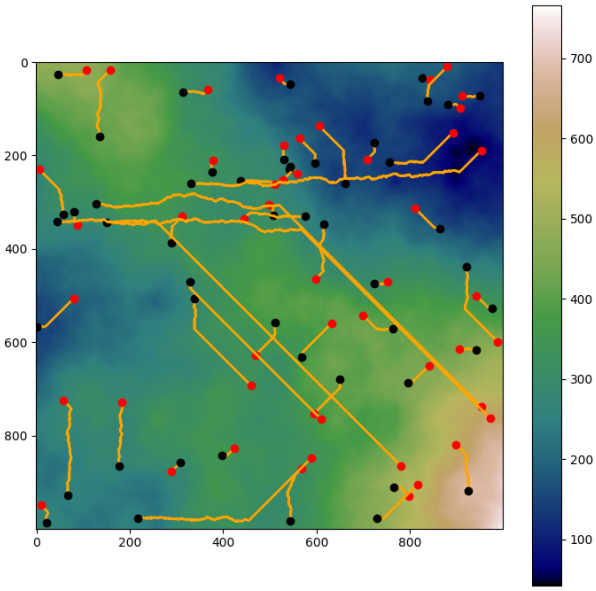


Рис. 13: Построенные пути для 50 роботов на карте размера 1000x1000

3.4 Исследования временной сложности алгоритмов

На Рис. 14 представлена плоскость отображающая зависимость времени вычислений от количества роботов и размера карты. На плоскости отображены только средние элементы измерений времени, полную информацию о измерении времени можно найти

в ПРИЛОЖЕНИЕ Б. Для наглядности ось по времени представлена в виде логарифмической шкалы.

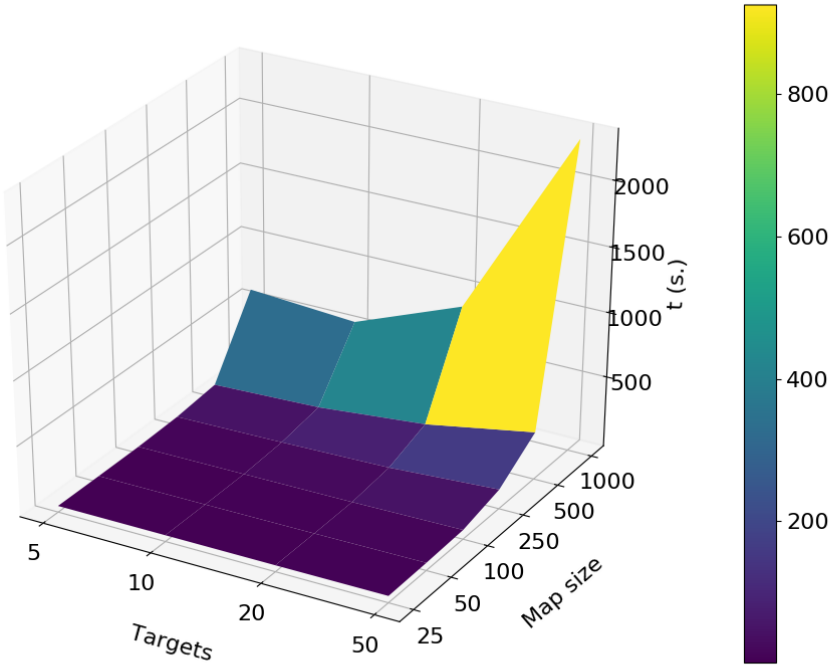


Рис. 14: Зависимость времени вычислений (с.) от размера карты и количества роботов

На Рис. 15 представлен график зависимости времени работы муравьиного алгоритма от размера карты при построении одного пути. Как можно заметить, временная сложность приблизительно квадратичная. Снижение сложности по сравнению с теоретической сложностью алгоритма аргументируется использованием эвристик и оптимального подбора параметров.

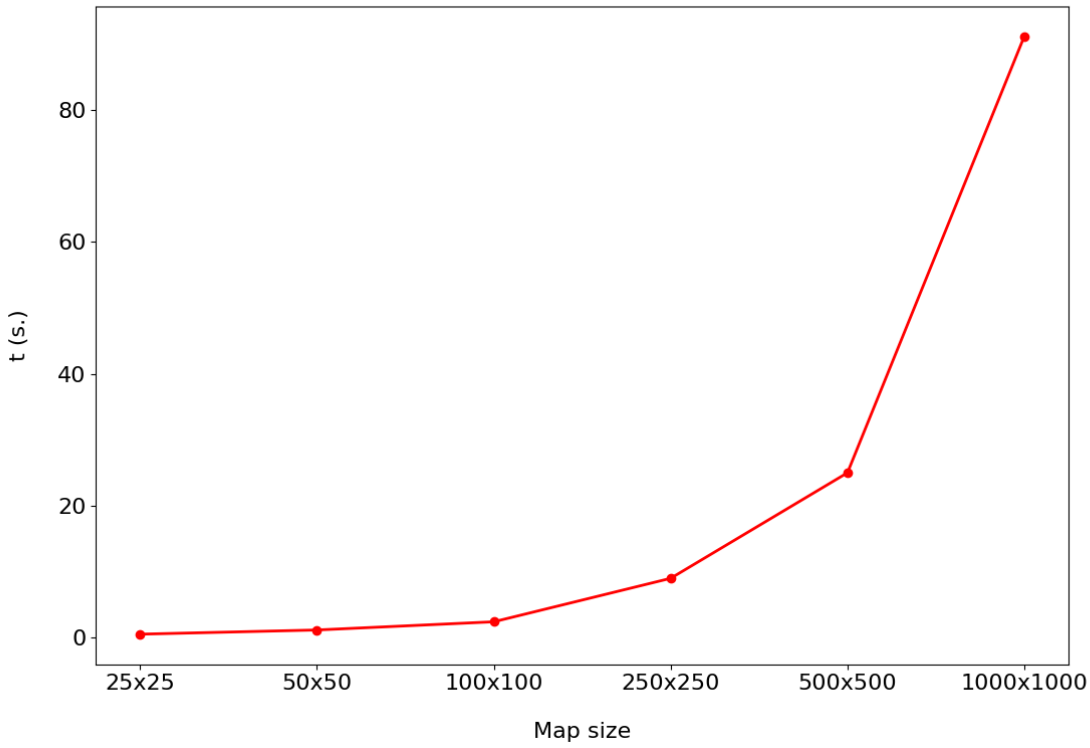


Рис. 15: Зависимость времени вычислений одного пути (с.) от размера карты

Практические результаты действительно отображают, что наибольшее влияние на время выполнение оказывает размер карты высот и меньшее влияние, однако все равно ощутимое, оказывает количество роботов. Общее время вычислений составило ≈ 14 часов. Так как время выполнения теста уже с таким количеством вариаций внушительное, то генерация 10-ти разных наборов данных для каждой карты и каждого количества роботов была опущена.

