Оглавление

1. Граф

Для хранения карты, т.е. совокупности точек и связей между ними, необходимо использовать графы. Граф - абстрактный математический объект, представляющий собой множество вершин графа и набор рёбер, то есть соединений между парами вершин.

В случае поиска маршрута, в замкнутом пространстве, точки на карте или лестницы будут множеством вершин графа, а связи между ними - ребра графа.

Графы разделяются на:

- Ориентированные и неориентированные
- Взвешенные и невзвешенные

Ориентированный граф — граф, рёбрам которого присвоено направление. Направленные рёбра именуются также дугами, а в некоторых источниках и просто рёбрами. Граф, ни одному ребру которого не присвоено направление, называется неориентированным графом.

Взвешенный граф — граф, каждому ребру которого поставлено в соответствие некое значение (вес ребра).

Вес ребра — значение, поставленное в соответствие данному ребру взвешенного графа. Обычно вес — вещественное число, в таком случае его можно интерпретировать как «длину» ребра.

Карта не идентифицируется как игровая, поэтому не нужны «одноразовые пути», следовательно, граф будет неориентированным. Для оптимального поиска пути, учитывающего различные факторы перемещения, например, приоритет лифта или лестницы, необходимо использовать веса. Итого: используемый граф будет взвешенным неориентированным.

1.1. Реализации хранения

Программно хранение графа может быть реализовано несколькими способами:

- в виде массива списков смежности («CListGraph»)
- в виде матрицы смежности («CMatrixGraph»)
- в виде массива хэш-таблиц («CSetGraph»)
- в виде одного массива пар («CArcGraph»)

Рассмотрим результаты тестирования реализаций. Граф генерируется программно по количеству вершин и плотности. Проверяются следующие методы интерфейса реализаций графа:

- занесение вершин и связей между ними
- получение вершин, доступных из текущей

Получение тестируется от 3 вершин: начальной, в середине и конечной, каждый тест проходит 1000 раз. По результатам высчитывается время прохождения теста, и средний «RPS» (requests per second).

2. Результаты для разреженного графа размером 1000 и количеством вершин 100000:

```
graph_size = 1000
Generation
        time: 2020 ms, (2020621 µs)
edges_count = 100000
```

(рис.1) *входные данные теста №1*

```
CListGraph
    CListGraph adding edges:
        time: 66 ms, (66837 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 0 ms, (2 µs)
        RPS = 500000.000
    get_next_vertices from middle:
        time: 0 ms, (9 µs)
        RPS = 500000.000
    get_next_vertices from back:
        time: 0 ms, (4 µs)
        RPS = 333333.333
```

```
CMatrixGraph
    CMatrixGraph adding edges:
        time: 5 ms, (5536 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 0 ms, (30 µs)
        RPS = 26315.789
    get_next_vertices from middle:
        time: 0 ms, (34 µs)
        RPS = 10638.298
    get_next_vertices from back:
        time: 0 ms, (276 µs)
        RPS = 15151.515
```

(рис.2) *результаты теста №1 для реализаций «CListGraph» и «CMatrixGraph»*

```
CSetGraph
    CSetGraph adding edges:
        time: 81 ms, (81385 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 0 ms, (15 µs)
        RPS = 250000.000
    get_next_vertices from middle:
        time: 0 ms, (7 µs)
        RPS = 125000.000
    get_next_vertices from back:
        time: 0 ms, (9 µs)
        RPS = 142857.143
```

```
CArcGraph
    CArcGraph adding edges:
        time: 15 ms, (15051 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 4 ms, (4348 µs)
        RPS = 651.890
    get_next_vertices from middle:
        time: 1 ms, (1184 µs)
        RPS = 896.861
    get_next_vertices from back:
        time: 1 ms, (1143 µs)
        RPS = 943.396
```

(рис.3) *результаты теста №1 для реализаций «CSetGraph» и «CArcGraph»*

3. Результаты для среднезаполненного графа размером 1000 и количеством вершин 600000:

```
graph_size = 1000
Generation
        time: 7387 ms, (7387158 µs)
edges_count = 600000
```

(рис.4) *Входные данные теста №2*

```
CListGraph
    CListGraph adding edges:
        time: 96 ms, (96678 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 0 ms, (2 µs)
        RPS = 166666.667
    get_next_vertices from middle:
        time: 0 ms, (7 µs)
        RPS = 166666.667
    get_next_vertices from back:
        time: 0 ms, (7 µs)
        RPS = 166666.667
```

```
CMatrixGraph
    CMatrixGraph adding edges:
        time: 20 ms, (20027 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 0 ms, (59 µs)
        RPS = 20000.000
    get_next_vertices from middle:
        time: 0 ms, (66 µs)
        RPS = 18518.519
    get_next_vertices from back:
        time: 0 ms, (66 µs)
        RPS = 16949.153
```

