**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ ФГБОУ ВО «БРЯНСКИЙ ГОСУДАРСТВЕНННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

## Кафедра «Информатика и программное обеспечение»

КУРСОВАЯ РАБОТА

**ПО ДИСЦИПЛИНЕ «СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА»** СРАВНИТЕЛЬНЫЙ АНАЛИЗ АЛГОРИТМОВ НЕИНФОРМИРОВАННОГО ПОИСКА И ЭВРИСТИЧЕСКОГО ПОИСКА

Студент О-20-МОА-тп-Б

Игнатьев И. П.

« » \_\_\_\_\_\_\_\_\_2023 г. Преподаватель

Бабурин А. Н.

« » \_\_\_\_\_\_\_\_\_2023 г.

**БРЯНСК 2023**

**СОДЕРЖАНИЕ**

[Введение 3](#_bookmark0)

1. [Аналитическая часть 4](#_bookmark1)
   1. [Поиск в ширину 4](#_bookmark2)
   2. [Поиск в глубину с итеративным углублением 5](#_bookmark3)
   3. [Эвристический (информативный) поиск 6](#_bookmark4)
   4. [Сравнительный анализ методов поиска 7](#_bookmark5)
2. [Конструкторская часть 9](#_bookmark6)
   1. [Описание задачи 9](#_bookmark7)
   2. [Описание разработанной программы 9](#_bookmark8)
   3. [Описание разработанных алгоритмов поиска 10](#_bookmark9)
      1. [Поиск в ширину 12](#_bookmark10)
      2. [Поиск в глубину с итеративным углублением 14](#_bookmark11)
      3. [Эвристический поиск 16](#_bookmark12)

[2.3.5 Сравнение эвристических функций информативного поиска 17](#_bookmark13)

1. [Тестирование алгоритмов поиска 18](#_bookmark14)

[Заключение 23](#_bookmark15)

[Список литературы 24](#_bookmark16)

# ВВЕДЕНИЕ

В данной работе будет проведен сравнительный анализ алгоритмов неинформативного и эвристического поиска с целью определение более эффективного подхода для различных ситуаций. Данные для анализа были получены при использовании лабораторных работ, выполненных ранее. Чтобы получить объективные выводы о эффективности алгоритмов, введем ключевые параметры для их оценки:

1. Количество итераций цикла поиска до нахождения решения;
2. Максимальное количество узлов, хранимое единовременно в памяти во время поиска решения;
3. Эффективный коэффициент ветвления (обозначающийся как b\*).

Сравнение будет проводиться на задачах разной сложности, где сложность оценивается глубиной решения задачи.

# АНАЛИТИЧЕСКАЯ ЧАСТЬ

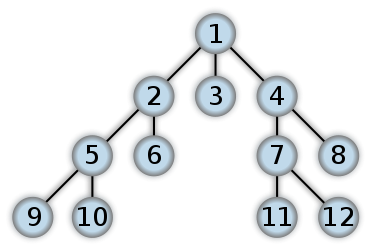
## Поиск в ширину

Поиск в ширину — один из простейших алгоритмов обхода графа, являющийся основой для многих важных алгоритмов для работы с графами. Он используется для исследования всех вершин графа путем поуровневого обхода каждой из них.

Поиск в ширину работает следующим образом: алгоритм начинает в заранее выбранной вершине и сначала «посещает» и отмечает всех соседей этой вершины (рис. 1). Потом он переходит к соседям посещенных вершин, затем — дальше по тому же принципу. Так продолжается до тех пор, пока все вершины не будут исследованы. Из-за характера распространения, похожего на волну, алгоритм еще называют волновым.

Для реализации алгоритма поиска в ширину принято использовать очередь: для каждой вершины, посещенной алгоритмом, все достижимые из нее соседи помещаются в очередь, из которой будет взята следующая вершина для дальнейшего обхода.

