ROBUST BARRIER COVERAGE IN THE INTERNET OF THINGS

Matheus Centa and João Oliveira

February 2019

"The real voyage of discovery consists not in seeking new landscapes, but in having new eyes." Marcel Proust

1 Introduction

In the development of this project, we have implemented several codes and packages for having a clear and readable model to the problem in analysis. We have used Eclipse IDE and JAVA 8 programming language for all implementations.

Chapter 7 of the poly was used as our primal source of inspiration. The Dinitz-Edmonds Karp algorithms and the classes to model the flow network was implemented having this as a reference.

Question 1. Let $S = (\mathcal{N}, \mathrm{R}, \mathrm{c}, \mathcal{A})$ be a sensor network and M be the maximum number of node-disjoint paths in the coverage graph Γ. The algorithm 1 takes S as an input and outputs M node-disjoint paths in Γ.

The code is implemented in the file `NodeDisjointPaths.java`. The pseudo-code is in page 2.

Question 2. Let n be the number of nodes in Γ and m the number of edges. The time complexity of creating the coverage graph Γ and the auxiliary graph G is $O(n+m)$ and recovering S from S' has time complexity $O(m)$ (the number of edges in the paths is smaller than the total number of edges in G, which is $O(m)$). As a result, the time complexity is dominated by the run-time of the algorithm used to calculate the maximum flow in G. The algorithm chosen is the Dinitz-Edmonds-Karp (DEK) algorithm, which has time complexity $O(m^2n)$. Finally, the run-time of algorithm 1 is $O(m^2n)$. It is easy to see that the space complexity is $O(m+n)$.

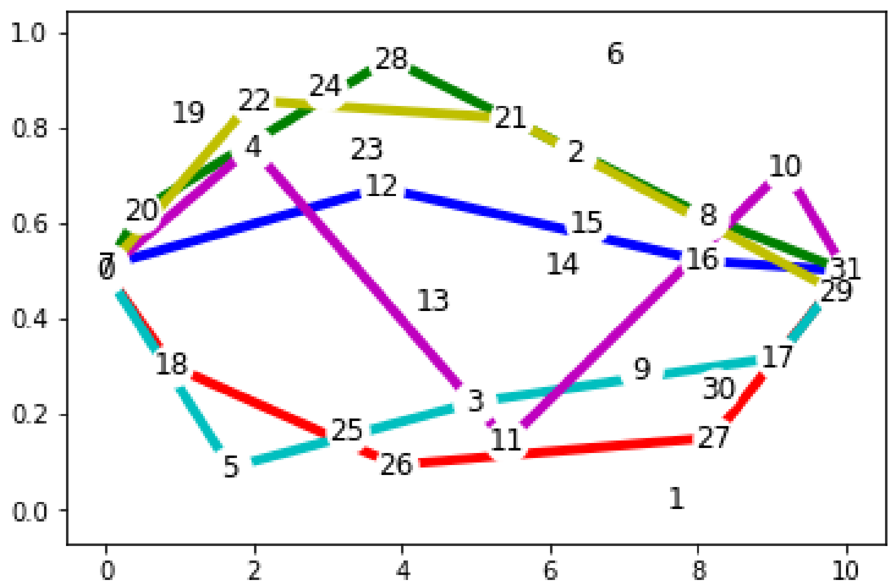
Question 3. By scheduling K node-disjoint paths simultaneously at each given time, we guarantee a robust coverage. For each interval of time with duration 1, schedule the sensors of K node-disjoint paths. This scheduling provides a lifetime of $\lfloor M/K \rfloor$, which is optimal.

Algorithm 1 Node-Disjoint Paths: an algorithm that takes as input a sensor network and outputs M node-disjoint paths from s to t.

