

АЛГОРИТМ АНАЛИЗА КРАТЧАЙШИХ МАРШРУТОВ В ДИНАМИЧЕСКИ РЕКОНФИГУРИРУЕМЫХ ГРАФОВЫХ СТРУКТУРАХ С ПЕРЕМЕННОЙ ТОПОЛОГИЕЙ: ОЦЕНКА СРЕДНЕГО ЧИСЛА ПРЕОБРАЗОВАНИЙ ДЛЯ ДОСТИЖЕНИЯ ЦЕЛЕВЫХ УЗЛОВ

Д. Рахмани, старший преподаватель

В.П. Суворов, студент

П.П. Михайлов, студент

Московский технический университет связи и информатики
(Россия, г. Москва)

DOI:10.24412/2500-1000-2025-3-1-253-263

Аннотация. В статье рассматривается задача поиска кратчайших маршрутов в динамично изменяемых графах с переменной топологией. Актуальность исследования обусловлена необходимостью разработки алгоритмов, способных работать в условиях изменяемой среды, таких как склады с подвижными стеллажами или лабиринты с перемещающимися стенами. Научная новизна заключается в предложении жадного алгоритма, который на каждом шаге выбирает оптимальное преобразование графа для приближения к целевой вершине. Алгоритм реализован на языке C# и протестирован на случайно генерируемых графах. Результаты экспериментов показали, что среднее количество преобразований зависит от конфигурации графа. Алгоритм продемонстрировал высокую эффективность в решении задач оптимизации логистики, робототехники и моделирования сложных систем. В дальнейшем предполагается улучшение алгоритма для работы с более сложными графами и топологиями.

Ключевые слова: динамические графы, поиск кратчайших маршрутов, жадные алгоритмы, оптимизация, C#, моделирование.

Современные задачи, связанные с оптимизацией маршрутов в динамически изменяемых системах, требуют разработки новых подходов к анализу графовых структур. Традиционные алгоритмы поиска кратчайших путей, такие как алгоритм Дейкстры или поиск в ширину, эффективны только в статических графах, где структура остается неизменной. Однако в реальных условиях, таких как крупные складские помещения или пространства, заполненные множеством перемещаемых объектов, графы, отображающие возможные пути, могут изменяться в процессе за счёт обстоятельств внешней силы или за счёт самого агента.

Актуальность данной работы обусловлена необходимостью создания алгоритмов, способных учитывать динамические изменения в графах и находить оптимальные пути в таких условиях, а также необходимостью оценки проходимости таких графов в целом. Научная новизна заключается в предложении реализации жадного алгоритма поиска пути и дальнейшей оценки графа на основании собранной статистики.

Целью исследования является разработка и анализ алгоритма для поиска кратчайших маршрутов в динамически изменяемых графах. Задачи работы включают: разработку алгоритма, учитывающего изменения в структуре графа, реализацию алгоритма на языке C#, проведение экспериментов для оценки эффективности алгоритма, анализ результатов и формулирование выводов.

Дан граф $G = (V, E)$, где V – вершины (тайлы), а E – рёбра (проходы) лабиринта. Рассматривается модель игры *Labyrinth* (1986), в которой игровое поле состоит из случайно расположенных тайлов с коридорами. В отличие от классических задач поиска пути (Дейкстра, BFS, A*), граф может быть несвязным и изменяемым, что важно для моделирования, например, складов с подвижными элементами. Граф G представлен как последовательность состояний G_1, G_2, \dots, G_k , где каждое G_i соответствует конфигурации лабиринта на i -м шаге. Изменения графа связаны с добавлением или удалением рёбер, вызванными перемещением строк/столбцов тайлов. Эти преобразования описываются опера-

торами $T_i: G_i \rightarrow G_{i+1}$, изменяющими структуру лабиринта.