Данный алгоритм наиболее часто применяется для поиска пути между двумя вершинами графа, так как поиск в ширину гарантирует нахождение кратчайшего пути в графе, а его эффективность не зависит от структуры самого графа.



*Рис. 1: Порядок рассмотрения вершин в поиске в ширину*

## Поиск в глубину с итеративным углублением

Алгоритм поиска в глубину с итеративным углублением используется для определения наилучшей глубины поиска. Постепенно увеличивая предел глубины (начиная с 0 и увеличивая на 1), алгоритм ищет целевую вершину. Когда предел глубины достигает глубины целевой вершины, алгоритм останавливается.

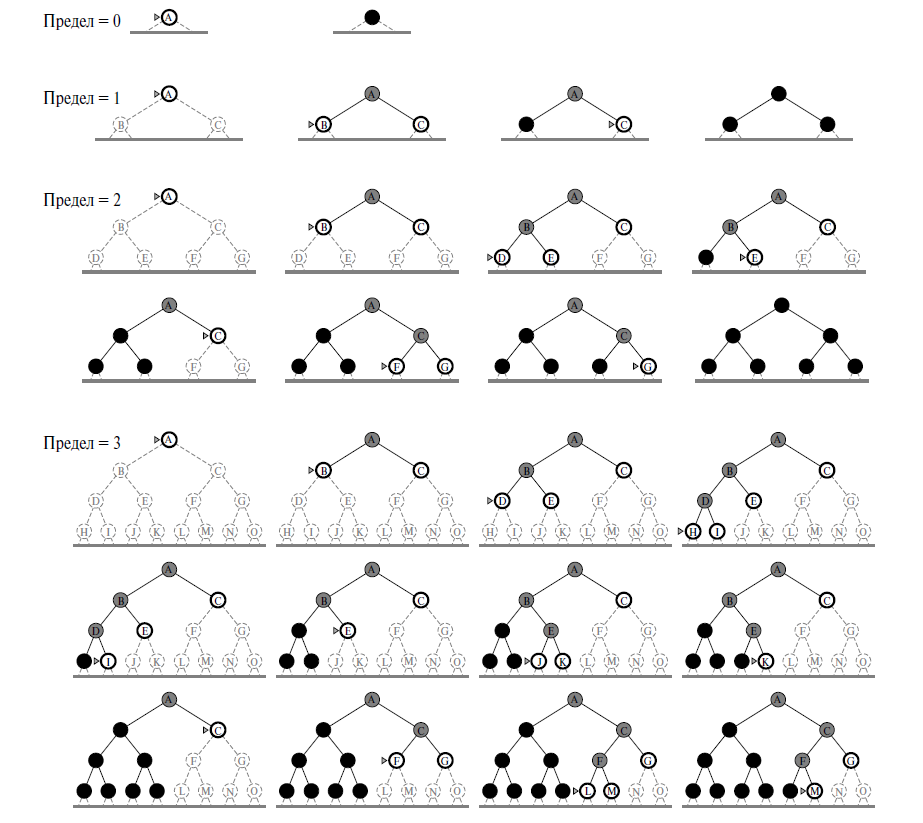
Этот алгоритм сочетает преимущества поиска в глубину и в ширину. Он имеет небольшие требования к памяти (O(bd)) и является полным, если коэффициент ветвления конечен, и оптимальным, если стоимость пути не убывает с глубиной узла. На рисунке 2 показаны четыре итерации поиска в глубину с итеративным углублением в бинарном дереве поиска, где целевая вершина найдена на четвертой итерации.

Рис. 2. Поиск в глубину с итеративным углублением

## Эвристический (информативный) поиск

Информативный поиск — стратегия поиска решений в пространстве состояний, в которой используются знания, относящиеся к конкретной задаче. Пространство состояний представляет собой граф, где узлы - состояния задачи, а ребра - операции, переводящие из одного состояния в другое. Информированные методы обычно обеспечивают более эффективный поиск по сравнению с неинформированными методами.

