МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования
«Чувашский государственный университет имени И.Н. Ульянова»

Факультет информатики и вычислительной техники

Кафедра вычислительной техники

СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

Расчетно-графическая работа

Исследование поиска с возвратом

Выполнил:
студент группы ИВТ-41-20
Галкин Д.С.

Руководитель:
доцент Павлов Л.А.

Чебоксары 2022

Оглавление

Задание к РГР (вариант 7)

Задача о ферзях. Ферзи, угроза всех полей

Найти все способы расстановки минимального числа ферзей на шахматной доске размером $n \times n$ полей так, чтобы они держали под угрозой все поля доски.

Исследовать асимптотическую временную сложность решения задачи в зависимости от n.

Введение

Цель работы – закрепление теоретических знаний, полученных по данному курсу и смежным дисциплинам, и приобретение практических навыков формализации поставленной задачи, создания и использования эффективных структур данных и алгоритмов в прикладных задачах, теоретических и экспериментальных оценок эффективности алгоритмов.

Поставленная задача о ферзях относится к классу комбинаторных задач, которые требуют исчерпывающего поиска множества всех возможных решений, а алгоритмы решения имеют экспоненциальную вычислительную сложность. Одним из общих методов организации такого поиска является *поиск с возвратом* (backtracking) [3; 4; 5], который можно взять за основу для решения поставленной задачи. Поскольку ферзь атакует все поля в своей строке, своем столбце и диагоналях, очевидно, что на шахматной доске можно расставить максимум n не атакующих друг друга ферзей. Таким образом, поставленная задача сводится к определению возможности минимальной расстановки n не атакующих друг друга ферзей и, если расстановка возможна, к определению, сколькими способами это можно сделать. Если же расстановка n ферзей невозможна, то решается задача расстановки $(n - 1)$ ферзей и т.д.

В процессе выполнения РГР необходимо [4]:

- формализовать поставленную задачу (перейти от словесной неформальной постановки задачи к математической формулировке);
- приспосабливать общие методы и алгоритмы решения классов задач к решению конкретной задачи;
- проводить сравнительную оценку различных вариантов с целью выбора наиболее эффективных структур данных и алгоритмов их обработки;
- исследовать и оценивать теоретически (аналитически) и экспериментально методы сокращения перебора в комбинаторных задачах;
- оценивать аналитически и экспериментально эффективность предложенных в работе алгоритмов (временную и емкостную сложности);
- программно реализовать разработанные структуры данных и алгоритмы на одном из алгоритмических языков программирования.

1. Формализация задачи

Поскольку стоит задача определения всех вариантов расстановки n не атакующих друг друга ферзей, можно взять за основу общий алгоритм поиска с возвратом [4], приведенный на рис. 1.

