NFA Threaded Router: Design, Documentation, and Scaling Studies

Author(s):
Ryan Jung
Henning S. Mortveit

NSSAC Technical Report: No. 2019-TBD

Status: *Approved for NSSAC Internal Release*

Contact:
Henning S. Mortveit (Henning.Mortveit@virginia.edu)

GitHub URL:
https://github.com/NSSAC/RE_Router

Network Systems Science and Advanced Computing
Biocomplexity Institute and Initiative
University of Virginia

NFA Threaded Router: Design, Documentation, and Scaling Studies

Ryan Jung[a] and Henning S. Mortveit[a,b,†*]

[a]Network Systems Science and Advanced Computing;
[b]Department of Engineering Science and Environment;
University of Virginia

October 30, 2019

Abstract

This technical report covers the threaded implementation of the regular expression (or NFA) constrained router of Jakob et al.

Previously built router using regular expression, was modified to be able to run parallel instances, so that multiple requests could be handled at once.

Introduction

The algorithm used in this software involved taking an existing NFA and a graph to produce an auxiliary graph. Using this, a Dijkstra's algorthm is used to traverse the graph, but only with the edges of the vertexes have the same edge label that follow with the NFA. This allows for not only traversal with only a specific set of nodes, but allows also for a number of patterns that the program can map out. [1].

ALGORITHM RE-CONSTRAINED-SHORT-PATHS

- *Input:* A Regular expression R, a directed weighted graph G, a source s, and a destination d
- 1. Construct an NFA
  2. Construct the NFA *M(G)* of G
  3. Construct *M(G) X M(R)*. The length of the edges in the product graph is chosen to be equal to the edges in the corresponding graph G
  4. Starting from state *(s_0, s)*, find a shortest path to the vertices *(f, d)*, where $f \in F$. Denote these paths by p_i, $l \leq i \leq w$. Also denote the cost of p_i by $w(p_i)$.
  5. $C^* := \min_{\text{pi}}$, $w(p_i)$; p^*: $w(p^*) = C^*$. (If p_* *is not uniquely determined, we choose an arbitrary one.*
- *Output:* The path p^* in G from$s$ to d of minimum length subject to the constraint that of $l(p) \in L(R)$

*† Corresponding author: Henning S. Mortveit;
email: henning.mortveit@virginia.edu;
github URL: https://github.com/NSSAC/RE_Router
NSSAC Technical Report: No. TR 2019–TBD

Report organization. Section 1 gives the software design of the solution Section 2 presents the methodology of the testing and Section 3 presents the data and possible explanations Section A providing complete user instructions as well as deploy constructions.

1 Design

The basis of the design takes from the original implementation. Quite simply, the focus is on the critical portion of the code base where:

- The NFA is created
- The trip requests are processed
- The program outputs all the processed request to a text file

Taking this, these implementations are stored in a callable method that can be stored in a thread. An additional method needed to be built that takes in the multiple thread requests and splits them up among however many threads the user decides to run. Alongside these changes, many legacy features were removed to help make the program more lightweight and simpler to understand. An option to allocate a certain number of cores was also added for testing purposes. The idea behind these changes is by allowing for threading and paralleling operations, the program can handle more requests more quickly and efficiently, making full use of a machine's capabilities.