Информация о конкретной задаче формулируется в виде эвристической функции. Эвристическая функция на каждом шаге перебора оценивает альтернативы на основании дополнительной информации с целью принятия решения о том, в каком направлении следует продолжать перебор.

В качестве примера эвристического поиска можно рассмотреть алгоритм A\*. A\* на каждом шагу просматривает все пути, ведущие от начальной вершины в конечную, пока не найдёт минимальный. Как и все информированные алгоритмы поиска, он просматривает сначала те маршруты, которые «кажутся» ведущими к цели. От жадного алгоритма, который тоже является алгоритмом поиска по первому лучшему совпадению, его отличает то, что при выборе вершины он учитывает, помимо прочего, весь пройденный до неё путь. Составляющая g(x) — это стоимость пути от начальной вершины, а не от предыдущей, как в жадном алгоритме.

В начале работы просматриваются узлы, смежные с начальным; выбирается тот из них, который имеет минимальное значение f(x), после чего этот узел раскрывается. На каждом этапе алгоритм оперирует с множеством путей из начальной точки до всех ещё не раскрытых (листовых) вершин графа

— множеством частных решений, — которое размещается в очереди с приоритетом. Приоритет пути определяется по значению f(x) = g(x) + h(x). Алгоритм продолжает свою работу до тех пор, пока значение f(x) целевой вершины не окажется меньшим, чем любое значение в очереди, либо пока всё дерево не будет просмотрено. Из множества решений выбирается решение с наименьшей стоимостью.

## Сравнительный анализ методов поиска

Сравнительный анализ перечисленных выше алгоритмов представлен в таблице 1.

***Таблица 1***

**Сравнение алгоритмов поиска**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Поиск в ширину | Поиск в глубину с итеративным углублением | Эвристический поиск |
| Тип | Неинформативный | Неиформативный | Информативный |
| Оптимальнос ть | Гарантировано | Гаратировано | Гарантировано |
| Время | O(n+m) | O(bd) | Зависит от эвристики и структуры пространства  состояний |
| Эффективнос ть | Эффективен для поиска в больших деревьях с неограниченной  памятью | Эффективен для поиска в больших графах с ограниченной памятью | Эффективен для задач с определенной целью и хорошей эвристикой |
| Хранение узлов | Очередь | Стек | Очередь с приоритетом |
| Поиск пути | Гарантирует нахождение оптимального пути | Гарантирует нахождение оптимального пути | Гарантирует поиск оптимального пути, при условии допустимой эвристики |

Исходя из сравнительной таблицы, можно сделать следующие выводы:

1. Поиск в ширину подходит для простых задач, но может быть неэффективен для больших деревьев и графов;
2. Поиск в глубину с итеративным углублением считается улучшенной версией поиска в глубину, так как гарантирует нахождение оптимального пути и эффективен для поиска в больших графах с ограниченной памятью.;
3. A\* является информированным (эвристическим) алгоритмом, который обеспечивает гарантированное нахождение оптимального пути при условии использования допустимой эвристики. A\* также эффективен для задач с определенной целью и хорошей эвристикой.

Таким образом, выбор конкретного алгоритма зависит от особенностей конкретной задачи.

