

**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.

## Оглавление

Задание к РГР (вариант 7).....	3
Введение.....	4
1. Формализация задачи.....	5
1.1. Абстрактные структуры данных для представления объектов.....	5
1.2. Анализ ограничений и усовершенствований.....	6
1.3. Разработка алгоритмов решения задачи .....	7
2. Исследование сложности выполнения алгоритмов .....	7
3. Программная реализация алгоритмов .....	10
3.1. Выбор языка и среды программирования.....	10
3.2. Разработка структурной схемы программы.....	10
3.3. Реализация структур данных и алгоритмов.....	11
Заключение.....	12
Список использованной литературы .....	12

### 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);
- приспособлять 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.

```

определить  $S_1 \subseteq A_1$ 
count  $\leftarrow 0$ 
 $k \leftarrow 1$ 

while  $k > 0$  do {
    while  $S_k \neq \emptyset$  do {
        {продвижение}
         $a_k \leftarrow$  элемент из  $S_k$ 
         $S_k \leftarrow S_k - \{a_k\}$ 
        count  $\leftarrow$  count + 1
        if  $(a_1, a_2, \dots, a_k)$  – решение
            then записать его
             $k \leftarrow k + 1$ 
        определить  $S_k \subseteq A_k$ 
    }
     $k \leftarrow k - 1$  {возвращение}
}
{все решения найдены}

```

Рис. 1. Общий алгоритм поиска с возвратом

В общем случае предполагается, что решение задачи представляет собой вектор  $(a_1, a_2, \dots)$  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  $A_1 \times A_2 \times \dots \times A_i$  для  $i = 0, 1, 2, \dots$  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  $(a_1, a_2, \dots, a_{k-1})$  до  $(a_1, a_2, \dots, a_{k-1}, a_k)$ . Если частичное решение  $(a_1, a_2, \dots, a_{k-1})$  не предоставляет возможностей для выбора элемента  $a_k$ , т. е.  $S_k = \emptyset$ , 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  $A_1 \times A_2 \times \dots \times A_i$  для  $i = 0, 1, 2, \dots$  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  $a_1, a_2, \dots, a_{k-1}$  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  $(a_1, \dots, a_n)$ . На первый взгляд, элемент  $a_k$  ( $1 \leq k \leq n$ ) 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  $(a_1, \dots, a_n)$ , в котором элемент  $a_k$  ( $1 \leq k \leq n$ ) 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.  $A_1 = \dots = A_n = A = \{1, \dots, n\}$ .

## 1.2. Анализ ограничений и усовершенствований

Рассмотрим свойства задачи 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  $i \neq j$ , то  $a_i \neq a_j$ . Наконец, поскольку ферзи могут атаковать друг друга по диагонали, мы должны иметь  $|a_i - a_j| \neq |i - j|$ , если  $i \neq j$ . Таким образом, для того чтобы определить, можно ли добавить  $a_k$  для расширения частичного решения  $(a_1, a_2, \dots, a_{k-1})$  до  $(a_1, a_2, \dots, a_{k-1}, a_k)$ , 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( $a, k$ ) min Boolean
false, если можно поставить ферзь в строку  $a$  столбца  $k$ 
   $px \leftarrow a[k]$ 
   $py \leftarrow k$ 
   $flag \leftarrow true$ 

  for  $i \leftarrow 1$  to  $k - 1$  do  $\left\{ \begin{array}{l} x \leftarrow a[i] \\ y \leftarrow i \end{array} \right.$ 
    if  $isStrike(x, y, px, py)$  then  $flag \leftarrow false$ 

   $flag \leftarrow false$ 
  STRIKE  $\leftarrow flag$ 

```

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  $(a_1, \dots, a_n)$  может быть решением только тогда, когда он является перестановкой элементов  $(1, 2, \dots, n)$ , 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  $S = (s_1, \dots, s_n)$ . Тогда проверке условия  $S_k \neq \emptyset$  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.

```

 $k \leftarrow 1$ 
 $cell[k] \leftarrow 0$ 
 $cell[0] \leftarrow 0$ 
 $count \leftarrow 0$ 

while  $k > 0$  do
     $cell[k] \leftarrow cell[k] + 1$ 
    if  $(k == N)$  then
        if  $cell[k] > N$  then
            while  $cell[k] > N$ 
                do  $cell[k] \leftarrow cell[k] - 1$ 
            else
                if strike( $cell, p$ )
                    then  $k++$ ;
        else
            if  $cell[k] > N$  then
                while  $cell[k] > N$ 
                    do  $cell[k] \leftarrow cell[k] - 1$ 
            else
                if strike( $cell, p$ )
                    then  $k++$ ;

```

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  $S = (s_1, \dots, s_n)$  на единицу, т. е.  $S = (s_1, \dots, s_{n+1})$ .

## 2. Исследование сложности выполнения алгоритмов

Аналитическое выражение для оценки вычислительной сложности алгоритмов решения комбинаторных задач удастся 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  $(a_1, a_2, \dots, a_{k-1})$  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  $x_1 + x_1x_2 + x_1x_2x_3 + x_1x_2x_3x_4 + \dots$  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  $S_k = \emptyset$  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  $a_k \leftarrow \text{rand}(S_k)$  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  $S_1 \subseteq A_1$ 
        k ← 1
        while  $S_k \neq \emptyset$  do
            {
                product ← product *  $|S_k|$ 
                sum ← sum + product
                 $a_k \leftarrow \text{rand}(S_k)$ 
                k ← k + 1
                определить  $S_k \subseteq A_k$ 
            }
        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  $11 \times 11$ , для которого выполнен как поиск с возвратом (определен фактический размер дерева поиска), так и оценка размеров дерева поиска методом Монте-Карло. Для размера задачи  $12 \times 12$  применен только метод Монте-Карло, который позволил определить, что ожидаемое время выполнения поиска для доски размера  $12 \times 12$  составит примерно 451 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