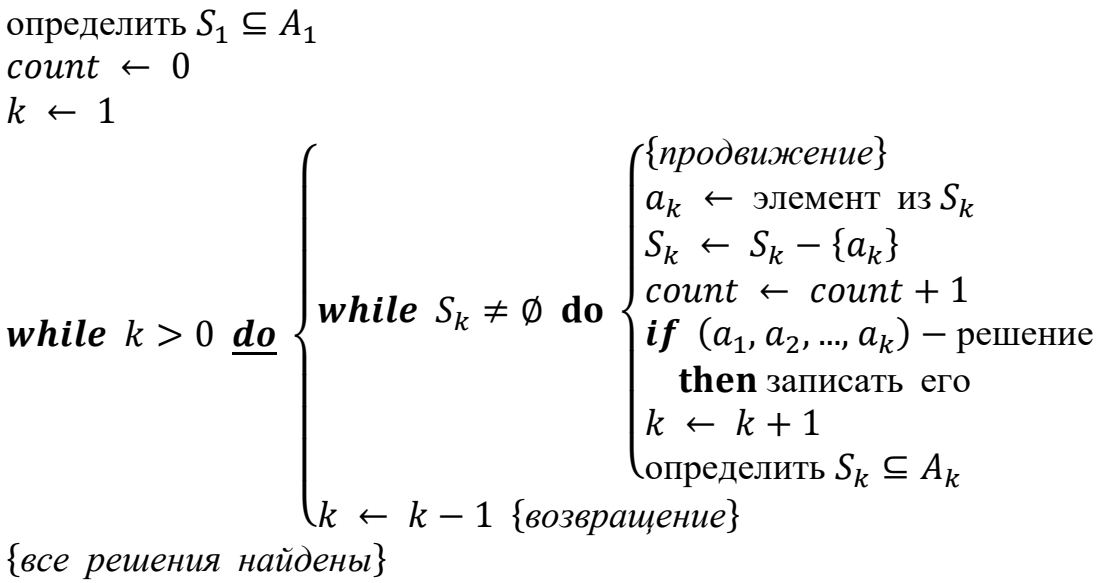


Рис. 1. Общий алгоритм поиска с возвратом

В общем случае предполагается, что решение задачи представляет собой вектор (a_1, a_2, …) конечной, но не определенной длины, удовлетворяющий некоторым ограничениям. Каждый элемент a_i является элементом конечного линейно упорядоченного множества A_i. Таким образом, при исчерпывающем поиске должны рассматриваться элементы множества $A_1 \times A_2 \times \ldots \times A_i$ для $i = 0, 1, 2, \ldots$ в качестве возможных решений. В качестве исходного частичного решения выбирается пустой вектор ( ) и на основе имеющихся ограничений определяется, какие элементы из множества A_1 являются кандидатами в a_1; подмножество таких кандидатов обозначим через S_1. В качестве a_1 выбирается наименьший элемент множества S_1; в результате получается частичное решение (a_1). В общем случае различные ограничения, описывающие решения, определяют, из какого подмножества S_k множества A_k должны выбираться кандидаты для расширения частичного решения от (a_1, a_2, …, a_{k-1}) до (a_1, a_2, …, a_{k-1}, a_k). Если частичное решение (a_1, a_2, …, a_{k-1}) не предоставляет возможностей для выбора элемента a_k, т. е. $S_k = \varnothing$, то необходимо вернуться и выбрать новый элемент a_{k-1}. Если новый элемент a_{k-1} выбрать невозможно, придется вернуться еще дальше и выбрать новый элемент a_{k-2} и т. д.

Процесс поиска с возвратом удобно представить в виде дерева поиска, в котором исследуемое подмножество множества $A_1 \times A_2 \times \ldots \times A_i$ для $i = 0, 1, 2, \ldots$ представляется следующим образом. Корню дерева (нулевой уровень) ставится в соответствие пустой вектор. Его сыновья образуют множество S_1 кандидатов для выбора a_1. В общем случае вершины k-го уровня образуют множества S_k кандидатов на выбор a_k при условии, что a_1, a_2, …, a_{k-1} выбраны так, как указывают предки этих вершин.

Переменная *count* в алгоритме не имеет принципиального значения для поиска, она носит информативный характер и служит для подсчета числа исследованных вершин в дереве поиска.

1.1. Абстрактные структуры данных для представления объектов

Сначала решим вопрос о представлении вектора решений. Очевидно, что все решения имеют одну и ту же фиксированную длину n, т. е. решение можно представить вектором (a_1, …, a_n). На первый взгляд, элемент a_k ($1 \le k \le n$) этого вектора должен представлять собой координату позиции, в которой размещается k-й ферзь, т. е. упорядоченную пару чисел, определяющих соответственно

номер строки и номер столбца. Однако поскольку в каждом столбце может находиться только один ферзь, то решение можно представить более простым вектором $(a_1, \ldots, a_n)$, в котором элемент a_k $(1 \le k \le n)$ есть номер строки ферзя, расположенного в столбце с номером k, т. е. координатой позиции является пара (a_k, k). Очевидно, что множества значений элементов a_k совпадают, т. е. $A_1 = \ldots = A_n = A = \{1, \ldots, n\}$.

1.2. Анализ ограничений и усовершенствований

Рассмотрим свойственные задаче ограничения. Одно ограничение, связанное с тем, что в столбце может находиться только один ферзь, учтено представлением вектора решения. Другое ограничение заключается в том, что в каждой строке может быть только один ферзь, поэтому если $i \ne j$, то $a_i \ne a_j$. Наконец, поскольку ферзи могут атаковать друг друга по диагонали, мы должны иметь $|a_i - a_j| \ne |i - j|$, если $i \ne j$. Таким образом, для того чтобы определить, можно ли добавить a_k для расширения частичного решения $(a_1, a_2, \ldots, a_{k-1})$ до $(a_1, a_2, \ldots, a_{k-1}, a_k)$, достаточно сравнить элемент a_k с каждым a_i, $i < k$. Эту проверку можно реализовать в виде функции STRIKE, представленной на рис. 2, которая отвечает на вопрос, включить данную позицию в подмножество S_k кандидатов на выбор a_k или нет.

```
function STRIKE(a, k) тип Boolean
false, если можно поставить ферзь в строку a столбца k
  px ← a[k]
  py ← k
  flag ← true
  for i ← 1 to k − 1 do
      x ← a[i]
      y ← i
      if isStrike(x, y, px, py) then flag ← true
  flag ← false
  STRIKE ← flag
```