# КОНСТРУКТОРСКАЯ ЧАСТЬ

## Описание задачи

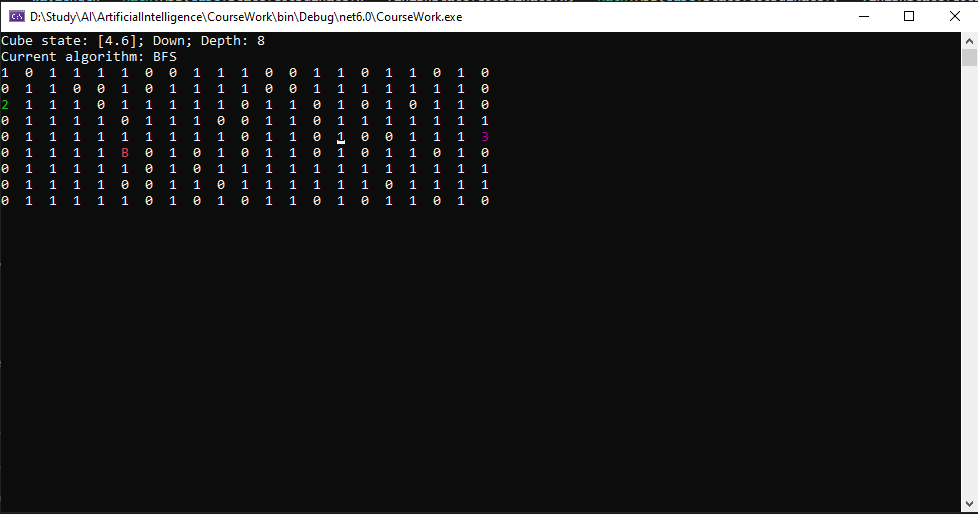
Необходимо разработать программу для нахождения решение в игре

«Кантуем куб».

Игра представляет из себя поле с неограниченным размером. Имеется 2 вида клеток: свободная, пропасть. В одном месте на свободных клетках расположен кубик, имеющий размеры 1x1x1. Одна из граней кубика красная. В другом месте поля стоит красная метка на одной из клеток. Цель – расположить кубик на помеченной клетке так, чтобы его красная грань была внизу. Куб можно только кантовать, т.е. «ронять» куб одной из граней на бок.

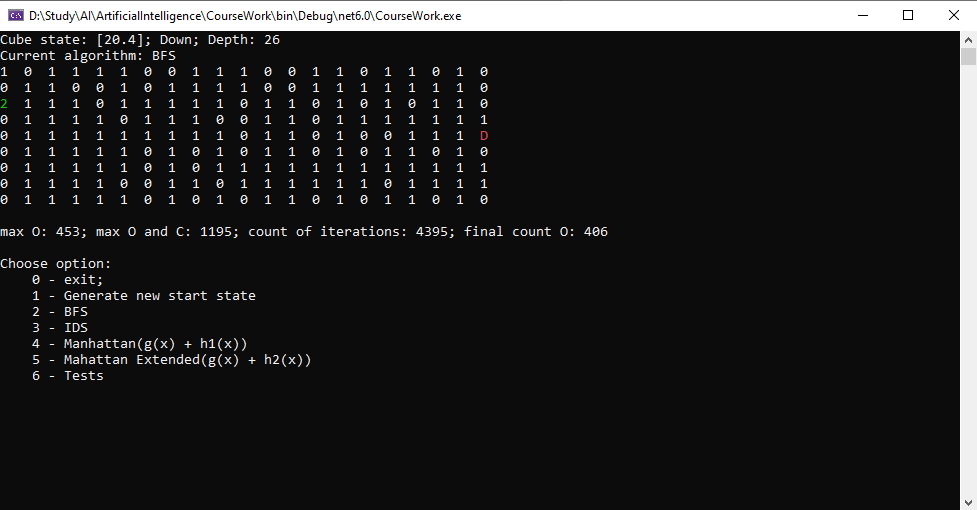
## Описание разработанной программы

Была разработана программа с графическим интерфейсом, представленным на рис. 3.



*Рис. 3. Графический интерфейс игры*

Для задания игровой карты необходимо изменить текстовый файл, идущий вместе с игрой, либо добавить новый и вписать его в коде программы.

При нахождении решения программа покажет путь от начального до конечного состояния, а также выведет всю необходимую для анализа информацию (рис. 4).

*Рис. 4. Результат поиска решения*

## Описание разработанных алгоритмов поиска

В ходе выполнения работы были разработаны 3 алгоритма поиска: 2 неинформативных и информативный (эвристический) с 2 различными эвристическими функциями.

