Задача
Симуляция аварий в оптической сети

Александр Максименко, Александр Курилкин, Кирилл Хлыновский

30 ноября 2023 г.

Аннотация

Оптическая сеть может быть представлена как граф. Каждое ребро этого графа — это оптический кабель (оптоволокно). Информация по нему передается в виде света на определенной частоте (длине волны). По одному и тому же оптоволокну можно одновременно передавать несколько десятков сигналов на разных частотах. Время от времени, некоторые кабели в сети могут выходить из строя. В таких случаях специальная программа-маршрутизатор (например, написанная в онлайн-марафоне, посвященному этой задаче) старается найти новые пути прохождения аварийно прерванных сигналов (сервисов). Алгоритм работы этой программы вам неизвестен. Вам нужно написать симулятор (генератор) аварий, который, в зависимости от текущего состояния сети, выбирает ребро в графе так, чтобы при выходе его из строя программа-маршрутизатор смогла восстановить как можно меньше сервисов.

Ваша задача — написать погрaмму, выбирающую в оптической сети следующее аварийное ребро так, чтобы максимально осложнить программе-маршрутизатору восстановление прерванных сигналов. Для успешного решения вам понадобится формулировка задачи, которую решает программа-маршрутизатор.

1 Задача программы-маршрутизатора

Дано:

1. Константа $W \in \mathbb{N}$ (число частот).

2. Неориентированный планарный граф $G = (V, E)$. В каждом ребре доступны W частот (каналов), пронумерованных $1, 2, \ldots, W$.

3. Множество сервисов D. Каждый сервис $d \in D$ это четверка (s_d, t_d, w_d^*, p_d^*), где $s_d \in V$ — *начало*, $t_d \in V$ — *конец*, $s_d \neq t_d$, $p_d^* \subseteq E$ — простой неориентированный путь из s_d в t_d, $w_d^* \in \{1, 2, \ldots, W\}$ — *частота* занимаемая

сервисом d в ребрах p_d^*. Далее, p_d^* называется *исходным путем*, а w_d^* — *исходной частотой* сервиса d.

В исходном состоянии, для каждого сервиса $d \in D$ его текущий путь p_d инициализируется исходным путем p_d^*, а текущая частота w_d — исходной частотой w_d^*.

Пусть d_1 и d_2 — два различных сервиса ($d_1 \neq d_2$). Если их текущие пути p_{d_1} и p_{d_2} используют общее ребро, то их текущие частоты должны быть различны: $w_{d_1} \neq w_{d_2}$. Другими словами, два сервиса *не могут* использовать одну и ту же частоту в одном и том же ребре.

В процессе симуляции, программа-маршрутизатор получает запросы по одному за раз (следующий запрос заранее неизвестен). Один запрос — это ребро $e \in E$, которое следует удалить. В следующих запросах, это ребро не восстанавливается (множество удаленных ребер увеличивается с каждым следующим запросом). Сервис d называется *поврежденным запросом* e, если $e \in p_d$ (текущий путь использует ребро e). При получении такого запроса, программа должна найти новые пути и частоты для поврежденных сервисов. Гарантируется, что после каждого запроса сеть остается связной (граф связен, но наличие незанятых частот не гарантируется).

Цель: Обработать последовательность запросов Q. Для каждого запроса $e \in Q$, определить подмножество поврежденных запросов $F \subseteq D$ и найти новые пути и частоты для них. Если программа не может найти новый путь для какого-нибудь сервиса, его текущий путь обновляется пустым множеством $\emptyset$.

Требования:

1. Число успешно проложенных сервисов нужно максимизировать. Результат обработки текущего запроса значительно важнее (приоритетнее), чем результат обработки следующего запроса. Более точную формулировку см. ниже.

2. Если текущий путь p_d сервиса d не пуст и не поврежден текущим запросом, то этот путь p_d и частота w_d не могут быть изменены (при обработке данного запроса).

3. Для любых двух сервисов d_1 и d_2, $d_1 \neq d_2$, если их текущие пути используют общее ребро ($p_{d_1} \cap p_{d_2} \neq \emptyset$), то они должны использовать разные частоты: $w_{d_1} \neq w_{d_2}$.

4. Пусть p_d^* и w_d^* — исходные путь и частота сервиса $d \in D$. Текущий путь $p_{d'}$ любого другого сервиса $d' \neq d$ не может использовать ресурсы исходного пути p_d^*, то есть $p_d^* \cap p_{d'} \neq \emptyset \Rightarrow w_d^* \neq w_{d'}$.

