Task
Rerouting in an optical network

Alexander Maximenko, Alexander Kurilkin

November 21, 2023

Abstract

An optical network may be represented as a graph. Every edge of this graph is an optical cable (fiber). An information is transmitted as a light with a specific wavelength (frequency). An optical fiber can transmit simultaneously multiple signals with different wavelengths. From time to time, some elements (fibers) of a network may fail and we need to find new routes for the interrupted services (connections). Moreover, there may be a random sequence of failures in a network. The goal of the task is to provide the functioning of as many services as possible.

1 Mathematical formulation

Given:

1. A constant $W \in \mathbb{N}$ (the number of wavelengths).
2. An undirected planar graph $G = (V, E)$. In every edge, there are W wavelengths (channels) enumerated by $1, 2, \ldots, W$.
3. A set of services D. Every service $d \in D$ is a tuple (s_d, t_d, w_d^*, p_d^*), where $s_d \in V$ is the *source*, $t_d \in V$ is the *destination*, $s_d \neq t_d$, $p_d^* \subseteq E$ is a simple undirected path from s_d to t_d, and $w_d^* \in \{1, 2, \ldots, W\}$ is the *wavelength* occupied by the service d in edges p_d^*. Further, p_d^* is called the *initial path* and w_d^* is called the *initial wavelength* of a service d.

At the start of network failures simulation, for each $d \in D$, we initialize the current path p_d by the initial path p_d^* and the current wavelength $w_d := w_d^*$.

Let d_1 and d_2 are some services ($d_1 \neq d_2$). If current paths p_{d_1} and p_{d_2} have a common edge, then $w_{d_1} \neq w_{d_2}$. In other words, any two services *can not* use the same wavelength in the same edge.

During simulation, you get several requests one-by-one (they are not known to you in advance). There are two types of requests: 1) restore the initial state, 2)

an edge $e \in E$ is failed. For a request of the first type, you need to restore the initial state of the network: all failed edges have to be restored, initial paths and wavelengths of all services have to be restored ($p_d := p_d^*$, $w_d := w_d^*$, $\forall d \in D$).

Let's consider a request of the second type: an edge $e \in E$ is failed. Further, we call it a *fault request.* A service d is called *failed for a request* e if $e \in p_d$ (the current path of d contains the failed edge). After each failure, we need to find new paths and wavelengths for failed services. In next fault requests, the failed edge is not restored (the set of failed edges increases with each fault request). The failed edges are restored only by requests of the first type. It's guaranteed that after each failure network stays connected (but availability of wavelengths is not guarateed).

Goal: Process a sequence Q of requests. For each fault request $e \in Q$, detect failed services $F \subseteq D$ and find new paths and wavelengths for them. If you can't find a new path for a service, then its current path should be assigned an empty set $\emptyset$.

Requirements:

1. The number of successfully rerouted services should be maximized. The result for the current fault request is much more important than the result of the next fault request. The priority (importance) of all fault requests that appear right after initial state (or after a request of the first type) are the same. For more formal clarification see Sec. 4 below.

2. If a service d has nonempty current path p_d and it is not failed for the current fault request, then it can not be rerouted (p_d and w_d can't be changed).

3. For any two services d_1 and d_2, $d_1 \neq d_2$, if their current paths have a common edge ($p_{d_1} \cap p_{d_2} \neq \emptyset$), then they must use different wavelengths $w_{d_1} \neq w_{d_2}$.

4. Let's consider the initial path p_d^* and wavelength w_d^* of a service $d \in D$. The current path of any other service $d' \neq d$ (and after any request) can't use resources of the initial path p_d^*: $p_d^* \cap p_{d'} \neq \emptyset \Rightarrow w_d^* \neq w_{d'}$.

2 Data format

The input data of open tests are given as text files with the following format.

The first line contains four integers separated by spaces: the number of nodes N, the number of edges M, the number of wavelengths W, the number of services K, $1 \leq N \leq 16$, $1 \leq M \leq 40$, $1 \leq W \leq 32$, $1 \leq K \leq 400$.

The next N lines contain an information about nodes. Each line contains three integers separated by a space. Node id $i \in [N]$, $[N] = \{1, 2, \ldots, N\}$ and its coordinates x_i, y_i. This information is used for internal needs by testing system and won't be available to your solution.

The next M lines contain an information about edges. Each line contains three integers separated by a space: edge id $i \in [M]$, and ids of two nodes v_i and u_i, $1 \le v_i < u_i \le N$.

The next K lines contain an information about services. Each line contains several integers separated by spaces: the service id d, the id of source node $s_d \in [N]$, the id of destination node $t_d \in [N]$, the initial wavelength $w_d \in [W]$, number of edges in the initial path p_d and edges ids of the initial path p_d.

The next line contains an integer R ($1 \le R \le 100$) — number of requests in the test.