Был разработан базовый класс Cube(см. листинг 1), который содержит в себе методы, описывающие логику передвижения куба.

***Листинг 1***

***Базовый класс Cube***

public class Cube

{

public State State;

public Cube(State state)

{

this.State = state;

}

public void Step(Coordinate coord)

{

if(coord.x < State.Coordinate.x && coord.y == State.Coordinate.y)

{

State.Direction = State.Direction switch

{

Direction.Up => Direction.Left,

Direction.Left => Direction.Down,

Direction.Down => Direction.Right,

Direction.Right => Direction.Up,

\_ => State.Direction

};

}

else if(coord.x > State.Coordinate.x && coord.y == State.Coordinate.y)

{

State.Direction = State.Direction switch

{

Direction.Up => Direction.Right,

Direction.Right => Direction.Down,

Direction.Down => Direction.Left,

Direction.Left => Direction.Up,

\_ => State.Direction

};

}

else if(coord.x == State.Coordinate.x && coord.y < State.Coordinate.y)

{

State.Direction = State.Direction switch

{

Direction.Up => Direction.Forward,

Direction.Forward => Direction.Down,

Direction.Down => Direction.Backward,

Direction.Backward => Direction.Up,

\_ => State.Direction

};

}

else if(coord.x == State.Coordinate.x && coord.y > State.Coordinate.y)

{

State.Direction = State.Direction switch

{

Direction.Up => Direction.Backward,

Direction.Backward => Direction.Down,

Direction.Down => Direction.Forward,

Direction.Forward => Direction.Up,

\_ => State.Direction

};

}

State.Coordinate = coord;

}

}

## Поиск в ширину

Реализация алгоритма поиска в ширину представлена в листинг 2. Для данного алгоритма были разработаны методы раскрытия узла, проверки на наличие рассматриваемого состояния в списках O и С.

***Листинг 2***

***Основной метод алгоритма поиска в ширину***

static Statistic FindWay(Cube cube, int maxDepth, bool isPrint = true)

{

cube.State.Depth = 0;

HashSet<State> closedStates = new HashSet<State>();

List<KeyValue> openedStates = new List<KeyValue>();

Statistic stat = new Statistic();

var neighbors = map.GetNeighbors(cube.State.Coordinate);

openedStates.Add(new KeyValue { Key = HeuristicFunction(cube), Value = new State { Coordinate = cube.State.Coordinate, Direction = cube.State.Direction } });

openedStates = openedStates.OrderBy(x => x.Key).ToList();

State tmpState = new State { Coordinate = new Coordinate { x = -1, y = -1 }, Direction = Direction.Forward };

List<State> finishStates = new List<State>();

while (openedStates.Count >= 0)

{

if(isPrint)

Print(map, cube, neighbors);

if (stat.Count == 0)

cube.State.ParentState = startState;

if (tmpState == finishState)

{

stat.IsHaveWay = true;

break;

}

if (stat.Count == 0)

{

tmpState = openedStates.First().Value;

openedStates.Remove(openedStates.First());

openedStates = openedStates.OrderBy(x => x.Key).ToList();

closedStates.Add(tmpState);

}

foreach (var neighbor in neighbors)

{

var neighborState = new State { Coordinate = neighbor, Direction = cube.DirectionAfterMove(neighbor), ParentState = cube.State, Depth = cube.State.Depth + 1 };

if (cube.State.Depth <= maxDepth)

{

var tmp = openedStates.Where(x => (x.Key >= 0) && (x.Key <= openedStates.Max(x => x.Key)) && (x.Key != HeuristicFunction(cube)) && (x.Value.ToString() == neighborState.ToString())).FirstOrDefault();

if (tmp != null)

{

if (tmp.Key > HeuristicFunction(cube))

{

tmp.Key = HeuristicFunction(cube);

openedStates.OrderBy(x => x.Key);

continue;

