МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ 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
Васильев Е. Ю

Руководитель:
доцент Павлов Л.А.

Чебоксары 2022

Оглавление

Задание к РГР (вариант 3)

Маршруты коня.

На шахматной доске размером $n \times n$ определить все возможные маршруты коня, начинающиеся на одном заданном поле шахматной доски и оканчивающиеся на другом. Никакое поле не должно встречаться в одном маршруте дважды.
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.

К одним из наиболее интересных задач относятся задачи, которые относятся к развитию интеллекта, в которых решение задачи ищется методом «проб и ошибок». При этом имеет место перебор при поиске решения, продолжительность которого может быть сокращена за счет применения тех или иных эвристических правил – методов оптимизации.

Была выбрана очень известная задача – обход шахматной доски конем.

Задача состоит в том, что шахматный конь ходит буквой «Г», т.е. две клеточки вверх или вниз в одну клеточку вправо или влево. Либо так: две клеточки вверх или вниз.

Необходимо найти для одного коня такой путь, чтобы он попал на другую клетку пустой доски размером $n \times n$, ни разу не попав туда, где он уже был. Разработаем программу, реализующую эту задачу и визуализирующую решение, т.е. выводящую на экран изображение доски с последовательным отображением позиций, посещаемых конем.

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

Эта глава посвящена самой интересной задаче о коне, пожалуй, вообще самой известной шахматно-математической задаче. Она заключается в нахождении маршрута коня на шахматной доске.

В литературе эту задачу обычно называют просто задачей о ходе коня. Особая популярность задачи объясняется тем, что в 18-19 веке ею занимались многие крупные математики, в том числе великий Леонард Эйлер, посвятивший ей большой мемуар. Хотя задача была известна и до Эйлера, лишь он впервые обратил внимание на ее математическую сущность, поэтому задачу часто связывают с его именем.

Значительно труднее проблема, состоящая не в отыскании определенного маршрута коня, а в нахождении всех маршрутов и подсчете их числа. Эта задача не решена до сих пор, и шансов на успех немного. Математик Ф.Миндинг, подошедший к проблеме с алгебраической точки зрения, предложил метод, позволяющий вывести формулу для числа всех решений, однако вычисления, которые следует при этом произвести, практически неосуществимы, и поэтому метод Миндинга представляет лишь теоретический интерес.

Литература, посвященная задаче о ходе коня, весьма обширна. Те или иные методы нахождения маршрутов можно найти в самых разнообразных источниках. Среди них следует выделить книгу Крайчика «Проблема коня», которая целиком посвящена этой теме. Поговорим о наиболее интересных, на мой взгляд, методах нахождения маршрутов коня.

Рамочный метод Мунка и Коллини. Шахматную доску делим на две части: внутреннюю, состоящую из 16 полей, и внешнюю, имеющую форму рамы и состоящую из 48 полей. На полях внутреннего квадрата запишем заглавные буквы A, B, C, D – так, чтобы каждая из них, повторенная четре раза, образовала квадрат или ромб, по всем сторонам которого может ходить конь. Те же буквы, только строчные, запишем на рамочных полях так, чтобы ходы коня по каждой из букв образовали замкнутые многоугольники, окаймляющие центральный квадрат.

Конь начинает ходить от какого-нибудь рамочного поля, проходит вокруг рамы по выбранной букве, и в 12 ходов исчерпывает эту букву. Затем он переходит во внутренний квадрат, на другую букву. Пройдя все поля, обозначенные этой буквой, конь снова переходит на раму – на букву, по которой еще не ходил, и вновь обегает рам, исчерпывая до конца эту букву, - и т.д., пока не пройдет по всей доске.

Метод Полиньяка и Роже – деление на четверти. Этот метод проще предыдущего, хотя и похож на него. Разделим доску крестов на четыре квадрата. Расставим буквы, конь начинает движение с любой буквы, проходит в выбранном квадрате по всем четырем полям с этой буквой, затем переходит на ту же букву соседнего квадрата, и т.д. Исчерпав все 16 полей, с одной буквой, он меняет ее и снова зигзагом обегает доску. После четырех таких кругов все поля будут пройдены.