procedure NodeDisjointPaths
Input: A sensor network $S = (\mathcal{N}, \mathrm{R}, \mathrm{c}, \mathcal{A})$
Output: A set of M node-disjoint paths in Γ from s to t.
Let $\Gamma = (W, F, c)$ be the coverage graph of S;
Let $G = (V, E, w)$ be a new directed graph with edge weights;
$V := V \cup \{s, t\}$
for each node $u \in W \setminus \{s, t\}$ **do:**
    $V := V \cup \{u_{IN}, u_{OUT}\}$;
    $F := F \cup \{(u_{IN}, u_{OUT})\}$;
    $w(u_{IN}, u_{OUT}) := 1$
for each edge $(u, v) \in F$ **do:**
    if $u \neq s$ and $v \neq t$ **then**:
        $F = F \cup \{(u_{OUT}, v_{IN})\}$;
    if $u = s$ **then**:
        $F = F \cup \{s, v_{IN})\}$;
    if $v = t$ **then**:
        $F = F \cup \{u_{OUT}, t)\}$;
    $w(u, v) = 1$
Let f be the maximum $s - t$ flow in G;
Let S' the set of all paths in G which only use edges $e \in F$ such that $f(e) > 0$;
Let S be a new set;
for each path $p' \in S'$ **do:**
    Create a new path p in Γ from p' by collapsing each two adjacent u_{IN}, u_{OUT} into the node $u \in W$;
    $S := S \cup \{p\}$;
return S;

Question 4. As specified before, the sensors are numbered by the reading order at the input, such that the first sensor to be read is numbered 1 and so on. The figure 1 presents all the node-disjoint paths in Γ for `sensornetwork0.doc`:

Figure 1: Node-Disjoint Paths in Γ.



The scheduling is given as follows: we first give an interval and then node-disjoint sensor paths that will be activated during that interval. The schedule for this problem is:

```
schedule:
( 0,  1): [7, 12, 16]
          [20, 28, 8]
          [18, 26, 27]
( 1,  2): [19, 13, 30]
          [5, 3, 17]
          [4, 11, 10]
```

and the network lifetime is 2.

The CPU time for this sensor network was 15 ms - it only counts the processing, not reading or printing.

Question 5. It is immediate the $F(p)$ is bounded by $\text{MaxFlowValue}(G)$ and, as a result, that Ψ tends to $-\infty$ as $p \to +\infty$. From the proprieties of Ψ from Lemma 3 and the fact that $\Psi(0) = 0$ and $r = \sup\{p \geq 0 \,|\, \Psi(p) = 0\}$ exists and is finite. Given that Ψ is also concave, r has the propriety that $\Psi(p) < 0, \forall p > r$.

Since an increase in p leads to either an increase in the capacities of G^p or no change, we can conclude that F is an increasing function. From these proprieties, we arrive at the following conclusion:

$$\max_{p \geq 0} \{F(p) \mid \Psi(p) \geq 0\} = F(r)$$

We will first the problem of finding r: given an upper bound on the value of r, we search for the root by reducing the upper bound, first decreasing over the multiples of 1, then 1/2, and so on until $1/|E|$. After an unsuccessful search using the multiples of $1/p$, we narrow the search space to $1/p$ which guarantees a $O(1)$ (either 1 or 2) calculations of Ψ when searching through the multiples of $1/(p+1)$.

The search stops when the root is contained within an interval of size $1/|E|^2$, because we can guarantee that the function is linear in this interval and the root can be found by linear interpolation. Note that using a binary search, for example, to reduce the search space to size $1/|E|^2$ and then interpolating won't yield correct results because we have no context of nearby breakpoints (e.g. the function will never be linear in an interval centered on a break-point).

The decrease upper bound approach (with upper bound MaxFlowValue$(G)/K$) was chosen because it tends to find correct values quicker than a similar approach in which we search the root by increasing an upper bound (with lower bound 0).

Algorithm 2 FindRoot: find the biggest root of Ψ

Input: A directed graph $G = (V, E, w)$ and an integer K.
Output: Biggest positive root of Ψ.
`upper` = MaxFlowValue(G) / K; # upper bound for the root
if $\Psi($`upper`$) = 0$ **then**
    return `upper`;
for each $r \in \{1, ..., |E|\}$ **do**:
    Let p be the smallest multiple of $1/r$ bigger than `upper`;
    while $\Phi(p) < 0$ and $p \geq 0$ **do**:
        $p := p$ - $1/r$;
    `upper` $:= p + 1/r$
    if $\Phi($`upper` $- 1/|E|^2) < 0$ **then**:
        Find the root by linear interpolation and **return** it;

Given the root r, the maximum flow f_r in G^r is the maximum K-route flow on G. Theorem 1 guarantees that the flow value is the maximum K-route flow value and the lemma 5 implies that this flow is a K-route flow (because the flow on each edge e is such that $0 \leq f_r(e) \leq r$ and the flow value is Kr by construction). As a result, the algorithm for finding the maximum K-route flow consisting of determining r and then returning the maximum flow for G^r. The pseudo-code is given by algorithm 3.