Рис. 2. Функция STRIKE

Оценим, как влияют эти ограничения на процесс поиска. Если нет никаких ограничений, то на доске размером $n \times n$ существует $\binom{n^2}{n}$ (для $n = 8$ около $4{,}4 \times 10^9$) возможных способов расстановки n ферзей. Тот факт, что в каждом столбце может находиться только один ферзь, дает n^n расстановок (для $n = 8$ около $1{,}7 \times 10^7$). То, что в строку можно поставить только одного ферзя, говорит о том, что вектор $(a_1, \ldots, a_n)$ может быть решением только тогда, когда он является перестановкой элементов (1, 2, ..., n), что дает $n!$ возможных расстановок (для $n = 8$ около $4{,}0 \times 10^4$). Требование, что на диагонали может находиться только один ферзь, еще больше сокращает число возможных расстановок. Для последнего ограничения аналитическое выражение, позволяющее оценить число возможных расстановок, получить трудно, поэтому необходима экспериментальная оценка размеров дерева поиска. Например, для $n = 8$ дерево поиска содержит только 2056 вершин.

Таблица 1

Сокращение перебора в задаче о ферзях

Усовершенствование	Аналитическое выражение для оценки	Количество возможных расстановок n ферзей для n=8
Нет никаких ограничений	$\binom{n^2}{n}$	около $4{,}4 \times 10^9$ расстановок
Каждый столбец содержит не более одного ферзя	n^n	около $1{,}7 \times 10^7$ расстановок
Каждая строка содержит не более одного ферзя	$n!$	около $4{,}0 \times 10^4$ расстановок
Каждая диагональ содержит не более одного ферзя	Нет	Дерево поиска имеет 2056 узлов. Оценка экспериментальная

1.3. Разработка алгоритмов решения задачи

Поскольку нет необходимости в явном хранении подмножеств S_k, вычисляемое текущее значение элемента множества S_k, обозначенное через s_k, является элементом вектора $S = (s_1, \ldots, s_n)$. Тогда проверке условия $S_k \neq \varnothing$ в общем алгоритме (см. рис. 1) будет соответствовать условие $s_k \leq n$. В результате процедуру нахождения всех решений задачи о не атакующих друг друга ферзях на доске размера $n \times n$ можно формально представить алгоритмом, приведенным на рис. 3.

$$
\begin{array}{l}
k \leftarrow 1 \\
cell[k] \leftarrow 0 \\
cell[0] \leftarrow 0 \\
count \leftarrow 0 \\
\textbf{while}\ \ k > 0\ \ \textbf{do}
\begin{cases}
\textbf{if}\ (k == N)\ \textbf{then}
\begin{cases}
cell[k] \leftarrow cell[k] + 1 \\
\textbf{if}\ cell[k] > N\ \textbf{then}
\begin{cases}
\textbf{while}\ cell[k]\ >\ N \\
\quad \textbf{do}\ p--
\end{cases} \\
\quad else
\begin{cases}
if\ strike(cell, p) \\
\quad then\ k+;
\end{cases}
\end{cases} \\
\quad \textbf{while}\ \ s_k \leq n\ \textbf{and not}\ STRIKE(s_k,\ k)
\end{cases}
\end{array}
$$

Рис. 3. Алгоритм решения задачи о ферзях

Следует отметить, что данный алгоритм корректно обрабатывает и ситуацию, когда $k = n + 1$, поскольку вычисляемое значение s_{n+1} не меньше, чем $n + 1$, и, следовательно, множество S_{n+1} всегда пусто. Включение во внутренний цикл специальной проверки для предотвращения ситуации, когда значение k становится больше n, будет слишком дорогостоящим с точки зрения времени работы алгоритма. Поскольку в соответствии с алгоритмом возможна ситуация, когда $k = n + 1$, следует увеличить размер вектора $S = (s_1, \ldots, s_n)$ на единицу, т. е. $S = (s_1, \ldots, s_{n+1})$.

2. Исследование сложности выполнения алгоритмов

Аналитическое выражение для оценки вычислительной сложности алгоритмов решения комбинаторных задач удается получить редко, так как трудно предсказать, как взаимодействуют различные ограничения по мере появления их при продвижении вглубь дерева поиска. В подобных случаях, когда построение аналитической модели является трудной или вовсе неосуществимой задачей, можно применить *метод Монте-Карло* (метод статистических испытаний). Смысл этого метода в том, что исследуемый процесс моделируется путем многократного повторения его случайных реализаций. Каждая случайная реализация называется *статистическим испытанием*.

