МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Тольяттинский государственный университет»

Кафедра «Прикладная математика и информатика»
(наименование)

01.03.02 «Прикладная математика и информатика»
(код и наименование направления подготовки / специальности)

Компьютерные технологии и математическое моделирование
(направленность (профиль) / специализация)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

на тему «Исследование и программная реализация алгоритма компьютерной игры»

Обучающийся Д.Р. Азаров _____
(Инициалы Фамилия) (личная подпись)

Руководитель д.т.н., доцент, С.В. Мкртычев
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Консультант к.п.н., доцент, С.А. Гудкова
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Тольятти 2024

Аннотация

Тема выпускной квалификационной работы – «Исследование и программная реализация алгоритма компьютерной игры».

Ключевые слова: алгоритм, компьютерная игра, исследование, программная реализация.

Объектом исследования бакалаврской работы является компьютерная игра «Пятнашки».

Предметом исследования бакалаврской работы является алгоритм компьютерной игры «Пятнашки».

Цель бакалаврской работы – исследование и программная реализация алгоритма компьютерной игры «Пятнашки».

Методы исследования – теория алгоритмов, методы и технологии разработки программного обеспечения.

Первая глава работы посвящена постановке задачи исследования алгоритма компьютерной игры «Пятнашки».

Вторая глава работы посвящена анализу алгоритма компьютерной игры «Пятнашки».

В третьей главе рассматривается процесс разработки и тестирования программы, реализующей алгоритм игры «Пятнашки».

В заключении описываются результаты выполнения выпускной квалификационной работы.

Выпускная квалификационная работа состоит из 50 страниц текста, 15 рисунков, 2 таблиц и 25 источников.

Abstract

The topic of the final qualifying work is 'Research and software implementation of a computer game algorithm'.

Keywords: algorithm, computer game, research, software implementation.

The object of the bachelor's thesis research is the computer game '15 Puzzle'.

The subject of the bachelor's thesis research is the algorithm of the computer game '15 Puzzle'.

The purpose of the bachelor's thesis is to study and implement the algorithm of the computer game '15 Puzzle'.

Research methods - algorithm theory, software development methods and technologies.

The first chapter of the work is devoted to the formulation of the problem of studying the algorithm of the computer game '15 Puzzle'.

The second chapter of the work is devoted to the analysis of the algorithm of the computer game '15 Puzzle'.

The third chapter discusses the process of developing and testing a program that implements the '15 Puzzle' game algorithm.

The conclusion describes the results of the final qualifying work.

The final qualifying work consists of 50 pages of text, 15 figures, 2 tables and 25 sources.

Оглавление

Введение

Компьютерные игры – это игры, в которые играют на электронных устройствах, таких как игровые консоли, смартфоны, планшеты, гарнитуры виртуальной реальности или персональные компьютеры. В них можно играть через Интернет, локальные сети или офлайн.

Как и обычные игры, компьютерные игры сильно различаются и включают в себя как сложные онлайн-миры с несколькими игроками (известные как многопользовательские онлайн-игры), так и простые головоломки для одного игрока. Последние очень популярны среди пользователей [3].

Головоломки очень хорошо развивают пространственное мышление, невероятно важное во многих сферах деятельности. В большей степени это касается художников, дизайнеров, архитекторов, инженеров, режиссеров, писателей и прочих творческих личностей [4].

Одной из таких компьютерных игр-головоломок является игра «Пятнашки», известная за рубежом под названием «Sliding puzzle».

Как правило, в основу любой компьютерной головоломки положен определенный алгоритм.

Исследование и программная реализация алгоритма игры «Пятнашки» представляет научный и практический интерес.

Объектом исследования бакалаврской работы является компьютерная игра «Пятнашки».

Предметом исследования бакалаврской работы является алгоритм компьютерной игры «Пятнашки».

Цель бакалаврской работы – исследование и программная реализация алгоритма компьютерной игры «Пятнашки».

Для достижения данной цели необходимо выполнить следующие задачи:

– выполнить постановку задачи исследования алгоритма

компьютерной игры «Пятнашки»;

- проанализировать алгоритм компьютерной игры «Пятнашки»;
- разработать и протестировать программу, реализующую алгоритм компьютерную игру «Пятнашки».

Методы исследования – теория алгоритмов, методы и технологии разработки программного обеспечения.

Практическая значимость бакалаврской работы заключается в разработке программы, позволяющий решить головоломку «Пятнашки».

Данная работа состоит из введения, трех глав, заключения и списка используемой литературы и используемых источников.

Первая глава работы посвящена постановке задачи исследования алгоритма компьютерной игры «Пятнашки».

Вторая глава работы посвящена анализу алгоритма компьютерной игры «Пятнашки».

В третьей главе рассматривается процесс разработки и тестирования программы, реализующей алгоритм игры «Пятнашки».

В заключении описываются результаты выполнения выпускной квалификационной работы.

Выпускная квалификационная работа состоит из 50 страниц текста, 15 рисунков, 2 таблиц и 25 источников.

Глава 1 Постановка задачи исследования алгоритма компьютерной игры «Пятнашки»

1.1 Математическое описание головоломки «Пятнашки»

«Пятнашки», скользящая головоломка, головоломка с раздвижными блоками или головоломка с раздвижными плитками – это комбинаторная головоломка, в которой игроку предлагается перемещать (часто плоские) части по определенным маршрутам (обычно на доске), чтобы установить определенную конечную конфигурацию [10].

Автором головоломки является Ной Чепмэн.

Части, которые нужно переместить, могут состоять из простых фигур или на них могут быть напечатаны цвета, узоры, части более крупного изображения (например, мозаика), цифры или буквы.

Правила игры просты: начиная с зашифрованного состояния, можно решить головоломку, неоднократно перемещая соседние плитки в пустое пространство, пока числа не вернутся в последовательность.

Таким образом, действие этой головоломки происходит в сетке размером $n \times m$.

Имеется $n \times m - 1$ пронумерованных плиток.

Цель состоит в том, чтобы переставить пронумерованные плитки так, чтобы они появлялись в возрастающем порядке.

Головоломка на 15 – частный случай этой головоломки.

Распространенными вариантами, используемыми в исследованиях, являются головоломки из 14 (3 × 5), головоломки из 19 (4 × 5) и головоломки из 24 (5 × 5).

Стандартный размер головоломки – 4 × 4, в ней 15 пронумерованных плиток и одна отсутствует.

Легко определить, разрешима ли данная головоломка.

Рассмотрим последовательность чисел в головоломке в том порядке, в

котором они встречаются на доске [24].

Головоломка разрешима тогда и только тогда, когда в последовательности имеется четное число инверсий (неупорядоченных пар).

Аналогично, одна конфигурация может достичь другой, если количество инверсий для обеих конфигураций имеет одинаковую четность.

Путем полного поиска в ширину было доказано, что ни одна конфигурация головоломки из 15 не требует более 80 ходов.

Точный размер пространства поиска равен $16!/2 \approx 10^{13}$.

Пространство состояний содержит только одно решение, но оптимальных последовательностей решений может быть несколько. Разные экземпляры (начальные позиции) принадлежат одному и тому же пространству поиска.

Принимая во внимание естественное обобщение головоломки на $n^2 - 1$ (сетка размером $n \times n$ с $n^2 - 1$ пронумерованными плитками) можно утверждать, что любое решение легко найти за полиномиальное время.

Поиск оптимального решения (наименьшего числа ходов) является NP-полным.

Требуемое количество ходов в худшем случае равно $\Theta(n^3)$.

Размер пространства поиска, включающий только достижимые состояния, равен $(n^2)!/2$.

Коэффициент ветвления варьируется от 2 до 4, в зависимости от положения пустого квадрата. Таким образом, эффективный коэффициент ветвления составляет от 1 до 3, поскольку отмена последнего хода никогда не рассматривается.

Сущность игры рассматриваемой игры состоит в следующем: пусть N - размер игрового поля (обычно N = 4 для классических "Пятнашек").

Игровое поле представляется в виде квадратной матрицы размером N x N, где каждая ячейка содержит число от 1 до N^2-1, а одна ячейка пустая.

Матрица обозначается как F, где F[i][j] - значение числа в ячейке с координатами (i, j), а F[x][y] = 0 - пустая ячейка.

Цель игры состоит в упорядочивании чисел по возрастанию слева направо, сверху вниз, с пустой ячейкой в нижнем правом углу (F[N-1][N-1] = 0).

Математические операции, связанные с игрой «Пятнашки», включают:

- перемещение чисел: число можно переместить из одной ячейки в другую, если эти ячейки соседние (граничат друг с другом по вертикали или горизонтали), а одна из ячеек пустая. После перемещения числа пустая ячейка занимает его место, а числовая ячейка становится пустой;
- генерацию начальной конфигурации: начальная конфигурация игры может быть сгенерирована случайным образом или с использованием других алгоритмов;
- проверку на решаемость: можно использовать различные методы для определения, является ли заданная конфигурация решаемой, то есть имеет ли она решение или нет;
- поиск оптимального решения: можно применять различные алгоритмы поиска, такие как поиск в ширину, алгоритм A*, эвристические алгоритмы и т.д., для нахождения оптимального решения игры «Пятнашки».