Algorithm 3 MaxKRouteFlow: given a directed graph G and an integer K with edge capacities, determine the maximum K-route flow from s to t in G.

1: **Input:** A directed graph $G = (V, E, s, t, w)$ and an integer K.
2: **Output:** A maximum K-route flow from s to t in G.
3: Let $r :=$ FindRoot(G, K);
4: **return** MaxFlow(G^p);

This algorithm is implemented on the `MaxKRouteFlow.java` file inside the algorithms package.

Question 6. Let m be the number of edges on the input graph and n be the number of nodes. After searching through the integers (multiples of 1/1), the algorithm calculates Ψ $O(m)$ times, as discussed on question 5. Let M_G be the initial upper bound chosen in the algorithm 2, then the initial search phase calculates Ψ at most $O(\lceil M_G/K \rceil)$ times. In the worst case we calculate Ψ at most $O(m + \lceil M_G/K \rceil)$ times. The complexity of the calculation of Φ(r) is dominated by the Dinitz-Edmods Karp algorithm $O(m^2 n)$, as the creation of the graph G^p takes $O(n + m)$ time. The total complexity of the algorithm is then $O\left(m^2 n(m + \lceil M_G/K \rceil)\right)$.

Question 7. Let G be the input directed graph with edge demands and capacities. To find a feasible flow we followed the strategy of creating a new graph $G' = (V', E', w')$, with a new source and sink nodes, and edges as detailed on Lemma 1. We can then recover a feasible flow from the maximum flow on G', as stated on Lemma 1.

The algorithm that realizes this task is `FeasibleFlow.java` and we can find it inside the algorithms package.

Question 8. Let $G = (V, E, s, t, c, d)$ be an directed graph with source s, target t, capacities $c : E \to \mathbb{R}$ and demands $d : E \to \mathbb{R}$. We propose a modification in the definition of the residual network G_f: we use the same definition for forward edges, but for backwards edges we propose the following definition:

$\diamond$ For each edge $e = (u, v)$ in G such that $f(e) > d(e)$, we have the backward edge (v, u) in G_f. Its capacity in G_f is $c'(v, u) = f(e) - d(e)$.

By starting the Dinitz-Edmonds Karp algorithm with a feasible flow (as given in Question 7) and using this new definition of residual network, we can show the the same `AugmentPath` function - without any changes - maintains the structure of the residual network and always produces viable flows. The algorithm with this modification terminates with a maximum flow for this flow-network with demands.

Question 9. For this algorithm, we use the lemma 4 to find elementary K-flows in the K-route flow f given. Then, the elementary K-flow that was found is subtracted from f and the process repeated until no more elementary K-flows are found.

To extract an elementary K-flow for a given K-route flow in a graph G, we create a maximum flow problem obtained by replacing the capacity on the edge (u, v) by $w'(u, v) = \lceil f(u, v)/v \rceil$ and imposing a demand of $d'(u, v) = \lfloor f(u, v)/v \rfloor$, as on the proof of the lemma 4. We also create a new sink t' and connect t to t' with an edge with capacity K. A maximum flow in this new problem is an elementary K-flow if the flow value is equal to K. This flow also is either 0 or 1 for each edge, because the MaxFlow algorithm always returns integer flows for integer capacities and demands.

By transferring the paths of this elementary K-flow to the original graph and assigning its value to the bottleneck (the minimum flow on all edges on this flow), we have extracted a elementary K-flow and its value.

The algorithm 4 gives the pseudo-code for this approach.

Algorithm 4 KRouteDecomposition: decomposes a K-route flow into elementary K-flows.

Input: An integer K and a K-route flow f on the graph $G = (V, E, s, t, c)$.
Output: A set of elementary K-flows.
Let S be an empty set;
while the last loop successfully extracted a elementary K-flow **do**
    Let G' be a new directed graph with edge demands d' and capacities w';
    for each edge $e \in E$ **do**
        $w'(e) := \lceil f(e)/v \rceil$
        $d'(e) := \lfloor f(e)/v \rfloor$
    Add a new sink t' to G' and connect t to t' with an edge with capacity K and demand 0;
    Let $f' :=$ MaxFlow(G');
    if the value of f' is less than K **then**
        Exit the loop; # *a elementary K-flow was not extracted*
    Remove the edge $\{t, t'\}$ from f';
    Assign the value of the flow f' to the bottleneck of f on its edges;
    Add f' (and its value) to S;
    Subtract f' from f;
return S;

An implementation of this algorithm can be found in `algorithms` package, on the file `KRouteDecomposition.java`.