Рассмотрим применение метода Монте-Карло для экспериментальной оценки размеров дерева поиска. Идея метода состоит в проведении нескольких испытаний, при этом каждое испытание представляет собой поиск с возвратом со случайно выбранными значениями a_i. Предположим, что имеется частичное решение $(a_1, a_2, \ldots, a_{k-1})$ и что число выборов для a_k, основанное на том, вводятся

ли ограничения или осуществляется склеивание, равно $x_k = |S_k|$. Если $x_k \neq 0$, то a_k выбирается случайно из S_k и для каждого элемента вероятность быть выбранным равна $1/x_k$. Если $x_k = 0$, то испытание заканчивается. Таким образом, если $x_1 = |S_1|$, то $a_1 \in S_1$ выбирается случайно с вероятностью $1/x_1$; если $x_2 = |S_2|$, то при условии, что a_1 было выбрано из S_1, $a_2 \in S_2$ выбирается случайно с вероятностью $1/x_2$ и т. д. Математическое ожидание $x_1 + x_1x_2 + x_1x_2x_3 + x_1x_2x_3x_4 + \ldots$ равно числу вершин в дереве поиска, отличных от корня, т. е. оно равно числу случаев, которые будут исследованы алгоритмом поиска с возвратом. Существует доказательство этого утверждения [5].

Общий алгоритм поиска с возвратом легко преобразуется для реализации таких испытаний; для этого при $S_k = \varnothing$ вместо возвращения просто заканчивается испытание. Алгоритм оценки размера дерева поиска [3; 4; 5] приведен на рис. 4. Он осуществляет N испытаний для вычисления числа вершин в дереве. Операция $a_k \leftarrow rand(S_k)$ реализует случайный выбор элемента a_k из множества S_k.

```
count ← 0   // суммарное число вершин в дереве
for i ← 1 to N do
    sum ← 0   // число вершин при одном испытании
    product ← 1   // накапливаются произведения
    определить S_1 ⊆ A_1
    k ← 1
    while S_k ≠ ∅ do
        product ← product * |S_k|
        sum ← sum + product
        a_k ← rand(S_k)
        k ← k + 1
        определить S_k ⊆ A_k
    count ← count + sum
average ← count/N   // среднее число вершин в дереве
```

Рис. 4. Метод Монте-Карло для поиска с возвратом

Таким образом, каждое испытание представляет собой продвижение по дереву поиска от корня к листьям по случайно выбираемому на каждом уровне направлению. Поскольку в методе Монте-Карло отсутствует возврат, оценка размеров дерева выполняется за полиномиальное время.

Вычисление по методу Монте-Карло можно использовать для оценки эффективности алгоритма поиска с возвратом путем сравнения его с эталоном, полученным для задачи с меньшей размерностью.

Конкретизация этого алгоритма для задачи о ферзях представлена на рис. 5.

Результаты экспериментальных исследований алгоритма решения задачи о ферзях (см. рис. 3) представлены в табл. 2. В качестве эталона взят размер задачи 11×11, для которого выполнен как поиск с возвратом (определен фактический размер дерева поиска), так и оценка размеров дерева поиска методом Монте-Карло. Для размера задачи 12×12 применен только метод Монте-Карло, который позволил определить, что ожидаемое время выполнения поиска для доски размера 12 × 12 составит примерно 1272 мс. Для каждого из исследованных методом Монте-Карло размеров задачи проведено $N = 1000$ испытаний.

```
count ← 0   // суммарное число вершин в дереве
for i ← 1 to N do
```

$$
\begin{cases}
sum \leftarrow 0 \quad // \textit{ число вершин при одном испытании} \\
product \leftarrow 1 \quad // \textit{ накапливаются произведения} \\
S_1 \leftarrow \{1,\ 2,\ \ldots,\ n\} \\
k \leftarrow 1 \\
\textbf{while}\ \ |S_k| > 0\ \ \textbf{do}
\begin{cases}
product \leftarrow product * |S_k| \\
sum \leftarrow sum + product \\
a_k \leftarrow rand(S_k) \\
k \leftarrow k + 1 \\
// \text{ определить } S_k \subseteq A_k \\
S_k \leftarrow \emptyset \\
j \leftarrow 1 \\
\textbf{while}\ \ j \le n\ \ \textbf{do}
\begin{cases}
\textbf{if}\ \ QUEEN(j,k)\ \ \textbf{then}\ \ S_k \leftarrow S_k \cup \{j\} \\
j \leftarrow j + 1
\end{cases}
\end{cases} \\
count \leftarrow count + sum
\end{cases}
$$
$$
average \leftarrow count/N \quad // \textit{ среднее число вершин в дереве}
$$

Рис. 5. Конкретизация метода Монте-Карло для задачи о ферзях

Таблица 2

Оценка времени выполнения

Размер задачи	Метод Монте-Карло		Фактически		
	Число узлов	Порядок роста	Число узлов	Время	Порядок роста
8×8	–	–	1 873	60 мс	–
9×9	–	–	7 690	100 мс	в 4,1 раза
10×10	–	–	34 091	180 мс	в 4,4 раза
11×11	158 466	в 4,6 раза	161 566	435 мс	в 4,7 раза
12×12	827 789	в 5,2 раза	–	–	–

