СИСТЕМНЫЙ АНАЛИЗ, УПРАВЛЕНИЕ И ОБРАБОТКА ИНФОРМАЦИИ / SYSTEM ANALYSIS, MANAGEMENT AND PROCESSING OF INFORMATION

DOI: https://doi.org/10.23670/IRJ.2023.135.82

СРАВНЕНИЕ МЕТОДОВ ПЕРЕСТАНОВКИ С ДВУМЯ ТИПАМИ СТРУКТУР ДАННЫХ

Научная статья

Лупин С.А.[1], Ай Мин Т.[2,] *

[1] ORCID : 0000-0001-8817-1362;

[2] ORCID : 0009-0004-8177-4266;

[1, 2] Национальный исследовательский университет "МИЭТ", Зеленоград, Российская Федерация

* Корреспондирующий автор (ayeminthike52[at]gmail.com)

Аннотация

Алгоритм полного перебора – широко используемый подход к решению задач дискретной оптимизации путем изучения всех возможных комбинаций параметров (векторов) для нахождения оптимального решения. В этой статье представлен анализ производительности различных методов перестановки, используемых для формирования векторов решений, (функция *next_permutation* в C++, метод *heap*, метод *next lexicographical permutation*) с двумя типами структур данных. Основная цель исследований заключается в повышении эффективности алгоритма полного перебора при решении задачи дискретной оптимизации. Метод *next lexicographical permutation* с использованием динамического массива показал лучшие результаты. Полученные результаты подтверждают возможность повышения эффективности алгоритма полного перебора за счет применения метода *next lexicographical permutation* для генерации векторов решений.

Ключевые слова: алгоритм полного перебора, функция next_permutation в C++, метод heap, метод next lexicographical permutation, структура данных, задача квадратичного назначения.

COMPARISON OF PERMUTATION METHODS WITH TWO TYPES OF DATA STRUCTURES

Research article

Lupin S.A.[1], Aye Min T.[2,] *

[1] ORCID : 0000-0001-8817-1362;

[2] ORCID : 0009-0004-8177-4266;

[1, 2] National Research University of Electronic Technology, Zelenograd, Russian Federation

* Corresponding author (ayeminthike52[at]gmail.com)

Abstract

The exhaustive permutation algorithm is a widely used approach to solve discrete optimization problems by exploring all possible combinations of parameters (vectors) to find the optimal solution. This work presents a performance analysis of different permutation methods used to generate solution vectors, (*next_permutation function* in C++, *heap* method, *next lexicographical permutation method*) with two types of data structures. The main objective of the research is to improve the efficiency of the exhaustive search algorithm in solving discrete optimization problem. The *next lexicographical permutation method* using dynamic set showed better results. The obtained results confirm the possibility of increasing the efficiency of the exhaustive search algorithm by using the *next lexicographical permutation method* for generating solution vectors.

Keywords: exhaustive algorithm, next_permutation function in C++, heap method, next lexicographical permutation method, data structure, quadratic assignment problem.

Введение

Генерация перестановок является важной задачей во многих областях, для которой известно множество методов решения. Классические методы, такие как подсчет, рекурсивный метод сверху вниз и итерационный метод снизу вверх [1], являются наиболее часто используемыми, но также существуют и другие алгоритмы, которые могут быть эффективными в определенных случаях.

Методы перестановки играют решающую роль в алгоритме полного перебора, определяющем порядок, в котором исследуются комбинации параметров. В статье [2] автор описывает сравнительное исследование производительности алгоритмов перестановок, таких как «Снизу вверх», лексикография и Johnson-trotter, которые были реализованы с использованием алгоритмов полного перебора (BFA) и ветвей и границ (BBA). Использование этих методов перестановки позволяет быстрее исследовать комбинации параметров, при этом гарантируя рассмотрение всех возможных решений [3]. Такая интеграция эффективных методов перестановки с алгоритмом полного перебора является новым подходом к снижению вычислительной сложности для крупномасштабных задач.

Структуры данных играют ключевую роль в хранении, доступе и обработке данных в памяти. Они являются важной частью решения научных задач, позволяя эффективно работать с большими объемами данных и выполнять операции оптимальным образом [4]. Выбор структуры данных может оказать значительное влияние на эффективность алгоритма. Оптимальная структура данных может сократить количество операций и повысить эффективность выполнения алгоритма [5]. Различные структуры данных обладают разными характеристиками, и выбор правильной из них может привести к повышению эффективности. В статье [6] автор проанализировал использование различных структур данных для повышения эффективности алгоритмов.