Из таблиц в ПРИЛОЖЕНИЕ Б также можно сделать вывод о том, что влияние оказываемое алгоритмом распределения целей на общую работу программы минимально и время его работы зависит от количества роботов.

ЗАКЛЮЧЕНИЕ

В рамках выполнения курсовой работы были выполнены следующие задачи:

- Генерация карт размеров 25x25, 50x50, 100x100, 250x250, 500x500, 1000x1000 с помощью алгоритма Diamond-Square;
- Были распределены цели по роботам используя алгоритм коллективного распределения целей для численностей 5, 10, 20 и 50;
- Для роботов были построены оптимальные пути используя муравьиный алгоритм.

В работе проведено исследование сложностей алгоритмов. Теоретическая сложность муравьиного алгоритма $O(t_n p_n n^4)$. Однако результат значительно улучшается с помощью эвристик. Эвристики необходимы для решения проблемы низкой сходимости муравьиного алгоритма. Также в ходе работы были подобраны оптимальные параметры для работы муравьиного алгоритма.

При измерении времени работы программы была выявлена линейная зависимость времени выполнения от размера карты высот. Учитывая предыдущий вывод о сложности муравьиного алгоритма можно сделать вывод, что алгоритм коллективного распределения целей значительно сэкономил время работы программы. Сложность этого алгоритма является пропорциональной количеству роботов (целей).

Таким образом, учитывая, что рассмотренный алгоритм распределения целей весьма низкозатратен (время выполнения менее доли секунды), но при этом значительно улучшает суммарное время работы программы, то его практическая применимость в подобного рода задачах не стоит под вопросом.

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.

[6] Alpa Reshamwala, Deepika P Vinchurkar. Robot Path Planning using An Ant Colony Optimization Approach: A Survey // (IJARAI) International Journal of Advanced Research in Artificial Intelligence, Vol. 2, No.3, 2013, pp. 7

[7] Sangita Sarangi. Optimization of Robot Motion Planning using Ant Colony Optimization // National Institute of Technology, Rourkela, 2011, pp. 81

ПРИЛОЖЕНИЕ А. Исходный код

Ниже приведен исходный код на языке Python

main.py

```
"""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 tools import plot_path
from graph import Graph
from ant import EAlg
from planning import planning

EALG_OBJ = EAlg(
    80,
    10,
    1.0,
    1.0,
    0.9,
    100
)

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"])
```

```
50                  full_times.append(alg_time["Whole"])
51                  file.write(str(map_it) + "\t" + str(alg_time["AntColony"])
                        ↪  + "\t" + str(alg_time["Planning"]) + "\t" + str(
                        ↪ alg_time["Whole"]) + "\n")
52              ant_times_sort = ant_times
53              ant_times_sort.sort()
54              ant_idx = ant_times.index(ant_times_sort[4])
55
56              plan_times_sort = plan_times
57              plan_times_sort.sort()
58              plan_idx = plan_times.index(plan_times_sort[4])
59
60              full_times_sort = full_times
61              full_times_sort.sort()
62              full_idx = full_times.index(full_times_sort[4])
63
64              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")
65              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")
66              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")
67              file.close()
68
69              plot_file_name = "data/mean_paths/" + str(size) + "x" + str(
                    ↪ size) + "/" + str(targets_num) + ".png"
70              plot_paths(graphs[full_idx], opt_paths[full_idx],
                    ↪ plot_file_name)
71
72  def examples_of_data():
73      """Necessary for report"""
74      sizes = [250, 500]
75      end_points = [[50, 80],
76                    [345, 345]]
77      for it, _ in enumerate(sizes):
78          graph = Graph(sizes[it], sizes[it], 0.3, 0.1)
79          graph.generate()
80          graph.init_pheromone_n_heuristics(end_points[it])
81          caption = str(sizes[it]) + "x" + str(sizes[it])
82          plot_heuristic_d(graph, "data/heuristics_example/heuristic_d_" +
                ↪ caption + ".png")
83          plot_pheromone(graph, "data/heuristics_example/pheromone_" +
                ↪ caption + ".png")
84          plot_map(graph, "data/maps_example/" + caption + ".png")
85
86  def dev_test():
87      """Function for development \n
88          I use it for testing components"""
89      size = 1000
90      graph = Graph(size, size, 0.3, 0.2)
91      graph.generate()
92      prog_bar_it = [0]
93      max_val = 50
```

```
94      bar_ = [progressbar.ProgressBar(maxval=max_val).start()]
95      opt_paths, _, alg_time = planning(prog_bar_it, bar_, graph, EALG_OBJ,
            ↪ 50)
96      print(alg_time)
97      plot_paths(graph, opt_paths, "data/mean_paths/test.png")
98
99  def single_path_test():
100     """plot single paths for different maps"""
101     sizes = [250, 500]
102     robots = [[50, 80],
103               [200, 460]]
104     targets = [[200, 180],
105                [300, 100]]
106     file = open("data/path_example/costs.data", "w+")
107     for it, _ in enumerate(sizes):
108         graph = Graph(sizes[it], sizes[it], 0.3, 0.2)
109         graph.generate()
110         path, cost = EALG_OBJ.get_path(graph, robots[it], targets[it])
111         caption = str(sizes[it]) + "x" + str(sizes[it])
112         plot_path(graph, path, "data/path_example/" + caption + ".png")
113         file.write(caption + ":\t" + str(cost) + "\n")
114     file.close()
115
116 def time_correlation():
117     """plot surface with mean times by ready data"""
118     sizes = (25, 50, 100, 250, 500, 1000)
119     targets_numbers = (5, 10, 20, 50)
120     plot_time_correlation(sizes, targets_numbers)
121
122 single_path_test()
123 # examples_of_data()
124 # dev_test()
125 # main_test()
126 # time_correlation()
```