(рис.5) *результаты теста №2 для реализаций «CListGraph» и «CMatrixGraph»*

```
CSetGraph
    CSetGraph adding edges:
        time: 225 ms, (225193 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 5 ms, (5291 µs)
        RPS = 52631.579
    get_next_vertices from middle:
        time: 0 ms, (28 µs)
        RPS = 55555.556
    get_next_vertices from back:
        time: 0 ms, (23 µs)
        RPS = 58823.529
```

```
CArcGraph
    CArcGraph adding edges:
        time: 55 ms, (55633 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 6 ms, (6438 µs)
        RPS = 152.765
    get_next_vertices from middle:
        time: 6 ms, (6374 µs)
        RPS = 156.128
    get_next_vertices from back:
        time: 6 ms, (6140 µs)
        RPS = 158.178
```

(рис.6) *результаты теста №2 для реализаций «CSetGraph» и «CArcGraph»*

4. Результаты для плотного графа размером 1000 и количеством вершин 1000000:

```
graph_size = 1000
Generation
        time: 12355 ms, (12355933 µs)
edges_count = 1000000
```

(рис.7) *Входные данные теста №3*

```
CListGraph
    CListGraph adding edges:
        time: 160 ms, (160745 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 0 ms, (2 µs)
        RPS = 90909.091
    get_next_vertices from middle:
        time: 0 ms, (26 µs)
        RPS = 83333.333
    get_next_vertices from back:
        time: 0 ms, (12 µs)
        RPS = 83333.333
```

```
CMatrixGraph
    CMatrixGraph adding edges:
        time: 40 ms, (40653 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 0 ms, (74 µs)
        RPS = 15873.016
    get_next_vertices from middle:
        time: 0 ms, (75 µs)
        RPS = 15873.016
    get_next_vertices from back:
        time: 0 ms, (63 µs)
        RPS = 15873.016
```

(рис.8) *результаты теста №3 для реализаций «CListGraph» и «CMatrixGraph»*

```
CSetGraph
    CSetGraph adding edges:
        time: 330 ms, (330421 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 0 ms, (43 µs)
        RPS = 40000.000
    get_next_vertices from middle:
        time: 0 ms, (33 µs)
        RPS = 32258.065
    get_next_vertices from back:
        time: 0 ms, (38 µs)
        RPS = 34482.759
```

```
CArcGraph
    CArcGraph adding edges:
        time: 83 ms, (83110 µs)
    --Vertices count = 1000
    get_next_vertices from begin:
        time: 11 ms, (11841 µs)
        RPS = 93.058
    get_next_vertices from middle:
        time: 10 ms, (10362 µs)
        RPS = 90.761
    get_next_vertices from back:
        time: 10 ms, (10402 µs)
        RPS = 91.971
```

(рис.9) *результаты теста №3 для реализаций «CSetGraph» и «CArcGraph»*

В нашем случае граф будет плотным, поэтому рассмотрим результаты теста №3 (количество вершин - 1000, связей - 1000000):

	CListGraph	CSetGraph	CMatrixGraph	CArcGraph
Заполнение графа, мс	160	330	40	83
Начало, запросы/с	90909	40000	15873	93
Середина, запросы/с	83333	32258	15873	91
Конец, запросы/с	83333	34482	15873	92

(таблица 1) *характеристики графов по реализациям*

Анализируя результаты, выстраивается четкая последовательность скорости реализаций: «CListGraph», «CSetGraph», «CMatrixGraph», «CArcGraph». Скорость заполнения не учитывается, т.к. она происходит лишь один раз при инициализации системы, а получения различных вершин - постоянно. Стоит отметить, что существенная разница, заметна только на больших графах с высокой плотностью, на низких объемах выбор реализации не критичен.