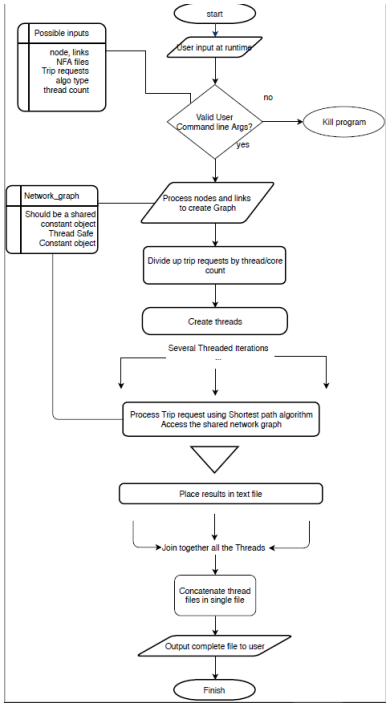
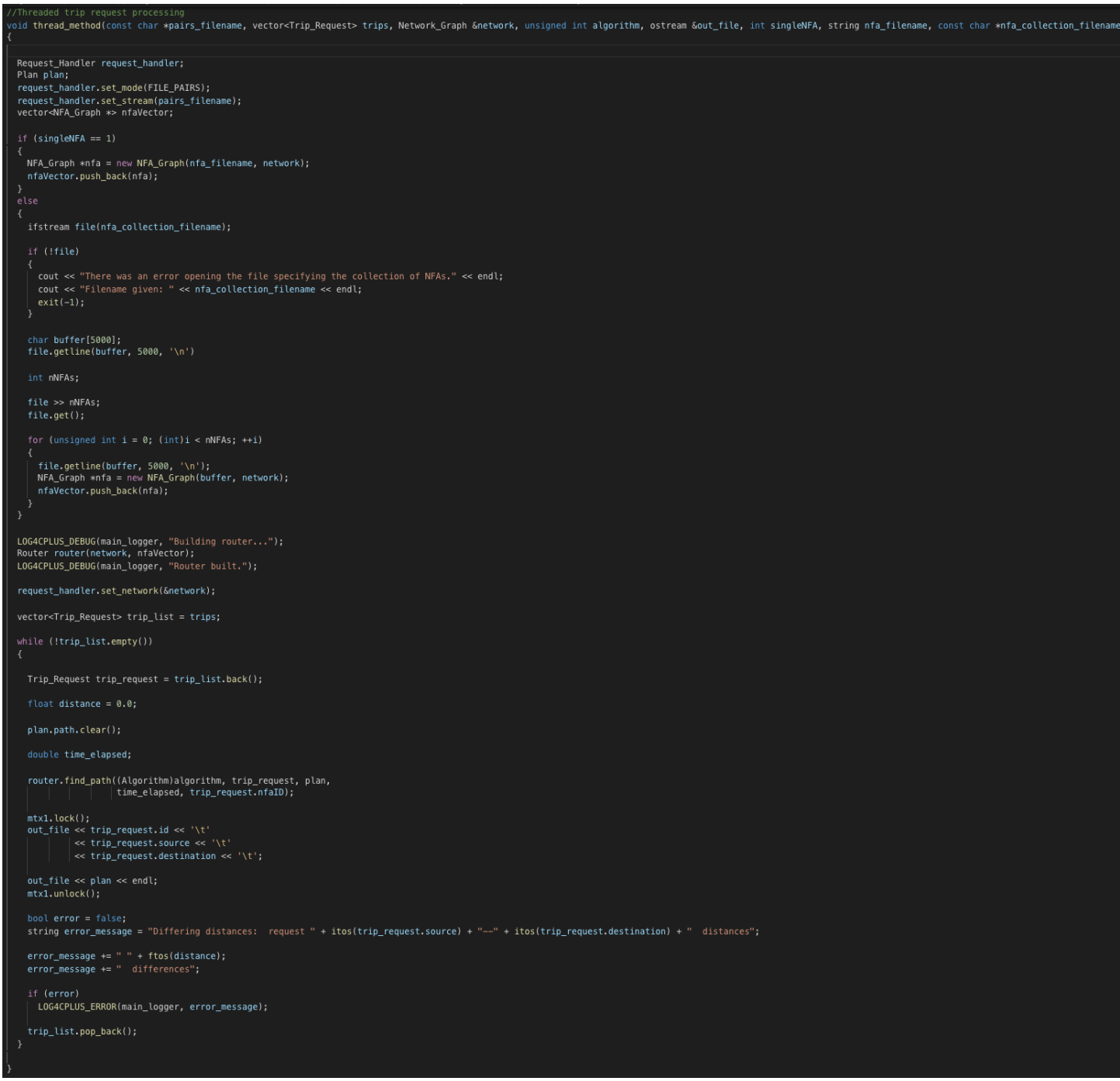


Figure 1: Design diagram

Once these ideas were firmly set out, the critical sections of the code are as follows.

