Анализ Алгоритмов

Лабораторная работа №6

По теме “*Муравьиный алгоритм и задача коммивояжера*”

Студент: Юмаев Артур

Группа: ИУ7-55

Оглавление

[Введение 3](#_Toc27010162)

[1. Аналитическая часть 4](#_Toc27010163)

[Обзор алгоритма 4](#_Toc27010164)

[Вывод 5](#_Toc27010165)

[2. Конструкторская часть 6](#_Toc27010166)

[3. Технологическая часть 7](#_Toc27010167)

[Поиск минимального расстояния методом Левенштейна матрично 7](#_Toc27010168)

[Поиск минимального расстояния методом Дамерау-Левенштейна матрично. 8](#_Toc27010169)

[Поиск минимального расстояния методом Левенштейна рекурсивно. 8](#_Toc27010170)

[Поиск минимального расстояния методом Дамерау-Левенштейна рекурсивно. 9](#_Toc27010171)

[Сравнительный анализ потребляемой памяти 9](#_Toc27010172)

[4. Исследовательская часть 11](#_Toc27010173)

[Заключение 13](#_Toc27010174)

[Литература 14](#_Toc27010175)

# Введение

Муравьиный алгоритм (алгоритм оптимизации подражанием муравьиной колонии, англ. ant colony optimization, ACO) — один из эффективных полиномиальных алгоритмов для нахождения приближённых решений задачи коммивояжёра, а также решения аналогичных задач поиска маршрутов на графах. Суть подхода заключается в анализе и использовании модели поведения муравьёв, ищущих пути от колонии к источнику питания. Первая версия алгоритма, предложенная доктором наук Марко Дориго[1][2] в 1992 году, была направлена на поиск оптимального пути в графе.

# 1. Аналитическая часть

В данном разделе будет дано полное описание муравьиного алгоритма и его математическое описание.

## Обзор алгоритма

Муравьи путешествуют из начальной точки в финальную, посещая все города. Они оставляют феромоны по пути назад. Также они оставляют больше феромонов на коротких дистанциях, чем на длинных и только на тех дистанциях, по которым ходят. Каждый конкретный муравей принимает решение в какой город пойти, основываясь на уровне феромонов на данном пути и дистанции до ближайшего города.

Муравей принимает решение по какому пути он пойдет от города *i* до города *j*, основываясь на следующей формуле.

|  |  |  |
| --- | --- | --- |
|  |  | (1) |

где – “вес” феромонов, – величина, определяющая “жадность” алгоритма, – количество феромона на пути *i*, – длина пути до города *j*, *a* – в данном случае представляет собой множество городов, которые муравей еще не посещал, т.е. суммирование идет по городам, которых нет в множестве посещенных, так как муравью нельзя возвращаться обратно по условию.

Например, если муравей в городе 2 и доступны для посещения города 4, 7 и 8 с весом перехода в каждый:

|  |  |  |
| --- | --- | --- |
|  |  | (2) |

то итоговая вероятность перехода в город 4 будет равна:

|  |  |  |
| --- | --- | --- |
|  |  | (3) |

и так далее. Чем больше вероятность перехода на пути *i*, тем больше вероятность того, что муравей выберет путь *i*.

Таким образом муравей продолжает путешествовать в соответствии с формулой вероятности (1) пока не посетит все города и не вернется обратно. На начальном этапе количество феромона полагается константным на всех путях.

На обратном пути все муравьи оставляют за собой след из феромона на *i*-том пути, который равен:

|  |  |  |
| --- | --- | --- |
|  |  | (4) |

где – расстояние между городом *i* и *j*.

Например, муравей путешествует по пути:

|  |  |  |
| --- | --- | --- |
|  |  | (5) |

при этом:

|  |  |  |
| --- | --- | --- |
|  |  | (6) |

тогда оставленный след из феромона будет увеличен как:

|  |  |  |
| --- | --- | --- |
|  |  | (7) |

Это исходит из того, что муравьи склонны выбирать более короткие маршруты.

Последнее, что нам нужно сделать, это учесть распад феромона. Программно это реализовывается следующим образом.

|  |  |  |
| --- | --- | --- |
|  | , | (8) |

где параметр *decay* отвечает за распад феромона. Обычно он равен 0.5.

## Вывод

В данном разделе было приведено общее и математическое описание муравьиного алгоритма, приведены примеры.

# 2. Конструкторская часть

В данном разделе будет приведена схема для муравьиного алгоритма (рисунок 1).

Рис. 1 – Схема муравьиного алгоритма

## Вывод

В данном разделе была приведена схема муравьиного алгоритма на псевдокоде.

# 3. Технологическая часть

В качестве языка программирования был выбран Python [3], так как имеется большой опыт работы с ним, в частности Python предоставляет быстрые и удобные средства для работы с многомерными массивами, что упрощает и ускоряет разработку.