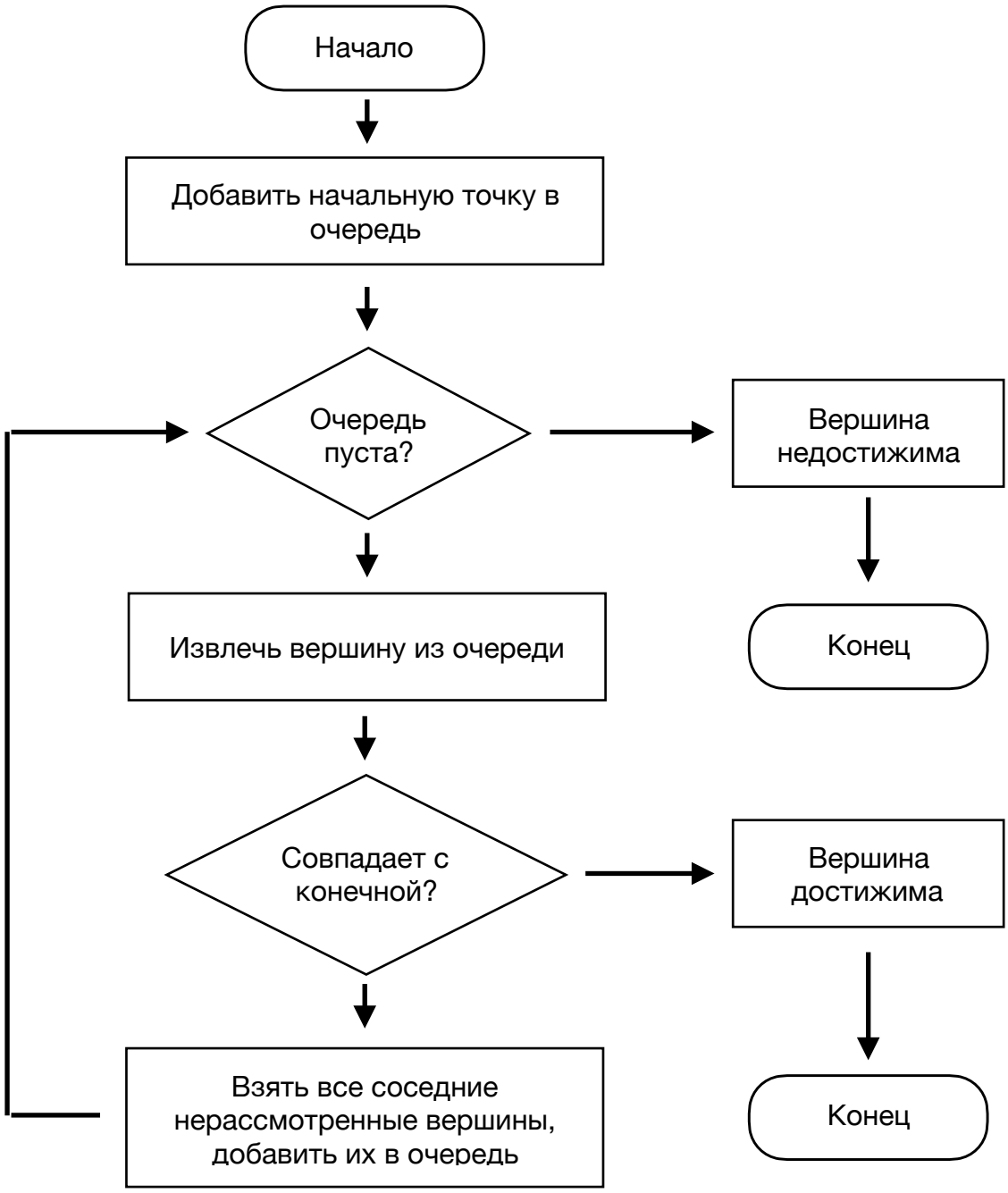
2. Алгоритмы поиска

Для выбора алгоритма необходимо определиться с представлением карты в виде графа. На выбор представляются 2 варианта: проставлять точки и соединять их вручную (трехмерный граф), или разделить граф на два (вертикаль и горизонталь). Первый вариант оптимален для поиска, но требует, много рутинной работы. Также присутствующие решения на рынке, использующие для определение местоположения пользователя маячки, применяют этот метод. Второй вариант требует минимального времени для заполнения, вся работа на отрисовку ложится на алгоритм, но из-за этого незначительно увеличивается время поиска.

Для поиска маршрута возможны алгоритмы

- «Поиск в ширину». Поиск в ширину выполняет исследование равномерно во всех направлениях. Это невероятно полезный алгоритм, не только для обычного поиска пути, но и для процедурной генерации карт, поиска путей течения, карт расстояний и других типов анализа карт
- «Алгоритм Дейкстры». Алгоритм Дейкстры (поиск с равномерной стоимостью) позволяет нам задавать приоритеты исследования путей. Вместо равномерного исследования всех возможных путей он отдаёт предпочтение путям с низкой стоимостью
- « A*». A* — это модификация алгоритма Дейкстры, оптимизированная для единственной конечной точки. Алгоритм Дейкстры может находить пути ко всем точкам, A* находит путь к одной точке. Он отдаёт приоритет путям, которые ведут ближе к цели.