Таким образом, определена сущность математической постановки задачи.

1.2 Методы решения головоломки «Пятнашки»

На примере скользящей головоломки можно доказать, что головоломку «Пятнашки» можно представить чередующейся группой A_{15}, т.к. комбинации головоломки «Пятнашки» могут быть созданы с помощью 3-х циклов [13].

Р.Е. Корф и др. предложил метод решения головоломки, основанный на использовании поиска в ширину [20].

Был выполнен полный поиск в ширину пространства поиска, используя

параллельную обработку и сохранение узлов на диск.

Средняя длина решения во всех состояниях составила 53 хода.

Дж. Калберсон использовал метод IDA* со следующими улучшениями [15]:

- базы данных шаблонов: это средство улучшения нижней границы эвристики. В базе данных собраны все решения подзадачи правильного размещения N плиток. Этот метод называется базами данных сокращения;
- таблицы транспозиции: отслеживаются ранее посещенные состояния;
- базы данных эндшпиля: хранятся все состояния, которые находятся не более чем на N шагах от целевого состояния. Когда поиск достигнет позиции в этом наборе, получить расстояние и прекратите поиск в этой ветви. В своем подходе авторы использовали N = 25;

Используя редукционные базы данных на 8 штук, авторы смогли сократить общее количество искомых узлов более чем в 1700 раз на стандартном тестовом наборе, содержащем 100 позиций.

Для решения скользящей головоломки также предлагается метод «Разделяй и властвуй (Divide and conquer)» [16].

Этот метод основан на разбиении сложной задачи на подзадачи, пока они не станут достаточно простыми, чтобы их можно было решить напрямую.

Особенность метода состоит в том, что он применимо к любому размеру головоломки: 3×3, 4×4, 5×5 и т. д.

Идея заключается в том, чтобы сначала решить углы, чтобы головоломка 5×5 превратилась в головоломку 4× 4, а затем 3x3.

Алгоритм, положенный в основу данного метода, состоит из следующих шагов:

- решите верхний угол (шаг 1 ниже);
- решите левый угол (шаг 2 ниже);
- повторяйте шаги 1 и 2, пока размер головоломки не уменьшится до

3×3;

– решите только верхний угол (шаг 1 ниже);
– решите оставшуюся головоломку 3×2 (шаг 4 ниже).

Чтобы переместить фигуру в любую позицию, проще всего:

– поместить пустой квадрат в целевую позицию;
– переместить свою плитку в заданном направлении, переместив все плитки на одну клетку вперед.
– снова поместить пустой квадрат перед своей плиткой, вращая плитки вокруг целевой плитки.

Повторяйте шаги 2 и 3, пока ваша плитка не достигнет цели.

На рисунке 1 показаны расположения плиток в процессе решения задачи.

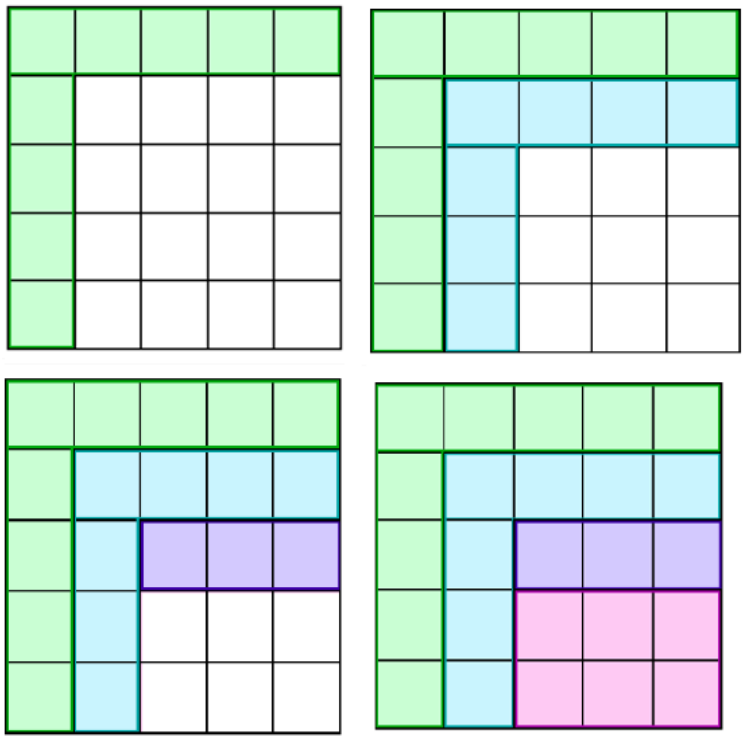

Рисунок 1 – Расположения плиток в процессе решения задачи методом «Разделяй и властвуй»

Поскольку метод состоит из решения углов один за другим, как только угол решен, он считается «замороженным», и его больше не следует трогать.

Это означает, что, когда вы вращаете плитки вокруг целевой плитки, вы никогда не должны нарушать «замороженную» область.

1.3 Анализ алгоритмов для решения скользящей головоломки

Для решения скользящей головоломки используются алгоритмы эвристического поиска. Они используются в ряде областей, включая робототехнику, видеоигры и обработку естественного языка [2].

Алгоритм эвристического поиска использует эвристическую функцию для оценки расстояния между текущим состоянием и целевым состоянием, чтобы определить кратчайший маршрут от начального состояния к целевому состоянию [17].

Существуют различные эвристические алгоритмы поиска, включая A*, поиск по единой стоимости (UCS), поиск в глубину (DFS) и поиск в ширину (BFS).

Рассмотрим характеристики популярных алгоритмов, используемых для решения скользящей головоломки [23].

1.3.1 Алгоритм поиска в ширину

«Поиск в ширину (Breadth-first search, BFS) – это алгоритм обхода графа, который начинает обход графа с корневого узла и исследует все соседние узлы. Затем он выбирает ближайший узел и исследует все неисследованные узлы.

При использовании BFS для обхода любой узел графа можно рассматривать как корневой узел.

Существует множество способов обхода графа, но среди них наиболее часто используемым является BFS» [14].

Это рекурсивный алгоритм поиска по всем вершинам структуры данных дерева или графа. BFS делит каждую вершину графа на две категории – посещенные и непосещенные. Он выбирает один узел в графе и после этого

посещает все узлы, соседние с выбранным узлом.

BFS можно использовать для поиска соседних местоположений из заданного исходного местоположения.

В одноранговой сети алгоритм BFS можно использовать в качестве метода обхода для поиска всех соседних узлов. Большинство торрент-клиентов, таких как BitTorrent, uTorrent и т. д., используют этот процесс для поиска «сидов» и «одноранговых узлов» в сети.

На рисунке 2 представлен псевдокод алгоритма BFS.

Algorithm ': Breadth-First Search

Data: s : the start node, *target*: the function that identifies a target node, *neighbors*: the function that returns the neighbors of a node.
Result: The shortest path between s and a target node, if one exists. Otherwise, the algorithm returns $\emptyset$.

```
if target(s) then
    return s
end
frontier ← a LIFO queue containing only s
explored ← an empty set
s.parent ← NULL
t ← NULL
while frontier isn't empty do
    u ← pop the oldest node
    Add u to explored
    for v ∈ neighbors(u) do
        v.parent ← u
        if u isn't in explored or frontier then
            if target(v) then
                t ← v
                break
            end
            Add v to frontier
        end
    end
    if target(v) then
        break
    end
end
if t is NULL then
    return ∅
end
else
    path ← reconstruct the path from s to t
    return path
end
```

Рисунок 2 – Псевдокод алгоритма BFS

BFS можно использовать в веб-сканерах для создания индексов веб-страниц. Это один из основных алгоритмов, который можно использовать для индексации веб-страниц. Он начинает переход с исходной страницы и следует по ссылкам, связанным с этой страницей. Здесь каждая веб-страница рассматривается как узел графа.

BFS используется для определения кратчайшего пути и минимального связующего дерева.

Алгоритм поиска в ширину состоит из нижеследующих шагов.

«Шаг 1: SET STATUS = 1 (состояние готовности) для каждого узла в G.

Шаг 2. Поставьте в очередь начальный узел A и установите его STATUS = 2 (состояние ожидания).

Шаг 3. Повторяйте шаги 4 и 5, пока ОЧЕРЕДЬ не опустеет.

Шаг 4: Выведите из очереди узел N. Обработайте его и установите его STATUS = 3 (состояние обработки).

Шаг 5. Поставьте в очередь всех соседей N, находящихся в состоянии готовности (СТАТУС = 1), и установите их СТАТУС = 2 (состояние ожидания).

(КОНЕЦ ЦИКЛА)

Шаг 6: ВЫХОД» [14].

Временная сложность BFS зависит от структуры данных, используемой для представления графа.