График, построенный по полученным экспериментальным данным, показан на рис. 6 (степенной изображена аппроксимирующая функция). Ось абсцисс – размер задачи, ось ординат – число размещений ферзей. Наиболее близкой аппроксимирующей функцией является функция $y = 0{,}0000x^{7.6488} = O(c^n)$ с величиной достоверности аппроксимации $R^2 = 0{,}961$. Полученные результаты подтверждают экспоненциальную вычислительную сложность решения задачи о ферзях. Что касается емкостной сложности, то она очевидна: для хранения вектора решений $(a_1, \ldots, a_n)$ требуется n ячеек памяти, для хранения вектора $S = (s_1, \ldots, s_{n+1})$ требуется $n + 1$ ячеек и одна ячейка для k, т.е. всего требуется памяти $n + n + 1 + 1 = 2n + 2 = O(n)$.

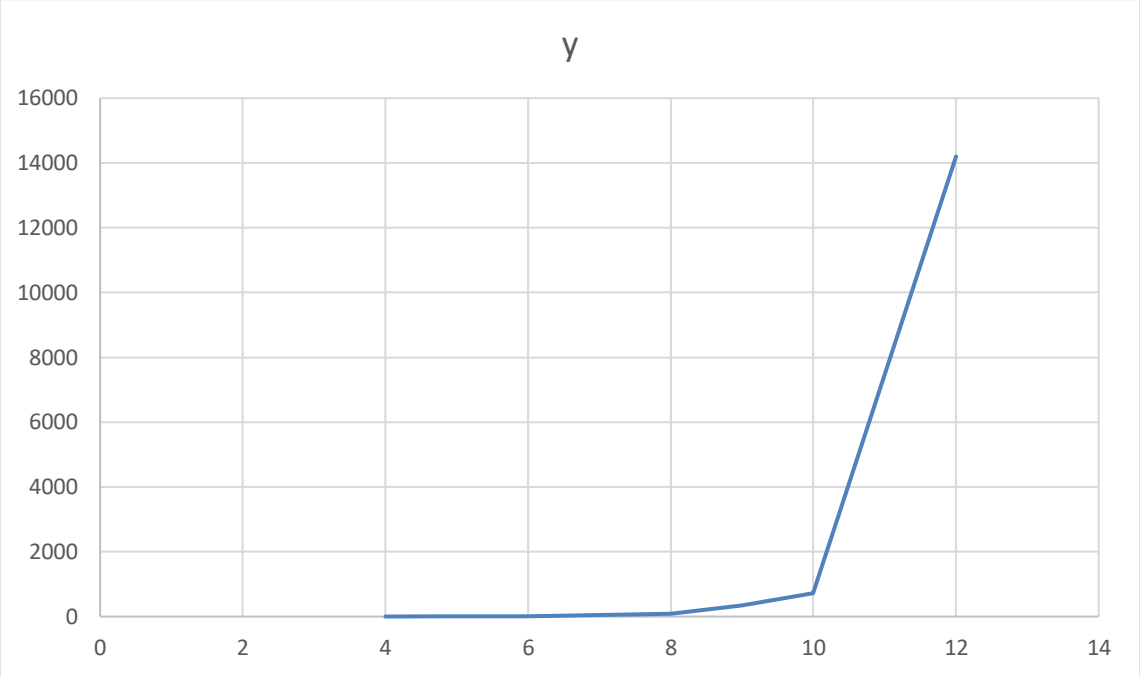


Рис. 6. График функции вычислительной сложности

3. Программная реализация алгоритмов

3.1. Выбор языка и среды программирования

В качестве среды разработки и языка программирования выбрана система программирования Java 11 SDK со средой разработки IntelliJ. Этот выбор сделан исходя из следующих соображений.

Прежде всего Java предназначен для профессиональных разработчиков, желающих очень быстро разрабатывать приложения, в частности, применяемый для создания серверных приложений. Сам язык полностью преобладает парадигмой объектно-ориентированного программирования.

Преимущества Java по сравнению с аналогичными программными продуктами:

- **Простота** – Четкие синтаксические правила и понятная семантика;
- **ООП** – В центре внимания находятся данные (объекты), интерфейсы;
- **Производительность** – Новые версии динамических компиляторов Java не уступают традиционным из других платформ;
- **Надежность** – Программы работают стабильно в любых условиях. Компилятор способен выявить ошибки до выполнения кода;
- **Независимость** – Важно лишь наличие исполняющей среды и JVM, запуск на любом ОС;

Основным конкурентом Java является .NET, технология работы с которой полностью совпадает с технологией Java. Язык C# практически полностью копирует основной функционал Java, но при этом переосмысляет их в своем виденье разработчиков.