2.1. Поиск в ширину



(рис. 10) *алгоритм поиска в ширину*

Поиск в ширину подходит для поиска маршрута только в невзвешенных графах из-за особенности алгоритма. В случае взвешенного графа найдет маршрут по числу ребер, но он не будет оптимальный, т.к. не учитываются веса ребер.

Так как в памяти хранятся все развёрнутые узлы, пространственная сложность алгоритма составляет $O(|V| + |E|)$, где V, E - число вершин и рёбер в графе соответсвенно.

Сложность алгоритма составляет $O(|V| + |E|)$, т.к. в худшем случае все узлы графа будут посещены.

2.2. Алгоритм Дейкстры

Аналогичен поиску в ширину и является его улучшением, использует очередь с приоритетом. Работает на взвешенных графах. Для поиска оптимального пути уже требуется обойти весь граф, т.к. кратчайшее расстояние определяется весом ребер, и первое вхождение в общем случае не является лучшим.

Оценка времени работы. Для реализации с двоичной кучей:

- Добавление узла: $O(logV)$. Не более V операций.
- Уменьшение значения ключа: $O(logV)$. Не более E операций.

Всего $T = O((E + V) * logV)$

Используемая память $M = O(V)$

2.3. Алгоритм A*

Тоже является улучшенной версией предыдущего алгоритма. Порядок обхода вершин определяется эвристической функцией «расстояние + стоимость» (обычно обозначаемой как f(x)). Эта функция — сумма двух других: функции стоимости достижения рассматриваемой вершины (x) из начальной (обычно обозначается как g(x) и может быть как эвристической, так и нет), и функции эвристической оценки расстояния от рассматриваемой вершины к конечной (обозначается как h(x)).

Функция h(x) должна быть допустимой эвристической оценкой, то есть не должна переоценивать расстояния к целевой вершине. Например, для задачи маршрутизации h(x)может представлять собой расстояние до цели по прямой линии, так как это физически наименьшее возможное расстояние между двумя точками.

Существуют различные эвристики, использующиеся в различных ситуациях. Например:

- Манхэттенское расстояние для перемещения в 4 направлениях плоскости

$$h(v) = |v.x - goal.x| + |v.y - goal.y|.$$

- Расстояние Чебышева применяется, когда к четырем направлениям добавляются диагонали

$$h(v) = max(|v.x - goal.x|, |v.y - goal.y|)$$

- Если передвижение не ограничено сеткой, то можно использовать евклидово расстояние по прямой

$$h(v) = \sqrt{(v.x - goal.x)^2 + (v.y - goal.y)^2}$$

В проекте были протестированы разные эвристики, например, для махэттенского расстояния поиск на сетке размером 1280x1080 недалеко стоящих точек занял 575 итераций, при спрямлении пути - 2141, а при евклидовом расстоянии уже 31918. Также усложнение эврестической функции на каждой итерации сильно отразилось на поиске маршрута в худшую сторону.

3. Поиск маршрута

В работе используется взвешенный неориентированный граф. Вся карта делится на 2 плоскости, горизонтальную и вертикальную. Вертикальный граф состоит из графа лестниц и связей между ними, поиск осуществляется алгоритмом Дейкстры. Горизонтальный - сетка, т.е. каждая вершина это пиксель и он связан с ближайшими. Размер горизонтального графа равен размеру изображения плана этажа, т.е. 1280 на 1080 пикселей. Поиск на этаже осуществляется алгоритмом A* с манхэтаннской эвристикой.

Алгоритм поиска состоит из комбинации двух алгоритмов, и позволяет прокладывать маршруты в ограниченных пространствах.

Неформальное описание:

1. Пользователь вводит 2 точки
2. Определяем координаты и этажи этих точек
3. Проверяем, находятся ли точки на одном этаже
    1. Да
        - Запускаем алгоритм A*
    2. Нет
        - Ищем ближайшие лестницы от начальной и конечной точек
        - Находим маршрут в вертикали по лестницам
        - Строим путь от конечной точки до конечной лестницы
        - Проходим по полученному маршруту, при совпадении этажей двух соседних точек, находим путь между этими точками в горизонтали. Добавляем его в список для отображения
        - Строим путь от начальной точки до начальной лестницы
        - Проходим полученный массив в обратную сторону и отображаем маршрут пользователю