graph.py

```
1   """Graph class"""
2   import math
3   import numpy as np
4   from tools import get_conj
5   from tools import get_distance_proj
6   from tools import get_distance
7   from tools import get_mean
8
9   class Graph:
10      """class for diamond square algorithm"""
11      def __init__(self, n, m, R, pheromone_eur_par):
12          self.n = n
13          self.m = m
14          self.max_element = pow(2, math.ceil(math.log(max(n, m) - 1, 2)))
15          self.matrix = np.zeros((self.max_element + 1, self.max_element +
                ↪ 1))
16          self.norm_matrix = np.zeros((self.max_element + 1, self.
                ↪ max_element + 1))
17          self.height = (n + m) / 2
```

```
18          self.pheromone_eur_par = pheromone_eur_par
19          self.max_dist_z = 0.0
20          self.max_dist_x_y = get_distance_proj([0, 0], [self.n - 1, self.m
                ↪ - 1])
21          self.available_moves = [[float(0) for x in range(n)] for y in
                ↪ range(m)]
22          self.heuristic_h = [[float(0) for x in range(n)] for y in range(m)
                ↪ ]
23          self.costs = [[float(0) for x in range(n)] for y in range(m)]
24          self.heuristic_d = np.zeros((n, m))
25          self.pheromone = np.ones((n, m))
26          self.R = R
27          self.first_call = True
28
29      def generate(self):
30          """main method"""
31          self.matrix[0][0] = np.random.uniform(low=0, high=self.height)
32          self.matrix[self.max_element][self.max_element] = np.random.
                ↪ uniform(low=0, high=self.height)
33          self.matrix[self.max_element][0] = np.random.uniform(low=0, high=
                ↪ self.height)
34          self.matrix[0][self.max_element] = np.random.uniform(low=0, high=
                ↪ self.height)
35
36          side_length = self.max_element
37          while side_length != 1:
38              x_1 = 0
39              y_1 = 0
40              x_2 = side_length
41              y_2 = side_length
42              while True:
43                  self.square(x_1, y_1, x_2, y_2)
44                  self.diamond(x_1, y_1, x_1, y_2)
45                  self.diamond(x_1, y_2, x_2, y_2)
46                  self.diamond(x_2, y_2, x_2, y_1)
47                  self.diamond(x_2, y_1, x_1, y_1)
48                  if y_2 == self.max_element:
49                      if x_2 == self.max_element:
50                          break
51                      else:
52                          x_1 += side_length
53                          x_2 += side_length
54                          y_1 = 0
55                          y_2 = side_length
56                  else:
57                      y_1 += side_length
58                      y_2 += side_length
59              side_length = int(side_length / 2)
60          self.matrix = self.matrix[0:self.n, 0:self.m]
61          self.matrix = np.around(self.matrix, decimals=3)
62          self.max_dist_z = np.amax(self.matrix) - np.amin(self.matrix)
63          self.norm_matrix = self.matrix - np.amin(self.matrix)
64          self.norm_matrix = self.norm_matrix / self.max_dist_z
65
66      def square(self, x_1, y_1, x_2, y_2):
67          """square step of algorithm"""
```

```
68          rad = (x_2 - x_1) / 2
69          center_x = int(x_1 + rad)
70          center_y = int(y_1 + rad)
71          vertexes = [self.matrix[x_1][y_1],
72                      self.matrix[x_2][y_2],
73                      self.matrix[x_1][y_2],
74                      self.matrix[x_2][y_1]]
75          self.matrix[center_x][center_y] = (get_mean(vertexes)) + np.random
               ↪ .uniform(low=(- self.R * rad * 2), high=(self.R * rad * 2))
76
77      def diamond(self, x_1, y_1, x_2, y_2):
78          """diamond step of algorithm"""
79          vertexes = []
80          x = 0
81          y = 0
82          rad = 0.0
83          if x_1 == x_2:
84              center_y = int((y_1 + y_2) / 2)
85              rad = abs(y_2 - center_y)
86              if x_1 not in (0, self.max_element):
87                  vertexes += [self.matrix[x_1][y_1],
88                               self.matrix[x_2][y_2],
89                               self.matrix[x_1 - rad][center_y],
90                               self.matrix[x_1 + rad][center_y]]
91              else:
92                  if x_1 == 0:
93                      vertexes += [self.matrix[x_1][y_1],
94                                   self.matrix[x_2][y_2],
95                                   self.matrix[x_1 + rad][center_y]]
96                  if x_1 == self.max_element:
97                      vertexes += [self.matrix[x_1][y_1],
98                                   self.matrix[x_2][y_2],
99                                   self.matrix[x_1 - rad][center_y]]
100             x = x_1
101             y = center_y
102         else:
103             center_x = int((x_1 + x_2) / 2)
104             rad = abs(x_2 - center_x)
105             if y_1 not in (0, self.max_element):
106                 vertexes += [self.matrix[x_1][y_1],
107                              self.matrix[x_2][y_2],
108                              self.matrix[center_x][y_1 - rad],
109                              self.matrix[center_x][y_1 + rad]]
110             else:
111                 if y_1 == 0:
112                     vertexes += [self.matrix[x_1][y_1],
113                                  self.matrix[x_2][y_2],
114                                  self.matrix[center_x][y_1 + rad]]
115                 if y_1 == self.max_element:
116                     vertexes += [self.matrix[x_1][y_1],
117                                  self.matrix[x_2][y_2],
118                                  self.matrix[center_x][y_1 - rad]]
119             x = center_x
120             y = y_1
121         self.matrix[x][y] = get_mean(vertexes) + np.random.uniform(low=(-
               ↪ self.R * rad * 2), high=(self.R * rad * 2))
```

```
122
123     def get_size(self):
124         """Get size of graph in format (,)"""
125         return (self.n, self.m)
126
127     def init_pheromone_n_heuristics(self, end_point):
128         """Init pheromone matrix, heuristic_d matrix, heuristic_h matrix
                ↪ of distances to available \n
129              moves by z and available_moves matrix of lists"""
130         if self.first_call:
131             for it in range(self.n):
132                 for jt in range(self.m):
133                     dist_to_conj = []
134                     conj_points = get_conj(self.norm_matrix, [it, jt])
135                     for point in conj_points:
136                         dist_to_conj.append(get_distance([it, jt], point,
                                ↪ self.matrix))
137                     self.costs[it][jt] = dist_to_conj
138                     self.available_moves[it][jt] = conj_points
139             self.first_call = False
140         else:
141             self.heuristic_h = [[float(0) for x in range(self.n)] for y in
                ↪  range(self.m)]
142             self.heuristic_d = np.zeros((self.n, self.m))
143             self.pheromone = np.ones((self.n, self.m))
144
145         end_point_val = max(self.n, self.m)
146         for it in range(self.n):
147             for jt in range(self.m):
148                 if not (it == end_point[0] and jt == end_point[1]):
149                     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))
150                     if pheromone < 0:
151                         self.pheromone[it][jt] = 0.0
152                     else:
153                         self.pheromone[it][jt] = pheromone
154                 z_dist_to_conj = []
155                 conj_points = get_conj(self.norm_matrix, [it, jt])
156                 for point in conj_points:
157                     z_dist_to_conj.append(1 - abs(self.norm_matrix[point
                            ↪ [0]][point[1]] - self.norm_matrix[it][jt]))
158                 self.heuristic_h[it][jt] = z_dist_to_conj
159                 self.heuristic_d[it][jt] = end_point_val - max(abs(it -
                    ↪ end_point[0]), abs(jt - end_point[1]))
160
161     def update_pheromone(self, pheromone_increment: np.array,
            ↪ evaporation_coef):
162         """Updates pheromone values"""
163         self.pheromone *= (1 - evaporation_coef)
164         self.pheromone += pheromone_increment
165
166     def get_pheromone(self, position):
167         """Returns pheromone value for position"""
168         return self.pheromone[position[0]][position[1]]
169
```

```
170     def get_heuristic_h(self, position):
171         """Returns list of distance by z to possible moves for ant"""
172         return self.heuristic_h[position[0]][position[1]]
173 
174     def get_available_moves(self, position):
175         """Returns list of possible moves for ant"""
176         return self.available_moves[position[0]][position[1]]
177 
178     def get_heuristic_d(self, position):
179         """Returns heuristic_d value for position"""
180         return self.heuristic_d[position[0]][position[1]]
181 
182     def get_cost(self, position):
183         """Returns list costs for conjugate positions"""
184         return self.costs[position[0]][position[1]]
185 
186     def get_pos_parameters(self, position):
187         """Returns parameters for possibility calculating \n
188         return available_moves, pheromones, heuristic_d, self.
               ↪ get_heuristic_h(position)"""
189         available_moves = self.get_available_moves(position)
190         heuristic_d = []
191         pheromones = []
192         for move in available_moves:
193             heuristic_d.append(self.get_heuristic_d(move))
194             pheromones.append(self.get_pheromone(move))
195         min_h_d = min(heuristic_d)
196         heuristic_d = [val - min_h_d for val in heuristic_d]
197         sum_d = sum([math.exp(w_d) for w_d in heuristic_d])
198         heuristic_d = [math.exp(w_d) / sum_d for w_d in heuristic_d]
199         return available_moves, pheromones, heuristic_d, self.
               ↪ get_heuristic_h(position), self.get_cost(position)
200 
201     def get_matrix(self):
202         """Returns surface in matrix formats"""
203         return self.matrix
```

