


# Graph traversal, Pt. 2

## Interlude

---

Это логическое продолжение первой части [Graph traversal, Pt. 1](#) , кто не читал — соболезную.

На просторах интернета есть много красивых анимашек, предлагаю начать, как всегда, с юзкейсов. В первой части за юзкейс была взята социальная сеть, пусть теперь это будет система дорог.

Я упоминал, что похожий граф используется в логистике Yandex, поэтому обойдемся без велосипедов, возьмем что есть с небольшими упрощениями для начала.

## How to

---

Небольшое введение в доменную область:

```
public class Stock
{
    public Stock(int id, string address, bool isActive)
    {
        Id = id;
        Address = address;
        IsActive = isActive;
    }

    public int Id { get; }

    public string Address { get; }

    public bool IsActive { get; set; }
}
```

```

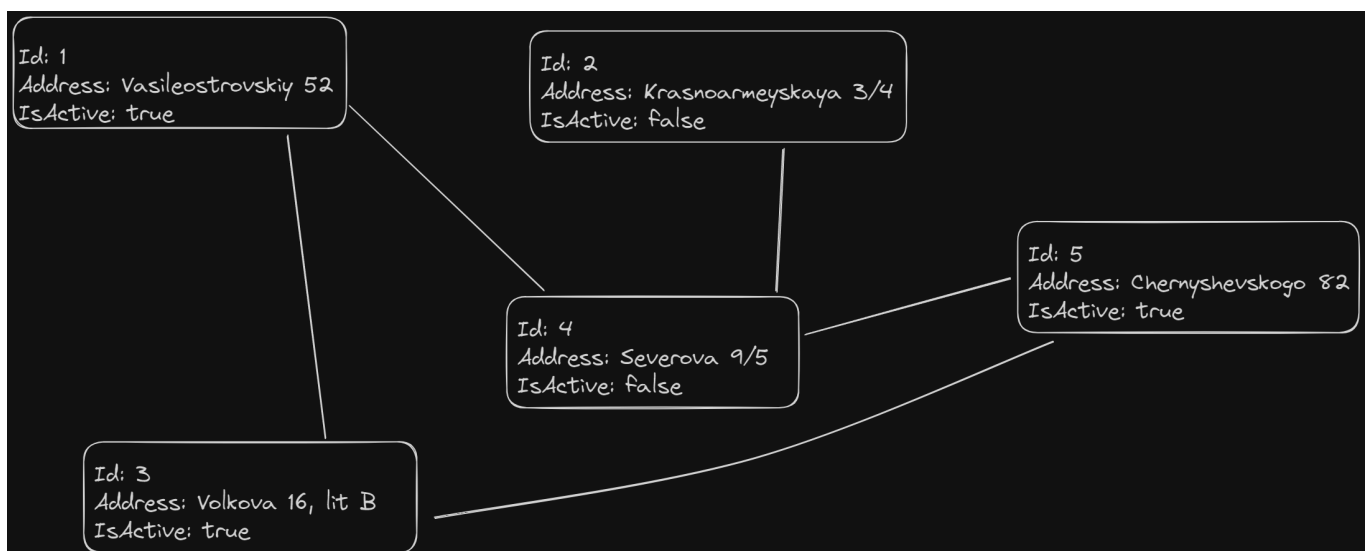
public class Road
{
    public Road(int fromId, int toId)
    {
        FromId = fromId;
        ToId = toId;
    }

    public int FromId { get; }

    public int ToId { get; }
}

```

В Санкт-Петербурге есть склады, которые посещаются ребятами из логистики. Будем хранить упрощенный вид карты — граф, в котором вершины это склады, а ребра дороги между этими складами.



Представим, что логистика обращается к диспетчеру со следующей проблемой. Стоки в данный момент сильно нагружены: какие-то из них сейчас неактивны, потому что заняты разгрузкой в соседние стоки. Какой из стоков имеет наибольшее количество соседних стоков, которые готовы принять разгрузку ( активных ).

