Санкт-Петербургский Политехнический Университет
Высшая школа прикладной математики и вычислительной физики, ФизМех
01.03.02 Прикладная математика и информатика

Лабораторная Работа №1
тема "Кодирование информации"
вариант "Алгоритм Фано"
дисциплина "Дискретная математика"

Выполнил студент гр. 5030102/20201 Буслама Анис

26 ноября 2024 г.

Содержание

1 Поставленная задача **3**
1.1 Постановка задачи: 3
1.2 Используемый язык программирования: 3
1.3 Ограничения 3

2 Алгоритм A* **3**
2.1 Псевдокод алгоритма A* для сетки с ограничениями 3

3 Пример работы алгоритма A* **6**
3.1 Ход работы алгоритма 6

4 Анализ сложности алгоритма A* **8**
4.1 Временная сложность 8
4.2 Пространственная сложность 8
4.3 Доказательство временной сложности 8
4.4 Заключение 9

5 Сравнение работы алгоритма A* на различных входных данных **10**
5.1 Графы, на которых алгоритм работает лучше всего 10
5.2 Графы, на которых алгоритм работает хуже всего 10
5.3 Графы, на которых алгоритм не работает 11
5.4 Выводы 11

6 Обоснование выбора способа представления графа **12**
6.1 Простота визуализации графа 12
6.2 Эффективность реализации 12
6.3 Универсальность представления 12
6.4 Обоснование выбора формата данных 12
6.5 Вывод 13

7 Вывод **13**

1 Поставленная задача

1.1 Постановка задачи:

На вход программе подаётся невзвешенный граф – представление координатной сетки с целыми координатами, начальная вершина и конечная. Рёбра в графе могут соединять только те вершины, которые соответствуют соседним ячейкам клетки. Переход от одной ячейки к другой можно осуществлять по четырём направлениям. Ребро между двумя вершинами в переданном в программу графе существует тогда и только тогда, когда разрешается совершать переход из одной ячейки в другую. Иными словами, должна быть возможность выставлять "стены" между соседними ячейками координатной сетки.

1.2 Используемый язык программирования:

C++ 20 вместе со стандартными библиотеками: STD, STL (Vector, Queue)

1.3 Ограничения

Граф должен быть связанным

2 Алгоритм A*

2.1 Псевдокод алгоритма A* для сетки с ограничениями

```
// Вход: grid - двумерный массив доступных клеток (true/false),
//       start - начальная клетка (x, y),
//       goal - конечная клетка (x, y),
//       directions - массив допустимых направлений движения,
//       walls - множество пар заблокированных переходов.
// Выход: путь от start до goal, либо пустой массив, если путь невозможен.
Function a_star(grid, start, goal, directions, walls)
    open_set := PriorityQueue()  // очередь с приоритетом (по f = g + h)
    g_score := Map()             // стоимость пути от start до текущей клетки
    f_score := Map()             // оценка стоимости пути через текущую клетку
    came_from := Map()           // хранит путь
    closed_set := Set()          // обработанные клетки

    open_set.push((start, 0.0, heuristic(start, goal))) // начальная клетка
    g_score[start] := 0.0
    f_score[start] := heuristic(start, goal)

    While not open_set.empty() do
        current := open_set.pop() // клетка с наименьшим f
        current_pos := current.position

        If current_pos = goal then
            return reconstruct_path(came_from, goal)
        End if

        closed_set.insert(current_pos)
```

```
        For each dir in directions do
            neighbor := current_pos + dir
            If not is_valid(neighbor, grid) or
               neighbor in closed_set or
               not can_move(current_pos, neighbor, walls) then
                continue
            End if

            tentative_g := g_score[current_pos] + 1.0 // стоимость соседней клетки

            If neighbor not in g_score or tentative_g < g_score[neighbor] then
                came_from[neighbor] := current_pos
                g_score[neighbor] := tentative_g
                f_score[neighbor] := tentative_g + heuristic(neighbor, goal)
                open_set.push((neighbor, g_score[neighbor], heuristic(neighbor, goal)))
            End if
        End for
    End while

    return [] // путь не найден
End function
```

Фун. 1. Реализация алгоритма A* для поиска пути в сетке

```
// Вход: from - текущая клетка (x, y),
//       to - соседняя клетка (x, y),
//       walls - множество запрещенных переходов (пар клеток).
// Выход: true, если переход возможен; false, если заблокирован.
Function can_move(from, to, walls)
    Return (from, to) not in walls and (to, from) not in walls
End function
```