Временная сложность алгоритма BFS равна O(V+E), где V и E – количество вершин и ребер графа, соответственно, поскольку в худшем случае алгоритм BFS исследует каждый узел и ребро.

Пространственную сложность алгоритма BFS равна O(V).

1.3.2 Алгоритм A*

A * – это хорошо известный эвристический метод поиска, который вычисляет расстояние между текущим состоянием и целевым состоянием с использованием эвристической функции. Метод поиска A* добавляет

фактическую стоимость от начального узла к текущему узлу и прогнозируемую стоимость от текущего узла к целевому узлу для определения стоимости каждого узла. Эвристическая функция, которая оценивает расстояния между текущим узлом и желаемым узлом, используется для определения предполагаемой стоимости. Затем алгоритм выбирает узел с наименьшей стоимостью, увеличивает его и продолжает делать это до тех пор, пока не достигнет узла назначения.

Пока эвристическая функция приемлема и непротиворечива, алгоритм поиска A* гарантирует нахождение кратчайшего пути к узлу назначения. Это делает его идеальным методом поиска.

На рисунке 3 представлен псевдокод алгоритма A*.

```
A* search {

closed list = [ ]
open list = [start node]

    do {
        if open list is empty then {
            return no solution
        }
        n = heuristic best node
        if n == final node then {
            return path from start to goal node
        }
        foreach direct available node do{
            if current node not in open and not in closed list do {
                add current node to open list and calculate heuristic
                set n as his parent node
            }
            else{
                check if path from star node to current node is
                better;
                if it is better calculate heuristics and transfer
                current node from closed list to open list
                set n as his parrent node
            }
        delete n from open list
        add n to closed list
    } while (open list is not empty)

}
```

Рисунок 3 – Псевдокод классического алгоритма A*

Эвристическая функция считается приемлемой, если она никогда не переоценивает расстояние до конечного узла. Согласно неравенству треугольника, согласованная эвристическая функция - это функция, в которой предполагаемая стоимость от текущего узла до целевого узла меньше или равна фактической стоимости плюс предполагаемая стоимость от следующего узла до целевого узла [1].

Итеративное углубление A* (IDA* или версия A* с ограниченным объемом памяти называется IDA*) – «это метод обхода графа и поиска пути, который может определить кратчайший маршрут во взвешенном графе между определенным начальным узлом и любым из группы целевых узлов.

Это своего рода итеративный поиск в глубину с углублением, в котором используется идея алгоритма поиска A* об использовании эвристической функции для оценки оставшихся затрат для достижения цели» [18].

Этот алгоритм выполняет все операции, которые выполняет A*, и обладает оптимальными возможностями для поиска кратчайшего пути, но занимает меньше памяти.

IDA* использует эвристику для выбора узлов для исследования и на какой глубине остановиться, в отличие от итеративного углубления DFS, которое использует простую глубину, чтобы определить, когда закончить текущую итерацию и продолжить с более высокой глубиной.

Ключевые особенности:

- алгоритм обхода графа;
- находит кратчайший путь во взвешенном графе между начальными и целевыми узлами, используя алгоритм поиска пути;
- альтернатива итеративному алгоритму поиска в глубину.
- использует эвристическую функцию, которая представляет собой метод, взятый из алгоритма A*;
- это алгоритм поиска в глубину, поэтому он использует меньше памяти, чем алгоритм A*;

– IDA* – допустимая эвристика, поскольку она никогда не переоценивает стоимость достижения цели;
– ориентирован на поиск наиболее перспективных узлов, поэтому не везде идет на одинаковую глубину.

Функция оценки в IDA* имеет следующий вид (1, 2):

$$f(n) = g(n) + h(n) \quad (1)$$

$$f(n) = Actual\ Cost + Estimated\ Cost, \quad (2)$$

где

$f(n)$ – функция оценки общей стоимости;

$g(n)$ – фактическая стоимость от начального узла до текущего узла;

$h(n)$ – эвристическая оценка стоимости перехода от текущего узла к целевому состоянию. он основан на аппроксимации по характеристикам задачи.

Рассмотрим свойства IDA [5].

Как и A*, IDA* гарантированно найдет кратчайший путь, ведущий от заданного начального узла к любому целевому узлу в графе задачи, если эвристическая функция h допустима, т.е. (3):

$$h(n) \leq h^*(n) \quad (3)$$

для всех узлов n, где h^* – истинная стоимость кратчайшего пути от n до ближайшей цели («идеальная эвристика»).

IDA* полезен, когда проблема связана с нехваткой памяти.

Поиск A* сохраняет большую очередь неисследованных узлов, которые могут быстро заполнить память. Напротив, поскольку IDA* не запоминает ни одного узла, кроме тех, которые находятся на текущем пути, ему требуется объем памяти, линейный только по длине строящегося решения.

Его временная сложность анализируется Корфом и др. в предположении, что эвристическая оценка стоимости h непротиворечива, а это означает, что (4):

$$h(n) \leq cost(n, n') \tag{4}$$

для всех узлов n и всех соседей n' из n.

Авторы исследования приходят к выводу, что по сравнению с поиском по дереву методом грубой силы для решения задачи экспоненциального размера, IDA* достигает меньшей глубины поиска (в постоянном коэффициенте), но не меньшего коэффициента ветвления.

Как и в случае A*, эвристика должна иметь определенные свойства, чтобы гарантировать оптимальность (кратчайшие пути).

С математической точки зрения игра представляет собой задачу, моделирования эвристических алгоритмов.

Есть три расстояния, которые можно использовать для измерения расстояния между состоянием головоломки и решением:

- расстояние Хэмминга – это общее количество неправильно размещенных плиток;
- манхэттенское расстояние – это сумма абсолютных разностей между двумя точками;
- линейный конфликт: плитки t1 и t2 находятся в линейном конфликте, если t1 и t2 являются одной линией, позиции ворот t1 и t2 находятся в этой линии, t1 находится справа от t2, а целевая позиция t1 находится слева от позиции t2.

Обычно задачу решают через количество перемещений и поиск манхетенского расстояния между каждой костяшкой и её позицией в собранной головоломке.

Целевая позиция вычисляется по формуле (5):

$$goalPosition = manhattanDistance + 2 * linearConflict. \quad (5)$$

Для сравнения характеристик алгоритмов решения скользящей головоломки составлена таблица 1.

Таблица 1 – Характеристики алгоритмов решения скользящей головоломки

Алгоритм	Преимущества	Недостатки
BFS	Находит кратчайший путь	Не знает, где цель
A* (классический)	Знает, где цель	Хранит слишком много состояний
IDA*	Не хранит слишком много состояний	Проходит одни и те же состояния

По результатам анализа выбираем алгоритм A* с эвристиками.

Таким образом, поставлена задача исследования и программной реализации алгоритм A* для решения головоломки «Пятнашки».

Выводы к главе 1

Результаты проделанной в главе 1 работы позволили сделать выводы:

- поиск оптимального решения (наименьшего числа ходов) является NP-полным;
- путем полного поиска в ширину было доказано, что ни одна конфигурация головоломки из 15 не требует более 80 ходов;
- для решения головоломки «Пятнашки» используются алгоритмы наподобие алгоритма A*;
- алгоритм IDA* полезен, когда проблема связана с нехваткой памяти. Как и в случае A*, эвристика должна иметь определенные свойства, чтобы гарантировать оптимальность (кратчайшие пути).

Поставлена задача исследования и программной реализации алгоритма A* для решения головоломки «Пятнашки».

Глава 2 Исследование возможности применения алгоритма A* для решения головоломки «Пятнашки»

2.1 Модель решателя головоломки «Пятнашки» на основе алгоритма A*

«Алгоритм A* предполагает наличие двух списков вершин графа: открытого и закрытого.

В первом находятся вершины, еще не проверенные алгоритмом, а во втором – те вершины, которые уже встречались в ходе поиска решения. На каждом новом шаге из списка открытых вершин выбирается вершина с наименьшим весом.

Вес F каждой вершины вычисляется как сумма расстояния от начальной вершины до текущей G и эвристического предположения о расстоянии от текущей вершины до терминальной H» [7].

Таким образом (6):

$$F_i = G_i + H_i, \tag{6}$$

где i – текущая вершина (состояние игрового поля).

Для «Пятнашек» можно сделать следующие эвристические предположения:

- для достижения терминальной вершины необходимо выполнить перемещений не меньше, чем сумма ортогональных расстояний каждой костяшки до своей терминальной позиции;
- для достижения терминальной вершины необходимо выполнить перемещений не меньше, чем количество костяшек, находящихся не на своих местах;
- для достижения терминальной вершины необходимо выполнить перемещений не меньше, чем сумма эвклидовых расстояний каждой

костяшки до своей терминальной позиции.

На основе этих предположения будем рассчитывать значение *H*.

Функция *G* рассчитывается как количество сделанных перестановок, считая от начальной вершины до текущего состояния.

«Расчёт значения *H* выполняется для каждой вершины при инициализации.