## Breadth first search

Алгоритм, который позволяет решить поставленную задачу называется Breadth First Search ( BFS ). План решения

следующий:

- Пройтись по всем стокам
- Для каждого стока посмотреть **только** на его соседей
- Посчитать количество активных соседних стоков  $k$
- Найти максимум по всем  $k$

```
(int, int) getMobileStock(Stack stock){
    var queue = []
    queue.push(stock)

    int maxActiveAround = 0
    int mostMobileStockId = stock.Id
    while( queue is not empty){
        auto current = queue.dequeue()

        if (current.Id is not stock.Id and current.IsActive)
// текущий сток не хочется считать
            maxActiveAround++

        for(auto edge : udg[id]){
            if (edge.FromId is not id)
// если вдруг начали смотреть дальше
                return (maxActiveAround, mostMobileStockId)
// соседей, то просто вернем ответ
            queue.push(stocks[edge.ToId])
        }
    }

    return (maxActiveAround, mostMobileStockId)
}

auto ans = -1
auto maxAround = 0
for(auto stock : stocks){
    var s = getMobileStock(stock)

    if (maxAround < s.maxActiveAround){
        maxAround < s.maxActiveAround
        ans = s.mostMobileStockId;
    }
}
```

```
}
```

```
return ans;
```

Данный алгоритм, можно немного модифицировать, чтобы получать несколько самых мобильных стоков. А ещё можно хранить данные компактнее. Если говорить про асимптотику то:

- Проходимся по каждой вершине `stocks` единожды
- Проходимся по всем инцидентным ребрам `udg[i]` единожды
- Размерности `stocks` и `udg` совпадают

Из чего следует, что время работы  $O(|V| + |E|)$ .

## What else ?

### Distance between vertices

Представим, что нам нужно понять сколько стоков нужно будет посетить, чтобы попасть из стока  $u$  в сток  $v$ .

```
int getDistanceBetween(int fromId, int toId){
    var queue = []
    queue.push(fromId)
    distances = [].fill(-inf)
    distances[fromId] = 0;

    while(queue is not empty){
        auto current = queue.dequeue()

        for(auto edge : udg[current])
            if (distances[edge.ToId] is -inf){
                queue.push(edge.ToId)
                distances[edge.ToId] = distances[current] + 1
            }
    }
}
```

```
    return distances[toId];  
}
```

Так выглядит классический BFS. Как правило, все модификации происходят в способе хранения графа, сигнатуре и условии обхода вершины.

## Path finding 🦶

Эта история до боли знакома тем, кто хоть когда то восстанавливал ответ в ДП.

```
(int, int[]) getDistanceBetween(int fromId, int toId){  
    var queue = []  
    queue.push(fromId)  
  
    parents = [].fill(-1)          // добавим вот это  
    distances = [].fill(-inf)  
  
    distances[fromId] = 0;  
  
    while(queue is not empty){  
        auto current = queue.dequeue()  
  
        for(auto edge : udg[current])  
            if (distances[edge.ToId] is -inf){  
                queue.push(edge.ToId)  
                distances[edge.ToId] = distances[current] + 1  
                parents[edge.ToId] = current    // и вот это  
            }  
    }  
  
    return (distances[toId], parents);  
}  
  
auto targetVertex = 5  
(_, parents) = getDistanceBetween(1, targetVertex)  
  
auto current = targetVertex  
path = []  
while (current is not -1){
```

```
    path.pushFront(current)    // здесь pushFront, потому
    что история такая же
    current = parents[current] // как с сортировкой графа
}
```

## Shortest path 🚊 🚊

---

Обычно, BFS не используют для поиска кратчайших путей, но в некоторых ситуациях это допустимо. Алгоритмы поиска кратчайших путей будут рассмотрены чуть позже, а пока что можно поиграться с тем, что есть