Фун. 2. Проверка возможности перехода между клетками

```
// Вход: pos - клетка (x, y), grid - двумерный массив доступных клеток.
// Выход: true, если клетка существует и доступна; иначе false.
Function is_valid(pos, grid)
    (x, y) := pos
    Return x >= 0 and y >= 0 and x < size(grid) and y < size(grid[0]) and grid[x][y]
End function
```

Фун. 3. Проверка валидности клетки

```
// Вход: came_from - карта путей (Map), current - конечная клетка (x, y).
// Выход: массив клеток, представляющий найденный путь.
Function reconstruct_path(came_from, current)
    path := []
    While current in came_from do
        path.insert(0, current)
        current := came_from[current]
```

```
    End while
    Return path
End function
```

Фун. 4. Восстановление пути из карты путей

```
// Вход: current - клетка (x1, y1), goal - клетка (x2, y2).
// Выход: эвристическое расстояние между клетками.
Function heuristic(current, goal)
    (x1, y1) := current
    (x2, y2) := goal
    Return sqrt((x2 - x1)^2 + (y2 - y1)^2) // Евклидово расстояние
End function
```

Фун. 5. Вычисление эвристического расстояния

```
// Вход: файл input.txt с параметрами: размеры сетки, начальная и конечная клетки,
//       описанием стен (запрещенных переходов).
// Выход: двумерный массив grid, начальная и конечная клетка, список направлений,
//        множество стен.
Function parse_input(filename)
    file := open(filename, "r")
    rows, cols := read_ints(file.readline())
    start := read_coords(file.readline())
    goal := read_coords(file.readline())

    grid := [[true for _ in range(cols)] for _ in range(rows)]
    walls := Set()
    directions := [(0, 1), (1, 0), (0, -1), (-1, 0), // четыре направления
                   (1, 1), (1, -1), (-1, 1), (-1, -1)] // диагонали

    For each line in file.readlines() do
        (x1, y1, x2, y2) := read_ints(line)
        walls.add(((x1, y1), (x2, y2)))
    End for

    return grid, start, goal, directions, walls
End function
```

Фун. 6. Разбор входных данных из файла

3 Пример работы алгоритма A*

Для демонстрации работы алгоритма A* возьмём следующую задачу. Имеется сетка 5×5, где некоторые клетки заблокированы. Начальная клетка находится в точке $(0, 0)$, конечная клетка — $(4, 4)$.

Допустим, заблокированные клетки (стены) определены следующими парами координат:

- $(1, 0) \leftrightarrow (1, 1)$
- $(2, 1) \leftrightarrow (3, 1)$
- $(3, 3) \leftrightarrow (4, 3)$

Сетка будет выглядеть так (где # — заблокированные клетки):

S				
#	#			
	#			
	#		#	
			#	G

Здесь S обозначает начальную клетку, а G — конечную.

3.1 Ход работы алгоритма

1. **Инициализация**: Алгоритм начинает с клетки $(0, 0)$. Вычисляется эвристическое расстояние h (евклидово расстояние) до конечной клетки $(4, 4)$:

$$h(0, 0) = \sqrt{(4 - 0)^2 + (4 - 0)^2} = \sqrt{32} \approx 5.66$$

   Стоимость пути $g(0, 0)$ равна 0, а $f(0, 0) = g + h = 5.66$. Клетка $(0, 0)$ добавляется в очередь.

2. **Выбор клетки**: Из очереди выбирается клетка с минимальным значением f. Текущая клетка — $(0, 0)$. Алгоритм проверяет соседей: $(0, 1)$ и $(1, 0)$ (диагональные движения в данном примере не используются).

3. **Обновление соседей**:

   - Для клетки $(0, 1)$:

$$g(0, 1) = g(0, 0) + 1 = 1, \quad h(0, 1) = \sqrt{(4 - 0)^2 + (4 - 1)^2} \approx 5.39, \quad f(0, 1) = 6.39$$

   - Для клетки $(1, 0)$ (блокирована): пропускается.

   Клетка $(0, 1)$ добавляется в очередь.

4. **Движение к следующей клетке**: Алгоритм переходит в клетку $(0, 1)$ и повторяет процесс. Обрабатываются соседи $(0, 2)$ и $(1, 1)$ (но $(1, 1)$ заблокирована).