Алгоритм минимизирует число преобразований T_1, T_2, \dots, T_n , необходимых для достижения вершины v_{end} из v_{start} :

$$\min_{T_1, T_2, \dots, T_n} n \quad (1)$$

На каждом шаге выбирается преобразование, приближающее агента к цели. Алгоритм оценивает все возможные шаги и выбирает

тот, который минимизирует эвристическую функцию $h(v, v_{end})$ – евклидово расстояние:

$$h(v, v_{end}) = \sqrt{(x_v - x_{end})^2 + (y_v - y_{end})^2} \quad (2)$$

где (x_v, y_v) и (x_{end}, y_{end}) – координаты вершин.

Преимуществами является простая реализация, эффективность приближения к цели, минимизация шагов, а недостатками – возможность застревания в локальных минимумах, решаемая повторными запусками разных конфигураций.

Алгоритм выполняет поиск оптимального шага на каждом этапе, что даёт сложность $O(k \cdot (n + m))$, где: n – количество вершин, m

– количество рёбер, k – количество преобразований графа. Экспериментально подтверждено, что в среднем $k \sim \log(n)$, что даёт сложность $O(n \cdot \log(n))$.

В худшем случае, если $k \sim n$, сложность возрастает до $O(n^2)$, но на практике k обычно меньше. График 1 показывает линейно-логарифмический рост времени работы, что соответствует $O(n \cdot \log(n))$.

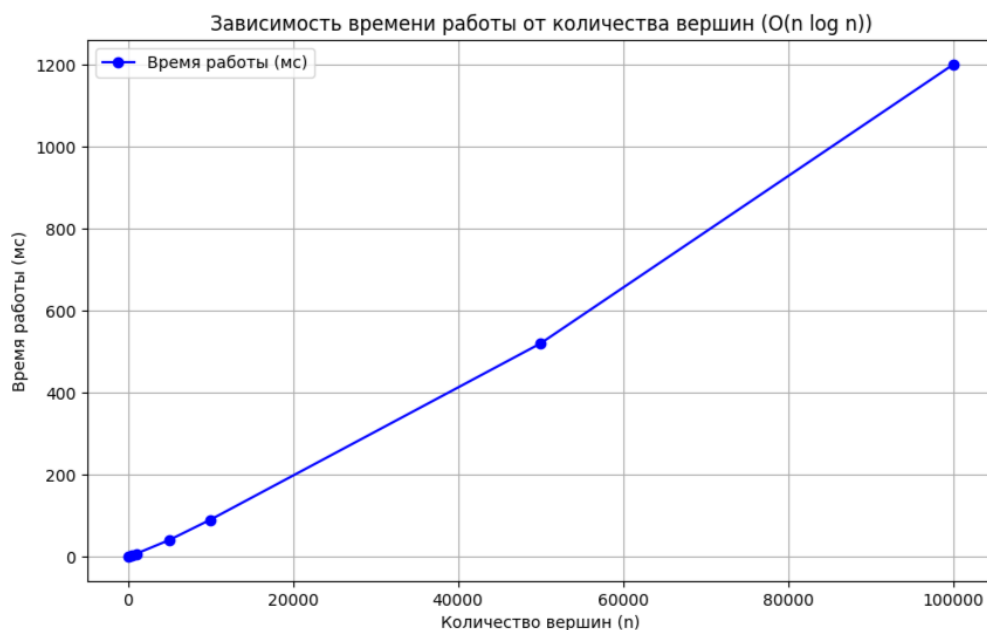


Рис. 1. График зависимости времени от количества вершин

Евклидово расстояние используется из-за его точности в непрерывных пространствах и вычислительной эффективности. Оно естественно отражает "прямое" расстояние между точками, что ускоряет поиск оптимального пути.

В открытых пространствах без препятствий евклидово расстояние даёт наиболее корректную оценку расстояния до цели. Оно также вычисляется быстрее, чем сложные эвристики, что делает его предпочтительным для задач с большим числом вершин.