Можно перечислить некоторые недостатки языка C# по сравнению с Java:

1. Сложная структура файлов проекта
2. Вечные проблемы с билдингом зависимостей в рантайме
3. VisualStudio – очень медленная

На выбор среды повлияло также то, что имеется опыт разработки других приложений в среде Java.

3.2. Разработка структурной схемы программы

Разработанная программа исследования алгоритма решения задачи о ферзях представляет собой единственный класс (**Chess**), включающий в себя метод решения самой задачи (Start) и метод

исследования методом Монте-Карло (**MethodMonteCarlo**). Эти методы между собой не взаимодействуют и запускаются отдельно при выборе соответствующего режима исследования. Таким образом, программа настолько проста, что нет необходимости в разработке специальной структурной схемы программы.

3.3. Реализация структур данных и алгоритмов

Программа реализована в виде единственного класса (**Chess**). Класс включает в себя независимые общедоступные методы:

- **Strike** – Проверяет различие координат между ферзями, чтобы не было наложения
- **Start** – Запускает основную часть алгоритма, решает данную задачу
- **MethodMonteCarlo** – Исследование методом Монте-Карло

Так же приватные методы, которые выполняют отдельные куски вычислений программного кода:

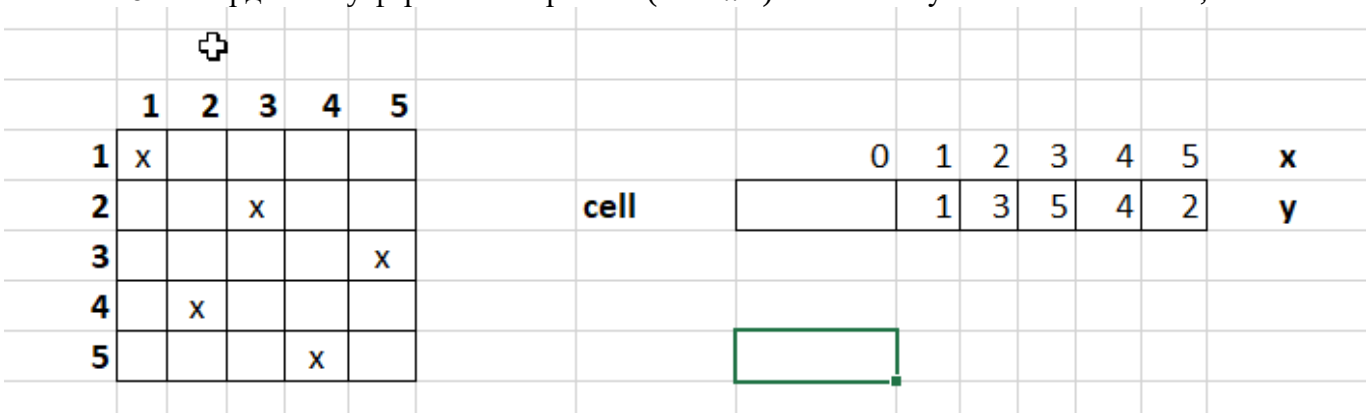
- **isStrike** – Проверка сражаются ли ферзи между собой
  - **isStrikeHorizontalAndVertical** – Проверка на сражение по вертикали и горизонтали
  - **isStrikeMainDiagonal** – Проверка на сражение по главной диагонали
  - **isStrikeSecondaryDiagonal** – Проверка на сражение по побочной диагонали
- **show** – вывод всех вариантов размещений

Поля, которые служат для реализации программного кода и хранения информации:

- cell – Представляет координаты правильного расположения ферзей;
- array – Представляет массив координат cell
- N – Размер доски
- countNode – Число узлов

В массиве **cell** координаты размещаемых ферзей формируются по следующему принципу:

- Координате x ферзя с номерами k (k = 1..N) соответствует значение cell[k];
- Координате y ферзя с номерами k (k = 1..N) соответствует само значение k;



p – В зависимости от ситуации, указатель будет движется право или влево по индексам x. Рассмотрим массив на основе указателя для N = 5;

- 1 – Начальное состояние. Это есть состояние массива cell, при котором начинается итерационный процесс;
- 2 – Промежуточное состояние. Это есть некоторый промежуточный вариант размещения, когда ещё не все ферзи размещены и размещаются k-й ферзь;
- 3 – Вариант размещения – этот вариант массива cell, в котором сформированно искомое размещение (случай, когда все N ферзей не бьют друг друга);
- 4 – Окончание итерационного процесса (p = 0). Итерационный процесс завершается, если после перебора всех возможных вариантов p двигается влево (p--) и становится равным 0;

Заключение

Результатами вычислений для доски размером 7 × 7 (n = 7) являются: всего решений 40; время решения 30 мс. В качестве примера на рис. 7 приведены 3 решения из 40.