Создадим ещё один доменный объект платной дороги:

```
public class Stock
{
    public Stock(int id, string address, bool isActive)
    {
        Id = id;
        Address = address;
        IsActive = isActive;
    }

    public int Id { get; }

    public string Address { get; }

    public bool IsActive { get; set; }
}

public class Road
{
    public Road(int fromId, int toId)
    {
        FromId = fromId;
        ToId = toId;
    }

    public int FromId { get; }
```

```

    public int ToId { get; }
}

public class TollRoad : Road
{
    public Road(int fromId, int toId, double fee) :
base(fromId, toId)
    {
        Fee = fee;
    }

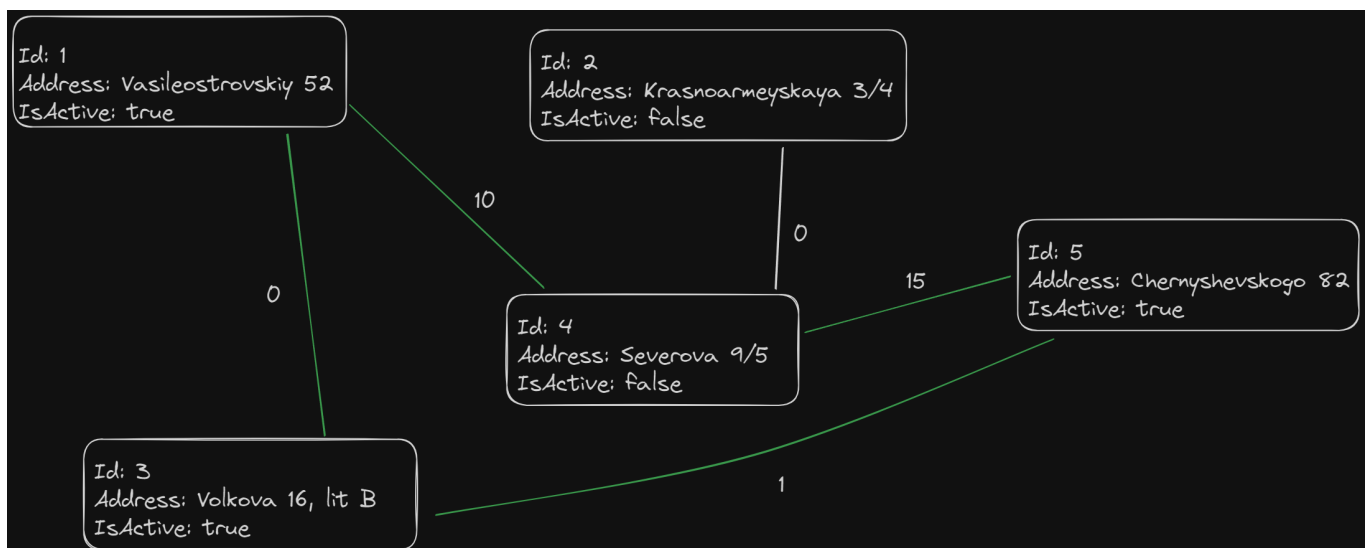
    public double Fee { get; }
}

```

Под платной дорогой можно понимать

- КАД или трассу, где за проезд буквально надо платить
- Какие-то исчисляемые ресурсы

Пусть на этом примере это будет стоимость дороги и логистика хочет получить информацию о путевых расходах до каждого стока.



```

int[] getCashExpenses(){
    var queue = []
    queue.push(0)
    distances = [].fill(-inf)
    distances[0] = 0;

    while(queue is not empty){

```

```

    auto current = queue.dequeue()

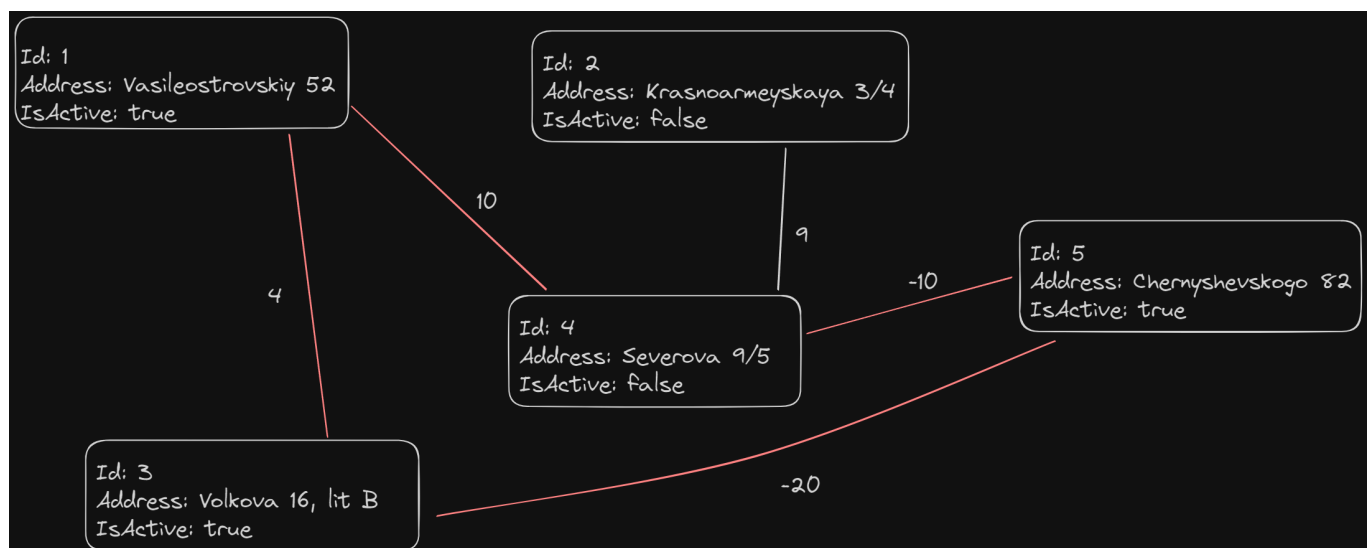
    for(auto edge : udg[current])
        if (distances[edge.ToId] > distances[current] +
edge.Fee){ // если можем потратить
            queue.push(edge.ToId)
// меньше
            distances[edge.ToId] = distances[current] +
edge.Fee
        }
    }

    return distances;
}