В отличие от манхэттенского расстояния, подходящего для решётчатых структур, евклидово расстояние обеспечивает более точные оценки в задачах, где движение возможно в любом направлении. В динамически изменяемых графах оно лучше адаптируется к изменениям, поскольку отражает реальное расстояние между точками.

Таким образом, евклидовая эвристика является оптимальным выбором благодаря своей точности, эффективности и универсальности.

Для лучшего понимания преимуществ и недостатков жадной стратегии, приведем сравнение с другими популярными алгоритмами поиска пути в таблице 1.

Таблица 1. Сравнение алгоритмов

Алгоритм	Сложность	Подходит для динамических графов?	Гарантия оптимального решения
Дейкстра	$O(V^2)$ или $O(E+V \log V)$	Нет	Да
A*	$O(E)$ (зависит от эвристики)	Частично	Да
BFS	$O(V+E)$	Нет	Нет
Жадный метод (из статьи)	$O(n \log n)$	Да	Частично

BFS гарантирует нахождение кратчайшего пути в невзвешенных графах с сложностью $O(V+E)$. Он прост в реализации, но неэффективен для больших графов и неприменим к взвешенным [1].

Дейкстра находит кратчайший путь в графах с неотрицательными весами, работает за $O(V^2)$ или $O(E+V \log V)$ с приоритетной очередью. Гарантирует оптимальность, но не поддерживает отрицательные веса и плохо адаптируется к динамическим изменениям.

A* сочетает Дейкстру и жадный поиск, оценивая стоимость пути с помощью эвристики. Его сложность $O(E)$, эффективность зависит от выбора эвристики. Гарантирует оптимальность при допустимой эвристике, но требует её тщательного подбора.

Жадный алгоритм быстро адаптируется к динамическим графам, принимает решения локально, минимизируя шаги. Прост в реализации и эффективен для задач реального времени, таких как маршрутизация. Однако не всегда находит глобально оптимальное решение. Жадный алгоритм подходит для адаптивных задач с ограниченным временем, а BFS, Дейкстра и A* предпочтительны, если требуется гарантированная оптимальность.

Адаптивные алгоритмы широко применяются в логистике, транспорте и робототехнике. Amazon Robotics использует их для управления 100 000+ роботов, что ускоряет обработку заказов на 25% и увеличивает пропускную способность на 40%. Uber и Яндекс.Такси снижают время поездки на 15-20% за счёт динамической маршрутизации. Tesla внедрила адаптивные конвейеры, повысив производительность на 20% и снизив простои на 30%.

По данным McKinsey, такие алгоритмы уменьшают операционные затраты на 10-15%, а International Federation of Robotics отмечает рост эффективности складской робототехники на 25-30%. Рынок адаптивных алгоритмов увеличивается на 20-25% ежегодно. За 5 лет их применение выросло на 40% в логистике, 35% в производстве и 50% в транспорте, подтверждая их значимость для повышения эффективности [2]. На рисунке 2 изображён график мировых поставок промышленных роботов, дающий представление о косвенном использовании адаптивных маршрутизационных алгоритмов в индустрии.



Рис. 2. График мировых поставок промышленных роботов

Алгоритм начинается с задания начальной вершины v_{start} и целевой вершины v_{end} , а также инициализации текущего состояния графа G_1 .

На каждом шаге он перебирает возможные преобразования T_i для графа G_i , вычисляет новое состояние G_{i+1} после их применения, оценивает расстояние до цели с помощью эвристической функции $h(v, v_{end})$ для достижимых вершин и выбирает преобразование, минимизирующее эту функцию [3].

Алгоритм завершается, когда достигает вершины v_{end} , либо сообщает о невозможности найти путь.

