Matteo Romiti

Daniel Velez

John Stiben Perez Brand

01NQQOC - Operations research:

Theory and Applications to Networking

Greedy heuristic

In this lab, we implemented our algorithms in python language.

1.

The greedy heuristic algorithm that we created and implemented can be summarized with the following steps:

* from the traffic matrix (Uniform(0.5,1.5)), create a vector in which each element had 3 sub-elements: flow to be exchanged, source node, destination node
* sort this vector in decreasing order of flow
* starting from the first element of the sorted vector, create a link between the source and the destination of the current element if the source still has a transmitter available and the destination still has a receiver available.

The process ends when all the elements of the vector have been considered.

For the nodes that do not have a direct link to exchange traffic we must find a multi-hop path on the existing logical topology. Here we used the following routing algorithm:

* check if the source *s* has a neighbor n1 which has a direct link with the destination *d*
* if so, add the flow to the links *s*-*n1* and *n1*-*d*
* otherwise check if *n1* has a neighbor *n2* which has a direct link with the destination *d*
* if so, add the flow to the links *s*-*n1*, *n1*-*n2*and *n2*-*d*
* go on until the destination is found.

It may be possible that the number of nodes and transmitter and receivers, together with the topology, do not allow us to find all the paths between all the sources and destinations. If this happens, we have to restart the process.

We compared our greedy algorithm with a random one, in which the vector containing flow, source and destination is not ordered but is randomly shuffled. Then we follow the same procedure: start assigning links from the first element of the vector and route the remaining traffic looking for a path in all the available links.

2.

Here, we report the results of the simulations for different values of number of nodes *N* and number of transmitter and receivers Δ.

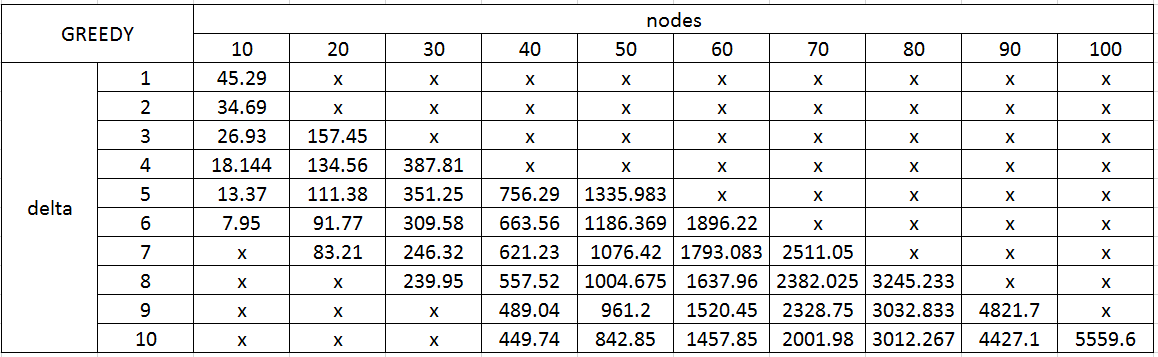


Figure – Greedy Algorithm

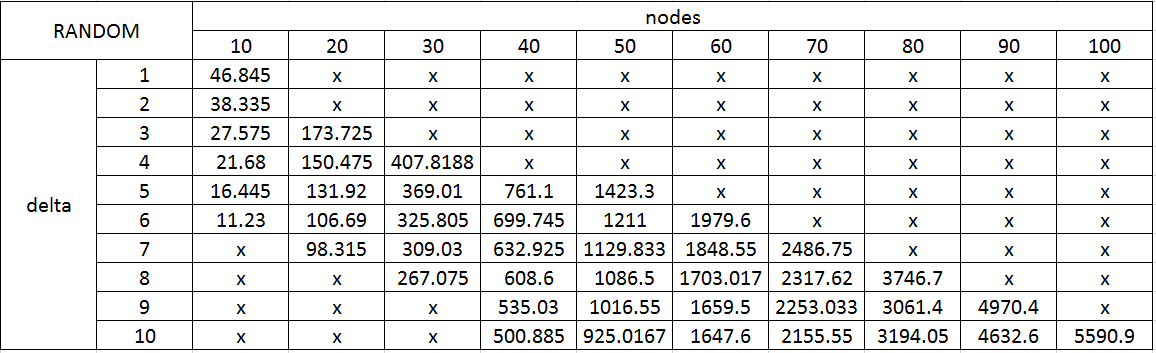


Figure – Random Algorithm

Cells containing “x” in the top-right half of the Figures 1 and 2 means that we were not able to find a topology allowing us to route all the traffic. Cells containing “x” in the bottom-left half means that we didn’t simulate that scenario was not of our interest (ratio between Δ and *N* was high). For each tested case, we repeated the simulation and computed the average fmax.

Comparing Figure 1 with Figure 2, we see that our greedy algorithm has a lower fmax for every tested case, giving us the right feedback and confirming that our implementation was correct. Also, keeping constant *N* and increasing Δ, we obtain a lower congestion. If we instead increase only *N*, we get more congestion as the exchanged traffic grows.

3.

We used the previous approach, but here we created two traffic matrices, high and low traffic, and the final traffic matrix had an element of the high traffic matrix with probability 0.1.