```

И все вроде бы классно, но я не зря пометил некоторые ребра зеленым. Посмотрим теперь на следующий пример, где платная дорога — это дорога у которой

- $Fee > 0$  , если время проведенное в дороге было потрачено на пробки
- $Fee < 0$  , если время проведенное в дороге не было потрачено в пробках



Применим тот же самый алгоритм, начнем из `id:1` , игнорируя `id:2` , и рассмотрим красный цикл:



distances

|   |     |     |     |     |
|---|-----|-----|-----|-----|
| 1 | 2   | 3   | 4   | 5   |
| 0 | inf | inf | inf | inf |

id: 1 -> id: 4  
inf > 0 + 10

|   |     |     |    |     |
|---|-----|-----|----|-----|
| 1 | 2   | 3   | 4  | 5   |
| 0 | inf | inf | 10 | inf |

id: 4 -> id: 5  
inf > 10 + (-10)

|   |     |     |    |   |
|---|-----|-----|----|---|
| 1 | 2   | 3   | 4  | 5 |
| 0 | inf | inf | 10 | 0 |

id: 5 -> id: 3  
inf > 0 + (-20)

|   |     |     |    |   |
|---|-----|-----|----|---|
| 1 | 2   | 3   | 4  | 5 |
| 0 | inf | -20 | 10 | 0 |

id: 3 -> id: 1  
inf > -20 + 4

|     |     |     |    |   |
|-----|-----|-----|----|---|
| 1   | 2   | 3   | 4  | 5 |
| -16 | inf | -20 | 10 | 0 |

id: 1 -> id: 4  
10 > -16 + 10

|     |     |     |    |   |
|-----|-----|-----|----|---|
| 1   | 2   | 3   | 4  | 5 |
| -16 | inf | -20 | -6 | 0 |

"кратчайший" путь начал перезаписываться,  
потому что имеется цикл отрицательного веса

**Циклом отрицательного веса** называется цикл, вес хотя бы одного из ребер которого отрицательный. В такой ситуации BFS никогда не выйдет и такого цикла.

Похожее ограничение можно будет увидеть, когда разберем алгоритмы поиска кратчайших путей в графе.

## Почему так никто не делает

BFS не создан для поиска кратчайших путей, в первую очередь — это алгоритм обхода. Всегда при выборе

алгоритма люди балансируют между затратами по памяти и времени.

В отличие от классической реализации BFS, здесь допускается повторный проход по ребрам и вершинам, что убивает оптимальность.

Оценка этой модификации по времени:

- В худшем случае каждое ребро может быть проверено каждый раз, когда вершина извлекается из очереди
  - Поскольку релаксация может произойти для каждого ребра, соединяющего текущую вершину с соседней, это может привести к повторному добавлению вершины в очередь
  - Каждая вершина может быть помещена в очередь  $V$  раз
- В итоге получаем  $O(V^2 * E)$ , что плохо как для разреженных так и для полных графов.

Оценка этой модификации по памяти  $O(V + E)$

## DFS vs BFS

---

Помимо обхода в глубину граф можно обходить и в ширину. Концептуальное отличие заключается в том, что в случае **DFS** обход идет в сторону самой удаленной достижимой вершины из текущей. В случае **BFS**, все вершины рассматриваются по мере поступления — все достижимые из текущей, а потом все достижимые из достижимых.

Как и с любым другим обходом, с помощью BFS можно делать много смешных вещей, но опять же есть свои ограничения, которые важно учитывать, как было видно из предыдущего параграфа.