Реализация выполнена на C# с использованием Windows Forms для визуализации лабиринта. Граф представлен списком вершин, соответствующих тайлам, и рёбер, обозначающих проходы между ними. Вершины содержат информацию о типе и повороте тайла, а рёбра создаются между соседними тайлами,

если между ними есть проход, что проверяется функцией HasPassage. Граф является неориентированным, а веса рёбер равны 1.

Генерация графа начинается с создания вершин для каждого тайла, после чего проверяются его соседи справа и снизу. Если между тайлом и его соседом есть проход, создаётся двустороннее ребро. На рисунке 3 представлена реализация генерации графа на C#.

Функция HasPassage проверяет, есть ли проход между двумя тайлами, учитывая их тип и поворот. Она принимает на вход тип и поворот первого и второго тайлов, а также их индексы в лабиринте. Логика проверки зависит от типа тайла. Например, для StraightCorridor проход возможен только если оба тайла повернуты определённым образом и находятся рядом. Функция возвращает true, если проход есть, и false, если прохода нет. Реализацию функции можно увидеть на рисунке 4.

```

1 private (List<GraphNode> graphNodes, List<GraphEdge> graphEdges) GenerateGraph(List<Tile> tiles, int columns, int rows)
2 {
3     List<GraphNode> graphNodes = new List<GraphNode>();
4
5     // Создаем вершины графа
6     foreach (Tile tile in tiles)
7     {
8         GraphNode node = new GraphNode(tile);
9         graphNodes.Add(node);
10    }
11
12    // Создаем рёбра графа
13    List<GraphEdge> graphEdges = new List<GraphEdge>();
14
15    for (int row = 0; row < rows; row++)
16    {
17        for (int col = 0; col < columns; col++)
18        {
19            int currentIndex = row * columns + col;
20            Tile currentTile = tiles[currentIndex];
21
22            // Проверяем соседей справа
23            if (col < columns - 1)
24            {
25                int neighborIndex = currentIndex + 1;
26                Tile neighborTile = tiles[neighborIndex];
27
28                if (HasPassage(currentTile.Type, currentTile.Rotation, currentIndex, neighborTile.Type, neighborTile.Rotation, neighborIndex))
29                {
30                    int weight = 1; // Вес ребра
31                    graphEdges.Add(new GraphEdge(graphNodes[currentIndex], graphNodes[neighborIndex], weight));
32                    graphEdges.Add(new GraphEdge(graphNodes[neighborIndex], graphNodes[currentIndex], weight)); // Обратное ребро
33                }
34            }
35
36            // Проверяем соседей снизу
37            if (row < rows - 1)
38            {
39                int neighborIndex = currentIndex + columns;
40                Tile neighborTile = tiles[neighborIndex];
41
42                if (HasPassage(currentTile.Type, currentTile.Rotation, currentIndex, neighborTile.Type, neighborTile.Rotation, neighborIndex))
43                {
44                    int weight = 1; // Вес ребра
45                    graphEdges.Add(new GraphEdge(graphNodes[currentIndex], graphNodes[neighborIndex], weight));
46                    graphEdges.Add(new GraphEdge(graphNodes[neighborIndex], graphNodes[currentIndex], weight)); // Обратное ребро
47                }
48            }
49        }
50    }
51
52    return (graphNodes, graphEdges);
53 }

```

Рис. 3. Листинг функции генерации графа

```

1 private bool HasPassage(TileType tileType1, int rotation1, int index1, TileType tileType2, int rotation2, int index2)
2 {
3     // Проверка прохода для StraightCorridor
4     if (tileType1 == TileType.StraightCorridor)
5     {
6         if (tileType2 == TileType.StraightCorridor)
7         {
8             if (Math.Abs(index1 - index2) == 1 && (rotation1 == 180 || rotation1 == 0) && (rotation2 == 180 || rotation2 == 0))
9             {
10                return true;
11            }
12            // Другие условия...
13        }
14        // Другие проверки...
15    }
16    // Аналогичные проверки для других типов тайлов...
17    return false;
18 }