Из начальной вершины проводятся рёбра в дочерние вершины, соответствующие конфигурациям поля, которые могут быть получены перемещением костяшек на пустую клетку. Каждая вершина имеет ссылку на родительскую вершину.

Выполняется перебор дочерних вершин.

Каждая дочерняя вершина проверяется на предмет наличия в списке закрытых. Если вершина не встречалась ранее, она перемещается в список открытых вершин. Для неё рассчитывается эвристическое расстояние до терминальной вершины. В качестве списка открытых вершин удобно использовать очередь с приоритетом, так как на каждом шаге нам нужно брать вершину с 130 наименьшим значением *F* (рисунок 4)» [7].

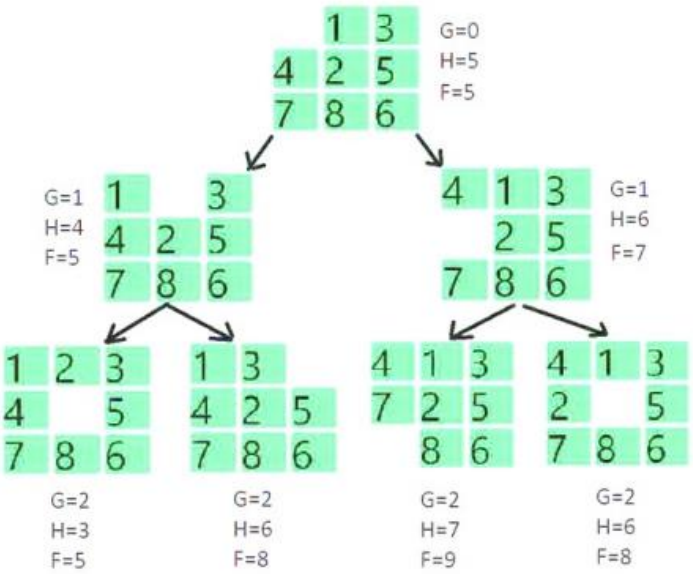

Рисунок 4 – Граф состояний игрового поля

В качестве списка закрытых вершин удобно использовать множество, так как нам нужно знать, обрабатывали ли мы данное состояние поля или нет.

Выбор этих структур данных значительно ускорит время работы алгоритма.

Алгоритм будет перебирать состояния до тех пор, пока из списка открытых вершин мы не извлечём терминальную. Но, прежде чем начать искать решение, необходимо проверить, имеет ли решение данная начальная конфигурация костяшек на поле.

Пусть:

– плитка с числом i расположена до (если считать слева направо и сверху вниз) n_i плиток с числами, меньшими i.
– e – номер ряда пустой клетки (считая с 1).

Вычислим сумму S (7):

$$S = N - \sum_{i=1}^{m} n_i + e \tag{7}$$

Как было отмечено выше, если S – нечетная, то решения головоломки не существует.

Если же перед нами поле с нечётной размерностью, то в формуле не нужно учитывать слагаемое e, то есть номер строки пустого элемента.

Тогда формула расчет суммы S будет иметь вид (8):

$$S = N - \sum_{i=1}^{m} n_i \tag{8}$$

Таким образом, алгоритм A* позволяет существенно сократить количество состояний для перебора, путем применения некоторой дополнительной информации, эвристики.

В качестве такой информации предлагается брать предполагаемое количество перестановок, необходимых для получения терминального состояния.

Р.Е. Корф и др. улучшили производительность в головоломке из 15 более чем в 2000 раз. Они достигли этого, используя непересекающиеся базы данных шаблонов – несколько подцелей, эвристические значения которых можно складывать вместе, гарантируя при этом, что сумма по-прежнему является допустимой эвристической функцией [19].

2.2 Методика разработки решателя «Пятнашек» на основе алгоритма A*

Рассмотрим пример разработки решателя «Пятнашек» [8].

Для упрощения используем версию головоломки для N=8 и размера доски 3x3.

Агентом, который мы будем разрабатывать, выполняется поиск в пространстве состояний (пространстве всех возможных конфигураций игрового поля) с использованием информированного поиска A*.

Поиск «информируется» при выборе следующего шага с учётом знаний предметной области, которые представлены значением функции, связанной с каждым состоянием задачи. Возможные состояния (вверх, вниз, влево и вправо) соответствуют направлениям перемещения пустой клетки (рисунок 5).

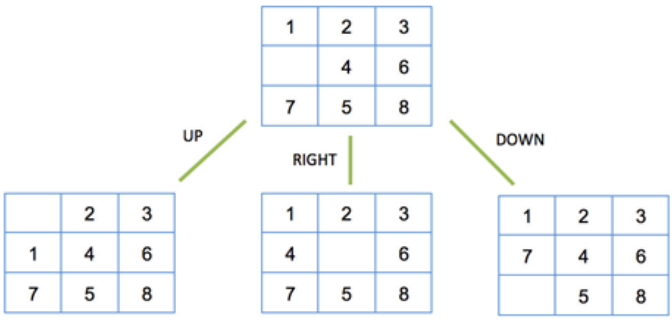

Рисунок 5 – Информационный поиск A*

Ключевой элемент A* находится в значении, которое присваивается каждому состоянию и даёт возможность резко сократить время на поиск целевого состояния из заданного начального состояния (рисунок 6).

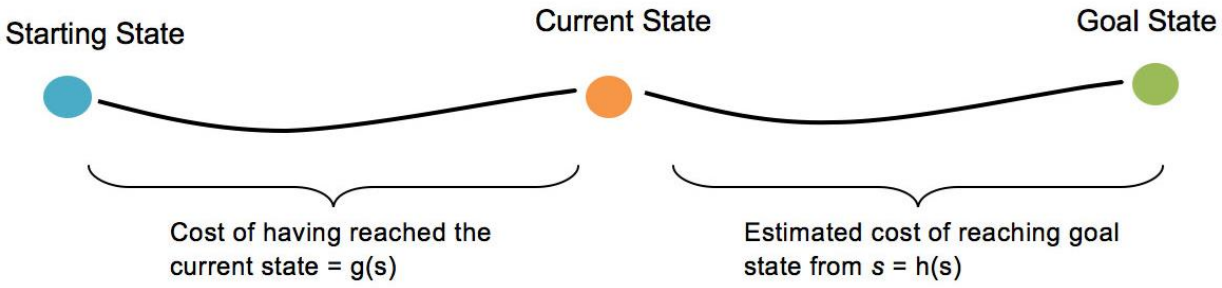

Рисунок 6 – Нахождение целевого состояния из заданного начального

«Эвристической функцией к агенту привязываются эмпирические правила. Поэтому, чем лучше добавляемые правила, тем скорее достигается целевое состояние.

Функция, с помощью которой выводится число неправильно расположенных клеток для состояния s, может считаться потенциально эвристической: по ней всегда понятно, насколько далеко мы от целевого состояния.

При создании или включении эвристики важна её допустимость, необходимая, чтобы гарантировать полноту и оптимальность алгоритма A*, то есть что алгоритм находит решение и это решение оптимальное.

Эвристика допустима, если минимальная стоимость достижения целевого состояния из узла s никогда не завышается: её значение не должно быть больше стоимости кратчайшего пути от s до целевого состояния. Эвристика неправильно расположенных клеток допустима, ведь каждая из них хотя бы раз должна быть перемещена, чтобы правильно расположить их» [8].

Как показано на рисунке 6, набор состояний можно смоделировать в виде графа, где дочерние элементы узла состояния получаются перемещением

пустой клетки во всех возможных направлениях.

Чтобы определить целевое состояние, в такой графовой задаче логично использовать алгоритм поиска по графу.

Основа алгоритма A* – это поиск в ширину. При поиске в ширину значение всех узлов равно 1, поэтому не важно, какой узел расширять: в A* всегда расширяется узел, которым максимизируется нужный результат. В случае с головоломкой это узел с минимальной стоимостью, то есть кратчайшим путём до целевого состояния.

На рисунке 7 показано выполнение алгоритма A* (в качестве эвристической функции используются неправильно расположенные клетки).