Листинг 1. Муравьиный алгоритма в виде Python класса

|  |
| --- |
| **class** AntColony(object):  **def** \_\_init\_\_(self, distances, n\_ants, n\_best, n\_iterations,  decay, alpha=1, beta=1):self.distances = distances  self.pheromone = np.ones(self.distances.shape) / len(distances)  self.all\_inds = range(len(distances))  self.n\_ants = n\_ants  self.n\_best = n\_best  self.n\_iterations = n\_iterations  self.decay = decay  self.alpha = alpha  self.beta = beta   **def** run(self):  shortest\_path = **None** *# Массив итогового решения и длина лучшего пути* all\_time\_shortest\_path = (**"placeholder"**, np.inf)   **for** i **in** range(self.n\_iterations):  *# Генерируем все пути i-го муравья* all\_paths = self.gen\_all\_paths()  self.spread\_pheronome(all\_paths,  self.n\_best,  shortest\_path=shortest\_path)  shortest\_path = min(all\_paths, key=**lambda** x: x[1])  **if** shortest\_path[1] < all\_time\_shortest\_path[1]:  all\_time\_shortest\_path = shortest\_path  self.pheromone \* self.decay  **return** all\_time\_shortest\_path   **def** spread\_pheronome(self, all\_paths, n\_best, shortest\_path):  sorted\_paths = sorted(all\_paths, key=**lambda** x: x[1])  **for** path, dist **in** sorted\_paths[:n\_best]:  **for** move **in** path:  self.pheromone[move] += 1.0 / self.distances[move]   **def** gen\_path\_dist(self, path):  total\_dist = 0  **for** ele **in** path:  total\_dist += self.distances[ele]  **return** total\_dist   *# Генерация всех путей из начальной точки* **def** gen\_all\_paths(self):  all\_paths = []  **for** i **in** range(self.n\_ants):  path = self.gen\_path(0)  all\_paths.append((path, self.gen\_path\_dist(path)))  **return** all\_paths   **def** gen\_path(self, start):  path = []  visited = set()  visited.add(start)  prev = start  **for** i **in** range(len(self.distances) - 1):  move = self.pick\_move(self.pheromone[prev],  self.distances[prev], visited)  path.append((prev, move))  prev = move  visited.add(move)  path.append((prev, start))**return** path   **def** pick\_move(self, pheromone, dist, visited):  pheromone = np.copy(pheromone)  pheromone[list(visited)] = 0   row = pheromone \*\* self.alpha \* (( 1.0 / dist) \*\* self.beta)   nor m\_row = row / row.sum()  move = np\_choice(self.all\_inds, 1, p=norm\_row)[0]  **return** move |

## Вывод

В данном разделе была приведена реализация муравьиного алгоритма на языке программирования Python.

# 4. Исследовательская часть

Замеры времени проводились на 64-битной операционной системе Windows 10 и на x64 процессоре Inter Core i7 с 4 гб оперативной памяти для диапозона городов от 1 до 10, так как на графе в 11 вершин мощности аппаратного обеспечения не хватало для решения задачи полным перебором. На момент замера времени работало в среднем 76 активных процессов.

Уже на графе с 9 вершинами видно, что решение полным перебором является неэффективным, несмотря на то, что в данном случае на количестве вершин < 9 полный перебор является более эффективным, так как меньше затраты алгоритма на память и операции.

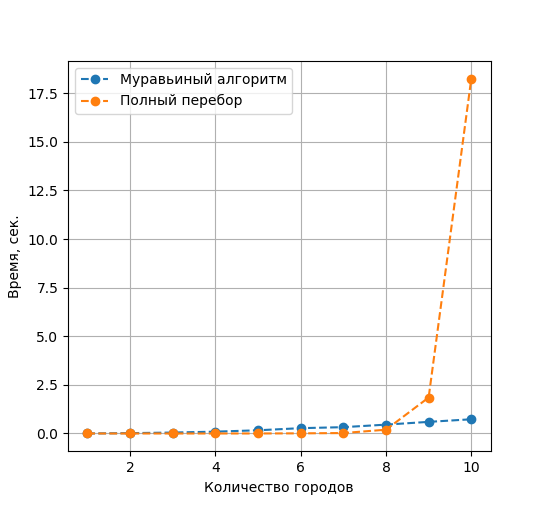


Рисунок 1. Сравнение времени работы муравьиного алгоритма с полным перебором в зависимости от количества вершин графа

Замер времени проводился с помощью стандартной библиотеки time в Python 3.7 и метода process\_time().

# Заключение

В результате выполнения данной работы рассмотрены и изучены понятия расстояния Левенштейна и расстояния Дамерау-Левенштейна. Реализованы два варианта алгоритма нахождения расстояния Левенштейна (рекурсивного и не рекурсивного вида). Сравнены их временные характеристики как следствие проведённых экспериментов. Реализован алгоритм нахождения расстояния Дамерау-Левенштейна. Были сделаны выводы об эффективности по времени рекурсивного и не рекурсивного вариантов алгоритмов. Применение рекурсивного варианта алгоритма нахождения расстояния Левенштейна неэффективно по времени. Рекомендуется использовать не рекурсивный алгоритм.

# Литература

[1] A. Colorni, M. Dorigo et V. Maniezzo, Distributed Optimization by Ant Colonies, actes de la première conférence européenne sur la vie artificielle, Paris, France, Elsevier Publishing, 134—142, 1991.

[2] M. Dorigo, Optimization, Learning and Natural Algorithms, PhD thesis, Politecnico di Milano, Italie, 1992.