```

Рис. 4. Листинг функции HasPassage

Алгоритм Дейкстры используется для поиска кратчайшего пути в графе от начальной вершины до всех остальных. На этапе инициализации массив расстояний заполняется бесконечностью (`int.MaxValue`), кроме начальной вершины, где расстояние равно 0. Массив предыдущих вершин заполняется значением -1. Используется приоритетная очередь для выбора вершины с минимальным расстоянием. Для каждой вершины обновляются рас-

стояния до её соседей, если найден более короткий путь. После завершения работы алгоритма путь восстанавливается с помощью массива предыдущих вершин. Возвращаемое значение – список вершин, представляющих кратчайший путь от начальной до конечной вершины. Алгоритм используется для поиска кратчайшего пути в той части графа, которая является доступной для агента. Реализацию можно увидеть на рисунке 5.

```

1 public static List<int> Dijkstra(List<GraphNode> graphNodes, List<GraphEdge> graphEdges, int startIndex, int endIndex)
2 {
3     int numVertices = graphNodes.Count;
4     int[] distance = new int[numVertices];
5     int[] previous = new int[numVertices];
6
7     for (int i = 0; i < numVertices; i++)
8     {
9         distance[i] = int.MaxValue;
10        previous[i] = -1;
11    }
12    distance[startIndex] = 0;
13    PriorityQueue<int, int> queue = new PriorityQueue<int, int>();
14
15    for (int i = 0; i < numVertices; i++)
16    {
17        queue.Enqueue(i, distance[i]);
18    }
19
20    while (queue.Count > 0)
21    {
22        int u = queue.Dequeue();
23
24        foreach (var edge in graphEdges.Where(e => graphNodes.IndexOf(e.StartNode) == u))
25        {
26            int v = graphNodes.IndexOf(edge.EndNode);
27            int alt = distance[u] + edge.Weight;
28
29            if (alt < distance[v])
30            {
31                distance[v] = alt;
32                previous[v] = u;
33                queue.Enqueue(v, alt);
34            }
35        }
36    }
37    return ReconstructPath(previous, endIndex);
38 }

```

Рис. 5. Листинг функции модифицированного алгоритма Дейкстры

Функция `findoptimalmove` реализует жадную стратегию для поиска оптимального преобразования графа, чтобы приблизиться к целевой вершине. Она перебирает все строки и столбцы лабиринта. Для каждого возможного сдвига строки или столбца выполняется сдвиг, вычисляются все достижимые вершины из текущей позиции, и выбирается вершина, ближайшая к цели, с помощью функции