Для задач небольшого размера, комбинаторные алгоритмы [7] могут быть достаточно эффективными и позволяют найти оптимальное решение. Однако, с увеличением размера задачи, комбинаторные алгоритмы могут стать слишком медленными и несостоятельными. В таких случаях применение быстрых алгоритмов оптимизации может быть более эффективным.

В нашей статье показано, что использование динамического массива в методах перестановок позволяет значительно повысить эффективность процесса поиска решения в BFA. Алгоритм может быть распараллелен для использования вычислительной мощности современных архитектур параллельных вычислений, повышая его эффективность при решении крупномасштабных задач.

Методы перестановки и структурирования данных

Функция *next_permutation* (метод_1), предоставляемая библиотекой стандартных шаблонов C++, является полезным инструментом для решения задач комбинаторной оптимизации [8], анализа алгоритмов, основанных на перестановках. Функция генерирует следующую лексикографически большую перестановку заданной последовательности [9] и изменяет входную последовательность, переставляя элементы для получения следующей лексикографически большей перестановки. Функция начинается с наименьшей возможной перестановки и рекурсивно перебирает все возможные перестановки, увеличивая элементы по порядку, пока не достигнет наибольшей перестановки, возвращая значение ложь, когда перестановок больше нет. Эта функция использует лексикографический порядок элементов для определения следующей перестановки.

Метод *heap* (метод_2) генерирует перестановки с использованием рекурсивного обратного отслеживания, если длина равна 1, он выводит текущую перестановку. В противном случае, он генерируется рекурсивным методом, при котором каждый элемент замещается последним, а затем происходит рекурсивное создание перестановок для остальных элементов [10]. Метод чередует генерацию перестановок с нечетной длиной и четной длиной, в зависимости от того, является ли длина вектора четной или нечетной.

Метод – *next lexicographical permutation* (метод_3) является общим подходом для генерации следующей лексикографически большей перестановки элементов последовательности [11]. Алгоритм работает путем нахождения крайнего правого элемента, справа от которого находится меньший элемент. Затем этот крайний правый элемент заменяется наименьшим элементом справа, который больше текущего элемента. После этого происходят изменения элементов справа от точки поворота [12]. Этот процесс осуществляется шаг за шагом и генерирует следующую лексикографически большую перестановку.

В таблице 1 представлено сравнение быстродействия различных методов перестановки с двумя структурами данных для векторов размерностью $10 \le N \le 14$. Во всех случаях число генерируемых векторов составляет $N!$.

Таблица 1 - Сравнение методов перестановки с двумя структурами данных

DOI: https://doi.org/10.23670/IRJ.2023.135.82.1

Размер *N*	Структура данных	Время генерации векторов (сек)		
		Метод_1	*Метод_2*	*Метод_3*
10	Вектор	2,36	0,38	0,36
	Динамический массив	0,65	0,13	0,08
11	Вектор	23,39	3,94	3,34
	Динамический массив	6,61	1,32	0,88
12	Вектор	279,3	45,43	38,2
	Динамический массив	76,6	14,38	8,72
13	Вектор	3730,86	585,4	492,99
	Динамический массив	980,79	183,96	110,51
14	Вектор	51322,04	8222,44	6999,98
	Динамический массив	14035,65	2571,32	1572,83

Динамический массив и вектор используются для хранения коллекций элементов в языке программирования C++. Динамический массив – это массив, размер которого может быть изменен во время выполнения, а вектор представляет собой специализированную реализацию динамического массива, которая обеспечивает дополнительную функциональность и безопасность, такие как проверка границ и автоматическое изменение размера при необходимости.

Динамические массивы имеют фиксированный объем, но мы можем увеличить или уменьшить их размер по мере необходимости, выделив новую память и скопировав элементы [13]. Вектор является частью библиотеки стандартных шаблонов (STL), и он автоматически обрабатывает выделение памяти и изменение размера [14]. Динамические

массивы требуют ручного выделения и освобождения памяти с помощью *new* и *delete*, в то время как векторы управляют памятью автоматически, и вам не нужно обрабатывать память явно.

Избегая операций изменения размера, динамический массив может быть быстрее вектора при генерации лексикографической перестановки. Вектор обеспечивает безопасность, удобство и более высокий уровень абстракции, в то время как динамический массив обеспечивает более низкий уровень управления и может быть немного быстрее в некоторых конкретных случаях использования. Наши эксперименты показали, что динамический массив работает быстрее, чем вектор, для всех возможных комбинаций перестановок из-за меньших затрат памяти, улучшенной локализации кэша и прямого доступа к памяти.