}

}

if (!openedStates.Contains(new KeyValue { Key = HeuristicFunction(cube), Value = neighborState }) && !closedStates.Contains(neighborState))

{

openedStates.Add(new KeyValue { Key = HeuristicFunction(cube), Value = neighborState });

openedStates = openedStates.OrderBy(x => x.Key).ToList();

}

}

}

if (openedStates.Count == 0)

break;

if (openedStates.Count == 0 && cube.State.Depth >= maxDepth)

break;

tmpState = openedStates.First().Value;

openedStates.Remove(openedStates.First());

openedStates = openedStates.OrderBy(x => x.Key).ToList();

closedStates.Add(tmpState);

if (stat.MaxO < openedStates.Count + 1)

stat.MaxO = openedStates.Count + 1;

if (stat.MaxOAndC < openedStates.Count + 1 + closedStates.Count())

stat.MaxOAndC = openedStates.Count + 1 + closedStates.Count();

cube.State = tmpState;

neighbors = map.GetNeighbors(cube.State.Coordinate);

stat.Count++;

}

stat.LastO = openedStates.Count;

return stat;

}

## Поиск в глубину с итеративным углублением

Для реализации данного алгоритма класс поиска решений был дополнен методом FindWayWithIterativeDepths, который вызывается в цикле for.

Алгоритма представлен в листинг 3.

***Листинг 3***

***Основной метод алгоритма поиска в глубину с итеративным углублением***

static Statistic FindWayWithIterativeDepths(Cube cube, int maxDepth, bool isPrint = true)

{

cube.State.Depth = 0;

HashSet<State> closedStates = new HashSet<State>();

Stack<State> openedStates = new Stack<State>();

Statistic stat = new Statistic();

var neighbors = map.GetNeighbors(cube.State.Coordinate);

openedStates.Push(new State { Coordinate = cube.State.Coordinate, Direction = cube.State.Direction });

State tmpState = new State { Coordinate = new Coordinate { x = -1, y = -1 }, Direction = Direction.Forward };

while (openedStates.Count >= 0)

{

if(isPrint)

Print(map, cube, neighbors);

if (stat.Count == 0)

cube.State.ParentState = startState;

if (tmpState == finishState)

{

stat.IsHaveWay = true;

break;

}

if (stat.Count == 0)

{

tmpState = openedStates.Pop();

closedStates.Add(tmpState);

}

foreach (var neighbor in neighbors)

{

var neighborState = new State { Coordinate = neighbor, Direction = cube.DirectionAfterMove(neighbor), ParentState = cube.State, Depth = cube.State.Depth + 1 };

if (cube.State.Depth <= maxDepth)

{

if (!openedStates.Contains(neighborState) && !closedStates.Contains(neighborState))

{

openedStates.Push(neighborState);

}

}

}

if (openedStates.Count == 0)

break;

if (openedStates.Count == 0 && cube.State.Depth >= maxDepth)

break;

tmpState = openedStates.Pop();

closedStates.Add(tmpState);

if (stat.MaxO < openedStates.Count() + 1)

stat.MaxO = openedStates.Count() + 1;

if (stat.MaxOAndC < openedStates.Count() + 1 + closedStates.Count())

stat.MaxOAndC = openedStates.Count() + 1 + closedStates.Count();

cube.State = tmpState;

neighbors = map.GetNeighbors(cube.State.Coordinate);

stat.Count++;

}

stat.LastO = openedStates.Count();

return stat;

}

## Эвристический поиск

Для вычисления эвристической функции разработано 2 метода: первый из них рассчитывает «Манхэттенское» расстояние текущей координаты куба до целевой (листинг 4), другая же также учитывает сколько дополнительных ходов нужно сделать кубу, что бы красная грань оказалась внизу (листинг 5).