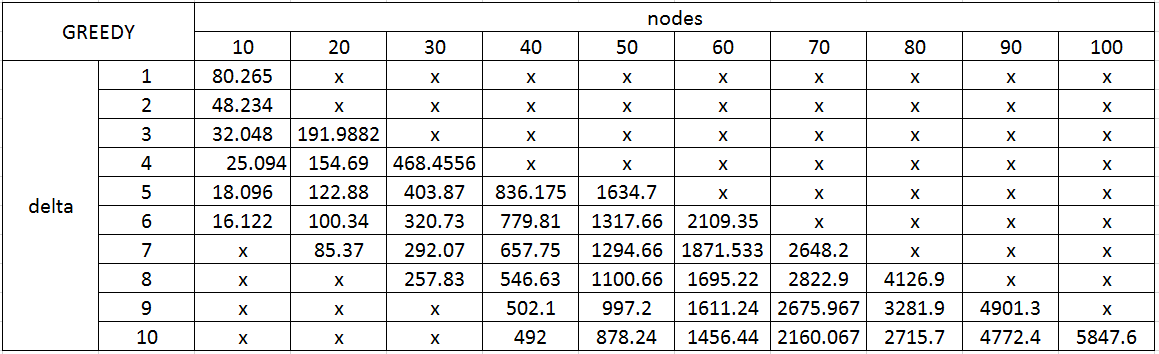


Figure - Greedy Algorithm with probability 0.1 of high traffic

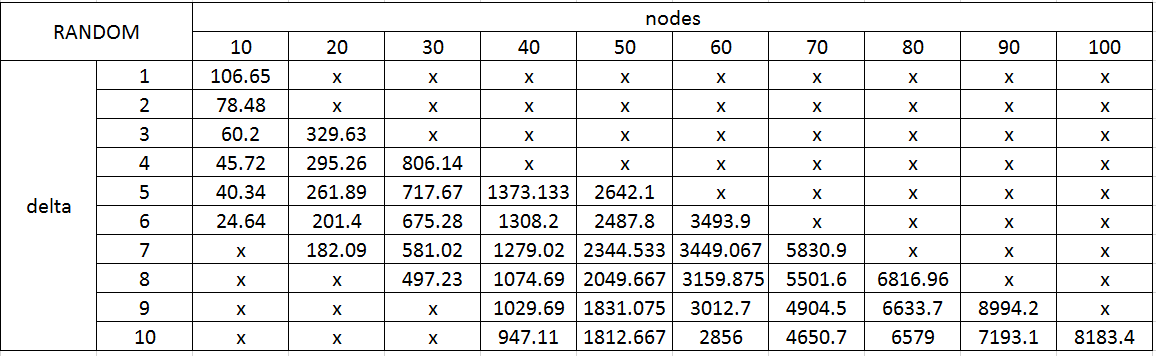


Figure - Random Algorithm with probability 0.1 of high traffic

Looking at Figure 3 and 4, the same observations as before can be made: lower fmax with the greedy algorithm, higher overall congestion if Δ decreases or *N* increases.

4.

The greedy heuristic algorithm that we created and implemented for the bidirectional Manhattan is the following:

* from the traffic matrix, compute the amount of traffic exchanged by each node, considering both its incoming and outgoing flows
* create a vector in which each element is made of two sub-elements: total traffic exchanged and node number
* sort this vector in decreasing order of total traffic exchanged
* starting from the first element of the ordered vector, find the place in the Manhattan that has the highest number of free neighboring places *p* and insert it there, if it has not been already inserted in the previous iterations
* find *p* nodes with which the inserted node has to exchange the highest amount of traffic, considering both incoming and outgoing flows and insert it next to it, if it was not already inserted in the previous iterations

The algorithm will stop as soon as all the nodes have been inserted.

In this way, it’s likely that flows with higher values will have short paths, hopefully just one hop.

To route the traffic, a source *s* checks whether the destination *d* is one, two, three or four hops away from it and then it sends the traffic through that path. This is done by comparing the indices (raw, column) of *s* and *d*.

In order to test our algorithm, we created a Manhattan in which the nodes are placed randomly and then we computed the fmax.

With 10000 simulations, we got an average value of fmax = 13.93326 in case of greedy algorithm and fmax = 14.17272 in case of random algorithm, which confirms that our implementation is a reasonable solution.

5.

In order to improve the performance of our solution, we implemented the simulated annealing algorithm. Starting from the Manhattan obtained at point 4, we randomly swap two nodes and re-compute the fmax. If it is smaller than the previous, we accept it, otherwise we accept it with a probability that decreases exponentially:

Here, we report the results repeating 10 times the simulation each time with a different traffic matrix, first with x [0,100], then with x [0,1000] and finally with x [0,10000].

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 100 iterations | | 1000 iterations | | 1000 iterations | |
| initial fmax | final fmax | initial fmax | final fmax | initial fmax | final fmax |
| 13.4 | 13.1 | 14.1 | 13.9 | 14.4 | 14 |
| 14 | 14.2 | 14.8 | 14.2 | 14.3 | 13.9 |
| 13.2 | 14.4 | 14.2 | 13.3 | 14.7 | 14.1 |
| 13.1 | 13.4 | 14.4 | 13.6 | 14.7 | 14 |
| 14 | 13.8 | 14.1 | 13.9 | 14 | 14.2 |
| 14.5 | 13.7 | 14.2 | 14.1 | 13.6 | 13.7 |
| 13.7 | 14.6 | 13.5 | 14.7 | 13.9 | 14.8 |
| 13.8 | 14.1 | 14.4 | 13.6 | 14.4 | 14.7 |
| 14 | 14 | 14.4 | 15.2 | 13.8 | 13.6 |
| 13.8 | 15 | 14.6 | 14.7 | 14.7 | 14 |

Figure – Results of the simulated annealing algorithm

Sometimes we got a lower fmax with our original nodes’ disposition, and some other times we had a better performance with the simulated annealing.