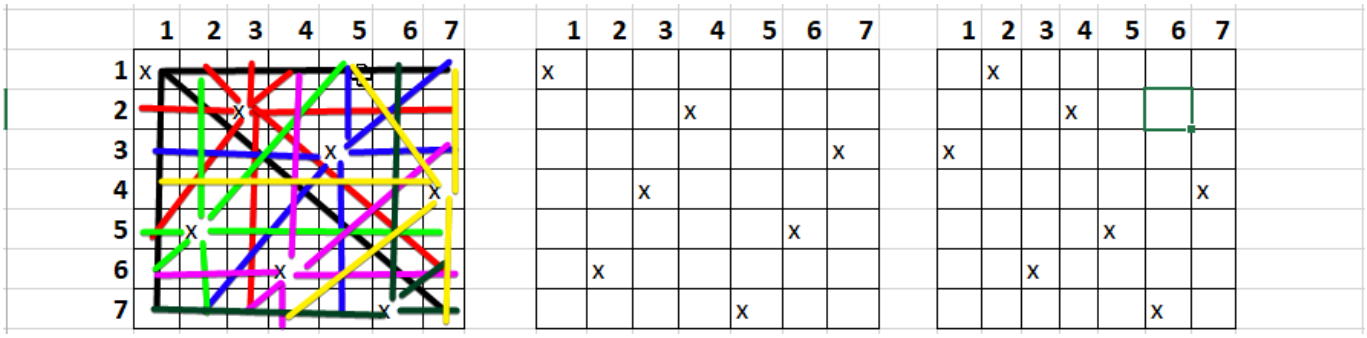


Рис. 7. Примеры трех решения задачи о ферзях для размера 7 × 7

В процессе выполнения расчетно-графической работы:

- формализована поставленная задача;
- общий алгоритм поиска с возвратом приспособлен к решению задачи о ферзях;
- проведена сравнительная оценка различных вариантов с целью выбора наиболее эффективных структур данных и алгоритмов их обработки;
- исследованы и оценены теоретически (аналитически) и экспериментально использованные методы сокращения перебора;
- экспериментально оценена эффективность предложенных в работе алгоритмов;
- программно реализованы разработанные структуры данных и алгоритмы.

В результате выполнения работы закреплены теоретические знания, полученных по данному курсу и смежным дисциплинам, приобретены практические навыки формализации задач, создания и использования эффективных структур данных и алгоритмов, теоретических и экспериментальных оценок эффективности алгоритмов.

Список использованной литературы

1. *Кевин, У.* Алгоритмы на Java / С. Роберт – 4-е изд. – М.: Вильямс, 2013.

3. *Павлов, Л.А.* Структуры и алгоритмы обработки данных: учеб. пособие / Л.А. Павлов. – Чебоксары: Изд-во Чуваш. ун-та, 2008. – 252 с.

4. Структуры и алгоритмы обработки данных: метод. указания к выполнению расчетно-графической работы / сост. Л.А. Павлов. – Чебоксары: Изд-во Чуваш. ун-та, 2014. – 24 с.

Приложение

```
/************************ Данные ************************/
/*
* Представляет координаты правильного расположения ферзей
* index = x
* item = y
 */
private int cell[];

/*
* Представляет массив координат cell[]
 */
private ArrayList<int[]> array;

/*
* Размерность доски
 */
private int N;

/*
* Число узлов
 */
private int countNode;
```

Рис. 8. Объявление всех переменных класса с их описанием

```java
/*
 * Проверка на различие координат между ферзями, чтобы не было наложения
 * @param cell[]
 * @param p - указатель на координаты
 * @return {Boolean}
 */
private boolean strike (int cell[], int p) {

    // доп.переменные для вычислений
    int px, py, x, y;

    px = cell[p];
    py = p;

    for (int i = 1; i <= p - 1; i++) {
        x = cell[i];
        y = i;
        if (isStrike(x, y, px, py)) return true;
    }
    return false;
}
```

Рис. 9. Метод strike – реализация и описание

```java
/*
* Проверка на сражение ферзей
* @param x1, y1 - координаты первого ферзя
* @param x2, y2 - координаты второго ферзя
* @return {Boolean}
 */
private boolean isStrike (int x1, int y1, int x2, int y2) {
    if (isStrikeHorizontalAndVertical(x1, y1, x2, y2)) return true;

    int tx = 0, ty = 0; // доп.переменные для вычислений

    if (isStrikeMainDiagonal(x1, y1, x2, y2, tx, ty)) return true;
    if (isStrikeSecondaryDiagonal(x1, y1, x2, y2, tx, ty)) return true;

    return false;
}
```