Compared to the original codebase, it is quite similar, with much of the components being adpated from the original. Major differences include the addition of locks around file output to ensure thread safety.



```
//Threaded trip request processing
void thread_method(const char *pairs_filename, vector<Trip_Request> trips, Network_Graph &network, unsigned int algorithm, ostream &out_file, int singleNFA, string nfa_filename, const char *nfa_collection_filename
{

  Request_Handler request_handler;
  Plan plan;
  request_handler.set_mode(FILE_PAIRS);
  request_handler.set_stream(pairs_filename);
  vector<NFA_Graph *> nfaVector;

  if (singleNFA == 1)
  {
    NFA_Graph *nfa = new NFA_Graph(nfa_filename, network);
    nfaVector.push_back(nfa);
  }
  else
  {
    ifstream file(nfa_collection_filename);

    if (!file)
    {
      cout << "There was an error opening the file specifying the collection of NFAs." << endl;
      cout << "Filename given: " << nfa_collection_filename << endl;
      exit(-1);
    }

    char buffer[5000];
    file.getline(buffer, 5000, '\n')

    int nNFAs;

    file >> nNFAs;
    file.get();

    for (unsigned int i = 0; (int)i < nNFAs; ++i)
    {
      file.getline(buffer, 5000, '\n');
      NFA_Graph *nfa = new NFA_Graph(buffer, network);
      nfaVector.push_back(nfa);
    }
  }

  LOG4CPLUS_DEBUG(main_logger, "Building router...");
  Router router(network, nfaVector);
  LOG4CPLUS_DEBUG(main_logger, "Router built.");

  request_handler.set_network(&network);

  vector<Trip_Request> trip_list = trips;

  while (!trip_list.empty())
  {

    Trip_Request trip_request = trip_list.back();

    float distance = 0.0;

    plan.path.clear();

    double time_elapsed;

    router.find_path((Algorithm)algorithm, trip_request, plan,
                     time_elapsed, trip_request.nfaID);

    mtx1.lock();
    out_file << trip_request.id << '\t'
             << trip_request.source << '\t'
             << trip_request.destination << '\t';

    out_file << plan << endl;
    mtx1.unlock();

    bool error = false;
    string error_message = "Differing distances:  request " + itos(trip_request.source) + "--" + itos(trip_request.destination) + "  distances";

    error_message += " " + ftos(distance);
    error_message += "  differences";

    if (error)
      LOG4CPLUS_ERROR(main_logger, error_message);

    trip_list.pop_back();
  }

}
```

Figure 2: Design diagram

```
do
{
    trip_stream >> t_id >> src >> dest >> t0 >> nfaID;
    if (trip_stream.eof() || trip_stream.bad())
    {
        requests_finished = true;
        continue;
    }

    trip_request.id = t_id;
    trip_request.source = src;
    trip_request.destination = dest;
    trip_request.start_time = t0;
    trip_request.nfaID = nfaID;
    request_vector.push_back(trip_request);
    //cout<<"okay"<<endl;
} while (!finished());
int cores = (int)std::thread::hardware_concurrency();
if (core_num != 0 && core_num <= cores)
{
    cores = core_num;
}
int vec_size = 0;
int test = request_vector.size();
if (test < cores)
{
    vec_size = 1;
}
else
{
    vec_size = test / cores;
}
```

Figure 3: Thread Request.

```
int count = 0;
vector<Trip_Request> temp;
vector<vector<Trip_Request> > big_list;
while (!request_vector.empty())
{

    if (count >= vec_size)
    {
        count = 0;
        big_list.push_back(temp);
        temp.clear();
    }
    else
    {
        temp.push_back(request_vector.back());
        request_vector.pop_back();
        count++;
    }
}
if (!temp.empty())
{
    big_list.push_back(temp);
    temp.clear();
}
return big_list;
```

Figure 4: Thread Request cont.

A custom "Thread Request" method was also created in to work in conjunction with the Thread method. The function of this is to split up a single request across an appropriate number of threads (can be user generated or defaulted to the number of cores existing on the machine) so that the work can be split up among them.