Метод Эйлера и Вандермонда. Этот метод, хотя и не является таким простым и эффективным, как два предыдущих, позволяет, в отличие от них, получать самые разнообразные маршруты коня.

В основе этого метода лежит возможность замены обратными всех ходов, начиная с поля, связанного ходом коня с конечным.

Основное достоинство этого метода в том, что он помогает завершить путь коня в тех случаях, когда двигались без всякой системы и попали в тупик – дальше идти некуда, а еще остались непройденные поля.

Правило Варнсдорфа. Это довольно эффективное правило заключается в следующем:

Первое: при обходе доски следует всякий раз ставить на поле, из которого он может сделать наименьшее число ходов на еще не пройденные поля;

Второе: если таких полей несколько, то разрешается выбирать любое из них.

Это правило предложено более 150 лет назад. Долгое время считалось, что оно действительно безукоризненно. Позднее было установлено, что его вторая часть не совсем точна. Если в распоряжении коня имеется несколько возможностей, упомянутых в первой части правила, то не все они являются равноценными. Полную ясность в этот вопрос удалось внести при помощи ЭВМ. Машинный эксперимент показал, что с какого б поля доски конь ни начал свой маршрут, можно так пользоваться второй частью правила Варнсдорфа, что он попадает в тупик, чем посетит все поля доски.

Правило Варнсдорфа, является достаточно эффективным не только для обычной шахматной доски, но и для других досок, на которых вообще имеется решение задачи.

Для нахождения решения задачи о ходе конем шахматной доски не было применено какого-то определенного метода.

Конь обходит все варианты до тех пор, пока не достигнет цели.

1.1. Анализ ограничений и усовершенствований

Рассмотрим следующие методы оптимизации:

- определение клеток, обход из которых невозможен (оптимизация 1);
- выявление заблокированных клеток (оптимизация 2);
- применение правила Варнсдорфа (оптимизация 3);
- использование различных массивов ходов коня (оптимизация 4).

1. Определение клеток, обход из которых невозможен

Если количество клеток доски нечетно (число белых и черных клеток отличается на единицу), то обход из некоторых клеток не существует. Отметим, что путь коня проходит по клеткам, чередующимся по цвету. Если общее число клеток доски нечетно, то первая и последняя клетки пути, пройденного конем, будут одного и того же цвета. Таким образом, обход будет существовать только тогда, когда он начнется клеткой того цвета, который имеют наибольшее число клеток.

Выполнение этого условия проверяется следующей функцией:

```
bool isSolutionImpossible()
{
  // Если произведение сторон доски нечетно и сумма координат начальной позиции
  // нечетна, то решения не существует.
  return ((size_x * size_y) % 2 != 0) && ((start_x + start_y) % 2 != 0) ;
}
```

Однако приведенное правило не охватывает всех клеток, для которых обхода не существует. Так, для доски размером 3x7, помимо тех клеток, для которых выполняется приведенное правило, обход невозможен также из клетки (2,4).

2. Выявление заблокированных клеток

Если при очередном ходе образуется клетка, куда можно войти, но откуда нельзя выйти, то такой ход недопустим, за исключением предпоследнего в обходе. Данный метод оптимизации позволяет значительно сократить число возвратов, в том числе и при совместном использовании с правилом Варнсдорфа.

Развитием этого метода является определение групп заблокированных клеток, связанных друг с другом, но отрезанных от остальной части доски. В рассматриваемой программе определяются группы из двух заблокированных клеток, что значительно уменьшает количество возвратов для небольших досок, а при использовании вместе с правилом Варнсдорфа - и для больших (например, размером 100*100 клеток).

3. Применение правила Варнсдорфа

Среди многих эвристических методов [5], используемых для сокращения перебора, правило Варнсдорфа является наиболее простым. В соответствии с ним при обходе доски коня следует каждый раз ставить на то поле, из которого он может сделать наименьшее число ходов по еще не пройденным полям. Если таких полей несколько, то можно выбрать любое из них, что может,

однако, завести коня в тупик и потребовать возврата [5]. Отметим, что наилучшим образом правило Варнсдорфа работает для угловых полей.