Рис. 9. Метод isStrike – реализация и описание

```
/*
* Горизонталь, вертикаль
* @param {isStrike}
* @return {Boolean}
 */
private boolean isStrikeHorizontalAndVertical (int x1, int y1, int x2, int y2) {
    return (x1 == x2) || (y1 == y2);
}
```

Рис. 9. Метод isStrikeHorizontalAndVertical – реализация и описание

```
/*
* Главная диагональ
* @param {isStrike}
* @return {Boolean}
 */
private boolean isStrikeMainDiagonal (int x1, int y1, int x2, int y2, int tx, int ty) {

    // Влево-вверх
    tx = x1 - 1;
    ty = y1 - 1;
    while ((tx ⩾ 1) && (ty ⩾ 1)) {
        if ((tx == x2) && (ty == y2)) return true;
        tx--;
        ty--;
    }

    // Вправо-вниз
    tx = x1 + 1;
    ty = y1 + 1;
    while ((tx ⩽ N) && (ty ⩽ N)) {
        if ((tx == x2) && (ty == y2)) return true;
        tx++;
        ty++;
    }
    return false;
}
```

Рис. 10. Метод isStrikeMainDiagonal – реализация и описание

```java
/*
 * Побочная диагональ
 * @param {isStrike}
 * @return {Boolean}
 */
private boolean isStrikeSecondaryDiagonal (int x1, int y1, int x2, int y2, int tx, int ty) {

    // Вправо-вверх
    tx = x1 + 1;
    ty = y1 - 1;
    while ((tx ⩽ N) && (ty ⩾ 1)) {
        if ((tx == x2) && (ty == y2)) return true;
        tx++;
        ty--;
    }

    // Влево-вниз
    tx = x1 - 1;
    ty = y1 + 1;
    while ((tx ⩾1) && (ty ⩽ N)) {
        if ((tx == x2) && (ty == y2)) return true;
        tx--;
        ty++;
    }
    return false;
}
```

Рис. 11. Метод isStrikeSecondaryDiagonal – реализация и описание

```java
/*
 * Запуск алгоритма
 * @param N - размерность доски
 */
public void start(int n) {
    N = n;
    cell = new int[n + 1];
    countNode = 0;
    int p; // Номер размещаемого ферзя
    int k; // кол-во вариантов размещения

    // Начальная настройка
    p = 1;
    cell[p] = 0;
    cell[0] = 0;
    k = 0;
```

Рис. 12. Метод start – инициализация промежуточных данных

```
// Цикл формирования размещения
while (p > 0) {
    cell[p] += 1;
    if (p == N) {
        if (cell[p] > N) {
            while (cell[p] > N) { p--; countNode++;}
        } else {
            if (!strike(cell, p)) {
                array.add(cell.clone());
                k++;
                p--;
            }
        }
    } else {
        if (cell[p] > N) {
            while (cell[p] > N) { p--; countNode++;}
        } else {
            if (!strike(cell, p)) {
                p++;
                cell[p] = 0;
            }
        }
    }
}
```

Рис. 13. Метод start – работа алгоритма

```
    if (k > 0) {
        System.out.println("Кол-во вариантов размещения = " + k);
    }

    System.out.println("Число узлов: " + countNode);
    show();

}
```

Рис. 14. Метод start – вывод результата

```
int methodMonteCarlo() {
    int count = 0; // суммарное число вершин в дереве
    countNode = 0;
    int n = 1000;
    cell = new int[N + 1];
    int sumTime = 0;
    for (int i = 1; i <= n; i++) {
        long sum = 0; // число вершин при одном испытании
        int product = 1; // накапливаются произведения
          var start = System.currentTimeMillis();
        int p = 1;
        countNode = 0;
        for (int j = 1; j < N + 1; j++) {
            cell[j] = 0;
        }
        cell[p] = 0;
        cell[0] = 0;
        int k = 0;
```

Рис. 15. Метод methodMonteCarlo – инициализация промежуточных данных

```
    while (p > 0) {

        product *= Math.abs(p);
        sum += product;

        cell[p] += 1;
        if (p == N) {
            if (cell[p] > N) {
                while (cell[p] > N) { p--; countNode++;}
            } else {
                if (!strike(cell, p)) {
                    array.add(cell.clone());
                    k++;
                    p--;
                }
            }
        } else {
            if (cell[p] > N) {
                while (cell[p] > N) { p--; countNode++;}
            } else {
                if (!strike(cell, p)) {
                    p++;
                    cell[p] = 0;
                }
            }
        }
    }
```

Рис. 16. Метод methodMonteCarlo – реализация алгоритма