5. Алгоритм продолжает строить путь, пока не дойдёт до клетки $(4, 4)$:

$$(0, 0) \to (0, 1) \to (0, 2) \to (1, 2) \to (2, 2) \to (3, 2) \to (4, 2) \to (4, 3) \to (4, 4)$$

6. **Восстановление пути**: Алгоритм строит путь, используя карту `came_from`. Результат:

$$\text{Путь: } (0,0) \to (0,1) \to (0,2) \to (1,2) \to (2,2) \to (3,2) \to (4,2) \to (4,3) \to (4,4)$$

Визуализация пути в сетке:

S	$*$	$*$		
#	#	$*$		
	#	$*$		
	#	$*$	#	
		$*$	$*$	G

Общий путь содержит 9 шагов, включая начальную и конечную клетку.

4 Анализ сложности алгоритма A*

Алгоритм A* представляет собой разновидность поиска по графу, сложность которого зависит от количества узлов, их связей и используемой эвристики. Разберём временную и пространственную сложность алгоритма.

4.1 Временная сложность

Пусть b — фактор ветвления графа (среднее количество соседей для каждого узла), а d — глубина пути от начальной точки до конечной. В худшем случае A* исследует почти все возможные узлы, что приводит к следующей оценке временной сложности:

$$O(b^d),$$

где:

- b^d соответствует числу узлов, которые могут быть рассмотрены до нахождения решения;
- использование эвристики помогает сузить область поиска, но в худшем случае алгоритм всё равно может исследовать почти все узлы.

При оптимальной эвристике (адекватно приближающей расстояние до цели) сложность может быть уменьшена до:

$$O(|E| + |V| \log |V|),$$

где $|E|$ — количество рёбер в графе, а $|V|$ — количество узлов. Это обусловлено использованием очереди с приоритетом (например, кучи).

4.2 Пространственная сложность

Пространственная сложность алгоритма A* определяется количеством узлов, которые хранятся в памяти. Алгоритм сохраняет:

- все узлы, которые находятся в очереди с приоритетом (Open List);
- все узлы, которые уже были посещены (Closed List).

Следовательно, пространственная сложность алгоритма также равна:

$$O(b^d),$$

поскольку в памяти может храниться до b^d узлов одновременно.

4.3 Доказательство временной сложности

Для доказательства временной сложности $O(b^d)$ рассмотрим процесс работы алгоритма:

1. Алгоритм начинает с начальной вершины и расширяет все её соседние узлы (всего b узлов).
2. На следующем уровне рассматриваются все узлы, доступные из уже посещённых. Их количество — b^2.
3. На k-м уровне число узлов, которые могут быть рассмотрены, составляет b^k.

4. На глубине d суммарное количество исследованных узлов оценивается как:

$$\sum_{k=0}^{d} b^k = \frac{b^{d+1} - 1}{b - 1},$$

   что в асимптотическом анализе упрощается до $O(b^d)$.

Если эвристика идеальна (точно соответствует реальной стоимости пути до цели), A* рассматривает только узлы на оптимальном пути, и сложность уменьшается до $O(d)$, но это идеализированный случай, который редко встречается на практике.

4.4 Заключение

Таким образом, временная и пространственная сложность алгоритма A* в общем случае равна $O(b^d)$. Однако, с использованием эффективной эвристики, сложность может быть уменьшена до $O(|E| + |V| \log |V|)$, что делает A* практичным для широкого класса задач.

5 Сравнение работы алгоритма A* на различных входных данных

Алгоритм A* демонстрирует разную эффективность в зависимости от структуры входного графа, используемой эвристики и особенностей задачи. Рассмотрим несколько типов графов и поведение алгоритма в каждом из случаев.

5.1 Графы, на которых алгоритм работает лучше всего

Алгоритм A* показывает наилучшую производительность на графах, где:

- **Эвристика точна**: эвристическая функция $h(n)$ точно оценивает минимальную стоимость пути от узла n до цели. Например, на двумерной сетке с евклидовой или манхэттенской метрикой (в зависимости от разрешённых направлений движения).
- **Граф имеет мало рёбер**: если фактор ветвления графа (b) низкий, алгоритм быстро находит путь, обходя минимальное количество узлов.
- **Структура графа проста**: если в графе нет циклов и минимальное расстояние до цели совпадает с числом переходов, A* находит путь с минимальным количеством операций.

Пример. Для двумерной сетки 5×5 без стен:

S				G

Алгоритм A* с манхэттенской эвристикой проходит оптимальный путь за $5 + 5 = 10$ шагов, практически не отклоняясь от целевого маршрута.