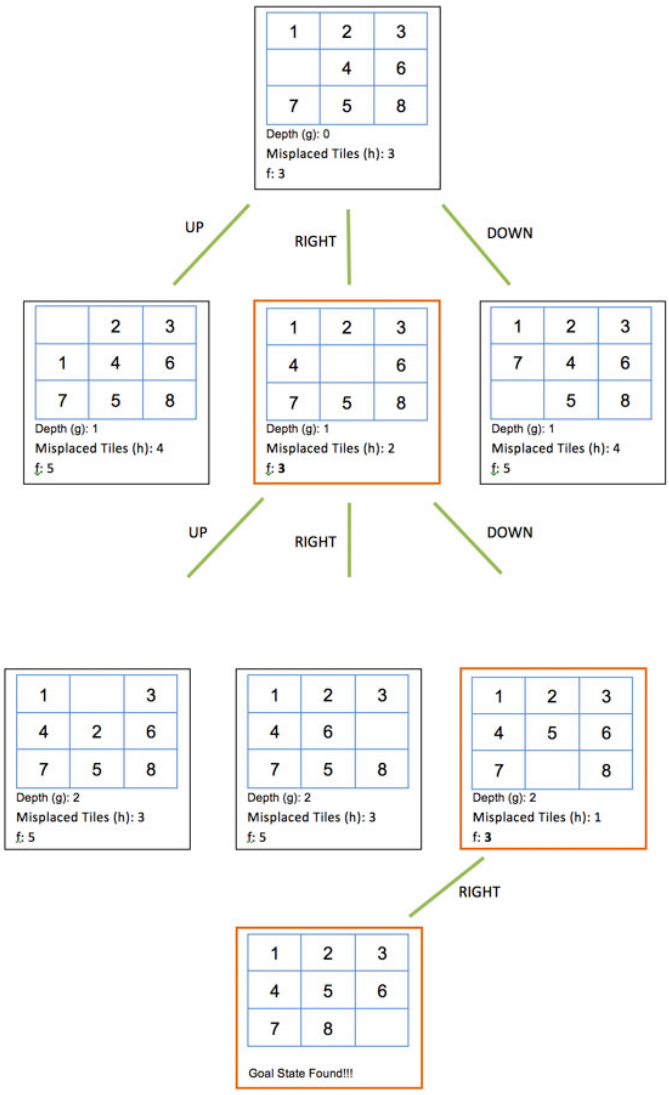

Рисунок 7 – Запуск алгоритма A* с неправильно расположенными клетками

Важно отметить, что при вычислении любой эвристики пустая клетка никогда не учитывается.

Иначе реальная стоимость кратчайшего пути до целевого состояния может быть завышена, а эвристика станет недопустимой.

Таким образом, результаты исследования подтвердили возможность использования алгоритма A* для решения головоломки «Пятнашки».

Выводы к главе 2

Результаты проделанной в главе 2 работы позволили сделать выводы:

- эвристический поиск – это метод информированного поиска. Эвристическое значение сообщает алгоритму, какой путь обеспечит решение как можно раньше. Эвристическая функция используется для генерации этого эвристического значения;
- ключевой особенностью алгоритма A* является то, что он отслеживает каждый посещенный узел, что помогает игнорировать уже посещенные узлы, экономя огромное количество времени. У него также есть список, содержащий все узлы, которые осталось изучить, и он выбирает наиболее оптимальный узел из этого списка, тем самым экономя время, не исследуя ненужные или менее оптимальные узлы.

Результаты исследования подтвердили возможность использования алгоритма A* для решения головоломки «Пятнашки».

Глава 3 Реализация и тестирование компьютерной игры «Пятнашки»

3.1 Выбор средства разработки программы

Для реализации игры «Пятнашки» выбран язык Python.

Для выбора среды разработки сравним характеристики двух сред разработки: Jupyter Notebook и Python IDLE.

Рассмотрим возможности среды Jupyter Notebook [21].

Среда Jupyter Notebook приобрела популярность среди ученых и исследователей данных благодаря своей интерактивной природе и возможности сочетать код, текст и визуализацию в одном документе.

Jupyter Notebook имеет несколько функций, которые делают его популярным инструментом для специалистов по данным и исследователей:

- интерактивный пользовательский интерфейс;
- выполнение кода в режиме онлайн;
- интеграция с несколькими языками программирования;
- легкое сотрудничество и обмен данными;
- визуализация данных и результатов;
- поддержка языка разметки и др.

Jupyter Notebook имеет ряд преимуществ и недостатков по сравнению с другими IDE, в том числе:

- интерактивное и исследовательское кодирование;
- визуализация и анализ данных в реальном времени;
- зависимость от веб-браузера;
- ресурсоемкий и медленный для больших наборов данных;
- ограниченные инструменты отладки;
- трудности с контролем версий;
- проблемы безопасности при запуске кода из ненадежных

источников.

Рассмотрим возможности среды Python IDLE [22].

Python IDLE – интегрированная среда разработки с языком программирования Python и базовым интерфейсом, который позволяет пользователям писать, выполнять и отлаживать код Python.

Python IDLE имеет несколько функций, которые делают его популярным среди начинающих и опытных разработчиков Python:

- базовый пользовательский интерфейс;
- инструменты отладки;
- подсветка синтаксиса;
- автозаполнение и др.

Преимущества и недостатки среды Python IDLE:

- простота и скорость;
- интегрированные инструменты отладки;
- ограниченные возможности и функциональность;
- нет поддержки совместной работы;
- ограниченная поддержка визуализации данных;
- нет поддержки языка разметки;
- нет интеграции с другими языками программирования.

Для сравнения рассмотренных сред программирования используем таблицу 2.

Таблица 2 – Сравнение характеристик сред разработки на языке Pyhon

Характеристика	Jupyter Notebook	Python IDLE
Интерфейс	Веб-интерфейс	Десктоп-приложение
Вывод	отображается в строке	в отдельном окне консоли
Совместная работа	Встроенная поддержка совместного использования и совместной работы	Нет встроенной функции совместной работы
Организация кода	Код организован в ячейках	Код, записан в одном файле
Отладка	Доступна ограниченная поддержка	Доступна поддержка отладки

Продолжение таблицы 2

Характеристика	Jupyter Notebook	Python IDLE
Визуализация	Отличная поддержка	Ограниченная поддержка
Кривая обучения	Крутая кривая обучения	Легко изучается
Варианты использования	Интерактивный анализ данных, создание прототипов и визуализация	Написание небольших скриптов и изучение синтаксиса Python

По результатам сравнения и с учетом предпочтений разработчика выбираем среду разработки Jupyter Notebook.

3.2 Реализация и тестирование компьютерной игры «Пятнашки»

Для представления структуры программы используем диаграмму классов UML [Леон].

Для построения диаграмм UML использован онлайн-сервис Visual Paradigm [25].

На рисунке 8 показана диаграмма классов UML игры «Пятнашки» [12].

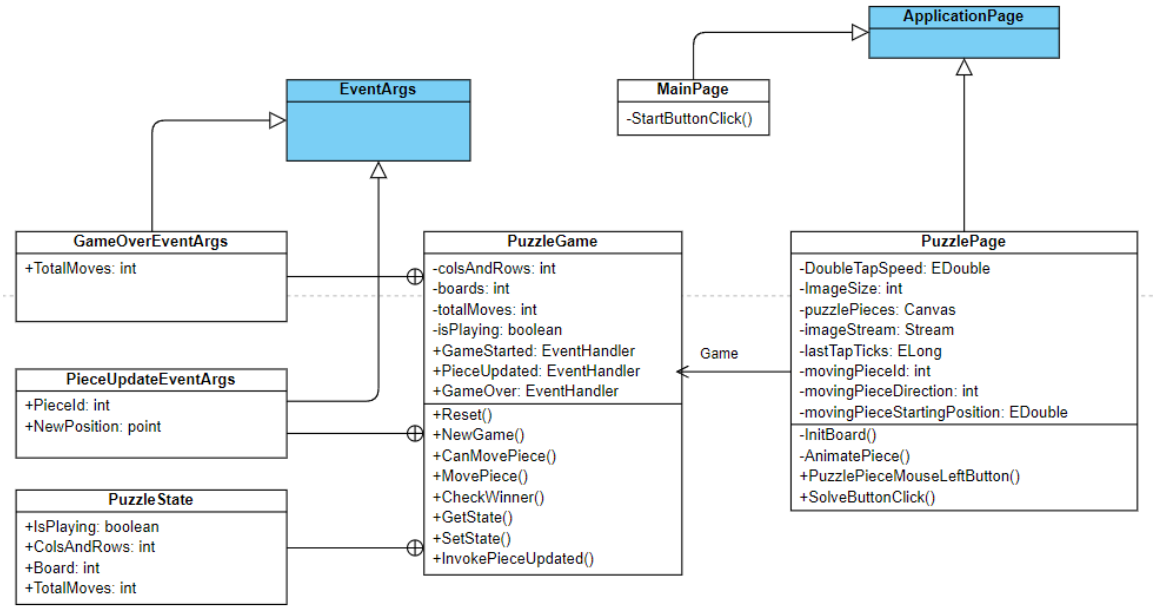

Рисунок 8 – Диаграмма классов игры «Пятнашки»

На этой диаграмме классы PuzzlePage и MainPage описаны как специализации класса ApplicationPage.

«Диаграмма классов также описывает две специализации класса EventArgs: GameOverEventArgs и PieceUpdateEventArgs. Аналогично, эта диаграмма также описывает ассоциативные отношения между PuzzlePage и PuzzleGame.

PuzzlePage отвечает за графический интерфейс и взаимодействие с пользователем. Этот класс расширяет ApplicationPage, который является классом, отвечающим за эти аспекты. Страница головоломки имеет множество атрибутов, таких как PuzzlePiece, imageStream, LastTapTicks, movingPieceId, movingPieceDirection и movingPieceStartingPosition, а также статические атрибуты, такие как DoubleTapSpeed и ImageSize.