***Листинг 4***

***Вычисление первой эвристической функции***

case Heuristic.Manhattan:

{

wayLength = Math.Abs(cube.State.Coordinate.x - finishState.Coordinate.x) + Math.Abs(cube.State.Coordinate.y - finishState.Coordinate.y);

result = cube.State.Depth + wayLength;

}break;

***Листинг 5***

***Вычисление второй эвристической функции***

case Heuristic.ManhattanExtended:

{

var tmpCube = new Cube(new State(cube.State));

while (tmpCube.State.Coordinate.x > finishState.Coordinate.x)

tmpCube.Step(new Coordinate { x = tmpCube.State.Coordinate.x - 1, y = tmpCube.State.Coordinate.y });

while (tmpCube.State.Coordinate.x < finishState.Coordinate.x)

tmpCube.Step(new Coordinate { x = tmpCube.State.Coordinate.x + 1, y = tmpCube.State.Coordinate.y });

while (tmpCube.State.Coordinate.y > finishState.Coordinate.y)

tmpCube.Step(new Coordinate { x = tmpCube.State.Coordinate.x, y = tmpCube.State.Coordinate.y - 1 });

while (tmpCube.State.Coordinate.y < finishState.Coordinate.y)

tmpCube.Step(new Coordinate { x = tmpCube.State.Coordinate.x, y = tmpCube.State.Coordinate.y + 1 });

wayLength = Math.Abs(cube.State.Coordinate.x - finishState.Coordinate.x) + Math.Abs(cube.State.Coordinate.y - finishState.Coordinate.y);

wayLength = tmpCube.State.Direction switch

{

Direction.Right => wayLength + 1,

Direction.Left => wayLength + 1,

Direction.Up => wayLength + 2,

Direction.Down => wayLength + 0,

Direction.Forward => wayLength + 1,

Direction.Backward => wayLength + 1

};

result = cube.State.Depth + wayLength;

}

break;

## Сравнение эвристических функций информативного поиска

Обе эвристики для вычисления своего значения используют «Манхэттенское» расстояние, однако вторая является более продуманной, так как она учитывает количество дополнительных ходов, которые придется совершить кубу для того, чтобы прийти в целевое состояние.

# ТЕСТИРОВАНИЕ АЛГОРИТМОВ ПОИСКА

На основе разработанной программы были проведены тесты для глубины решения d = 4-13.

Для каждой глубины были рассмотрены 10 различных состояний, полученных с помощью генератора состояний с заданной глубиной и взяты средние результаты, приведенные в таблицах 2, 3, 4.

***Таблица 2***

**Результаты стоимости поиска**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Стоимость поиска N | | | |
| d | BFS | IDS | A\*(h1) | A\*(h2) |
| 4 | 66 | 31 | 38 | 28 |
| 5 | 130 | 71 | 69 | 55 |
| 6 | 182 | 82 | 98 | 65 |
| 7 | 197 | 86 | 103 | 79 |
| 8 | 290 | 151 | 109 | 66 |
| 9 | 380 | 197 | 144 | 97 |
| 10 | 478 | 235 | 167 | 94 |
| 11 | 588 | 328 | 197 | 121 |
| 12 | 740 | 358 | 236 | 154 |
| 13 | 979 | 466 | 287 | 163 |

***Таблица 3***

**Результаты количества итераций**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Количество итераций | | | |
| d | BFS | IDS | A\*(h1) | A\*(h2) |
| 4 | 33 | 55 | 16 | 12 |
| 5 | 74 | 180 | 34 | 27 |
| 6 | 114 | 208 | 51 | 32 |
| 7 | 140 | 312 | 58 | 44 |
| 8 | 229 | 599 | 62 | 34 |
| 9 | 367 | 916 | 93 | 57 |
| 10 | 542 | 1342 | 106 | 51 |
| 11 | 625 | 2299 | 126 | 68 |
| 12 | 887 | 2555 | 154 | 89 |
| 13 | 1333 | 3616 | 196 | 87 |