`ChooseNearestRoom`. Выбирается преобразование, которое максимально приближает агента к цели, и оно применяется. Логика выбора преобразования заключается в том, что для каждого сдвига вычисляется новое состояние графа, и выбирается сдвиг, который минимизирует расстояние до цели. Реализацию можно увидеть на рисунке 6.

```

1 void findoptimalmove(int currentRoomIndex, int endIndex, List<int>? moveblerows = null, List<int>? moveblecolumns = null)
2 {
3     int rows = int.Parse(textBox1.Text);
4     int columns = int.Parse(textBox2.Text);
5     int tempcurrentroomindex = currentRoomIndex;
6     int nearestRoomIndex = 0;
7
8     List<int> AllCalculatedVertices = new List<int> { };
9     List<string> AllCalculatedVerticesSetups = new List<string> { };
10
11     // Перебор всех строк
12     int movingrowindex = 1;
13     while (movingrowindex <= rows)
14     {
15         if (moveblerows == null || moveblerows.Contains(movingrowindex))
16         {
17             movingRowRight(movingrowindex, false);
18             List<int> ConnectedVertices = GetConnectedVertices(tempgraphnodesNEW, tempgraphedgesNEW, tempcurrentroomindex);
19             int nextRoomIndex = ChooseNearestRoom(endIndex, ConnectedVertices);
20             AllCalculatedVertices.Add(nextRoomIndex);
21             AllCalculatedVerticesSetups.Add(Convert.ToString(movingrowindex) + 'r');
22
23             ConnectedVertices.Clear();
24
25             movingRowLeft(movingrowindex, false);
26             ConnectedVertices = GetConnectedVertices(tempgraphnodesNEW, tempgraphedgesNEW, tempcurrentroomindex);
27             nextRoomIndex = ChooseNearestRoom(endIndex, ConnectedVertices);
28             AllCalculatedVertices.Add(nextRoomIndex);
29             AllCalculatedVerticesSetups.Add(Convert.ToString(movingrowindex) + 'l');
30         }
31         movingrowindex++;
32     }
33     int movingcolumnindex = 1;
34     while (movingcolumnindex <= columns)
35     {
36         if (moveblecolumns == null || moveblecolumns.Contains(movingcolumnindex))
37         {
38             movingColumnUp(movingcolumnindex, false);
39             List<int> ConnectedVertices = GetConnectedVertices(tempgraphnodesNEW, tempgraphedgesNEW, tempcurrentroomindex);
40             int nextRoomIndex = ChooseNearestRoom(endIndex, ConnectedVertices);
41             AllCalculatedVertices.Add(nextRoomIndex);
42             AllCalculatedVerticesSetups.Add(Convert.ToString(movingcolumnindex) + 'u');
43
44             ConnectedVertices.Clear();
45
46             movingColumnDown(movingcolumnindex, false);
47             ConnectedVertices = GetConnectedVertices(tempgraphnodesNEW, tempgraphedgesNEW, tempcurrentroomindex);
48             nextRoomIndex = ChooseNearestRoom(endIndex, ConnectedVertices);
49             AllCalculatedVertices.Add(nextRoomIndex);
50             AllCalculatedVerticesSetups.Add(Convert.ToString(movingcolumnindex) + 'd');
51         }
52         movingcolumnindex++;
53     }
54
55     // Выбор оптимального преобразования
56     nearestRoomIndex = ChooseNearestRoom(endIndex, AllCalculatedVertices);
57     int IndexOfSetup = AllCalculatedVertices.IndexOf(nearestRoomIndex);
58     string CurrentSetup = AllCalculatedVerticesSetups[IndexOfSetup];
59     switch (CurrentSetup[1])
60     {
61     case 'r':
62         movingRowRight(Convert.ToInt32(CurrentSetup[0])-'0');
63         break;
64     case 'l':
65         movingRowLeft(Convert.ToInt32(CurrentSetup[0])-'0');
66         break;
67     case 'u':
68         movingColumnUp(Convert.ToInt32(CurrentSetup[0])-'0');
69         break;
70     case 'd':
71         movingColumnDown(Convert.ToInt32(CurrentSetup[0])-'0');
72         break;
73     };
74 }

```

Рис. 6. Листинг функции Findoptimalmove

Функция `ChooseNearestRoom` выбирает вершину, ближайшую к целевой, на основе евклидова расстояния. Для каждой вершины вычисляется её позиция в лабиринте (строка и столбец). Затем вычисляется евклидово расстояние между текущей вершиной и целевой. Евклидово расстояние между двумя точками

(x_1, y_1) и (x_2, y_2) вычисляется по формуле: $distance = (x_2 - x_1)^2 + (y_2 - y_1)^2$. Возвращается вершина с минимальным расстоянием. Реализацию можно увидеть на рисунке 7.

```

1  private int ChooseNearestRoom(int endIndex, List<int> ConnectedVertices)
2  {
3      int columns = int.Parse(textBox2.Text);
4      double minDistance = double.MaxValue;
5      int VerticeOfminDistance = -1;
6
7      foreach (int Vertice in ConnectedVertices)
8      {
9          int currentRow = (Vertice / columns) + 1;
10         int currentColumn = (Vertice % columns) + 1;
11         int endRow = (endIndex / columns) + 1;
12         int endColumn = (endIndex % columns) + 1;
13
14         double rowDifference = Math.Abs(endRow - currentRow);
15         double columnDifference = Math.Abs(endColumn - currentColumn);
16         double distance = Math.Sqrt(
17             Math.Pow(rowDifference, 2) + Math.Pow(columnDifference, 2)
18         );
19
20         if (distance < minDistance)
21         {
22             minDistance = distance;
23             VerticeOfminDistance = Vertice;
24         }
25     }
26
27     return VerticeOfminDistance;
28 }