4. Использование различных массивов ходов коня

Ходы коня могут быть заданы, например, в виде следующего массива:

```
int possibleMoves[][2] = {
{-1, -2}, {-2, -1}, {-2,  1}, { 1, -2},
{-1,  2}, { 2, -1}, { 1,  2}, { 2,  1} };
```

Каждый его элемент определяет один возможный ход коня и содержит изменения его координат на доске при совершении хода. При использовании различных массивов для ходов коня количество возвратов может различаться. В программе применяются пять эвристически выбранных массивов, содержащих возможные ходы коня в различном порядке. Также задается максимальное число возвратов (GOOD_RET), и когда оно будет достигнуто, поиск пути начинается заново с использованием уже другого массива. При поиске обхода с применением последнего массива количество возвратов ограничивается значением MAX_RET. Если при совместном использовании всех предложенных методов оптимизации установить значение GOOD_RET равным единице, то для досок, близких к квадратным, можно строить обходы без единого возврата для всех клеток, из которых существует обход. Обход без единого возврата из каждой клетки не удается получить для "вытянутых" досок (например, 14*3) и для больших досок, например для доски 100*100 клеток.

1.2. Разработка алгоритмов решения задачи

Наиболее известное решение для задачи обхода конем – рекурсивное. Для выполнения ргр алгоритм был переработан. При этом структура метода, выполняющего перебор, имеет следующий вид:

```
public List<Desk> FindPossibleMoves(int endX, int endY, Desk startDesk)
{
    var desks = new List<Desk>();

    var queue = new Queue<Desk>();
    queue.Enqueue(startDesk);

    while (queue.Count != 0)
    {
        Desk desk = queue.Dequeue();
        desk.turnCount++;

        int currentX = desk.StartX;
        int currentY = desk.StartY;

        if (isEnd(currentX, currentY, endX, endY))
        {
            desks.Add(desk);
        }

        AddToVisitedCords(currentX, currentY, desk.visitedCoords);

        if (!visitedDesks.Contains(desk))
        {
            visitedDesks.Add(desk);

            for (int i = 0; i < possibleXTurns.Length; i++)
            {
                int dx = currentX + possibleXTurns[i];
                int dy = currentY + possibleYTurns[i];

                if (isValidTurn(dx, dy, desk))
                {
                    queue.Enqueue(new Desk(desk, dx, dy, desk.turnCount));
                }
            }
        }
    }

    return desks;
}
```

- На каждом шаге ищется фрагмент пути, начинающийся из текущей клетки и не включающий уже пройденные;

- Если ход возможен, из массива возможных ходов извлекается очередной элемент, который приводит в незанятую клетку, находящуюся в пределах доски;
- Поиск пути считается успешным тогда, когда текущие координаты становятся равны координатам точки назначения. Если из начальной клетки перебраны все возможные ходы, то пути не существует.