2 Ваша задача: выбор аварийного ребра

На этапе инициализации, вы получаете в точности тот же набор исходных данных, что и программа-маршрутизатор (см. выше), текущие пути и частоты сервисов инициализируются их исходными путями.

Основная задача — реализовать функцию выбора ребра `gen_request()`, в зависимости от текущего состояния сети. А именно, входом этой функции будет набор текущих путей и частот всех запросов, выходом — ребро, которое будет аварийным в следующем запросе для программы-маршрутизатора.

При тестировании одного набора исходных данных, функция `gen_request()` будет вызвана не более 10 раз. После каждого вызова, соответствующее аварийное ребро передается в программу-маршрутизатор, которая выбирает для поврежденных сервисов новые пути и частоты. В свою очередь, обновленные пути и частоты всех сервисов передаются в `gen_request()` в следующем вызове. При этом набор аварийных ребер монотонно увеличивается (аварийные ребра не восстанавливаются).

Требование: После удаления всех аварийных ребер граф должен оставаться связным.

3 Формат данных

Исходные данные открытых тестов будут представлены в виде текстовых файлов со следующим содержанием.

Первая строка содержит четыре натуральных числа, разделенных пробелами: число узлов N, число ребер M, число частот W, число сервисов K, $1 \le N \le 16$, $1 \le M \le 40$, $1 \le W \le 32$, $1 \le K \le 400$.

Следующие N строк содержат информацию о координатах узлов. Это служебная информация, которая будет недоступна вашему решению.

Далее идут M строк, содержащих информацию о ребрах. Каждая строка содержит три натуральных числа, разделенных пробелами: идентификатор ребра $i \in [M]$, $[M] = \{1, 2, \ldots, M\}$, и идентификаторы двух узлов v_i и u_i, $1 \le v_i < u_i \le N$.

Следующие K строк содержат информацию о сервисах. Каждая строка содержит несколько натуральных чисел, разделенных пробелами: идентификатор сервиса d, идентификатор узла-начала $s_d \in [N]$, идентификатор узла-конца $t_d \in [N]$, исходная частота $w_d \in [W]$, число ребер исходного пути p_d и список идентификаторов ребер этого пути.

Последняя строка содержит два натуральных числа R ($1 \le R \le 10$) и id — кол-во раз, которое будет вызываться процедура `gen_request()` (для всех тестов, кроме примеров, $R = 10$), и идентификатор программы-маршрутизатора для этого теста. Это служебная информация, которая будет недоступна вашему решению.

4 Формат решения

Ваше решение должно выглядеть следующим образом:

```
#include "rerouting2.h"
...

void init(int N, int M, int W, int K,
          std::vector<Edge> initEdges,
          std::vector<Service> initServices) {
    ...
}

int gen_request(std::vector<Route> currentRoutes){
    int edge_id;
    ...
    return edge_id;
}
```

Массив `currentRoutes` содержит ровно K элементов — текущий путь и частоту для каждого сервиса. Перед тем, как этот массив передастся в процедуру `gen_request()`, он считывается со стандартного потока ввода в следующем формате.

На вход программе подается K строк. Для каждого сервиса, соответствующая строка содержит несколько чисел, разделенных пробелами: идентификатор сервиса d, текущая частота $w_d \in [W]$, число ребер текущего пути p_d и идентификаторы ребер, составляющих этот путь. Если путь не был найден программой-маршрутизатором, то $w_d = 0$ и $p_d = \emptyset$.

Получив массив текущих путей, процедура `gen_request()` должна вернуть номер ребра e ($1 \leq e \leq M$), которое будет аварийным на следующем шаге маршрутизатора. Оно выведется в стандартный поток вывода.

В случае, если вы вернули номер ребра, несоответствующий условию (например, уже удаленное ребро или ребро, удаление которого делает граф несвязным), ваша программа немедленно завершится.

Процедура `init()` будет вызвана ровно один раз для каждого теста и передаст в вашу программу исходные данные. Далее, процедура `gen_request()` будет вызвана не более 10 раз. С каждым следующим разом, множество удаленных ребер будет увеличиваться. Поэтому вам следует хранить его в глобальной (по отношению к вашим процедурам) структуре данных (например, в массиве).