Повышение эффективности BFA с помощью next lexicographical permutation

В статье [15] проанализированы рекурсивные и итерационные реализации алгоритмов поиска. В статье [16] автор представил сравнительный анализ между рекурсивным и итерационным методами генерации перестановок для решения задачи коммивояжера и показал, что итерационный алгоритм работает в 8-16 раз быстрее, чем рекурсивный алгоритм.

В нашей статье было исследовано применение *next lexicographical permutation* с итерационной технологией и динамическим массивом для повышения эффективности метода BFA при решении задачи квадратичного назначения (QAP). Данный метод генерирует перестановки с использованием итерационного подхода для поиска перестановки следующей непосредственно за данной перестановкой в лексикографическом порядке. Она включает в себя идентификацию элементов, которые необходимо заменить, а также реверсирования частей последовательности.

Алгоритм полного перебора и QAP

Алгоритм полного перебора гарантирует нахождение точного решения задачи квадратичного назначения путем оценки всех возможных комбинаций [17]. BFA легко реализуется и не требует сложных модификаций или настройки параметров, специфичных для конкретной задачи. Этот алгоритм универсален и может быть применен в различных областях, включая расположение объектов, планирование, оптимизацию сетей и кластеризацию данных [18].

В качестве оптимизационной задачи в наших исследованиях использовалась задача квадратичного назначения. Её математическая формулировка и условия задачи описаны в работе [19].

Повышение скорости генерации вариантов решений за счёт совершенствования операции перестановки снижает вычислительную сложность BFA при решении QAP.

Ниже представлен фрагмент псевдокода, который демонстрирует использование метода *next lexicographical permutation* с итерацией и динамическим массивом для эффективного перебора всех возможных комбинаций элементов вектора решения. Такой подход значительно уменьшает количество вариантов, которые необходимо рассмотреть.

```
1: void reverseOrder(int * p, int start, int end){
2:   while(start < end) {
3:     swap(p[start], p[end]);
4:       start++; end--;}}
//вычислительная функция для каждой перестановки
1: void calculationfun(int * p, int ** D, int ** R, int n){
2: computevalue = 0;
3:   for i = 0 to n do {
4:     for j = i + 1 to n do {
5:       computevalue+= R[i][j] * D[p[i]][p[j]];}}
6:   if(min ≥ computevalue){ min = computevalue }}
// Функция перестановки
1: void permutationfun(int * p, int ** D, int ** R, int n){
2: calculationfun(p, D, R, n); computevalue = 0; int k = -1
3: for i = n - 2 to i ≥ 0 do { if(p[i] < p[i + 1]) {
4:    k = i; break;}}
5:     if (k == -1) { return false; }
6:     int m = n - 1; while (p[k] ≥ p[m]) { m--; }
7:     swap(p[k], p[m]);
8:     reverseOrder(p, k + 1, n - 1);
9:     calculationfun(p, D, R, n);
10:    return true; }
```

Рисунок 1 - Фрагмент псевдокода

DOI: https://doi.org/10.23670/IRJ.2023.135.82.2

Основные результаты

Для проведения вычислительных экспериментов был использован персональный компьютер с процессором Intel® Core™ i3 9th поколения с тактовой частотой 3.1 ГГц. Эксперименты были выполнены в среде Visual Studio 2019. В таблице 2 показано время, затраченное на решение задачи с использованием метода BFA с различными методами перестановок, при решении QAP с использованием итерации и структур данных.

Таблица 2 - Время решения QAP с использованием BFA

DOI: https://doi.org/10.23670/IRJ.2023.135.82.3

Размер *N*	Структура данных	Время решения (сек)		
		Метод_1	*Метод_2*	*Метод_3*
14	Вектор	87,724	54,174	53,264
	Динамический массив	17,772	12,176	11,323

Полученные результаты подтверждают, что меньшее время решения обеспечивает метод *next lexicographical permutation* с динамическими массивами.

Заключение

Проведенные исследования показали важность операции перестановки элементов вектора решения для алгоритма полного перебора. Использование эффективного метода *next lexicographical permutation* с динамическим массивом, позволяет снизить вычислительную нагрузку BFA, делает его более практичным для реальных задач высокой размерности.

Конфликт интересов

Не указан.

Рецензия

Сообщество рецензентов Международного научно-исследовательского журнала
DOI: https://doi.org/10.23670/IRJ.2023.135.82.4

Conflict of Interest

None declared.