PuzzleGame отвечает за управление игрой, обеспечивая такие операции, как Reset, NewGame, CanMovePiece, MovePiece, CheckWinner, GetState, SetState и ignorePieceUpdated.

Этот класс имеет множество атрибутов, таких как colsAndRows, board, totalMoves, isPlaying, а также атрибуты событий, такие как GameStarted, PieceUpdated и GameOver» [9].

На рисунке 9 представлен фрагмент программного кода игры «Пятнашки» (полный код программы расположен в Приложении А).

```
1  from random import shuffle
2  from tkinter import Canvas, Tk
3
4  BOARD_SIZE = 4
5  SQUARE_SIZE = 80
6  EMPTY_SQUARE = BOARD_SIZE ** 2
7  root = Tk()
8  root.title("Игра Пятнашки")
9  c = Canvas(root, width=BOARD_SIZE * SQUARE_SIZE,
10             height=BOARD_SIZE * SQUARE_SIZE, bg='#808080')
11 c.pack()
```

Рисунок 9 – Листинг игры «Пятнашки»

В процессе реализации программы использованы следующие модули библиотеки языка Python:

– random – модуль, реализующий генераторы псевдослучайных чисел для различных распределений;
– tkinter – стандартный интерфейс Python для набора инструментов Tcl/Tk GUI [11].

На рисунках 10 и 11 показаны скриншоты игры «Пятнашки».

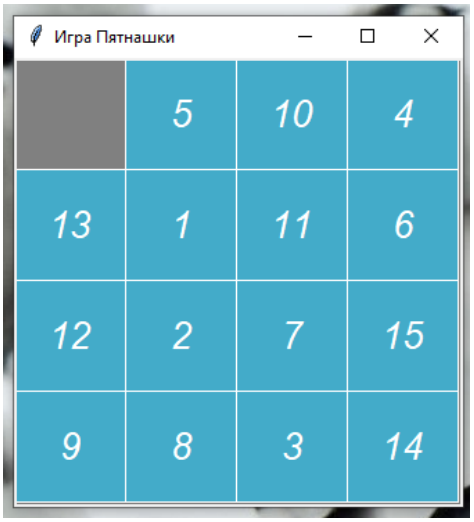

Рисунок 10 – Скриншот начального состояния игры «Пятнашки»

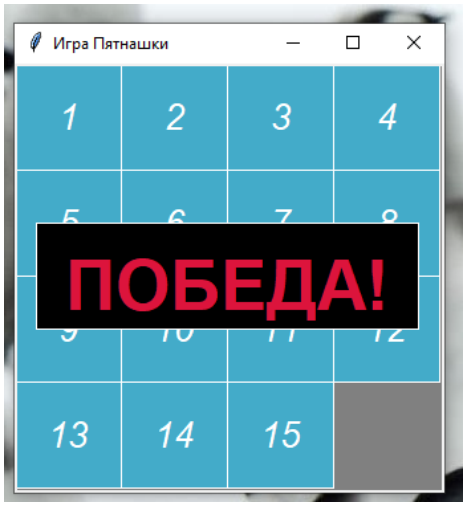

Рисунок 11 – Скриншот успешного завершения игры «Пятнашки»

Таким образом, тестирование программы игры «Пятнашки» подтвердило ее работоспособность.

3.3 Реализация и тестирование решателя игры «Пятнашки»

Как было отмечено выше, задачи, решаемые с помощью алгоритма A*, отличаются определением вершин графа (или состояниями).

Определение вершин графа (или состояний): в алгоритме A* каждая вершина графа представляет собой некоторое состояние задачи, которую мы пытаемся решить. Эти состояния могут быть самыми разными в зависимости от конкретной задачи: они могут описывать положение робота на карте, возможные ходы в шахматной партии, состояние логической схемы и т.д.

На рисунке 12 представлена диаграмма классов решателя игры «Пятнашки» [6].

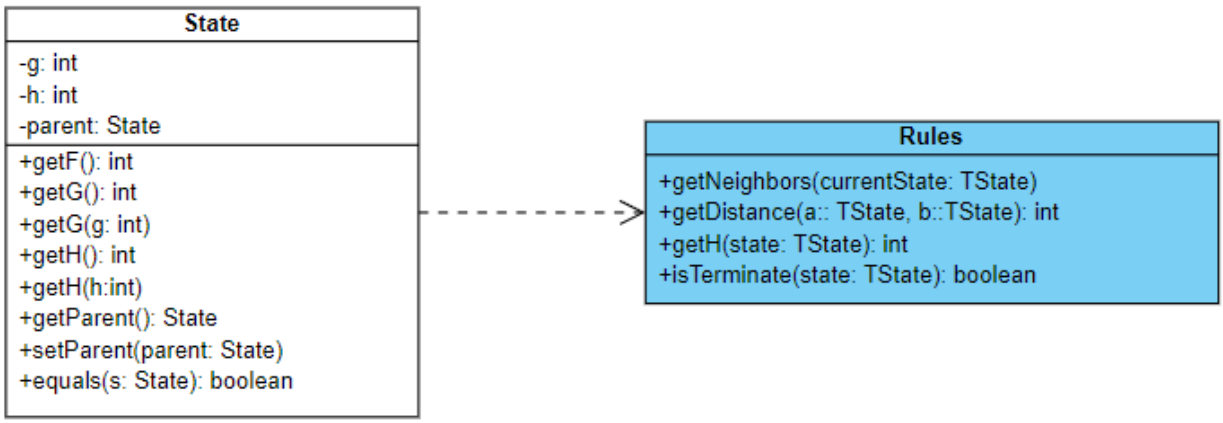

Рисунок 12 – Диаграмма классов решателя игры «Пятнашки»

Ключевым моментом является то, что каждая вершина должна содержать всю необходимую информацию для описания текущего состояния задачи.

Для того, чтобы обеспечить единообразное представление состояний (вершин) графа, используется абстрактный класс State. Этот класс определяет

базовые свойства и методы, которые будут общими для всех состояний, такие как уникальный идентификатор состояния, методы сравнения состояний, вычисления хэш-кода и т.д.

Конкретные реализации состояний (вершин) будут наследоваться от этого абстрактного класса и добавлять свою специфическую логику, связанную с предметной областью задачи.

Одним из ключевых аспектов алгоритма A* является правила, по которым из текущей вершины (состояния) порождаются дочерние вершины (новые состояния). Эти правила определяют, какие возможные переходы из текущего состояния в новые состояния являются допустимыми в рамках решаемой задачи.

Алгоритм A* использует фактическое расстояние (стоимость) от начальной вершины до текущей вершины для оценки пути. Это расстояние рассчитывается на основе правил перехода между вершинами, определенных в предыдущем пункте.

Для ускорения поиска оптимального пути алгоритм A* также использует эвристическую оценку расстояния от текущей вершины до целевой (терминальной) вершины. Эта эвристическая оценка должна быть корректной (не завышать реальное расстояние) и как можно более точной, чтобы направлять поиск в наиболее перспективном направлении.

Для инкапсуляции всех вышеперечисленных особенностей (правила порождения дочерних вершин, расчет расстояний и эвристических оценок) используется интерфейс Rules. Этот интерфейс определяет контракт, который должны реализовывать конкретные реализации правил для различных задач.

Таким образом, алгоритм A* может работать с любой задачей, реализующей интерфейс Rules, не зависимо от конкретных деталей предметной области.

На рисунке 13 представлен фрагмент программного кода решателя игры «Пятнашки» (полный код программы расположен в Приложении Б).

```python
import random, time

DIFFICULTY = 40 # Со скольких случайных слайдов начинается головоломка.
SIZE = 4 # Доска имеет размер SIZE x SIZE клеток.
random.seed(1) # Выберите, какую головоломку нужно решить

BLANK = 0
UP = 'наверх'
DOWN = 'вниз'
LEFT = 'налево'
RIGHT = 'направо'

def displayBoard(board):
    """Отобразить плитки, хранящиеся в 'доске', на экране"""
    for y in range(SIZE): # Iterate over each row.
        for x in range(SIZE): # Iterate over each column.
            if board[y * SIZE + x] == BLANK:
                print('__ ', end='') # Display blank tile.
            else:
                print(str(board[y * SIZE + x]).rjust(2) + ' ', end='')
        print() # Print a newline at the end of the row.


def getNewBoard():
    """Возвращает список, представляющий новую головоломку с плиткой"""
    board = []
    for i in range(1, SIZE * SIZE):
        board.append(i)
    board.append(BLANK)
    return board
```

Рисунок 13 – Листинг игры «Пятнашки»

В процессе реализации программы использован модуль time, который предоставляет функции для решения задач, связанных со временем. В данном программном коде представлен фрагмент, который ведет отсчет времени выполнения решателя (рисунок 14).

```python
start_time = time.time()  # Запускаем таймер

    while queue:
        board, path = queue.popleft()

        if is_winning(board):  # Если текущее состояние поля выигрышное, возвращаем путь
            end_time = time.time()  # Останавливаем таймер
            execution_time = end_time - start_time
            print(f"Время выполнения: {execution_time} секунд")
            return path

        visited.add(tuple(map(tuple, board)))  # Добавляем текущее состояние поля в множество посещенных

        next_states = generate_next_states(board)  # Генерируем возможные следующие состояния поля

        for state in next_states:
            if tuple(map(tuple, state)) not in visited:  # Если следующее состояние поля не посещено
                queue.append((state, path + [state]))  # Добавляем его в очередь с обновленным путем

    end_time = time.time()  # Останавливаем таймер
```

Рисунок 14 – Фрагмент кода по работе с таймером

На рисунке 15 показан скриншот результата работы решателя игры «Пятнашки».