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 S1 ⊆ A1
    k ← 1
    while Sk ≠ ∅ do
        product ← product * |Sk|
        sum ← sum + product
        ak ← rand(Sk)
        k ← k + 1
        определить Sk ⊆ Ak
    count ← count + sum
average ← count / N    // среднее число вершин в дереве
```

Рис. 1. 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.

Оценка времени выполнения

Размер задачи	Стартовое поле	Назначение	Время выполнения
8×8	0x0	8×8	1,154 с
9×9	0x0	9×9	1,855с
10×10	0x0	10×10	2,069с
11×11	0x0	11×11	9,937с
12×12	0x0	12×12	9,556с

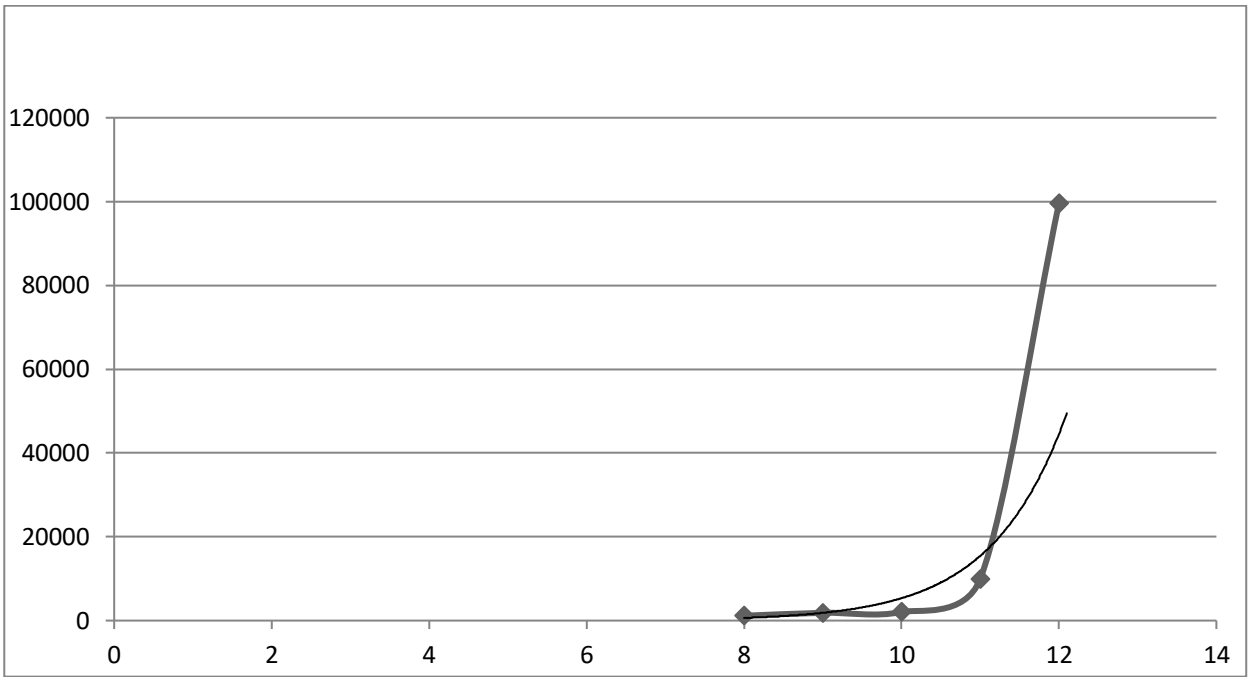


График функции вычислительной сложности

3. Программная реализация алгоритмов

3.1. Выбор языка и среды программирования

В качестве языка программирования был выбран язык c#, среда разработки – Visual Studio Этот выбор сделан исходя из следующих соображений.

Прежде всего c# предназначен для профессиональных разработчиков, желающих очень быстро разрабатывать приложения. C# производит небольшие по размерам высокоэффективные исполняемые модули. С другой стороны небольшие по размерам и быстро исполняемые модули означают, что требования к клиентским рабочим местам существенно снижаются.

Преимущества c# по сравнению с аналогичными программными продуктами:

- высокая производительность разработанного приложения;
- низкие требования разработанного приложения к ресурсам компьютера;
- возможность разработки новых компонентов и инструментов собственными средствами c# (существующие компоненты и инструменты доступны в исходных кодах);
- удачная проработка иерархии объектов.
- Получение опыта разработки в данной среде, на языке c#

Подводя итоги, можно сделать вывод о том, что язык C++ в качестве основного инструмента для разработки программного приложения является подходящим выбором.

3.2. Разработка структурной схемы программы

Разработанная программа исследования алгоритма решения задачи о ходе коня включает в себя 3 файла: Desk.cs, Horse.cs и файл Program.cs, содержащий метод Main, запускающий программу. Первый файл содержит класс Desk, реализующий шахматную доску. Второй файл содержит класс Horse, реализующий фигуру коня.

3.3. Реализация структур данных и алгоритмов

Первый модуль Desk.cs содержит класс Desk, в котором хранится:
информация о шахматной доске

```
private List<List<string>> board = new List<List<string>>();
```

информация о пройденных полях

```
public List<List<int>> visitedCoords = new List<List<int>>();
```

Доска строится с помощью конструктора