```



```

sum ← 0 // число вершин при одном испытании
product ← 1 // накапливаются произведения
S1 ← {1, 2, ..., n}
k ← 1
while |Sk| > 0 do
    {
        product ← product * |Sk|
        sum ← sum + product
        ak ← rand(Sk)
        k ← k + 1
        // определить Sk ⊆ Ak
        Sk ← ∅
        j ← 1
        while j ≤ n do {if STRIKE(j, k) then Sk ← Sk ∪ {j}}
        j ← j + 1
    }
count ← count + sum
average ← count/N // 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	—	—	2 056	1,14 мс	—
9×9	—	—	8 393	4,57 мс	в 4 раза
10×10	—	—	35 538	19,40 мс	в 4,2 раза
11×11	161 124	в 4,5 раза	166 925	92,04 мс	в 4,7 раза
12×12	825 987	в 4,9 раза	—	—	—

График, построенный по полученным экспериментальным данным, показан на рис. 6 (тонкой линией изображена аппроксимирующая функция). Ось абсцисс – размер задачи, ось ординат – размер дерева поиска. Наиболее близкой аппроксимирующей функцией является функция  $y = 0,012e^{1,4982n} = O(c^n)$  с величиной достоверности аппроксимации  $R^2 = 0,9992$ . Полученные результаты подтверждают экспоненциальную вычислительную сложность решения задачи о ферзях. Что касается емкостной сложности, то она очевидна: для хранения вектора решений  $(a_1, \dots, a_n)$  требуется  $n$  ячеек памяти, для хранения вектора  $S = (s_1, \dots, s_{n+1})$  требуется  $n + 1$  ячеек и одна ячейка для  $k$ , т.е. всего требуется памяти  $n + n + 1 + 1 = 2n + 2 = O(n)$ .

**Примечание для студента.** В качестве оси ординат можно использовать время вычислений. В данном отчете используется размер дерева поиска, поскольку время вычислений при поиске с возвратом примерно пропорционально числу вершин в дереве поиска.

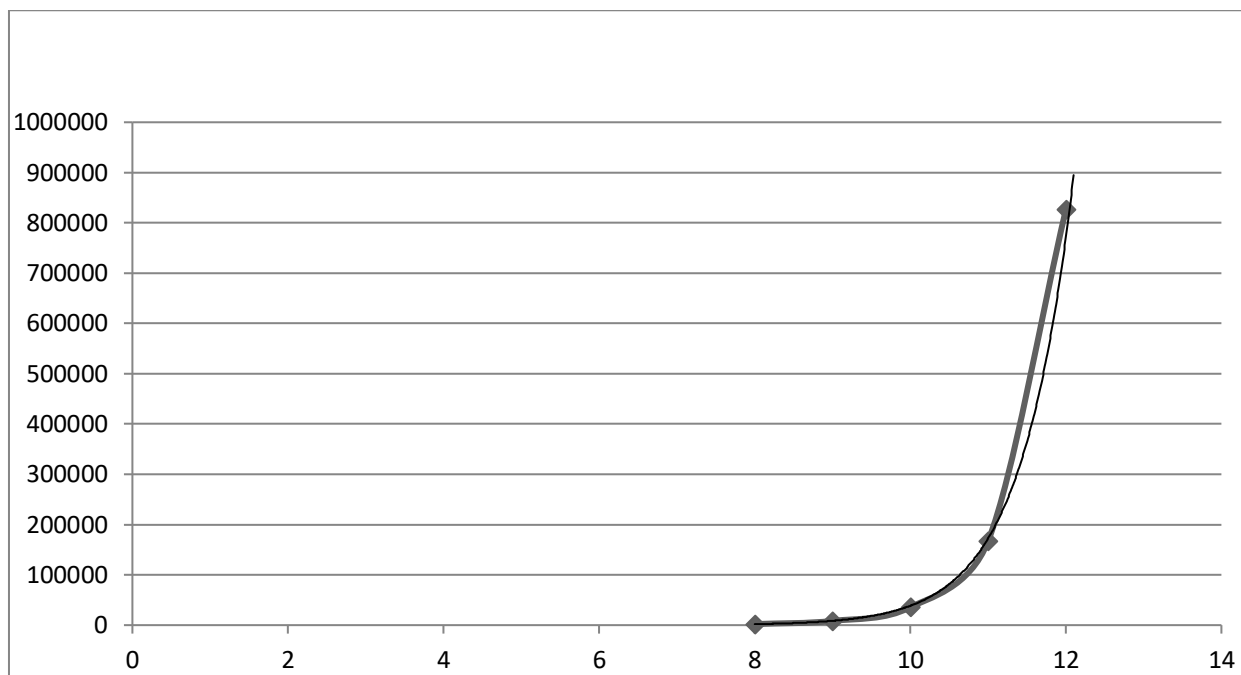


Рис. 6. График функции вычислительной сложности

### 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 программный модуль (**unit** Queen), включающий в себя процедуру решения

самой задачи (QueenTask) и процедуру исследования методом Монте-Карло (MonteCarlo). Эти процедуры 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 модуля (**unit** Queen). Модуль включает в себя независимые процедуры:

**procedure** QueenTask – решение задачи о ферзях;

**procedure** MonteCarlo – исследование методом Монте-Карло.

В главной (и единственной) форме модуля предусмотрен визуальный компонент edSize для ввода размера шахматной доски (свойство Text компонента типа TEdit). Для отображения результатов предусмотрены визуальные компоненты:

- edMonte – число исследованных вершин методом Монте-Карло (свойство Text компонента типа TEdit);

- edNode – количество фактически исследованных вершин (свойство Text компонента типа TEdit);

- edRez – количества решений (свойство Text компонента типа TEdit);

- edTime – время вычислений (свойство Text компонента типа TEdit);

- memoRez – решения (свойство Lines компонента типа TMemo).

Для вызова процедур QueenTask и MonteCarlo предусмотрены соответствующие кнопки «Задача» и «Монте-Карло» (компоненты типа TButton).

Вектор решения  $(a_1, \dots, a_n)$  и подмножества кандидатов для включения в решение представлены соответственно как массивы a и s:

a: **array** [1..MaxSize] **of** integer;

s: **array** [1..MaxSize+1] **of** integer;

Именованная константа MaxSize устанавливает предельно допустимый размер шахматной доски.

Алгоритм функции *QUEEN* (см. рис. 2), которая отвечает на вопрос, включить данную позицию в подмножество кандидатов на выбор  $a_k$  или нет, реализован функцией

**function** QueenPlace (ak, k: integer): Boolean;

## Заключение

Результатами вычислений для доски размером  $8 \times 8$  ( $n = 8$ ) являются: всего решений 92; количество исследованных вершин дерева поиска 2056; время решения 1,14 мс. В качестве примера на рис. 7 приведены 3 решения из 92.

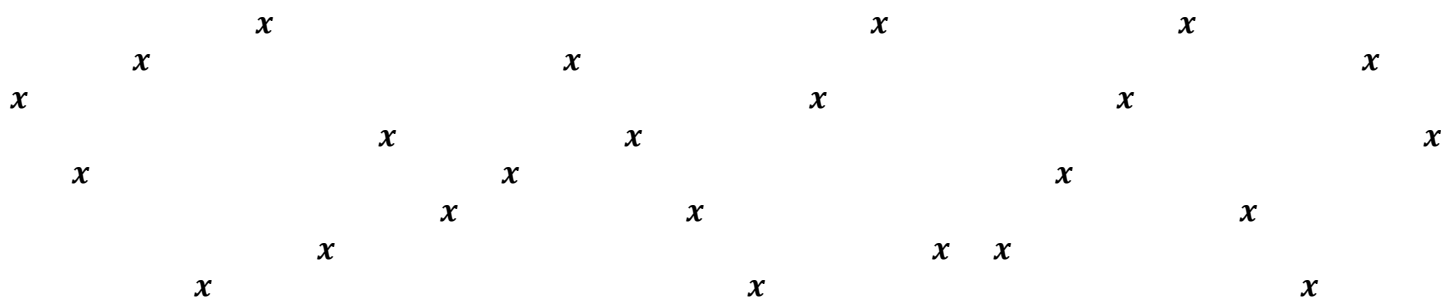


Рис. 7. Примеры трех решения задачи о ферзях для размера  $8 \times 8$

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. Кормен, Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. – 2-е изд. – М.: Вильямс, 2011. – 1296 с.
2. Липский, В. Комбинаторика для программистов / В. Липский. – М.: Мир, 1988. – 213 с.
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.
5. Рейнгольд, Э. Комбинаторные алгоритмы. Теория и практика / Э. Рейнгольд, Ю. Нивергельт, Н. Део. – М.: Мир, 1980. – 476 с.

**Примечание для студентов.** В данном отчете нет текстов программ в качестве приложения. У Вас приложение должно быть обязательно. При этом следует включать в приложение модули (процедуры, функции, объявления переменных, определение классов), связанные с решением задачи, а не включать все подряд, что генерирует визуальная среда.