

# Time To Live (TTL): A Dynamic Routing Algorithm For Sensor Networks

CPE 400

<https://github.com/alexander-novo/CPE400>

Alexander Novotny

May 6, 2021

## Contents

<b>1. Motivation</b>	<b>1</b>
<b>2. Time to Live Algorithm</b>	<b>1</b>
<b>3. Implementation</b>	<b>2</b>
3.1. Simulator . . . . .	2
3.2. TTL Algorithm . . . . .	2
<b>4. Results</b>	<b>3</b>
<b>A. Network Diagrams</b>	<b>6</b>
<b>B. Configuration</b>	<b>6</b>

## 1. Motivation

Suppose you have a network of sensors. Each sensor is responsible for observing and reporting some data randomly with some average rate, but it might be expensive, unreliable, or otherwise infeasible to connect every sensor to an external gateway to record and/or process this data. Instead, a limited number of sensors (called ‘escape nodes’) are given the capability to connect to an external gateway, and every other sensor must route through these escape nodes. As well, each sensor has a limited capability to send data due to something such as battery life, and the data from each sensor is only useable if all data from other sensors is up to date (so as soon as a packet is lost or can’t be sent, all other packets are useless) - think of dual sensors used in tandem for object recognition on a car: if one sensor goes out, the other sensor’s data is useless. In this situation, established routing algorithms like OSPF won’t be able to effectively route packets due to their emphasis on link weights and broadcasting.

## 2. Time to Live Algorithm

The time to live algorithm relies on each node calculating its expected time to live (TTL). Since each node  $n$  observes data randomly with some rate  $\lambda_n$ , this amount of time between observations is a random variable  $X_n \sim \text{Exp}(\lambda_n)$  and the expected time between observations  $\mathbb{E}[X_n] = \frac{1}{\lambda_n}$ . Then the expected time to live is

$$\text{TTL}_n = \frac{e_n}{e_{\text{obs}}\lambda_n}, \quad (1)$$

where  $e_n$  is the amount of energy that node  $n$  currently has and  $e_{\text{obs}}$  is the energy it takes to transmit a single observation packet. However, this only takes into account immediately transmitting through an external gateway - if a node is surrounded by low energy neighbors, then it won’t be able to route its packet to an external gateway and its TTL should be similarly low. To account for this, we use a slightly modified metric

$$\text{TTL}'_n = \begin{cases} \infty, & n \text{ is an escape node} \\ \max_{i \in \mathcal{A}_n} \left\{ \frac{e'_i}{e_{\text{obs}}(\lambda'_i + \lambda_n)} \right\}, & \text{otherwise} \end{cases}, \quad (2)$$

where  $\mathcal{A}_n$  is the set of all adjacent nodes to  $n$  (nodes which are 1 hop away), and  $e'_i, \lambda'_i$  are the ‘limiting’ energy and observation rate of node  $i$  (more on these in a second).  $\text{TTL}'_n = \infty$  when  $n$  is an escape node since it is adjacent to the external gateway, whose TTL is  $\infty$ . We can take the maximum over all neighbors because we can simply choose to route to the neighbor which gives us the highest TTL. A node’s actual TTL is the minimum of eqs. (1) and (2) - a node must be able to survive sending the observations it itself generates, but it almost must have somewhere to send it. In the case that  $\text{TTL}_n$  is the minimum, then the bound energy  $e'_n = e_n$  and the bound rate  $\lambda'_n = \lambda_n$  since node  $n$ ’s time to live is ‘bound’ by its own parameters. In the other case, where  $\text{TTL}'_n$  is the minimum, then  $e'_n = e'_i$  and  $\lambda'_n = \lambda'_i + \lambda_n$ , since node  $n$ ’s time to live is bound by the same parameters that bind its neighbor.

Since the TTL for each escape node is bound by its own parameters (they never have to worry about having somewhere to send packets), we know that every other node’s TTL is eventually bound by some node’s parameters, and in the case where each node starts with the same amount of energy and has the same observation rate, every non-escape node’s TTL is bound by the parameters of an escape node, decreasing based on the number of hops from the nearest escape node. In this way, the initial routing setup is the same as OSPF. To update it, each node simply keeps track of a sliding window of packets it has received from its neighbors and uses this window to update  $\lambda_n$  to not only include the observations it itself is generating, but also those that it is receiving from its neighbors. Then it uses this new  $\lambda_n$  to calculate its TTL as in eqs. (1) and (2) and updates its neighbors.