ant.py

```
1  """Evolution algorithm implementation"""
2  import numpy as np
3  from graph import Graph
4  from tools import choice
5 
6  class Ant:
7      """Single ant behavior"""
8      def __init__(self, graph: Graph, start, alpha, beta, q):
9          self.graph = graph
10         self.position = start
11         self.alpha = alpha
12         self.beta = beta
13         self.path = [start]
14         self.path_length = 0.0
15         self.last_cost = 0.0
16         self.q = q
17         self.increase = [0.0]
```

```
18          self.iteration = 0
19          self.fail = False
20
21      def get_pos(self):
22          """Get ant's position"""
23          return self.position
24
25      def move(self):
26          """Move ant in next graph's point"""
27          available_moves, moves_pheromones, moves_heuristic_d,
               ↪ moves_heuristic_h, moves_costs = self.graph.
               ↪ get_pos_parameters(self.position)
28          weights = []
29          sum_w = 0.0
30          for it in range(len(available_moves)):
31              weight = moves_pheromones[it] ** self.alpha *
                   ↪ moves_heuristic_d[it] ** self.beta * moves_heuristic_h[it
                   ↪ ]
32              sum_w += weight
33              weights.append(weight)
34          weights = [w / sum_w for w in weights]
35          choosen_idx = choice(weights)
36          self.position = available_moves[choosen_idx]
37          self.path.append(self.position)
38          # if moves_costs[choosen_idx] == 0.0: # was Loch Ness bug and this
               ↪  is for safety
39          #     moves_costs[choosen_idx] += 0.01
40          self.increase.append(moves_costs[choosen_idx])
41          self.path_length += moves_costs[choosen_idx]
42          self.iteration += 1
43
44      def get_position(self):
45          """Returns position of ant"""
46          return self.position
47
48      def get_pheromone_increase(self, idx):
49          """Return pheromone increase for idx move"""
50          return self.q / self.increase[idx]
51
52      def get_path_length(self):
53          """Returns path length"""
54          return self.path_length
55
56      def get_path(self):
57          """Returns path"""
58          return self.path
59
60      def delete_loops(self):
61          """Delete loops from path"""
62          for it in self.path:
63              if self.path.count(it) > 1:
64                  idx = self.path.index(it)
65                  for jt in range(idx, len(self.path) - 1 - self.path[::-1].
                       ↪ index(it)): #last idx
66                      self.path.pop(idx)
67                      self.path_length -= self.increase[idx]
```

```
68                      self.increase.pop(idx)
69
70  class EAlg:
71      """Evolution algorithm"""
72      def __init__(self, pop_size, iter_size, alpha, beta, rho, q):
73          self.pop_size = pop_size
74          self.iter_size = iter_size
75          self.alpha = alpha
76          self.beta = beta
77          self.rho = rho
78          self.q = q
79
80      def get_path(self, graph: Graph, start: [], end_point: []):
81          """Main method of algorithm, which find best path\n
82          It returns cost and path
83          """
84          path = []
85          path_length = float('inf')
86          graph.init_pheromone_n_heuristics(end_point)
87          lim = 0
88          if (graph.get_size()[0] + graph.get_size()[1]) / 2 < 100:
89              lim = 10000
90          else: lim = graph.get_size()[0] * graph.get_size()[1] / 10
91          for it in range(self.iter_size):
92              pheromone_increment = np.zeros(graph.get_size())
93              for ant_it in range(self.pop_size):
94                  ant = Ant(graph, start, self.alpha, self.beta, self.q)
95                  while ant.get_pos() != end_point:
96                      ant.move()
97                      if ant.iteration == lim:
98                          ant.fail = True
99                          break
100                     pos = ant.get_pos()
101                     pheromone_increment[pos[0]][pos[1]] += ant.
                            ↪ get_pheromone_increase(len(ant.get_path()) - 1)
102                 if not ant.fail:
103                     ant.delete_loops()
104                     if ant.get_path_length() < path_length:
105                         path = ant.get_path()
106                         path_length = ant.get_path_length()
107             if not ant.fail:
108                 graph.update_pheromone(pheromone_increment, self.rho)
109
110         return path, path_length
```