```
public Desk(int size)
    {
        this.size = size;
        turnCount = 0;

        for (int i = 0; i < size; i++)
        {
            var row = new List<string>();

            for (int j = 0; j < size; j++)
            {
                row.Add("0");
            }

            board.Add(row);
        }
    }
```

Метод для вывода доски на экран

```
public void PrintDesk()
    {
        for (int i = 0; i < size; i++)
        {
            for (int j = 0; j < size; j++)
            {
                Console.Write($"{board[i][j], 2}" + " ");
            }
            Console.WriteLine();
        }
    }
```

Модуль Horse.cs содержит класс Desk, в котором хранится:
информация о возможных ходах коня

```
private readonly int[] possibleXTurns = new int[] { 2, 2, -2, -2, 1, 1, -1, -1 };
private readonly int[] possibleYTurns = new int[] { -1, 1, 1, -1, 2, -2, 2, -2 };
```

Вспомогательные методы **isNotVisited** – посещено поле или нет, **isValidTurn** – правильный ли ход, **isEnd** – дошли ли до конечного поля

AddToVisitedCords добавляет координаты в список посещенных координат.

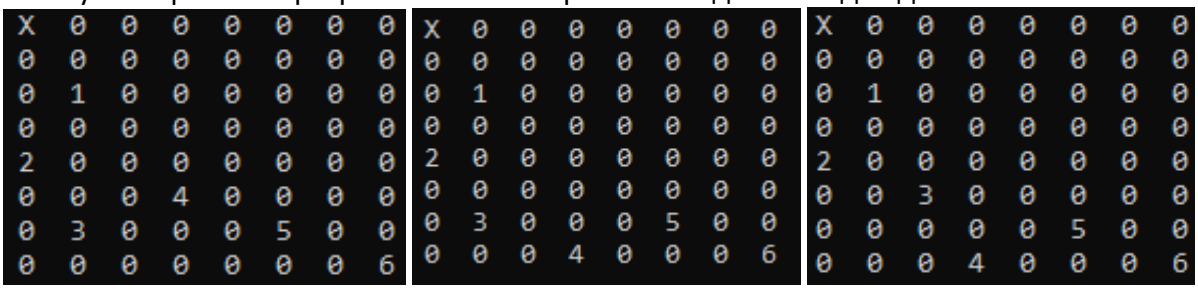
Метод **FindPossibleMoves** находит все возможные ходы коня.

Реализация метода:

```
public List<Desk> FindPossibleMoves(int endX, int endY, Desk startDesk)
    {
      endX -= 1;
      endY -= 1;

      var desks = new List<Desk>();

      var queue = new Queue<Desk>();
      queue.Enqueue(startDesk);

      while (queue.Count != 0)
      {
        Desk desk = queue.Dequeue();
        desk.turnCount++;

        int currentX = desk.StartX;
        int currentY = desk.StartY;

        if (isEnd(currentX, currentY, endX, endY))
        {
          desks.Add(desk);
        }

        AddToVisitedCords(currentX, currentY, desk.visitedCoords);

        if (!visitedDesks.Contains(desk))
        {
          visitedDesks.Add(desk);

          for (int i = 0; i < possibleXTurns.Length; i++)
          {
            int dx = currentX + possibleXTurns[i];
            int dy = currentY + possibleYTurns[i];

            if (isValidTurn(dx, dy, desk))
            {
              queue.Enqueue(new Desk(desk, dx, dy, desk.turnCount));
            }
          }
        }
      }

      return desks;
    }
```

Заключение

Результат работы программы. Часть вариантов ходов коня для доски 8x8



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.
-

Вывод: 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. *А. Шалыто.* Задача о ходе коня.// Н. Туккель, Н. Шамгунов// Мир ПК. – 2013г.

2. *Е.Я. Гик*. Математика на шахматной доске// Москва: Наука – 1976г.

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

Модуль **Desk.cs**