The energy needed to send TTL updates is considered negligible compared to the energy needed to transmit observations (and in fact this can be enforced by ‘grouping’ observations before transmitting),



## 4. Results

A comparison of the algorithms using the simple network shown in the assignment can be found in listings 1 and 2. Note the the 50% increase in uptime - this is largely due to the nodes surrounding the escape node (nodes 1, 5, and 6) being more evenly utilized in the TTL algorithm.

Listing 1.: Output of the simulation after using OSPF on the network in fig. 2.

```
Collected 2315 observations in 66.4965s. Stopped after node 3 made an observation
↪ and attempted to route through node 1, which did not have enough energy to
↪ route it.
```

Node	Remaining energy	Observations made	Observations routed
0	1685.000000	307	2008
1	0.000000	330	670
2	329.000000	352	319
3	681.000000	319	0
4	669.000000	331	0
5	327.000000	342	331
6	665.000000	335	0

Listing 2.: Output of the simulation after using TTL on the network in fig. 2.

```
Collected 3402 observations in 98.9487s. Stopped after node 6 made an observation
↪ and attempted to route through node 6, which did not have enough energy to
↪ route it.
```

Node	Remaining energy	Observations made	Observations routed
0	598.000000	472	2930
1	27.000000	480	493
2	28.000000	515	457
3	543.000000	457	0
4	511.000000	489	0
5	43.000000	491	466
6	0.000000	499	501

Another comparison can be found in listings 3 and 4, this time with the larger network with 2 escape nodes depicted in fig. 3. Once again, we notice an almost 50% increase in uptime and much more even energy utilization.

Listing 3.: Output of the simulation after using OSPF on the network in fig. 3.

Collected 4605 observations in 65.9894s. Stopped after node 2 made an observation  
 ↳ and attempted to route through node 1, which did not have enough energy to  
 ↳ route it.

Node	Remaining energy	Observations made	Observations routed
0	1680.000000	323	1997
1	0.000000	330	670
2	329.000000	339	332
3	668.000000	332	0
4	671.000000	329	0
5	335.000000	336	329
6	1715.000000	342	1943
7	34.000000	345	621
8	379.000000	306	315
9	685.000000	315	0
10	675.000000	325	0
11	338.000000	337	325
12	665.000000	335	0
13	688.000000	312	0

Listing 4.: Output of the simulation after using TTL on the network in fig. 3.

Collected 6689 observations in 95.7407s. Stopped after node 2 made an observation  
 ↳ and attempted to route through node 13, which did not have enough energy to  
 ↳ route it.

Node	Remaining energy	Observations made	Observations routed
0	684.000000	489	2827
1	37.000000	481	482
2	11.000000	500	489
3	511.000000	489	0
4	532.000000	468	0
5	76.000000	488	436
6	627.000000	499	2874
7	50.000000	474	476
8	85.000000	440	475
9	525.000000	475	0
10	535.000000	465	0
11	71.000000	475	454
12	65.000000	464	471
13	0.000000	483	517

A final comparison can be found in listings 5 and 6, this time with a network with many links and 3 escape nodes. There is still a 50% increase in uptime and much more even energy utilization in the TTL algorithm's results than OSPF, leading to a conclusion of a succesful implementation.

Listing 5.: Output of the simulation after using OSPF on the network in fig. 4.

Collected 4949 observations in 66.4453s. Stopped after node 1 made an observation  
 ↳ and attempted to route through node 1, which did not have enough energy to  
 ↳ route it.

Node	Remaining energy	Observations made	Observations routed
0	1000.000000	333	1667
1	0.000000	348	652
2	333.000000	347	320
3	663.000000	337	0
4	684.000000	316	0
5	680.000000	320	0
6	356.000000	299	345
7	655.000000	345	0
8	678.000000	322	0
9	357.000000	321	322
10	1360.000000	322	1318
11	326.000000	341	333
12	667.000000	333	0
13	681.000000	319	0
14	1691.000000	347	962

Listing 6.: Output of the simulation after using TTL on the network in fig. 4.