Question 10. Let n be the number of nodes on the G, m be the number of edges in G and m_f be the number of edges with positive flow on the given K-route flow f.

For every iteration of the while loop in algorithm 4, at least one edge which had positive flow becomes an edge with no flow (because we subtract the elementary K-flow from f and its value is the bottleneck of f on its edges the edges that cause this bottleneck will then have flow equals to 0). The work done inside the while loop is dominated by the Dinitz-Edmonds Karp algorithm, which has run-time $O(m^2 n)$. We can then assert that the run-time of this algorithm is $O(m_f \times m^2 n)$ or $O(m^3 n)$ (because $m_f \leq m$). As usual, the space-complexity is

$O(m+n)$.

Question 11. We'll adapt the approach of the question 1 with some small adjustments: we will create an auxiliary graph G and

- For each node u of Γ that is not s or t, we'll split the node into two nodes u_{IN} and u_{OUT} and connect them with an edge with capacity $w(u_{IN}, u_{OUT}) = c(u)$ (the lifetime of the sensor u);

- For each edge (u, v) in Γ, create (u_{OUT}, v_{IN}) and (v_{OUT}, u_{IN}) in the directed graph G, except for the edges parting from s or going into t for, which we only create the edge in one direction. All these edges have capacities set to $+\infty$ (or any values that doesn't restrict flow).

An elementary K-flow f_e in G represents K node-disjoint paths that provide robust K-coverage and can be scheduled for value(f_e) units of time. To maximize the network lifetime we must then seek a maximum K-route flow f in G. We can provide a schedule by decomposing f into elementary K-flows. The pseudo-code is given on algorithm 5.

Algorithm 5 HomogeneusSolver: returns an optimal scheduling for a sensor network with heterogeneous sensor lifetimes.

```
Input: A sensor network S = (𝒩, R, c, K, 𝒜).
Output: S: A scheduling of sensors which provides optimal network lifetime.
Let Γ = (W, F, c) be the coverage graph of S;
Let G = (V, E, w) be a new directed graph with edge weights;
V := V ∪ {s, t}
for each node u ∈ W \ {s, t} do
    V := V ∪ {u_IN, u_OUT};
    F := F ∪ {(u_IN, u_OUT)};
    w(u_IN, u_OUT) := c(u)
for each edge (u, v) ∈ F do
    if u ≠ s and v ≠ t then:
        F = F ∪ {(u_OUT, v_IN)};
    if u = s then:
        F = F ∪ {s, v_IN)};
    if v = t then:
        F = F ∪ {u_OUT, t)};
    w(u, v) = +∞
Let f := MaxKRouteFlow(G, K);
Let S = KRouteDecomposition(f, K, G);
for each elementary K flow f_e in S do
    Schedule the K node-disjoint paths in f_e for value(f_e) units of times;
```

An implementation of this algorithm can be found on the `sensor_network` package in the `HeterogeneousSolver.java`.

Question 12. The resulting schedule for the file `sensornetwork1.doc` is:

```
schedule:
( 0,  1): [20, 24, 14, 17]
          [18, 23, 2, 10]
          [5, 3, 30]
( 1,  3): [7, 4, 3, 30]
          [18, 24, 14, 17]
          [19, 23, 2, 10]
( 3,  4): [20, 4, 3, 30]
          [18, 24, 14, 17]
          [19, 23, 2, 10]
( 4,  8): [20, 4, 3, 30]
          [18, 25, 14, 17]
          [19, 23, 2, 10]
( 8,  9): [20, 4, 3, 30]
          [18, 25, 14, 17]
          [19, 12, 2, 10]
```

In the output, first the interval is given (e.g. the interval from time 0 to time 1 in the first line) and then the list of node-disjoint paths to be turned on for the duration of this interval. The network lifetime is 9 and the processing time for this file was 56 ms.