The last line contains R integers separated by a space: a sequence of requests.

A request is an integer $q \in \{0, 1, \ldots, M\}$. If $q = 0$, then you should restore the initial state. If $q \in [M]$, then q is the id of the failed edge. The next request is sent only after you finish processing of the current request.

There are no more than 100 requests for one testcase. The sequence of requests can be splitted into *subsequences* by requests of the first type. For example, sequence

```
3 1 0 2 0 1 4 2
```

is splitted into three subsequences of fault requests: `3 1`; `2`; `1 4 2`. The number of fault requests in a subsequence can't be greater than 10. We say that the first request in each subsequence has a position number 1, the second one has a position number 2, etc. So, in the sequence above, there are 3 fault requests with the position number 1, 2 requests with the position number 2, and 1 request with the position number 3. The position number $b(q)$ of a request q is used in the score calculation.

During testing, the results of request processing are stored in a text file with the following format. The number of lines are equal to the number of services K in the input. For each service, the appropriate line contains several integers separated by spaces: the service id d, the current wavelength $w_d \in [W]$, number of edges in the current path p_d and edges ids of the current path p_d. If you have not found a new path for a service, then set $w_d := 0$ and $p_d := \emptyset$.

3 Solution format

Your solution will look like this one:

```
#include "rerouting.h"
...

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

```
std::vector<Route> request(int r){
    std::vector<Route> currentRoutes; // Current answer state
    ...
    return currentRoutes;
}
```

All read/write operations are done by our testing system (grader). Your solution shouldn't use standard input and output streams! The grader will call `init()` and `request()` procedures from your solution. The procedure `init()` will be called only once for one test. It makes all the needed initialization to process the following requests. Then requests are followed one-by-one, each request r is passed in function `request` which returns `std::vector` of K elements: current routes for services after processing r.

The file `rerouting.h` contains following lines:

```
#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> E, std::vector<Service> D);

std::vector<Route> request(int r);
```

We've prepared for you a sample solution. You can find it in eJudge testing system in `samples.zip` along with the grader and `rerouting.h` to be compiled locally.

4 Scoring

1. There are three sets of tests: examples, open set and closed set. Examples are open and they are not counted in the scoring.

2. Open set and closed set are similar to each other in general, but different in details, since they were generated randomly. Open set contains 10 tests, closed set contains 50 tests. In the open set, tests are opened for all participants and can be found in `samples.zip`. Tests of closed set are not available to participants.

3. Open set is used for scoring during the online marathon. Participants can view their scores for each of 10 tests.

4. Open set is available in `Samples ZIP` in eJudge testing system. Along with the tests itself their graph vizualization is available.

5. Closed set is used for final scoring of the last success solutions of participants.

6. A test is a network and no more than a 100 requests.

7. The time limit for one test (one call of `init()` and 100 calls of `request()`) is 30 seconds. The testing of 10 tests may take up to 5 minutes.

8. Only one CPU core is used (multithreading has no sense). No more than 1 GB of RAM can be used. The number of submissions is not limited. You may make a new submission after the testing of your previous submission has been finished.

9. If the solution violates time limit, memory limit, or problem requirements, then it gets 0 score.

10. For a fault request q, the result of rerouting is scored with the number

$$C(q) = \lfloor 4^{10-b(q)} \cdot 100 \cdot X/K \rfloor,$$

where $b(q)$ is the position number of q ($1 \le b(q) \le 10$), X is the total number of unsuccessful services (services with $p_d = \emptyset$), K is the total number of services in the input.

11. The scores of all requests Q of one test t are summed up and are subtracted from the constant $M = 3 \cdot 10^7$:

$$C(t) = \max\left(1, M - \sum_{q \in Q} C(q)\right).$$

The total score for a submission is average of scores on all tests (except examples). The greater score, the better.

12. The last successful solution of participant is used in the final testing (scoring). It will be tested for closed set. The solution with the highest score wins. If two solutions have the same score, the fastest solution wins.

5 Examples

Here are listed simple examples of an input and output data.

5.1 Example 1

The network consists of 4 nodes and 5 edges. Three wavelentghs are available. Initial paths of 3 services are shown in the left part of Fig. 1.

The first request is the edge 1–2. The algorithm finds new path 1–3–4 with wavelengths 2 and 3 for both failed services (see the middle part of Fig. 1). The score is $4^{10-1} \cdot 100 \cdot 0/3 = 0$.

The second request is the edge 3–4. The algorithm finds new paths for 2 out of 3 failed services (see the right part of Fig. 1). Only 2 of 3 services can be rerouted, since the initial path of the blue service uses wavelength 1 in the edge 1–3 and the initial path of the green service uses wavelength 1 in the edge 2–4. The new path for the orange service uses resources of the initial path (wavelength 2 in the edge 2–4). The score is $4^{10-2} \cdot 100 \cdot 1/3 = 2\,184\,533$.

The 3rd request restores the initial state of the network. The score is 0.

The total score for this test is $3 \cdot 10^7 - (0 + 2\,184\,533 + 0) = 27\,815\,467$.

The **input** (see Fig. 1):