planning.py

```
1   """Planning algorithm"""
2   from time import time
3   import numpy as np
4   from graph import Graph
5   from ant import EAlg
6   from tools import get_distance_proj
7
8   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)]
    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)
```

```
62
63      for pair in opt_pairs:
64          prog_bar_it[0] += 1
65          bar_[0].update(prog_bar_it[0])
66          path, cost = model.get_path(graph, robots[pair[0]], targets[pair
               ↪ [1]])
67          opt_paths.append(path)
68          opt_costs.append(cost)
69
70
71
72      time_dic["AntColony"] = time() - ant_start
73      time_dic["Whole"] = round(time_dic["AntColony"] + time_dic["Planning"
           ↪ ], 3)
74
75      return opt_paths, opt_costs, time_dic
```

tools.py

```
1   """usefull functions"""
2   import math
3   import bisect
4   import numpy as np
5   import matplotlib.pyplot as plt
6   from matplotlib import rcParams
7   from mpl_toolkits.mplot3d import Axes3D
8
9   def plot_surface(matrix, sizes, targets_numbers, file_name):
10      """Plot 3d surface"""
11      rcParams.update({'font.size': 16})
12      (x, y) = np.meshgrid(np.arange(matrix.shape[1]), np.arange(matrix.
           ↪ shape[0]))
13      fig = plt.figure()
14      ax = fig.add_subplot(111, projection='3d')
15      surf = ax.plot_surface(x, y, np.log(matrix), cmap=plt.get_cmap("
           ↪ viridis"))
16      ax.set_xlabel('Targets', labelpad=20)
17      ax.set_ylabel('Map size', labelpad=20)
18      ax.set_zlabel('ln(t)', labelpad=10)
19      plt.xticks(range(len(targets_numbers)), targets_numbers)
20      plt.yticks(range(len(sizes)), sizes)
21      fig.colorbar(surf)
22      fig.set_size_inches(12.5, 8.5)
23      fig.savefig(file_name, dpi=100)
24      plt.close(fig)
25
26  def plot_time_correlation(sizes, targets_numbers):
27      """tool for plot surface from time data"""
28      matrix = np.zeros((len(sizes), len(targets_numbers)))
29      for it, _ in enumerate(sizes):
30          root = "data/time/" + str(sizes[it]) + "x" + str(sizes[it]) + "/"
31          for jt, _ in enumerate(targets_numbers):
32              file_path = root + str(targets_numbers[jt]) + ".data"
33              file = open(file_path)
34              mean_time = float(file.readlines()[15].rstrip().rsplit("\t")
                   ↪ [3])
```

```
35              matrix[it][jt] = round(mean_time, 3)
36      plot_surface(matrix, sizes, targets_numbers, "data/time/mean_surface.
            ↪ png")
37
38  def plot_map(graph, file_name):
39      """Plot map in heatmap format"""
40      plot_heatmap(graph.get_matrix(), file_name)
41
42  def plot_paths(graph, paths, file_name):
43      """Plot paths on map"""
44      fig = plt.figure()
45      ax = fig.add_subplot(111)
46      pl = ax.imshow(graph.get_matrix(), cmap=plt.get_cmap("gist_earth"))
47      fig.colorbar(pl)
48      fig.set_size_inches(8.5, 8.5)
49      for path in paths:
50          ax.plot([x for x, y in path], [y for x, y in path], linewidth=2.0,
                ↪  c="orange")
51          ax.plot(path[0][0], path[0][1], "ro", c="black")
52          ax.plot(path[len(path) - 1][0], path[len(path) - 1][1], "ro", c="
                ↪ red")
53      fig.savefig(file_name, dpi=100)
54      plt.close(fig)
55
56  def plot_path(graph, path, file_name):
57      """Plot single path on map"""
58      fig = plt.figure()
59      ax = fig.add_subplot(111)
60      pl = ax.imshow(graph.get_matrix(), cmap=plt.get_cmap("gist_earth"))
61      fig.colorbar(pl)
62      fig.set_size_inches(8.5, 8.5)
63      ax.plot([x for x, y in path], [y for x, y in path], linewidth=2.0, c="
            ↪ orange")
64      ax.plot(path[0][0], path[0][1], "ro", c="black")
65      ax.plot(path[len(path) - 1][0], path[len(path) - 1][1], "ro", c="red")
66      fig.savefig(file_name, dpi=100)
67      plt.close(fig)
68
69  def plot_heuristic_d(graph, file_name):
70      """Plot heuristic by distance in heatmap format"""
71      plot_heatmap(graph.heuristic_d, file_name)
72
73  def plot_pheromone(graph, file_name):
74      """Plot pheromone heatmap"""
75      plot_heatmap(graph.pheromone, file_name)
76
77  def plot_heatmap(matrix, file_name):
78      """Plot 2d heat map"""
79      fig = plt.figure()
80      ax = plt.imshow(matrix, cmap=plt.get_cmap("gist_earth"))
81      fig.colorbar(ax)
82      fig.set_size_inches(8.5, 8.5)
83      fig.savefig(file_name, dpi=100)
84      plt.close(fig)
85
86  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):
```

```
143     """distance between two point by x and y using Euclid metric"""
144     return math.sqrt((x[0] - y[0]) ** 2 + (x[1] - y[1]) ** 2)
145 
146 def get_distance(x, y, matrix):
147     """distance between two point by x, y and z using Euclid metric"""
148     return math.sqrt((x[0] - y[0]) ** 2 + (x[1] - y[1]) ** 2 + (matrix[x
        ↪ [0]][x[1]] - matrix[y[0]][y[1]]) ** 2)
149 
150 def get_mean(some_list):
151     """Returns mean value of list"""
152     return sum(some_list) / len(some_list)
```