5.2 Графы, на которых алгоритм работает хуже всего

Наибольшие сложности возникают в следующих случаях:

- **Эвристика недооценивает путь**: если $h(n)$ сильно меньше реальной стоимости пути, A* сводится к обычному алгоритму Дейкстры. Это увеличивает число узлов, которые необходимо обработать.
- **Граф имеет высокую степень ветвления (b)**: большое количество соседних узлов для каждого узла увеличивает размер очереди с приоритетом и время обработки.
- **Плотные препятствия**: если граф содержит множество блокированных клеток, A* вынужден обходить их, часто исследуя лишние пути.

Пример. Для сетки 5×5 с множеством препятствий:

S	#	#	#	#
	#	#	#	
		#	#	
			#	G

A* придётся исследовать множество обходных путей, прежде чем найти путь к цели. Эвристика в таких случаях теряет эффективность.

5.3 Графы, на которых алгоритм не работает

Алгоритм A* может не найти путь, если:

- **Граф несвязен**: если начальный и конечный узлы находятся в разных компонентах графа, алгоритм завершится без нахождения пути.
- **Эвристика неадекватна**: если $h(n)$ переоценивает путь ($h(n) >$ реальный путь), алгоритм может пропустить оптимальное решение.

Пример. В несвязной сетке:

S	#	#	#	#
#	#	#	#	#
#	#	#	#	G

Алгоритм не сможет найти путь, так как целевая клетка G полностью отделена от начальной клетки S.

5.4 Выводы

Алгоритм A* работает эффективно на графах с:

- правильной структурой и допустимой эвристикой;
- небольшой степенью ветвления и низкой плотностью препятствий.

Алгоритм менее эффективен на графах с высокой степенью ветвления, плотными препятствиями или при использовании эвристики, которая недооценивает или переоценивает стоимость пути. На несвязных графах алгоритм не работает вовсе.

6 Обоснование выбора способа представления графа

Для реализации алгоритма A* в программе граф был представлен в виде двумерной координатной сетки. Такой способ представления был выбран по следующим причинам:

6.1 Простота визуализации графа

Граф, представленный в виде координатной сетки, интуитивно понятен:

- Узлы соответствуют клеткам сетки с целыми координатами (x, y).
- Рёбра графа определяются соседними клетками, что позволяет легко представить граф в двумерной плоскости.
- Структура такого графа хорошо соответствует практическим задачам, таким как навигация в сетке (например, карты, робототехника).

6.2 Эффективность реализации

Координатная сетка позволяет эффективно управлять данными:

- Для хранения рёбер используется информация о соседях каждой клетки (например, вверх, вниз, влево, вправо или по диагоналям). Это уменьшает количество лишних связей.
- Стены между узлами представлены булевыми значениями или списками запрещённых переходов. Это упрощает проверку доступности соседних клеток.
- Поиск соседей узла и проверка их доступности выполняются за $O(1)$ для каждой клетки, так как достаточно проверить фиксированное количество направлений.

6.3 Универсальность представления

Представление графа в виде сетки является универсальным и подходит для большинства задач, связанных с A*:

- Оно легко масштабируется на более сложные задачи: можно увеличить размер сетки или изменить правила переходов.
- Разрешение движения по 4 или 8 направлениям (на выбор) можно легко реализовать за счёт изменения набора допустимых соседей.
- Это представление подходит как для задач на плоскости, так и для обобщений на трёхмерные пространства.

6.4 Обоснование выбора формата данных

Для хранения графа использовались:

- Двумерный массив для представления клеток сетки. Каждый элемент массива содержит информацию о доступности данной клетки.
- Списки для хранения координат начального и конечного узлов.
- Структуры данных, такие как очереди с приоритетом, для реализации механизма поиска.

Такой подход позволил достичь оптимального баланса между простотой реализации и эффективностью выполнения алгоритма.

6.5 Вывод

Выбор представления графа в виде двумерной сетки обусловлен следующими факторами:

- Простота и наглядность;
- Эффективность работы с соседними узлами;
- Лёгкость в реализации различных правил переходов;
- Универсальность и масштабируемость.

Это делает такое представление наиболее подходящим для задач, решаемых алгоритмом A*.

7 Вывод

Алгоритм A* является мощным методом поиска кратчайшего пути в графе, который комбинирует преимущества жадного подхода и алгоритма Дейкстры. Он эффективно использует эвристическую функцию для направления поиска, что позволяет значительно сократить количество обрабатываемых узлов. Однако производительность алгоритма зависит от качества эвристики: при недооценке он становится медленнее, а при переоценке может не найти оптимальный путь. Сложность алгоритма составляет $O(b^d)$ в худшем случае, где b — фактор ветвления, а d — глубина решения, что делает его менее подходящим для графов с высокой ветвистостью или некачественными эвристиками.