Collected 7002 observations in 93.9109s. Stopped after node 3 made an observation  
 ↳ and attempted to route through node 1, which did not have enough energy to  
 ↳ route it.

Node	Remaining energy	Observations made	Observations routed
0	555.000000	477	1968
1	0.000000	486	514
2	32.000000	479	489
3	480.000000	491	29
4	564.000000	436	0
5	548.000000	452	0
6	0.000000	438	562
7	538.000000	462	0
8	532.000000	468	0
9	41.000000	439	520
10	548.000000	483	1969
11	31.000000	469	500
12	523.000000	477	0
13	346.000000	454	200
14	895.000000	492	1613

## A. Network Diagrams

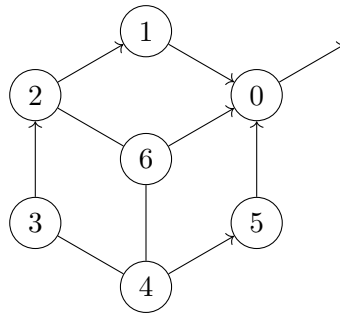


Figure 2: A simple example of a sensor network with one escape node. OSPF-chosen links are marked with arrow heads.

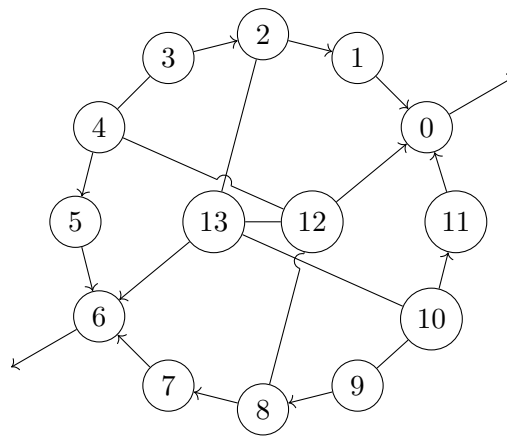


Figure 3: A more complex, but similar, example of a sensor network with 2 escape nodes. OSPF-chosen links are marked with arrow heads.

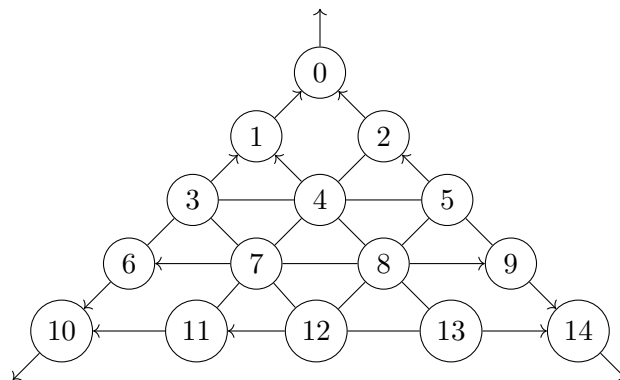


Figure 4: A large network with 3 escape nodes and many links. OSPF-chosen links are marked with arrow heads.

## B. Configuration

You can make your own sensor networks by simply providing a network configuration file. An example is given in listing 7. Each network has a set of nodes (with energy values, observation rates, and

whether or not it is an escape node) and a set of links. You can also define how much energy is used per transmitted observation.

Listing 7.: Configuration file for generating the network depicted in fig. 2.

```
{
  "energyPerTransmit": 1,
  "nodes": [
    {
      "edge": true,
      "energy": 4000,
      "observationRate": 5
    },
    {
      "edge": false,
      "energy": 1000,
      "observationRate": 5
    },
    {
      "edge": false,
      "energy": 1000,
      "observationRate": 5
    },
    {
      "edge": false,
      "energy": 1000,
      "observationRate": 5
    },
    {
      "edge": false,
      "energy": 1000,
      "observationRate": 5
    },
    {
      "edge": false,
      "energy": 1000,
      "observationRate": 5
    }
  ],
  "links": [
    [
      0,
      1
    ],
    [
      1,
      2
    ],
    [
```



```
    2,  
    3  
  ],  
  [  
    3,  
    4  
  ],  
  [  
    4,  
    5  
  ],  
  [  
    5,  
    0  
  ],  
  [  
    6,  
    0  
  ],  
  [  
    6,  
    2  
  ],  
  [  
    6,  
    4  
  ]  
]  
}
```