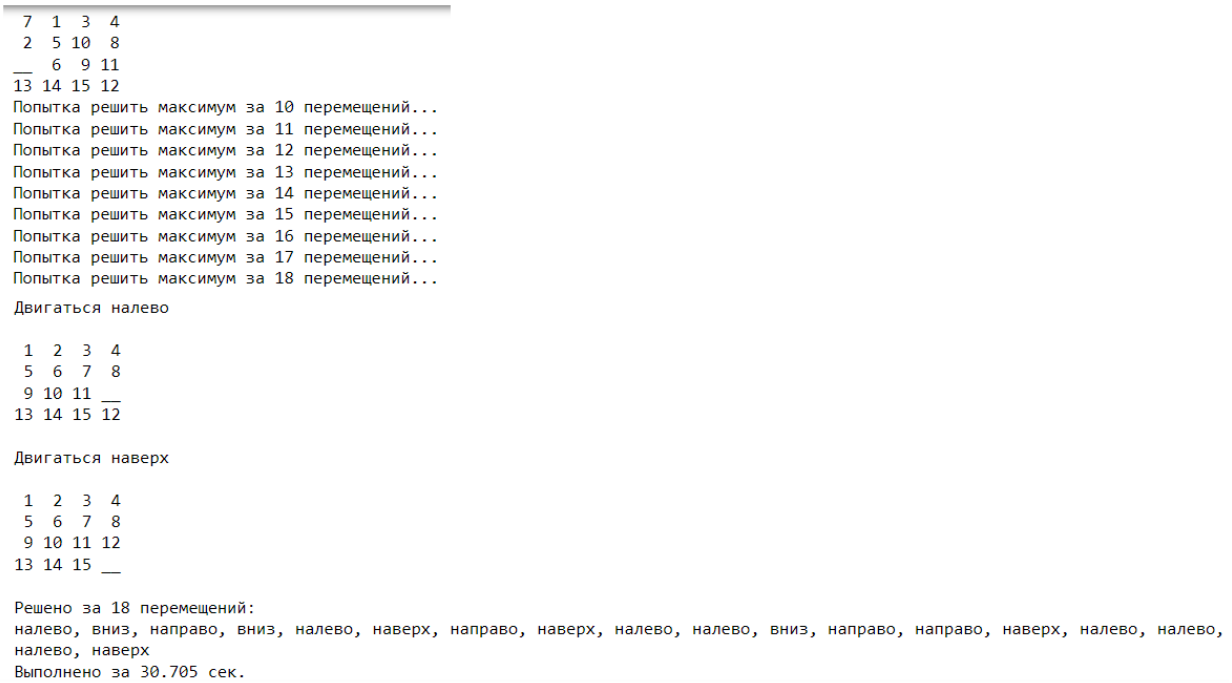

Рисунок 15 – Скриншот результата работы решателя игры «Пятнашки»

Таким образом, тестирование программы решателя игры «Пятнашки» подтвердило ее работоспособность.

Выводы по главе 3

Для реализации игры «Пятнашки» выбраны язык Python и среда Jupyter Notebook.

Тестирование программ игры «Пятнашки» и ее решателя «Пятнашки» подтвердили их работоспособность.

Заключение

Бакалаврская работа посвящена актуальной проблеме исследования и программной реализации алгоритма компьютерной игры на примере головоломки «Пятнашки».

Выполненные в рамках бакалаврской работы задачи представлены следующими основными результатами:

- произведена постановка задачи исследования алгоритма компьютерной игры «Пятнашки». Как показал анализ, поиск оптимального решения (наименьшего числа ходов) является NP-полным. Головоломка разрешима тогда и только тогда, когда в последовательности имеется четное число инверсий (неупорядоченных пар). Путем полного поиска в ширину было доказано, что ни одна конфигурация головоломки из 15 не требует более 80 ходов. Для решения головоломки «Пятнашки» используются алгоритмы наподобие алгоритма A*. Алгоритм IDA* полезен, когда проблема связана с нехваткой памяти. Как и в случае A*, эвристика должна иметь определенные свойства, чтобы гарантировать оптимальность (кратчайшие пути). Поставлена задача исследования и программной реализации алгоритма A* для решения головоломки «Пятнашки»;
- исследована возможности применения алгоритма A* для решения головоломки «Пятнашки». Как показал анализ, эвристический поиск – это метод информированного поиска. Эвристическое значение сообщает алгоритму, какой путь обеспечит решение как можно раньше. Эвристическая функция используется для генерации этого эвристического значения. Ключевой особенностью алгоритма A* является то, что он отслеживает каждый посещенный узел, что помогает игнорировать уже посещенные узлы, экономя огромное количество времени. У него также есть список, содержащий все

узлы, которые осталось изучить, и он выбирает наиболее оптимальный узел из этого списка, тем самым экономя время, не исследуя ненужные или менее оптимальные узлы. Результаты исследования подтвердили возможность использования алгоритма A* для решения головоломки «Пятнашки»;

- выполнены реализация и тестирование программ компьютерной игры «Пятнашки» и ее решателя. Для реализации программ выбраны язык Python и среда Jupyter Notebook. В процессе реализации программы использованы модули библиотеки языка Python random, tnicker и time. Построены диаграммы классов программ и описана их спецификация. Выполнено функциональное тетсирование разработанных программ. Тестирование программ игры «Пятнашки» и ее решателя «Пятнашки» подтвердили их работоспособность.

Результаты бакалаврской работы могут представлять интерес для разработчиков и специалистов, занимающимся разработкой компьютерных игр-головоломок.

Список используемой литературы и используемых источников

1. Введение в алгоритм A* [Электронный ресурс]. URL: https://habr.com/ru/articles/331192/ (дата обращения: 20.04.2024).

2. Воронина В. В. Программирование игр: алгоритмы и технологии : учебное пособие. Ульяновск : Ульяновский государственный технический университет, 2017. 306 с. URL: https://www.iprbookshop.ru/106113.html (дата обращения: 26.04.2024).

3. Глозман Ж. М., Курдюкова С.В., Сунцова А.В. Развиваем мышление. Игры, упражнения, советы специалиста. Саратов : Вузовское образование, 2013. 78 c. URL: https://www.iprbookshop.ru/11270.html (дата обращения: 26.04.2024).

4. Есть ли польза от головоломок? [Электронный ресурс]. URL: https://cccstore.ru/blog/articles/est-li-polza-ot-golovolomok/ (дата обращения: 20.04.2024).

5. Златопольский Д. М. Программирование: типовые задачи, алгоритмы, методы. Москва : Лаборатория знаний, 2020. 224 c. URL: https://www.iprbookshop.ru/12264.html (дата обращения: 204.04.2024).

6. Интеллектуальные системы. Алгоритм A* и игра «Пятнашки» [Электронный ресурс]. URL: https://www.dokwork.ru/2012/03/blog-post.html (дата обращения: 20.04.2024).

7. Исхаков И.И., Сазонов Д.В., Фолунин В.А. Алгоритм поиска A* на примере игры «Пятнашки» // Информатика, моделирование, автоматизация проектирования (ИМАП-2016). VIII Всероссийская школа-семинар аспирантов, студентов и молодых ученых : сборник научных трудов. 2016 г. С. 129-134.

8. Как написать решатель «Пятнашек» на C# [Электронный ресурс]. URL: https://habr.com/ru/companies/skillfactory/articles/655629/ (дата обращения: 20.04.2024).

9. Леоненков А. В. Объектно-ориентированный анализ и

проектирование с использованием UML и IBM Rational Rose : учебное пособие. М. : ИНТУИТ, Ай Пи Ар Медиа, 2020. 317 с. [Электронный ресурс]. URL: https://www.iprbookshop.ru/97554.html (дата обращения: 20.04.2024).

10. Пятнашки – история и описание игры [Электронный ресурс]. URL: https://www.i-igrushki.ru/igrushkapedia/pyatnashki.html (дата обращения: 20.04.2024).

11. Сузи Р. А. Язык программирования Python : учебное пособие. Москва : Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2020. 350 с. URL: https://www.iprbookshop.ru/97589.html (дата обращения: 26.04.2024).

12. Abilio G. Parada et al. Automating mobile application development: UML-based code generation for Android and Windows Phone, RITA, Vol. 22(2). 2015. P. 32-50.