ПРИЛОЖЕНИЕ Б. Таблицы измерений времени

Ниже приведены измерения 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 измерениыя 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

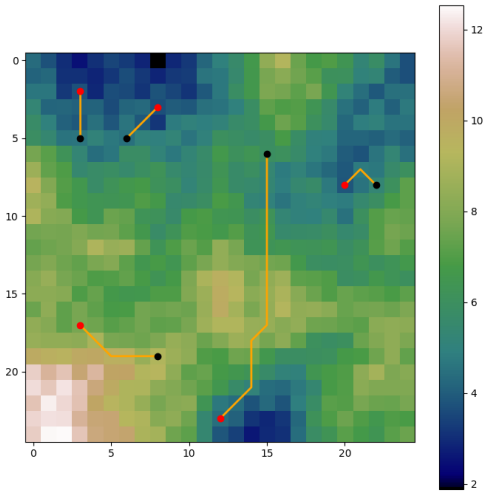
Размер карты: 500x500

№ 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

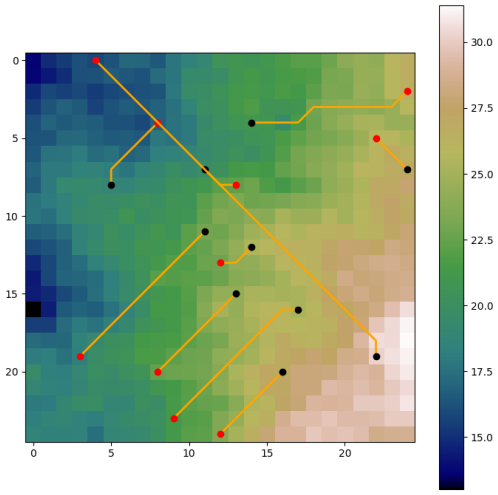
Размер карты: 1000x1000

ПРИЛОЖЕНИЕ В. Графики решений

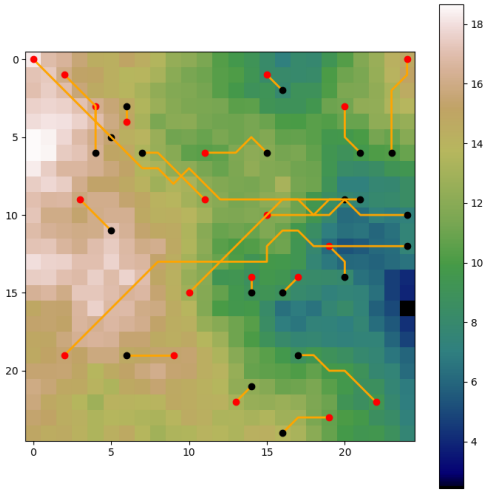
Ниже представлены построенные пути с средним временем выполнения для разного числа роботов и разных размеров матриц (черным отмечены роботы, красным - цели):