```

Рис. 7. Листинг функции `ChooseNearestRoom`

Функция `GetConnectedVertices` возвращает все вершины, достижимые из текущей, с использованием алгоритма поиска в глубину (DFS). На этапе инициализации создаётся список для хранения достижимых вершин и множество для отслеживания посещённых

вершин. Рекурсивный обход графа начинается с текущей вершины, и для неё добавляются все её соседи, если они ещё не посещены. DFS позволяет обойти все вершины, достижимые из текущей, без необходимости обхода всего графа. Листинг можно увидеть на рисунке 8.

```

1 public List<int> GetConnectedVertices(List<GraphNode> graphNodes, List<GraphEdge> graphEdges, int targetIndex)
2 {
3     List<int> connectedVertices = new List<int>();
4     HashSet<int> visited = new HashSet<int>();
5
6     DFS(graphNodes, graphEdges, targetIndex, connectedVertices, visited);
7
8     return connectedVertices;
9 }
10
11 private void DFS(
12     List<GraphNode> graphNodes,
13     List<GraphEdge> graphEdges,
14     int currentIndex, List<int> connectedVertices,
15     HashSet<int> visited
16 )
17 {
18     visited.Add(currentIndex);
19     connectedVertices.Add(currentIndex);
20
21     foreach (var edge in graphEdges)
22     {
23         if (graphNodes.IndexOf(edge.StartNode) == currentIndex && !visited.Contains(graphNodes.IndexOf(edge.EndNode)))
24         {
25             DFS(graphNodes, graphEdges, graphNodes.IndexOf(edge.EndNode), connectedVertices, visited);
26         }
27         else if (graphNodes.IndexOf(edge.EndNode) == currentIndex && !visited.Contains(graphNodes.IndexOf(edge.StartNode)))
28         {
29             DFS(graphNodes, graphEdges, graphNodes.IndexOf(edge.StartNode), connectedVertices, visited);
30         }
31     }
32 }

```

Рис. 8. Листинг функции GetConnectedVertices

Для оценки сложности алгоритма и среднего числа изменений, необходимых для достижения цели, используется статистический подход. Метод основан на многократном запуске алгоритма и анализе собранных данных, что позволяет определить зависимость количества преобразований от параметров графа.

Пусть n – число преобразований, необходимых для достижения вершины B из A . Оно

зависит от структуры графа G (количества вершин $|V|$ и рёбер $|E|$) и стратегии поиска. Для оценки сложности алгоритма выполняется N запусков на случайных графах с фиксированными параметрами. В каждом запуске фиксируется число преобразований n_i , после чего формируется выборка $\{n_1, n_2, \dots, n_N\}$.

Среднее количество преобразований вычисляется как:

$$\bar{n} = \frac{1}{N} \sum_{i=1}^N n_i \quad (3)$$

Эта величина позволяет оценить ожидаемое число изменений графа [4]. Изменяя параметры графа (например, $|V|$ и $|E|$), можно построить зависимость $\bar{n}(|V|, |E|)$ и определить асимптотическую сложность алгоритма.

Линейный рост \bar{n} указывает на линейную сложность, квадратичный или экспоненциальный – на более сложные зависимости. Для оценки точности вычисляется доверительный интервал:

$$\bar{n} \pm z * \frac{\sigma}{\sqrt{N}} \quad (4)$$

где z – квантиль стандартного нормального распределения (например, 1.96 для 95% доверия), а σ – стандартное отклонение.