13. Beeler R.“The Fifteen Puzzle: A Motivating Example for the Alternating Group”, East Tennessee State University. Retrieved, 2020-12-26.

14. BFS algorithm [Электронный ресурс]. URL: https://www.javatpoint.com/breadth-first-search-algorithm (дата обращения: 20.04.2024).

15. Culberson J.C., Schaeffer J. Efficiently searching the 15- puzzle. Technical report 94-08 (unpublished), 1994.

16. How to solve any slide puzzle regardless of its size [Электронный ресурс]. URL: https://www.kopf.com.br/kaplof/how-to-solve-any-slide-puzzle-regardless-of-its-size/ (дата обращения: 20.04.2024).

17. Introduction to Heuristic Search Algorithms [Электронный ресурс]. URL: https://www.javatpoint.com/iterative-deepening-a-algorithm (дата обращения: 20.04.2024).

18. Iterative Deepening A* algorithm (IDA*) [Электронный ресурс]. URL: https://www.geeksforgeeks.org/iterative-deepening-a-algorithm-ida-artificial-intelligence/ (дата обращения: 20.04.2024).

19. Korf R.E., Felner A. Disjoint pattern database heuristics. Artificial

Intelligence, volume 134, January 2002, pp.9-22, 2002.

20. Korf R.E., Schultze P. Large-scale parallel breadth-first search. Proceedings of the national conference on artificial intelligence, 2005.

21. Project Jupyter [Электронный ресурс]. URL: https://jupyter.org/ (дата обращения: 20.04.2024).

22. Python IDLE [Электронный ресурс]. URL: https://docs.python.org/3/library/idle.html (дата обращения: 20.04.2024).

23. Sliding puzzle solver [Электронный ресурс]. URL: https://github.com/gojkovicmatija99/Sliding-puzzle-solver?tab=readme-ov-file (дата обращения: 20.04.2024).

24. Spaans R. Solving sliding-block puzzles. URL: https://www.pvv.ntnu.no/~spaans/spec-cs.pdf (дата обращения: 20.04.2024).

25. Visual Paradigm: Online Productivity Suite [Электронный ресурс]. URL: https://online.visual-paradigm.com/ (дата обращения: 20.04.2024).

Приложение А

Программный код игры «Пятнашки»

```
import pygame
import heapq

# Размер окна
WINDOW_SIZE = (400, 400)
# Цвета
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
BLUE = (0, 0, 255)
# Размер ячейки
CELL_SIZE = WINDOW_SIZE[0] // 4
# Задержка между ходами (в миллисекундах)
DELAY = 500

# Класс для представления состояния игры
class GameState:
    def __init__(self, board, parent=None, move=""):
        self.board = board
        self.parent = parent
        self.move = move
        self.g = 0  # Стоимость пути от начального состояния до текущего состояния
        self.h = 0  # Эвристическая оценка оставшейся стоимости до целевого состояния

    def __lt__(self, other):  # Метод для сравнения состояний игры
        return self.g + self.h < other.g + other.h
```

```
    def is_goal(self, target):
        return self.board == target

    def generate_moves(self):
        moves = []
        empty_pos = self.board.index(0)
        row, col = empty_pos // 4, empty_pos % 4

        if row > 0:
            moves.append((row - 1, col, "Up"))
        if row < 3:
            moves.append((row + 1, col, "Down"))
        if col > 0:
            moves.append((row, col - 1, "Left"))
        if col < 3:
            moves.append((row, col + 1, "Right"))

        return moves

    def make_move(self, move):
        new_board = self.board[:]
        empty_pos = new_board.index(0)
        row, col = empty_pos // 4, empty_pos % 4
        new_row, new_col, move_name = move
        new_pos = new_row * 4 + new_col
        new_board[empty_pos], new_board[new_pos] = new_board[new_pos],
new_board[empty_pos]
```

```
        return GameState(new_board, self, move_name)

    def calculate_heuristic(self, target):
        count = 0
        for i in range(16):
            if self.board[i] != target[i]:
                count += 1
        return count

    def get_solution(self):
        moves = []
        current_state = self
        while current_state.parent is not None:
            moves.append(current_state.move)
            current_state = current_state.parent
        moves.reverse()
        return moves


def solve_puzzle(initial_state, target):
    open_set = []
    closed_set = set()

    initial_state.h = initial_state.calculate_heuristic(target)
    heapq.heappush(open_set, initial_state)

    while open_set:
        current_state = heapq.heappop(open_set)
```

```
            closed_set.add(tuple(current_state.board))

            if current_state.is_goal(target):
                return current_state.get_solution()

            for move in current_state.generate_moves():
                new_state = current_state.make_move(move)
                if tuple(new_state.board) not in closed_set:
                    new_state.g = current_state.g + 1
                    new_state.h = new_state.calculate_heuristic(target)
                    heapq.heappush(open_set, new_state)

        return None


    def draw_board(screen, board):
        screen.fill(WHITE)
        for i in range(4):
            for j in range(4):
                num = board[i * 4 + j]
                if num != 0:
                    pygame.draw.rect(screen, BLUE, (j * CELL_SIZE, i *
CELL_SIZE, CELL_SIZE, CELL_SIZE))
                    font = pygame.font.Font(None, 36)
                    text = font.render(str(num), True, WHITE)
                    text_rect = text.get_rect(center=(j * CELL_SIZE + CELL_SIZE //
2, i * CELL_SIZE + CELL_SIZE // 2))
                    screen.blit(text, text_rect)
```

```
    pygame.display.flip()

def visualize_solution(board, solution):
    pygame.init()
    screen = pygame.display.set_mode(WINDOW_SIZE)
    pygame.display.set_caption(«Пятнашки»)

    clock = pygame.time.Clock:

    draw_board(screen, board)
    pygame.time.wait(DELAY)

    for move in solution:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                return

        if move == "Up":
            board.move_up()
        elif move == "Down":
            board.move_down()
        elif move == "Left":
            board.move_left()
        elif move == "Right":
            board.move_right()

        draw_board(screen, board)
```

```
            pygame.time.wait(DELAY)

        while True:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    return


if __name__ == "__main__":
    initial_board = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0]
    target_board = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0]

    initial_state = GameState(initial_board)
    solution = solve_puzzle(initial_state, target_board)

    if solution:
        print("Решение найдено:")
        for move in solution:
            print(move)
        visualize_solution(initial_board, solution)
    else:
        print("Решение не найдено.")
```

Приложение Б

Программный код решателя игры «Пятнашки»

```
from collections import deque

# Проверка, является ли состояние поля выигрышным
def is_winning(board):
    nums = [num for row in board for num in row]
    return nums == sorted(nums)

# Находим позицию пустой ячейки на игровом поле
def find_empty_cell(board):
    for i in range(4):
        for j in range(4):
            if board[i][j] == 0:
                return i, j

# Генерируем возможные следующие состояния поля
def generate_next_states(board):
    next_states = []
    i, j = find_empty_cell(board)

    if j > 0:
        next_board = [row[:] for row in board]
        next_board[i][j], next_board[i][j-1] = next_board[i][j-1], next_board[i][j]
        next_states.append(next_board)

    if j < 3:
        next_board = [row[:] for row in board]
```

```
            next_board[i][j], next_board[i][j+1] = next_board[i][j+1], next_board[i][j]
            next_states.append(next_board)

        if i > 0:
            next_board = [row[:] for row in board]
            next_board[i][j], next_board[i-1][j] = next_board[i-1][j], next_board[i][j]
            next_states.append(next_board)

        if i < 3:
            next_board = [row[:] for row in board]
            next_board[i][j], next_board[i+1][j] = next_board[i+1][j], next_board[i][j]
            next_states.append(next_board)

        return next_states

    # Решатель игры «Пятнашки» с использованием алгоритма поиска в ширину (BFS)
    def solve_puzzle(initial_board):
        queue = deque([(initial_board, [])])  # Очередь состояний поля для обхода
        visited = set()  # Множество посещенных состояний поля

        while queue:
            board, path = queue.popleft()
```

```
            if is_winning(board):  # Если текущее состояние поля выигрышное, возвращаем путь
                return path

            visited.add(tuple(map(tuple, board)))  # Добавляем текущее состояние поля в множество посещенных

            next_states = generate_next_states(board)  # Генерируем возможные следующие состояния поля

            for state in next_states:
                if tuple(map(tuple, state)) not in visited:  # Если следующее состояние поля не посещено
                    queue.append((state, path + [state]))  # Добавляем его в очередь с обновленным путем

        return None  # Если решение не найдено

    # Пример использования решателя
    initial_board = [
        [1, 2, 3, 4],
        [5, 6, 0, 8],
        [9, 10, 7, 12],
        [13, 14, 11, 15]
    ]

    solution = solve_puzzle(initial_board)
    if solution:
```

```
    print("Решение найдено:")
    for step, board in enumerate(solution):
        print(f"Шаг {step + 1}:")
        for row in board:
            print(row)
        print()
else:
    print("Решение не найдено.")
```