Эти две процедуры будут вызываться тестирующей системой. В своих процедурах вы не можете использовать стандартные потоки ввода/вывода!

Файл `rerouting2.h` содержит следующие определения:

```
#include <vector>
```

```
struct Service {
    int id, s, t, w;
    std::vector<int> p;
};

struct Route {
    int service_id, w;
    std::vector<int> p;
};

struct Edge {
    int id, u, v;
};

void init(int N, int M, int W, int K,
          std::vector<Edge> initEdges,
          std::vector<Service> initServices);

int gen_request(std::vector<Route> currentRoutes);
```

5 Подсчет очков

1. Для проверки вашего решения будут использоваться три различных программы-маршрутизатора.

2. Будут использованы три набора тестов: простые примеры, открытый набор, закрытый набор. Примеры открыты и не учитываются при начислении очков. Для их тестирования используется пример маршрутизатора, который вы можете найти в eJudge в `Samples ZIP`. Открытый набор содержит три группы, в каждой по 10 тестов, суммарно 30 тестов. Их также можно найти в eJudge в `Samples ZIP`. Тесты в группах идентичны, за исключением того, что в каждом используется своя программа-маршрутизатор, код которых вы не знаете. Закрытый набор устроен аналогично, за исключением того, что в каждой группе по 20 тестов (вместо 10).

3. Открытый и закрытый наборы в целом схожи, но различаются в деталях, так как они генерируются случайным образом.

4. Открытый набор используется для подсчета очков во время основной части соревнования. Участники могут видеть актуальные результаты тестирования.

5. Закрытый набор используется для финального скоринга решений участников, отправленных последней посылкой.

6. Суммарное время работы ваших процедур для одного теста (один вызов процедуры `init()` и 10 вызовов процедуры `request`) не может превышать 3 секунды. (Суммарно, проверка 30 тестов может занимать около трех минут, так как включает запуск программы-маршрутизатора).

7. Многопоточные решения запрещены. Разрешено использовать не более 1 GB оперативной памяти. Количество посылок неограничено. Вы можете отправлять новое решение сразу после завершения тестирования вашего предыдущего решения.

8. Если решение нарушает ограничения по времени, памяти, или какие либо требования, прописанные в условии, то оно получает 0 баллов.

9. Для запроса q, результат работы программы маршрутизации оценивается числом
$$C(q) = \lfloor 4^{10-b(q)} \cdot 100 \cdot X/K \rfloor,$$
где $b(q)$ — порядковый номер запроса (от 1 до 10), X — общее число сервисов, для которых не удалось найти новые пути (т.е. $p_d = \emptyset$), K — общее число всех сервисов в исходных данных теста. Таким образом, результат, полученный для текущего запроса, гораздо важнее, чем результат следующего запроса (в одном и том же тесте).

10. Результаты всех запросов складываются. Чем больше сумма, тем лучше результат.

11. Для финального скоринга будет использовано последнее успешное решение каждого участника. Тестирование будет выполняться на закрытом наборе. Побеждает решение с наибольшим средним очков по всем тестам. Если два решения набирают одинаковое число очков, выигрывает наиболее быстрое решение.

6 Примеры

6.1 Пример 1

Сеть состоит из 4 узлов и 5 ребер. Доступны три частоты. Исходные пути трех сервисов изображены в левой части рис. 1.

В качестве первого запроса выбираем ребро 1–2 (его используют два сервиса). Программа-маршрутизатор перепрокладывает оба поврежденных запроса по пути 1–3–4, занимая частоты 2 и 3 (см. среднюю часть рис. 1). Число набранных очков $4^{10-1} \cdot 0/3 = 0$.

В качестве второго запроса выбираем ребро 3–4 (ребро 1–3 мы выбрать не можем, так как иначе граф перестанет быть связным). Программа-маршрутизатор перепрокладывает 2 сервиса из 3 (правая часть рис. 1). Только 2 из 3 сервисов могут быть проложены, так как исходный путь синего сервиса использует частоту 1 в ребре 1–3, а исходный путь зеленого сервиса использует частоту 1 в ребре 2–4. Новый путь оранжевого сервиса использует ресурсы исходного пути этого же сервиса (частота 2 в ребре 2–4). Число набранных очков $4^{10-2} \cdot 100 \cdot 1/3 = 2\,184\,533$. Дальнейшее удаление ребер в этом примере недопустимо, так как граф перестанет быть связным. Итоговый результат для этого теста: $0 + 2\,184\,533 = 2\,184\,533$.