```
public class Desk
   {
      private readonly int size;

      private int startX;
      private int startY;

      private List<List<string>> board = new List<List<string>>();

      public List<List<int>> visitedCoords = new List<List<int>>();
      public int turnCount = 0;

      public int Size => size;
      public int StartX => startX;
      public int StartY => startY;

      public Desk(int size)
      {
         this.size = size;
         turnCount = 0;

         for (int i = 0; i < size; i++)
         {
            var row = new List<string>();

            for (int j = 0; j < size; j++)
            {
               row.Add("0");
            }

            board.Add(row);
         }
      }

      public Desk(Desk desk, int x, int y, int turnCount)
      {
         this.size = desk.size;
         this.visitedCoords = desk.visitedCoords;
         this.turnCount = turnCount;

         for (int i = 0; i < size; i++) // jija
         {
            var row = new List<string>();

            for (int j = 0; j < size; j++)
            {
               row.Add("0");
            }

            board.Add(row);
         }

         for (int i = 0; i < size; i++)
         {
            for (int j = 0; j < size; j++)
            {
               board[i][j] = desk.board[i][j];
            }
         }

         this.board[x][y] = $"{turnCount}";
         this.startX = x;
         this.startY = y;
      }
```

```
    public void SetStartPosition(int x, int y)
    {
        board[x][y] = "X";
        startX = x;
        startY = y;
    }

    public void PrintDesk()
    {
        for (int i = 0; i < size; i++)
        {
            for (int j = 0; j < size; j++)
            {
                Console.Write($"{board[i][j], 2}" + " ");
            }
            Console.WriteLine();
        }
    }
}
```

Модуль **Horse.cs**

```
public class Horse
{
    private int startX;
    private int startY;

    private readonly int[] possibleXTurns = new int[] { 2, 2, -2, -2, 1, 1, -1, -1 };
    private readonly int[] possibleYTurns = new int[] { -1, 1, 1, -1, 2, -2, 2, -2 };

    private List<Desk> visitedDesks = new List<Desk>();

    public Horse(int startX, int startY, Desk desk)
    {
        this.startX = startX - 1;
        this.startY = startY - 1;

        desk.SetStartPosition(this.startX, this.startY);
    }

    private bool isNotVisited(int a, int b, Desk desk)
    {
        if ((a == startX) & (b == startY))
        {
            return false;
        }

        foreach (var elem in desk.visitedCoords)
        {
            if ((elem[0] == a) && (elem[1] == b)) {
                return false;
            }
        }

        return true;
    }

    private bool isValidTurn(int currentX, int currentY, Desk desk)
    {
        return (currentX >= 0 && currentX < desk.Size) & (currentY >= 0 && currentY < desk.Size) & isNotVisited(currentX, cur-
rentY, desk);
    }

    private bool isEnd(int currentX, int currentY, int endX, int endY)
    {
```

```
        return (currentX == endX) & (currentY == endY);
    }

    private static void AddToVisitedCords(int a, int b, List<List<int>> list)
    {
        var row = new List<int>();

        row.Add(a);
        row.Add(b);

        list.Add(row);
    }

    public List<Desk> FindPossibleMoves(int endX, int endY, Desk startDesk)
    {
        endX -= 1;
        endY -= 1;

        var desks = new List<Desk>();

        var queue = new Queue<Desk>();
        queue.Enqueue(startDesk);

        while (queue.Count != 0)
        {
            Desk desk = queue.Dequeue();
            desk.turnCount++;

            int currentX = desk.StartX;
            int currentY = desk.StartY;

            if (isEnd(currentX, currentY, endX, endY))
            {
                desks.Add(desk);
            }

            AddToVisitedCords(currentX, currentY, desk.visitedCoords);

            if (!visitedDesks.Contains(desk))
            {
                visitedDesks.Add(desk);

                for (int i = 0; i < possibleXTurns.Length; i++)
                {
                    int dx = currentX + possibleXTurns[i];
                    int dy = currentY + possibleYTurns[i];

                    if (isValidTurn(dx, dy, desk))
                    {
                        queue.Enqueue(new Desk(desk, dx, dy, desk.turnCount));
                    }
                }
            }
        }

        return desks;
    }

}
```

Модуль **Program.cs**

public class Program

```
{
    public static void Main()
    {
        var desk = new Desk(size:8);
        var horse = new Horse(startX:1, startY:1, desk);
        var desks = new List<Desk>();

        desks = horse.FindPossibleMoves(endX:8, endY:8, desk);

        Console.WriteLine();

        int count = 0;
        foreach (var elem in desks)
        {
            count++;
            elem.PrintDesk();
            Console.WriteLine();
        }

        Console.WriteLine($"Number of options: {count}");
    }
}
```