ЛЕКЦИЯ 19
Обход графа в ширину (BFS)

1. Визуализация алгоритма BFS (пожар на графе)

Данный алгоритм можно сравнить с разгорающимся лесом (это отразилось на названии переменных в коде, см. ниже). Зажжем первую вершину (выбираем какую-то стартовую). От нее, как по мостикам, по ребрам графа огонь (наш алгоритм) переходит на соседние вершины. После этого сами соседние вершины уже зажигают своих соседей. При этом, чтобы наш алгоритм постоянно не записывал уже зажженные вершины, нам нужно, чтобы вершины «догорали» (необходимо убирать вершины, соседей которых уже проверили, из списка зажженных вершин). Т.е. этот алгоритм вкратце можно описать так:
1) Вершина графа загорается.
2) Передает огонь на все своих соседей.
3) Догорает.
И так происходит до того момента, пока мы не пройдем по всем соседям.

Более подробная визуализация изложена в википедии.

Анимация BFS

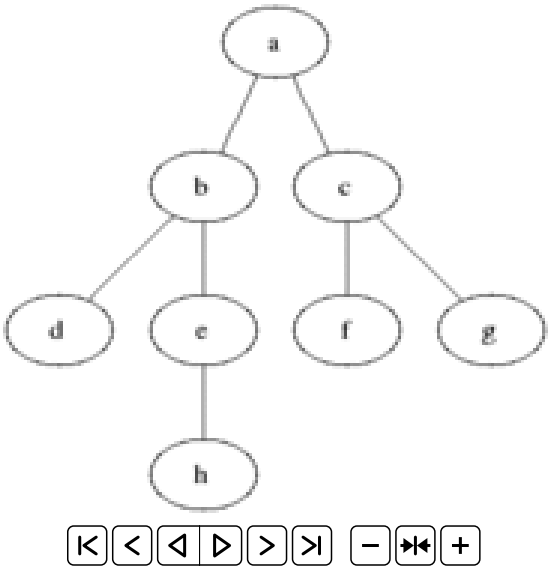

Рис. 1. Белый — вершина, которая еще не обнаружена. Серый — вершина, уже обнаруженная и добавленная в очередь. Черный — вершина, извлечённая из очереди.

Прагматический смысл обхода в ширину: *алгоритм позволяет находить расстояния на неориентированном графе от каждой вершины.* При этом остовное дерево BFS имеет наименьший диаметр для данной точки A.

Применение алгоритма:

- Построение остовного дерева
- Поиск расстояния (для не взвешенного графа)
- Обнаружение циклов наименьшей длины (когда происходит попытка зажечь вершину, которая уже горит, причем не ту, из которой позвали)
- Подсчет компонент связности

2. Реализация алгоритма BFS

Программа №2.1. Реализация алгоритма BFS

```
1    def bfs_fire(G, start, fired = None):
2        if fired is None:
3            fired = set()
4        fired.add(start)
5        time = {start: 0} # Хранение времен их добывания
```

```
6         Q = [start]
7         while Q:
8             current = Q.pop(0) # Для списка это не эффективно
9             for neighbour in G[current]:
10                if neighbour not in fired:
11                    fired.add(neighbour)
12                    Q.append(neighbour)
13                    print(current, neighbour) # Для построения остовного дерева
14                    time[neighbour] = time[current] + 1
```

Построчный комментарий кода:
2) Задаём `fired` как пустое множество (по умолчанию этого делать нельзя, т.к. тогда это станет глобальной переменной).
4) Добавляем стартовую вершину в «догоревшие».
5) Хранение времени, за которое мы доходим до вершины (это позволяет находить расстояние до вершины от стартовой).
6) Добавление вершины в очередь «горящих».
7) Цикл работает, пока есть хотя одна горящая вершина.
8) Берем первую вершины из очереди.
9) Проходим по всем соседям вершины.
11) Добавляем нашу вершину в очередь «зажженных».
12) Добавляем в «догоревших» (чтобы снова не записывать его в очередь «зажженных»).
13) Это позволяет получить остовное дерево.
14) Так и получаем расстояние до вершины.

3. Алгоритм Дейкстры

Данный алгоритм необходим для поиска кратчайшего маршрута от исходной вершины до всех вершин графа. Его метод работы основан на обходе графа в ширину (как мы помним, данный алгоритм позволяет находить расстояния в невзвешенном графе). Идея аналогичная — «зажигаем» вершины, после этого проходим по соседям «зажженной» вершины. Но теперь в нашем проходе появляется определенный порядок — вначале мы идем к вершинам, путь к которым наиболее краток. Это позволяет раньше добавить вершины в список «сгоревших» и закончить вычислять длину маршрута к ним.

Подробная визуализация алгоритма изложена в википедии.

Условие на граф: не должно быть ребер с отрицательным весом.

4. Реализация алгоритма Дейкстры

Программа №4.1. Реализация алгоритма Дейкстры (не эффективен)

```
1     def dijkstra(G, start): # G - словарь словарей с весами
2         d = {v: float('+inf') for v in G}
3         d[start] = 0
4         used = set()
5         while len(used) != len(G):
6             min_d = float('+inf')
7             for v in d:
8                 if d[v] < min_d and v not in used:
```

```
 9                     current = v
10                     min_d = d[v]
11             for neighbour in G[current]:
12                 l = d[current] + G[current][neighbour]
13                 if l < d[neighbour]:
14                     d[neighbour] = l
15             used.add(current)
16         return d # Алгоритм не эффективен
```

Построчный комментарий кода:
1) G — словарь словарей (каждой вершине соответствует список вершин, каждой из которых поставили в соответствие длину ребра).
2) Задаем изначальный словарь длин расстояний до вершин.
3) Задаем длину расстояния до исходной вершины (она ноль, очевидно).
4) Задаем множество пройденных вершин.
5) Пока не все вершины использованы.
6) Задаем расстояние (делаем его бесконечным, чтобы потом можно было найти минимальное).
7-10) Находим вершину с минимальным путем до нее. При первом проходе это будет начальная вершина (до нее путь ноль). При втором проходе она уже будет смотреть и выбирать из соседей исходной вершины и т.д.
11-14) Пробегаемся по соседям нашей вершины и рассчитываем путь до нее. Если он меньше того пути, который сейчас соответствует этой вершине, то мы записываем новое значение — это путь до той вершины, по соседям которой мы пробегаемся плюс путь от нее до соседа.
15) Добавляем проверенную вершину в использованное.

Программа №4.2. Реализация алгоритма Дейкстры

```
 1    from heapq import*
 2    def dijkstra(G, start): # G - словарь словарей с весами
 3        d = {v: float('+inf') for v in G}
 4        d[start] = 0
 5        Q = [(0, start)]
 6        used = set()
 7        while len(used) != len(G):
 8            d_c, current = heappop(Q)
 9            if d_c != d[current]:
10                continue
11            for neighbour in G[current]:
12                l = d[current] + G[current][neighbour]
13                if l < d[neighbour]:
14                    d[neighbour] = l
15                    heappush(Q, (l, neighbour))
16            used.add(current)
17        return d
```

Данная программа имеет такую же схему работы, как и предыдущая, но для большей эффективности использована пирамида кортежей.

Г. С. Демьянов, VK
С. С. Клявинек, VK