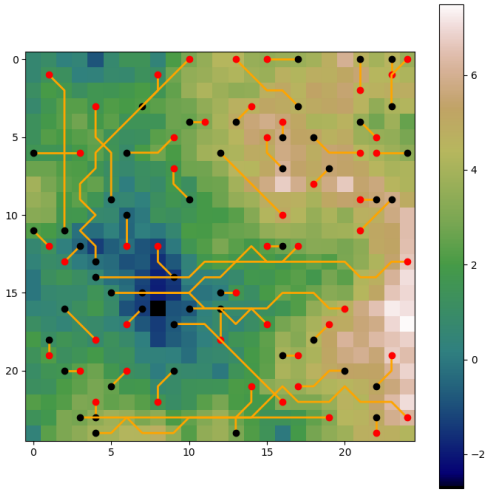
5 роботов



10 роботов

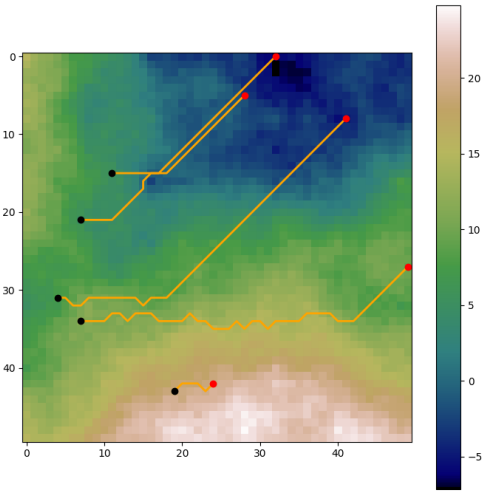


20 роботов

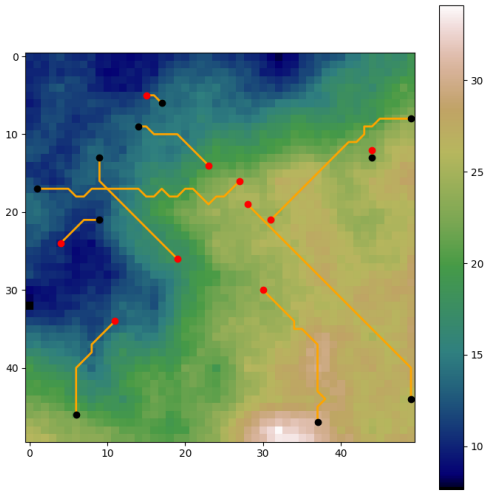


50 роботов

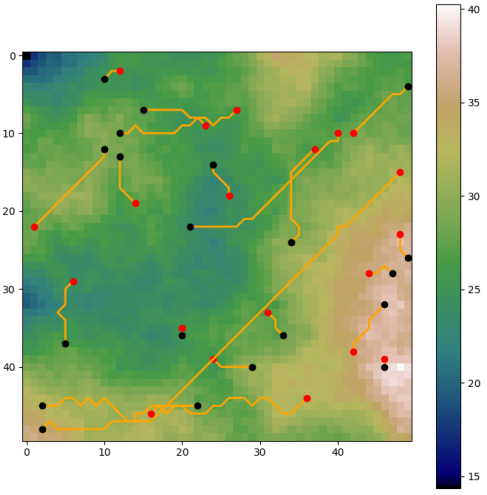
Размер карты: 25x25



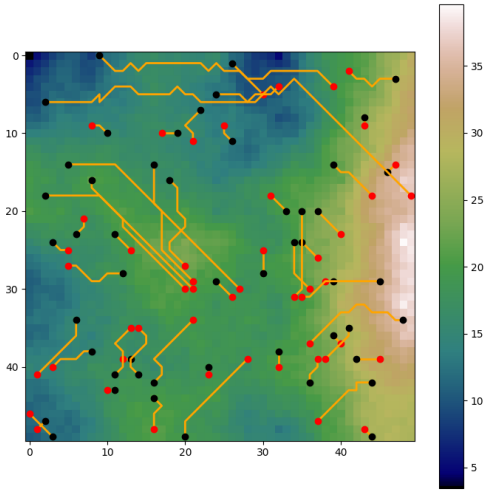
5 роботов



10 роботов

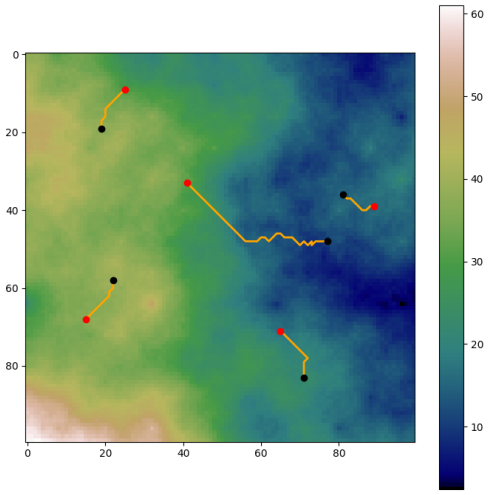


20 роботов

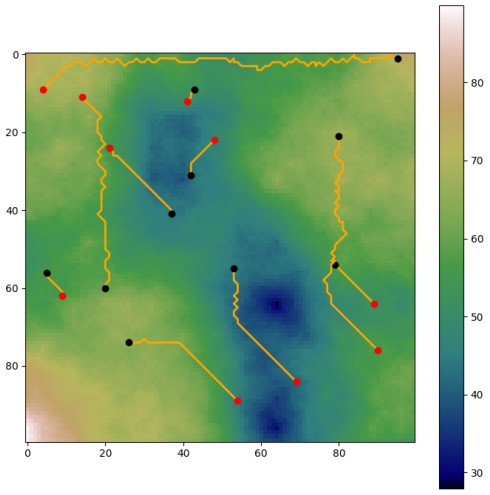


50 роботов

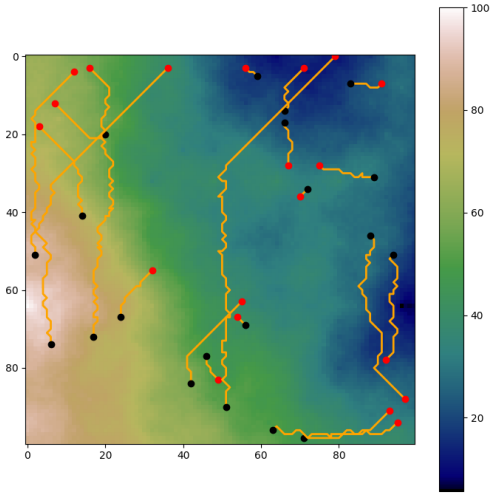
Размер карты: 50x50



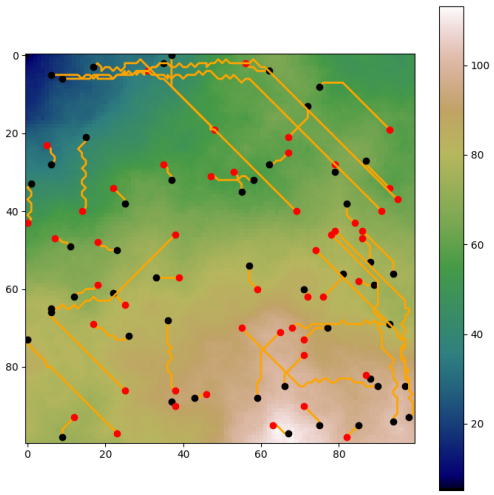
5 роботов



10 роботов

