Анализ Алгоритмов

Лабораторная работа №7

По теме “*Методы поиска подстроки в строке*”

Студент: Юмаев Артур

Группа: ИУ7-55

Оглавление

[Введение 3](#_Toc27400905)

[1. Аналитическая часть 4](#_Toc27400906)

[Обзор алгоритма 4](#_Toc27400907)

[Вывод 5](#_Toc27400908)

[2. Конструкторская часть 6](#_Toc27400909)

[Вывод 6](#_Toc27400910)

[3. Технологическая часть 7](#_Toc27400911)

[Вывод 8](#_Toc27400912)

[4. Исследовательская часть 9](#_Toc27400913)

[Заключение 11](#_Toc27400914)

[Литература 12](#_Toc27400915)

# Введение

Поиск подстроки в строке — одна из простейших задач поиска информации. Применяется в виде встроенной функции в текстовых редакторах, СУБД, поисковых машинах, языках программирования и т. п. В задачах поиска традиционно принято обозначать шаблон поиска как needle (с англ. — «иголка»), а строку, в которой ведётся поиск — как haystack (с англ. — «стог сена»). Обычно через Σ обозначается алфавит, на котором проводится поиск.

# 1. Аналитическая часть

В данном разделе будeт дано полное описание алгоритма Кнута-Морриса-Прата и алгоритма Бойера-Мура и их математические описания.

## Обзор алгоритма Кнута-Мориса-Прата

В задачах поиска информации одной из важнейших задач является поиск точно заданной подстроки в строке. Примитивный алгоритм поиска подстроки в строке основан на переборе всех подстрок, длина которых равна длине шаблона поиска, и посимвольном сравнении таких подстрок с шаблоном поиска.

Алгоритм Кнута-Морриса-Пратта является одним из первых алгоритмов с линейной оценкой в худшем случае.

Обозначим через:

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

где означает длину слова *word*.

Префикс-функция строки – это длина наибольшего префикса строки , который не совпадает с этой строкой и одновременно является ее суффиксом. Проще говоря, это длина наиболее длинного начала строки, являющегося также и ее концом. Для строки *S* удобно представлять префикс функцию в виде вектора длиной . Можно рассматривать префикс-функцию длины , положив . Пример приведен на таблице 1.

Таблица 1

Пример префикс функции для строки «abcdabca»

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | a | b | c | d | a | b | c | a |
|  | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 1 |

Наиболее полный алгоритм вычисления префикс-функции на псевдокоде показан на листинге 1.

|  |
| --- |
| **алг** префикс\_функция (**арг** строка S[1:i])  π(S,i)=k.  **Если** S[i+1]=S[k+1], то  π(S,i+1)=k+1.  **Иначе**  **Если** k=0, то  π(S,i+1)=0.  **Иначе**  Положить k:=π(S,k) и перейти к шагу 1. |