Review

International Research Journal Reviewers Community
DOI: https://doi.org/10.23670/IRJ.2023.135.82.4

Список литературы на английском языке / References in English

1. Song Xiaomei. Research on Permutation Generation Algorithm Based on Sorting / Song Xiaomei, Changxiu Cheng, Zhou Chenghu // The 2nd International Conference on Information Science and Engineering, 4-6 December 2010. — 2010. — pp. 7410-7413.

2. Youssef B. A Comparative Study on the Performance of Permutation Algorithm / Bassil Youssef // Journal of Computer Science & Research(JCSCR). — 2012. — vol. 1, no. 1. — pp. 7-19.

3. Naidan B. Permutation Search Methods are Efficient, Yet Faster Search is Possible / Bilegsaikhan Naidan, Leonid Boytsov, Eric Nyberg // Proceedings of the VLDB Endowment. — 2015. — vol. 8, no. 12. — pp 1-14.

4. How do data structures affect the design and implementation of algorithms in software engineering? // BGI Bhopal. — URL: https://bgibhopal.com/blog/how-do-data-structures-affect-the-design-and-implementation-of-algorithms-in-software-engineering/ (accessed 08.06.2023)

5. Khalfallah H.B. Data Structures and Their Impact on Performance / Héla Ben Khalfallah // medium.com. — URL: https://medium.com/coderbyte/data-structures-and-their-impact-on-performance-86ab8c8cb006 (accessed 08.06.2023)

6. Martins L. R. Algorithms and How to Choose the Right Data Structure / Leonardo Rodrigues Martins. — URL: https://blog.bitsrc.io/how-the-choice-of-data-structure-impacts-your-code-220d06c4ab96 (accessed 08.06.2023)

7. Loiola E. M. A Survey for the Quadratic Assignment Problem / Eliane Maria Loiola, Nair Maria Maia de Abreu, Paulo Oswaldo Boaventura-Netto [et al.] // European Journal of Oprational Research. — 2005. — pp. 657-690.

8. Find the next permutation in c++ // Prepbytes. — URL: https://www.prepbytes.com/blog/cpp-programming/find-the-next-permutation-in-cpp/ (accessed 08.06.2023)

9. Heap B.R. Permutations by Interchanges / B.R.Heap // The Computer Journal. — 1963. — vol. 6, no. 3. — pp. 293-298.

10. Verma N. Generating Permutations with Recursion / N. Verma. — URL: https://mathsanew.com/ (accessed 08.06.2023)

11. Hanafi A. Lexicographically Next Permutation (javascript version) / Abderrahmen Hanafi // Pulse. — URL: https://www.linkedin.com/pulse/lexicographically-next-permutation-javascript-version-hanafi (accessed 08.06.2023)

12. Genitrini A. Lexicographic Unranking of Combinations Revisited / Antoine Genitrini, Martin Pepin // MDPI Journals. — 2021. — pp. 1-25.

13. Thompson B. C++ Dynamic Allocation of Arrays with Example / Barbara Thompson. — URL: https://www.guru99.com/cpp-dynamic-array.html (accessed 08.06.2023)

14. Great Learning Team Vectors in C++ STL. — URL: https://www.mygreatlearning.com/blog/vectors-in-c/ (accessed 08.06.2023)

15. McCauley R. Recursion vs Lteration: An empirical study of comprehension revisited / Renee McCauley, Brain Hanks, Sue Fitzgerald [et al.] // 46th ACM Technical Symposium on Computer Science Education. — 2015. — pp. 350-355.

16. Kralev V. An Analysis of a Recursive and an Iterative Algorithm for Generating Permutations Methods for Traveling Salesman Problem / Velin Kralev // International Journal on Advanced Science Engineering and Information Technology. — 2017. — vol. 7, no. 5. — pp. 1685-1692.

17. Darwin, Implementation of Brute Force Algorithm in Quadratic Assignment Problem. — URL: https://dokumen.tips/documents/implementation -of-brute-force-algorithm-in-quadratic-rinaldimunirstmik.html?page=2 (accessed 08.06.2023)

18. Neris Y.G. Solving Quadratic Assignment Problem for Survivable Optical Networks with Genetic Algorithm / Yrui Giovan Neris, Marcia Helena Moreira Paiva, Claunir Pavan // 2019 SBMO/IEEE MTT-S International Microwave and Optoelectronics Conference (IMOC). — 2019.

19. Thike A. M. The Features of Using a Brute Force Algorithm for Solving a Quadratic Assignment Problem / Aye Min Thike, S.A. Lupin, S.A. Balabaev, // International Journal of Open Information Technologies. — 2023. — Vol. 11, No. 7. — pp. 60-66.