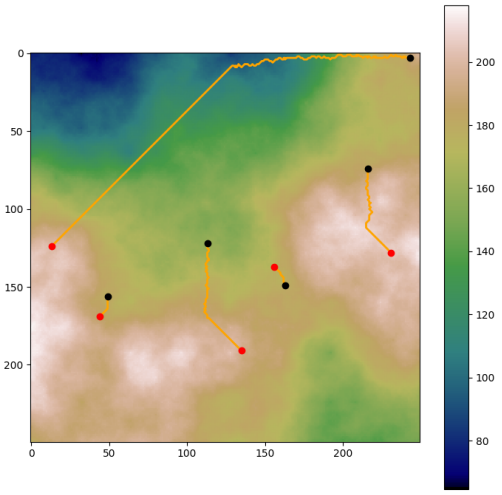


20 роботов

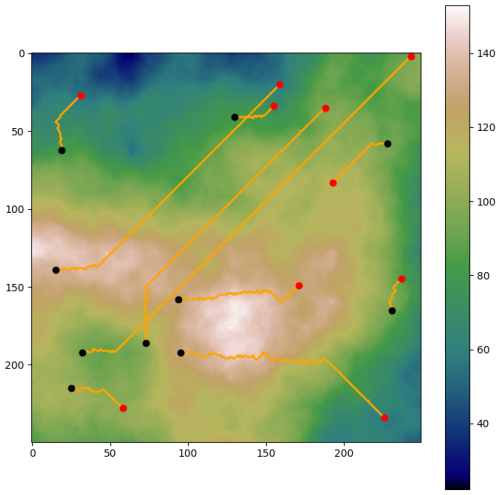


50 роботов

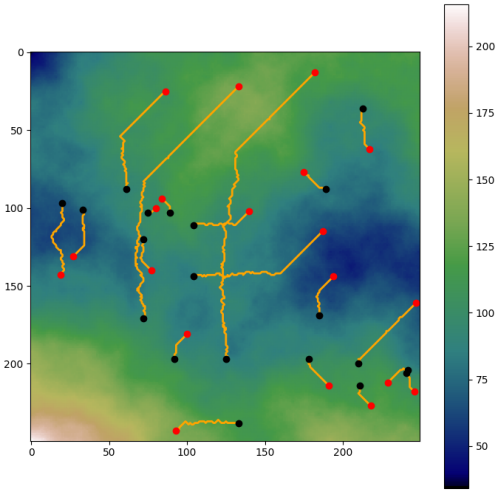
Размер карты: 100x100



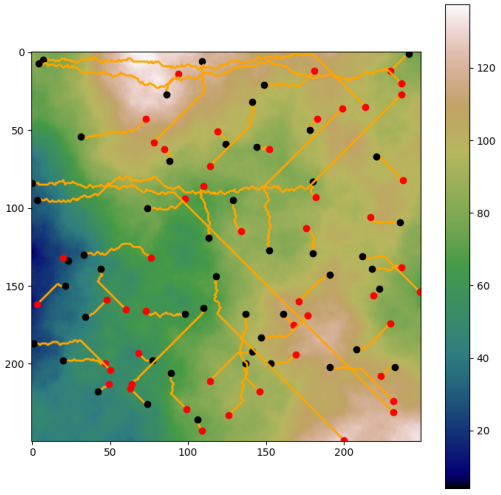
5 роботов



10 роботов

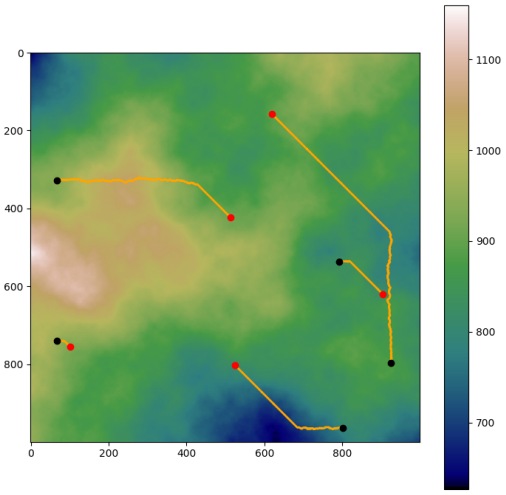


20 роботов

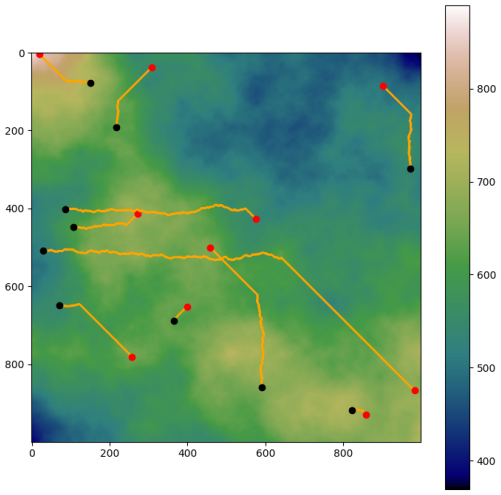


50 роботов

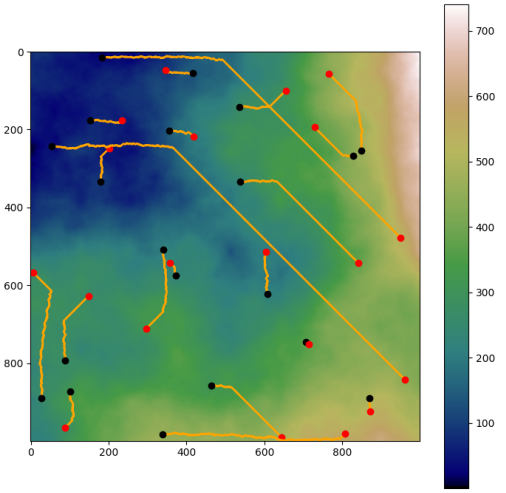
Размер карты: 250x250



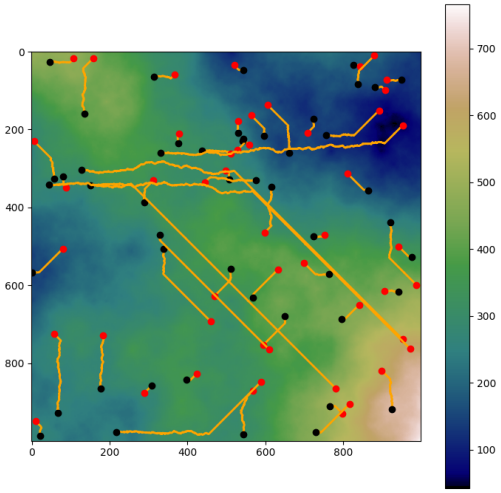
5 роботов



10 роботов



20 роботов



50 роботов

Размер карты: 1000x1000