2 Scaling Studies

Outline of work:

- Construct e.g. a Python tool that takes as argument the node file of a network, and an integer N, and that produces a complete trip request file containing N random trips. They will all have travel mode 0 which corresponds to automobile. It will likely be useful to have trip request files with 1,000, 10,000, and 100,000, requests. For testing, use the smaller request files; for the serious scaling studies, use the larger one.
- Create a timing framework that can time and report the execution of the router. It may be helpful to separately time initialization code such as construction of networks.
- Create a timing diagram giving time needed for computation as a function of the number of requested compute threads. Conclusions? Linear scaling? If not, why not? What happens if you request more threads than there are available cores? For each timing run, there may be fluctuations due to other computations running on your computer/Rivanna. It will likely be useful to at least conduct two runs per setting.
- If time permits, conduct a scaling study on Rivanna using multiple nodes.

3 Results

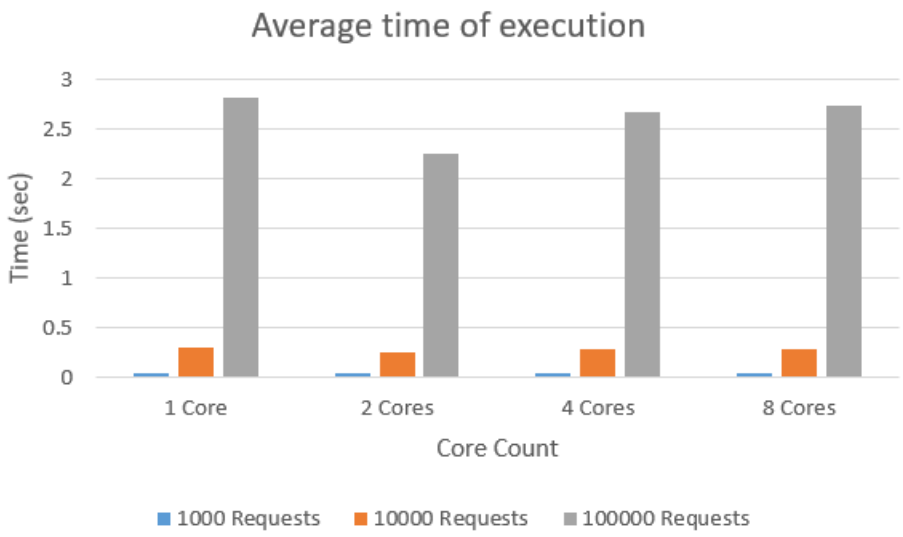


Figure 5: Design diagram

Interestingly, the results show that across the board utilizing only 2 cores would produce the best results. All these tests were performed on a local machine and the only explanation that could be thought of is that the OS is utilizing the other cores for other overhead operation which could cause a slowdown. Further testing on Rivanna with dedicated nodes and cores performing the tests seems like it would be necessary. This was not the expected result where it was initially hypothesized that the relationship between average time and core count would be logarithmic. Still using multiple cores is almost always faster than using one core (save for the test of 1000 trip requests).

References

[1] Chris Barret, Rico Jacob, and Madhav Marathe *Formal Language Constrained Path Problems* Society for Industrial and Applied Mathematics, 2000

A RE_Router User Documentation

```
./new_main -g <graph> -c <coords> -N <NFAFILE> -s <cores> -t <time> -f <pairs>

  -c <coords>: coordinates (vertex) file

  -g <graph>: pairs from file

  -N <NFAFile>:graph (edge) file

  -s <cores>: specifiying how many cores (and consequently threads) are used

  -t <time>: time of departure

 plans.txt: output file returning request path traversal

./trip_maker.py: prompts user for a number and creates that many trip requests

./super_script.sh:runs program with newly created pairs file from trip_maker.py with 4 cores

./testing_frame: prompts user for number of requests and number of cores.
   Runs 100 trials and returns the average time and all trial times to a .csv file

./data_collect.py: runs 3 tests of 1000, 10,000, and 100,000 requests. Returns 3
   .csv files with results
```

A.1 Input Files

The router uses the following input files:

- Network node file: The text file with specifications of node id and travel options
- Network link file: The text file that puts together the nodes
- Trip Request file: The text file that puts out specific travel routes