Предположим, что алгоритм запускается $N=1000$ раз на графах с $|V|=100$ вершинами и $|E|=300$ рёбрами. Для каждого запуска фиксируется количество преобразований n_i . После проведения экспериментов вычисляется:

$$15.3 \pm 1.96 * \frac{2.1}{\sqrt{1000}} \approx 15.3 \pm 0.13 \quad (5)$$

В статье представлен жадный алгоритм поиска кратчайших маршрутов в динамично изменяемых графах с переменной топологией. Алгоритм эффективно решает задачи, такие как поиск оптимальных путей в средах с подвижными элементами (например, склады или лабиринты с перемещающимися стенами), адаптируясь к изменениям структуры графа и выбирая локально оптимальные шаги. Эксперименты показали, что количество преобразований зависит от конфигурации графа и его сложности, подтверждая эффективность алгоритма для реальных задач, таких как оптимизация логистики и моделирование сложных систем.

Алгоритм решает задачу поиска кратчайших маршрутов в динамично изменяемых

- Среднее количество преобразований \bar{n} .
- Дисперсия σ^2 .
- Доверительный интервал для \bar{n} .

Если, например, $\bar{n} = 15.3$ с $\sigma = 2.1$, то с 95% уверенностью можно утверждать, что среднее количество преобразований лежит в интервале:

графах, подходит для задач оптимизации логистики, робототехники и моделирования. Среднее количество преобразований зависит от сложности и конфигурации графа, что подтверждает необходимость адаптивного подхода. Дальнейшие исследования могут включать улучшение алгоритма для более сложных графов и интеграцию дополнительных эвристик для повышения точности и скорости. Также возможно применение алгоритма в анализе транспортных сетей, управлении беспилотниками и проектировании гибких производственных систем. Алгоритм является эффективным инструментом для анализа маршрутов в изменяемых графах и может быть рекомендован для прикладных задач.

Библиографический список

1. Новиков Ф.А., Поздняков С.Н. Жадные алгоритмы // КИО. – 2005. – №2.
2. Севостьянов А.К., Витковская А.А. Промышленные роботы в машиностроении // Евразийский научный журнал. – 2019. – №7.
3. Гуляевский С.Е. Системы и подходы для обработки информации, представленной большими динамическими графами // Программные продукты и системы. – 2022. – №1.
4. Сперанский Д.В. Поиск оптимальных путей в нечетких графах // Автоматика на транспорте. – 2022. – №4.

**ANALYSIS ALGORITHM OF SHORTEST PATHS IN DYNAMICALLY RECONFIGURABLE
GRAPH STRUCTURES WITH VARIABLE TOPOLOGY: EVALUATION
OF THE AVERAGE NUMBER OF TRANSFORMATIONS TO REACH TARGET NODES**

J. Rahmani, *Senior Lecturer*

V.P. Suvorov, *Student*

P.P. Mikhailov, *Student*

**Moscow Technical University of Communications and Informatics
(Russia, Moscow)**

***Abstract.** This article addresses the problem of finding the shortest paths in dynamically changing graphs with variable topology. The relevance of the research is driven by the need to develop algorithms capable of operating in changing environments, such as warehouses with movable shelves or mazes with shifting walls. The scientific novelty lies in the proposal of a greedy algorithm that selects the optimal graph transformation at each step to approach the target vertex. The algorithm is implemented in C# and tested on randomly generated graphs. Experimental results show that the average number of transformations depends on the graph configuration. The algorithm demonstrated high efficiency in solving problems related to logistics optimization, robotics, and complex systems modeling. Future work will focus on improving the algorithm to handle more complex graphs and topologies.*

***Keywords:** dynamic graphs, shortest path search, greedy algorithms, optimization, C#, modeling.*