Исходные данные:

```
4 5 3 3
1 0 0
2 1 1
3 1 -1
4 2 0
1 1 2
2 1 3
3 2 3
4 2 4
5 3 4
1 1 4 1 2 1 4
2 1 4 2 2 1 4
3 1 4 1 2 2 5
2 0
```

Выполнены два запроса: 1 и 5

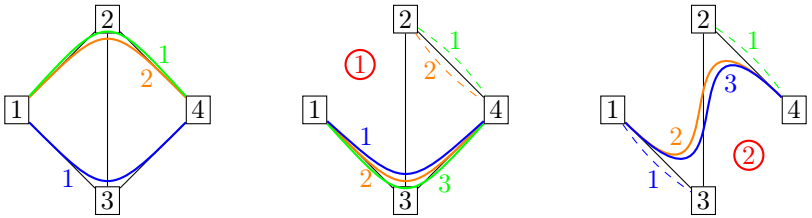

Рис. 1: Пример 1: исходные данные и результаты поиска новых путей для двух запросов. Исходная частота для зеленого и синего сервисов равна 1, для оранжевого сервиса — 2. После первого запроса частота зеленого сервиса меняется на 3. После второго запроса, проложены два из трех сервисов.

6.2 Тест 2

Сеть состоит из 5 узлов и 7 ребер. Доступны три частоты. Исходные пути семи сервисов изображены на рис. 2.

Первый запрос — ребро 2–5 (см. левую часть рис. 3). Новый путь для зеленого сервиса найти невозможно, так как частоты 1 и 2 заняты оранжевыми сервисами, а частота 3 — синим и фиолетовым сервисами. Число набранных очков $4^{10-1} \cdot 100 \cdot 1/7 = 3\,744\,914$.

Второй запрос — ребро 5–3 (см. среднюю часть рис. 3). Синий сервис не может быть проложен, так как частоты 1 и 2 заняты оранжевыми и коричневыми сервисами, а частота 3 — фиолетовым сервисом. Число набранных очков $4^{10-2} \cdot 100 \cdot 2/7 = 1\,872\,457$.

Третий запрос — ребро 3–4 (правая часть рис. 3). Только один из двух коричневых сервисов может быть восстановлен. Число набранных очков $4^{10-3} \cdot 100 \cdot 3/7 = 702\,171$.

Итоговый результат для этого теста: $3\,744\,914 + 1\,872\,457 + 702\,171 = 6\,319\,542$.

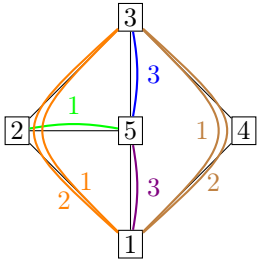

Рис. 2: Пример 2: исходное состояние сети.

Исходные данные:

```
5 7 3 7
1 0 -1
2 -1 0
3 0 1
4 1 0
5 0 0
1 1 2
2 1 4
3 1 5
4 2 3
5 2 5
6 3 4
7 3 5
1 2 5 1 1 5
2 3 5 3 1 7
3 1 5 3 1 3
4 1 3 1 2 1 4
5 1 3 2 2 1 4
6 1 3 1 2 2 6
7 1 3 2 2 2 6
3 3
```

Выполнены три запроса: 5, 7, 6.

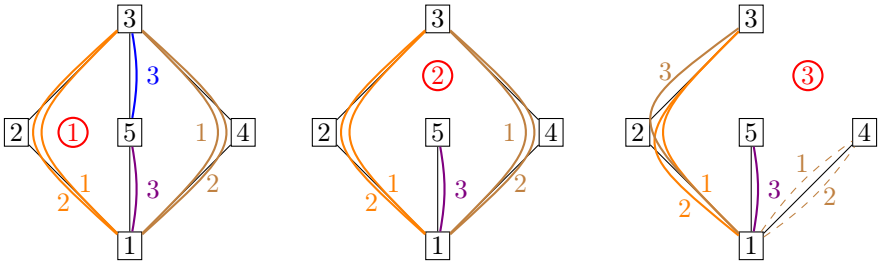

Рис. 3: Пример 2: результаты трех запросов на удаление ребра.