Кратчайшие пути в невзвешенных графах

1 Постановка задачи

Дан простой связный неориентированный граф и задана стартовая вершина s. Необходимо найти кратчайшее расстояние из этой вершины до всех остальных. Задача с одной стороны кажется нетривиальной, однако есть алгоритм, который ее решает за довольно небольшое время. Сразу стоит упомянуть, что рассматриваемый алгоритм применим к графам с петлями и кратными ребрами. Также не влияет отсутствие связности и ориентированность ребер. При любой из этих модификаций алгоритм остается неизменным.

2 Волновой алгоритм

Немного другую задачу. Пусть граф представлен прямоугольным полем размера $N \times M$. Клетки являются смежными, если у них есть общая сторона или общий угол. Необходимо найти длину кратчайшего пути из клетки s в клетку t. Одним из вариантов решения задачи является волновой алгоритм (алгоритм Ли). Для того, чтобы понять принцип работы алгоритма, промоделируем мысленно следующий процесс. В каждой ячейке поля расположена спичка. В нулевой момент времени в клетке s спичку поджигают. За единицу времени спичка полностью сгорает, а все смежные с ней спички загораются. И так продолжается, пока огонь не дойдет до клетки t. Тогда ответом на вопрос о длине кратчайшего пути будет затраченное время. Сам путь восстановить легко и данная часть задачи будет рассмотрена позже. Однако, заметим, что пока мы считали длину пути до клетки t, м посчитали длину пути и для других, не интересующих нас клеток. Если бы мы позволили процессу продолжаться, он бы закончился тогда, когда все поле бы сгорело. Сам алгоритм не эффективен при поиске пути до конкретной клетки, но позволяет искать кратчайшие пути до всех клеток сразу. Рассмотрим сам алгоритм.

Алгоритм 1 Волновой алгоритм (алгоритм Ли)

```
1: procedure LEE(N, M, s, t)
2:     d = Matrix[N, M]
3:     d[i, j] = +∞, ∀i, j
4:     d[x_s, y_s] = 0
5:     nodes = s
6:     while nodes ≠ ∅ do
7:         next = ∅
8:         for (x, y) in nodes do
9:             for (i, j) in adjacent(x, y) do
10:                if d[i, j] == +∞ then
11:                    d[i, j] = d[x, y] + 1
12:                    next = next ∪ (i, j)
13:                    if (i, j) == (x_t, y_t) then
14:                        return d[i, j]
15:                    end if
16:                end if
17:            end for
18:        end for
19:        nodes = next
20:    end while
21:    No route
22: end procedure
```

Стоит учесть, что если целевая клетка не достижима, то расстояние до нее останется $+\infty$. Такое может произойти если граф не связный (поле разделено на отдельные участки непроходимыми клетками).

2.1 Упражнение №1

Даны 2 числа N и M, задающие размер поля. В следующие двух строках даны пары чисел, координаты стартовой клетки и целевой клетки. После идет строка с одним числом K. В следующих K строках идут пары чисел, обозначающие непроходимые клетки. За какое наименьшее число шагов можно добраться до целевой клетки. Если пути в нее нет, то выведите -1. Клетки являются смежными, если имеют общую сторону или общий угол.

Пример

Вход:	Выход:
6 8	5
1 1	
4 6	
0	

Вход:	Выход:
6 8	6
1 1	
4 6	
3	
1 3	
2 3	
3 3	

3 Поиск в ширину

Вернемся к исходной задаче. Как было видно в предыдущей секции, волновой алгоритм решает поставленную задачу. Только у нас граф произвольный и, соответственно код перебора ребер выглядит по другому. Однако принцип остается тот же. Такой алгоритм называется обходом в ширину (breadth-first search, bfs).

Алгоритм 2 Обход в ширину

```
1:  procedure BFS(V, E, s)
2:      N = |V|
3:      d = Vector[N]
4:      d[i] = +∞, ∀i
5:      d[s] = 0
6:      queue.push(s)
7:      while queue.size() > 0 do
8:          u = queue.pop()
9:          for (u, v) ∈ E do
10:             if d[v] == +∞ then
11:                 d[v] = d[u] + 1
12:                 queue.push(v)
13:             end if
14:         end for
15:     end while
16:     return d
17: end procedure
```

Одной из отличительных особенностей является то, что у нас пропал один цикл, который пробегался по всем вершинам, горящим в данный момент. Его роль на себя берет внешний цикл ‘while‘. Так как для хранения вершин тут мы используем очередь, то обходить будем вершины в том порядке, в котором они загораются.

Аналогично волновому алгоритму, bfs работает на несвязных графах. Целевая вершина может быть недостижима и расстояние до нее останется $+\infty$. Также такое возможно, если граф является ориентированным.

Рассмотренный выше волновой алгоритм основан на обходе в ширину. Сам bfs можно применять так же для поиска кратчайшего пути до конкретной вершины. Кроме того он применяется для поиска компонент связности или как часть других алгоритмов.

3.1 Упражнение №2

Дано два числа - N вершин и M ребер. В следующих M строках заданы пары чисел - неориентированные ребра графа. В последней строке заданы два числа s и t. Необходимо определить кратчайшее расстояние из s в t. Нумерация вершин с 0.

Пример

Вход:	Выход:
5 3	3
0 1	
1 2	
0 4	
2 4	

4 Восстановление пути

Теперь, когда мы научились определять длину кратчайшего пути, научимся определять сам путь. Восстановление пути в bfs не отличается от восстановления пути в задачах на ДП. В дополнение к массиву расстояний d будем хранить массив p. $p[u] = v$ будет означать, что мы пришли в вершину u из вершины v. Для случая с волновым алгоритмом размер массива p будет совпадать с размером поля, и в ячейках будут храниться координаты в виде пары чисел. После того, как мы нашли путь до целевой вершины t, начнем обратный проход по массиву p от вершины t. Мы просто будем идти по этому массиву, используя записанные в нем значения. Остановимся мы тогда, когда дойдем до стартовой вершины s. Пройденный таким образом "обратный"путь просто запишем в массив и перед выводом развернем. Таким образом получим искомый кратчайший путь.

4.1 Упражнение №3

Решая упражнение №1, выведите кратчайший путь.

Вход:	Выход:
6 8	6
1 1	1 1
4 6	1 2
4	0 3
1 3	1 4
2 3	2 5
3 3	3 6
	4 6

4.2 Упражнение №4

Решая упражнение №2, выведите кратчайший путь.
Пример

Вход:	Выход:
5 3	3
0 1	2 1 0 4
1 2	
0 4	
2 4	