***Таблица 4***

**Результаты эффективного коэффициента ветвления**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Эффективный коэффициент ветвления b\* | | | |
| d | BFS | IDS | A\*(h1) | A\*(h2) |
| 6 | 3,27 | 2,86 | 2,52 | 2,3 |
| 7 | 3,02 | 2,54 | 2,35 | 2,2 |
| 8 | 2,8 | 2,4 | 2,31 | 2,13 |
| 9 | 2,6 | 2,2 | 2,15 | 1,91 |
| 10 | 2,52 | 2,07 | 2,07 | 1,9 |
| 11 | 2,53 | 2,06 | 2,05 | 1,79 |
| 12 | 2,42 | 2 | 2,01 | 1,73 |
| 13 | 2,33 | 1,95 | 1,88 | 1,81 |
| 14 | 2,24 | 1,89 | 1,81 | 1,66 |
| 15 | 2,21 | 1,82 | 1,85 | 1,63 |

Из данных таблиц можно сделать вывод, что эвристические алгоритмы поиска гораздо превосходят неинформированные алгоритмы.

Экспериментальным путём выяснено, что алгоритм A\*(h2) лучше, чем A\*(h1), что и ожидалось на этапе разработки, но всё же оба эти алгоритма лучше IDS (поиск в глубину с итеративным углублением).

Для наглядности можно привести следующие графики (рис 6, 7, 8):

*Рис. 5. График стоимости пути*

*Рис. 6. График количества итераций*

*Рис. 7. График эффективного коэффициента ветвления b\**

Из графиков видны колоссальный рост стоимости поиска и количества итераций для стандартного метода поиска в ширину, небольшой рост для поиска в глубину с итеративным углублением и стабильность информированных методов. Также для информированных алгоритмов поиска эффективный коэффициент ветвления b\* стремится к 1.

# ЗАКЛЮЧЕНИЕ

В заключении работы можно сделать следующие вывод: аналитическое сравнение алгоритмов неинформированного и эвристического поиска позволило выявить превосходство эвристического подхода в решении поставленной задачи.

Исследование основывалось на данных из предыдущих лабораторных исследований, анализ был обоснован и объективен.

При сравнении алгоритмов они оценивались по трем ключевым параметрам: количество итераций цикла поиска, максимальное количество узлов, хранимых в памяти, и эффективный коэффициент ветвления. Эти параметры позволили точно оценить эффективность каждого алгоритма в различных ситуациях.

Сравнение было проведено на задачах разной сложности, где сложность определялась глубиной оптимального решения. Это позволило проверить применимость каждого алгоритма в разных сценариях.

В результате исследования напрашивается вывод, что эвристический подход является широко применимым в решении задач. Этот вывод подкреплен полученными экспериментальными данными и аналитическими вычислениями. Таким образом, результаты работы говорят в пользу использования эвристического поиска в решении задач поиска кратчайшего пути и являются

основанием для дальнейших исследований в этой области.

# СПИСОК ЛИТЕРАТУРЫ

* + - 1. Расселл, С., & Норвиг, П. (2016). Искусственный интеллект: Современный подход. Pearson Education.
      2. Нильссон, Н. Й. (1998). Искусственный интеллект: Новый синтез. Morgan Kaufmann.
      3. Лугер, Г. Ф., & Стаблфилд, У. А. (2004). Искусственный интеллект: Структуры и стратегии для решения сложных проблем. Pearson Education.
      4. Перл, Дж. (1984). Эвристика: Интеллектуальные стратегии поиска решений компьютерных проблем. Addison-Wesley.
      5. Харт, П. Е., Нильссон, Н. Й., & Рафаэль, Б. (1968). Формальная основа эвристического определения путей минимальной стоимости. IEEE Transactions on Systems Science and Cybernetics, 4(2), 100-107.
      6. Ньюэлл, А., & Саймон, Х. А. (1972). Человеческое решение проблем. Prentice-Hall.