```
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
3
1 5 0
```

The output for the **first** request:

```
1 3 2 2 5
2 2 2 2 5
3 1 2 2 5
```

The score is 0.

The output for the **second** request:

```
1 0 0
2 2 3 2 3 4
3 3 3 2 3 4
```

The score: 2 184 533.

The output for the **third** request:

```
1 1 2 1 4
2 2 2 1 4
3 1 2 2 5
```

The total score: 27 815 467.

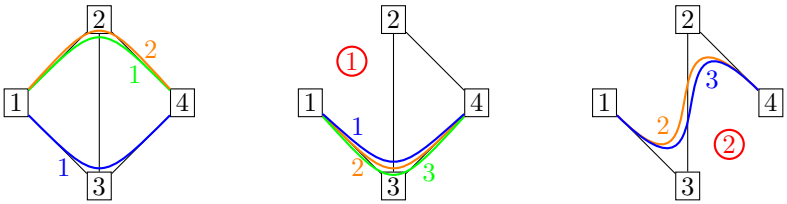

Figure 1: Example 1: the input and output for 2 fault requests. The initial wavelength for the green service is 1, for the orange service is 2, for the blue service is 1. The current wavelength for the green service is changed to 3 after the first request. After the second request, only 2 out of 3 services are routed.

5.2 Example 2

The network consists of 5 nodes and 6 edges. Two wavelentghs are available. Initial paths of 3 services are shown in the left part of Fig. 2.

The first request is the edge 3–4. The algorithm finds the new path 4–5–3 with wavelengths 1 for the green service (see the middle part of Fig. 2). All services have paths. The score is 0.

The second request is the edge 2–5 (see the right part of Fig. 2). The blue service can not be rerouted, since the wavelength 2 in the edge 1–3 is occupied (by the orange service) and the wavelength 1 in the edge 3–5 is occupied (by the green service). The score is $4^{10-2} \cdot 100 \cdot 1/3 = 2\,184\,533$.

The 3rd request restores the initial state of the network. The score is 0.

The 4th request is the edge 3–4 (see the left part of Fig. 3). The algorithm finds the new path 4–5–3 with wavelengths 1 for the green service. All services have paths. The score is 0.

The 5th request is the edge 1–3 (see the right part of Fig. 3). The algorithm finds the new path 1–2–5–3 with wavelengths 2 for the orange service. All services have paths. The score is 0.

The total score for this test is $3 \cdot 10^7 - (0 + 2\,184\,533 + 0 + 0 + 0) = 27\,815\,467$.

The **input** (see Fig. 2):

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

The output for the **first** request:

```
1 1 2 5 6
2 1 1 3
3 2 1 2
```

The score is 0.
The output for the **second** request:

```
1 1 2 5 6
2 0 0
3 2 1 2
```

The score: 2 184 533.
The output for the **third** request:

```
1 1 1 4
2 1 1 3
3 2 1 2
```

The score is 0.
The output for the **4th** request (Fig. 3):

```
1 1 2 5 6
2 1 1 3
3 2 1 2
```

The score is 0.
The output for the **5th** request:

```
1 1 2 5 6
2 1 1 3
3 2 3 1 3 5
```

The score is 0.
The total score: 27 815 467.

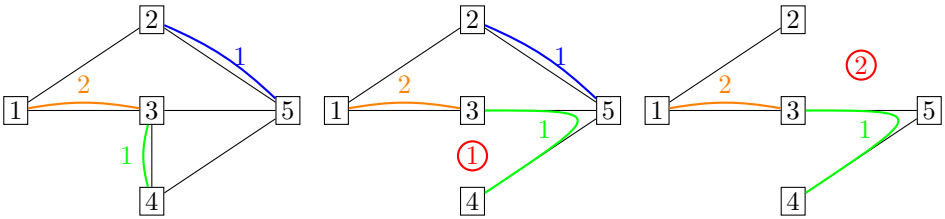

Figure 2: Example 2: the input and output for 2 fault requests in the first subsequence `4 3`.

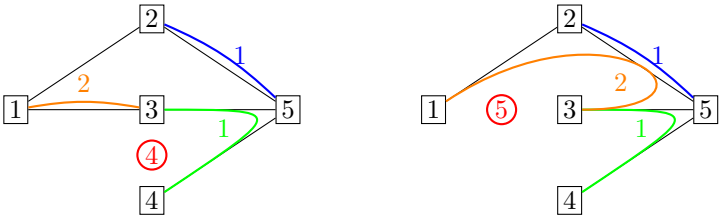

Figure 3: Example 2: the output for 4th and 5th requests.