## Обзор алгоритма Бойера-Мура

## 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).

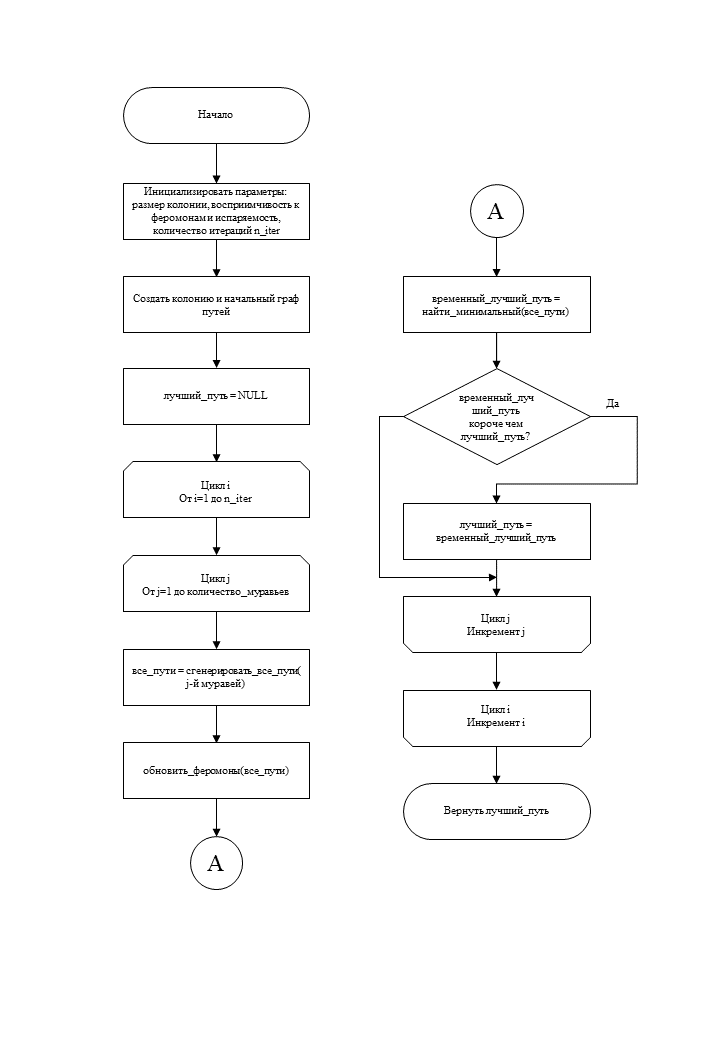


Рис. 1 – Схема муравьиного алгоритма

## 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 вершин графа 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.

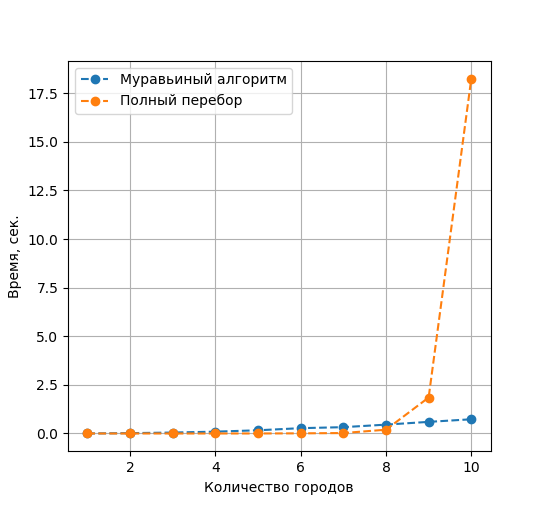


Рисунок 1. Сравнение времени работы муравьиного алгоритма с полным перебором в зависимости от количества вершин графа

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

В качестве параметризации были такие параметры, как скорость испарения феромонов и параметр , отвечающий за то, насколько муравьи чувствительны к феромонам. Исследование проводилось на предмет сходимости алгоритма, выражающееся в количестве операций, необходимых для нахождения оптимального пути обхода всех вершин графа. Исследование проводилось на графе в 9 вершин с 9 муравьями.

Таблица 1

Параметризация по параметрам “скорость испарения” и

|  |  |  |
| --- | --- | --- |
| Скорость испарения |  | Итераций до сходимости |
| 0.0 | 0.0 | 0 |
| 0.0 | 0.5 | 5 |
| 0.0 | 1.0 | 26 |
| 0.25 | 0.0 | 0 |
| 0.25 | 0.5 | 5 |
| 0.25 | 1.0 | 5 |
| 0.75 | 0.0 | 0 |
| 0.75 | 0.5 | 4 |
| 0.75 | 1.0 | 1 |
| 1.0 | 0.0 | 0 |
| 1.0 | 0.5 | 1 |
| 1.0 | 1.0 | 1 |

Как видно из таблице, чем меньше скорость испарения, тем в среднем быстрее муравьи находят самый короткий путь. Если муравьи сильно восприимчивы к феромонам и скорость испарения не высока, они в среднем в 9 раз быстрее находят оптимальный путь.

## Вывод

В данном разделе был проведен анализ задачи коммивояжера при решении разными способами. Опыт выявил, что при небольшом количестве городов () быстрее работает метод полного перебора, но при количестве городов больше 9, муравьиный алгоритм начинает выигрывать по времени. Также опыт показал, что при большом количестве городов (> 100), алгоритм не гарантирует нахождение наикратчайшего пути.

# Заключение

В результате выполнения данной работы были рассмотрены способы решения задачи коммивояжера 2 способами: с помощью полного перебора и с помощью муравьиного алгоритма. Результаты опытов приведены в разделе 4. В целом для количества городов ( рекомендуется использовать перебор, иначе муравьиный 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.