Optimal NIDS Sensor Placement

Joshua (Alex) Rinaldi
AFIT/ENG CSCE 686 Advanced Algorithms
Final Project

Introduction

Inevitably one issue that Cyber Defense teams encounter is determining where to place sensors on a network when setting up monitoring tools. Most of these teams typically have a limited number of sensors available to place on the network, and they need to ensure that they are monitoring the most likely route that an attacker might take from an edge node to a target machine on the network. This is the problem I aim to solve with the algorithm proposed below; given a knowledge of network topology, along with vulnerabilities in the machines on the network, produce a list of connections on the network to monitor that maximizes the likelihood of detecting an adversary moving laterally through the network.

Problem Domain Requirements Specification

Input Domain D_i:
For this problem, we will need to supply network information. This will be represented as a graph:
$G\left(V, E, N_E, N_T\right)$
Where:
- V is the set of all vertices in the graph (e.g., the machines in the network)
- E is the set of edges connected those vertices (e.g., the communications pathways flowing between machines)
- N_E is the set of edge nodes in the network, which are most likely to provide an entry point into the network for an attacker
- N_T is the set of target nodes in the network, the machines which are critical to the network and are in need of protecting from the outside world.
- Every edge, $e \in E$, will have some weight w such that $w\left(a, b\right)$ will be equal to the number of vulnerabilities present on machine b.
    - If $w(*, b) = 1$, it means that the node $b \in V$ does not have any known vulnerabilities, however it is still given a weight of 1 to account for the potential of a zero day exploit.

Additionally, we will need a number n of the available sensors that we can place on the network.

Output Domain D_O:
The output of this problem should be a set S of edges would should be monitored in order to maximize the likelihood of detecting an intrusion in the network.

Input Conditions:
- $\left(N_E, N_T\right) \in V$
    - the target and edge nodes should exist in the list of vertices in the graph.

Output Conditions:

- $||S|| = n$
  - The number of edges in the solution set S should be equal to the number of sensors that are available
- $(\forall e \in S) \subset (\forall e \in P)$
  - for some set of paths P which could be taken from any N_E to any N_T, all edges e in the solution set S should appear in the paths contained in P
- Edges $e \in S$ should come from the most likely paths $p \in P$

Discussion on Complexity

This problem is one that can be viewed as a combination of two other problems: the K-Cut problem, which is NP-Hard under the correct circumstances, and the Stable Paths Problem, which is NP-Complete.

Per [3] the Multi-Cut Problem is a problem that is able to be solved in polynomial time. However, certain variations of the problem are considered to be in NP-Hard. One such variation is the K-Cut problem, where we wish to split the graph into K separate components. Ordinarily this problem is also solvable in polynomial time, however, when the number of cuts, K, that need to be made is included as a part of the input, the problem becomes NP-Hard.

When looking at the problem presented in this paper, it can be said that finding the locations on the network to place sensors is analogous to finding where to cut the graph in order to separate all $v \in N_E$ from $v \in N_T$. Because the number of edge nodes, $v \in N_E$, will determine the number of possible paths to a target node, $v \in N_T$, it will also be a factor in the number of cuts that need to be made, therefore, the condition that K is a part of the input is met, making this specific instance of the K-Cut problem NP-Hard.

The other portion of the problem presented in this paper is that of finding the most likely path an attacker would take to travel from $v \in N_E$ to $v \in N_T$. One thing to keep in mind is that an attacker would likely be moving to those machines that have a great number of known vulnerabilities, and will not likely be moving into those machines that do not have any known vulnerabilities, e.g. $(v \in V) \ni w(*, v) = 1$. Per [5], a common technique used when hacking a computer is to use a machine as a "pivot" where the attacker will move from one machine on the network they have exploited into another. Because the machines that the hacker has exploited might normally talk to one another, this can help any malicious traffic to be hidden amongst traffic that is considered legitimate.

Because each machine that a hacker uses as a pivot can be viewed as sort of router, this brings us around to the Stable Paths Problem, which is a problem proposed in [2] to the best path through to route data across the internet. Per [4], the Stable Paths Problem, and it's variants will always be in NP-Complete.

Therefore, because of the combined complexity of these two problems, we defer to the higher of the two complexity classes and say that the problem presented intros paper exists in NP-Hard space.

Problem Domain/Algorithm Domain Integration Specification

Initialization
After the network graph has been ingested, it will be stored as a list of structs, the specifics of which will be discussed in further detail later in this paper. The vertices contained within N_E will be placed inside of a separate data structure.

Next State Generator
The next vertex that will be visited will come from the input domain. This vertex will specifically be chosen from the neighbors of the current vertex.

Selection Function
When looking at which neighboring vertex to pivot to next, we will consider a few factors:

- The vulnerability score of that particular neighbor, v
- The number of unexplored vertices v has as neighbors
- The vulnerability scores of the unexplored neighbors of v
- Whether the neighboring vertex is in the path to the current vertex

These will be fed as input into a likelihood function, and the neighboring vertexes will be explored in the order of their likelihood.

Feasibility Function
A neighboring vertex will only be considered to be a feasible next hop if it is has a non-zero likelihood.

Objective Function
This function will check to see if the current path ends on a target node, if it does, the overall probability of an adversary traversing the path will be determined and the path will be added to the set of viable paths for the current starting vertex of the path (which will be an edge node). This updated set of viable paths will then be used to update the set of edges that will reliably provide visibility on traffic coming from an edge node to a target node.

Delay Termination
Termination of the algorithm will be delayed in order to ensure that all potential paths from all edge nodes to any of the target nodes are considered. Only after all of these potential paths have been explored, and the solution set of edges to monitor has been fully updated will the algorithm be terminated and a solution returned.

Algorithm Domain Design Specification Refinement

Initialization and Data Structures
When the network graph is ingested, each vertex will be stored in a struct. This structs will be stored in a list V and contain:

- v: Vertex Name
- λ: Vulnerability Score
    - This value is actually the weight of all edges $e(*, v)$ where v is the current vertex. Because $w(*, v)$ will be the same for all edges terminating at v, these edge weights will be stored as a part of v for the sake of space efficiency.
- A_v: Neighboring Vertexes & their Vulnerability scores
    - This will be stored as key-value pairs, and used to help determine the most stable paths

Next State Generator

$\left(D_i\right) \to v_{next} \in A_{v_{cur}}$

From our input domain we will pull some vertex v_{next} to pivot to next from the list of neighbors, A, of current vertex v_{cur}.

Selection Function

When looking at which neighbors of the current vertex v_{cur} to pivot to next, the following likelihood value is calculated for each neighbor is considered:

$$L\left(v_{next} \in A_{v_{cur}}\right) = \varphi_{v_{next}}\lambda_{v_{next}}\left(\frac{\sum_{i=1}^{A_{v_{next}}} \lambda_i\varphi_i}{\sum_{i=1}^{A_{v_{next}}} i\varphi_i}\right)$$

Where:

- v_{next} is the neighboring vertex being considered
- $A_{v_{cur}}$ is the matrix of neighbors of v_{cur}
- $A_{v_{next}}$ is the matrix of neighbors of v_{next}
- $\varphi = \begin{cases} 1 & \text{if } v \text{ is in current path} \\ 0 & \text{if } v \text{ is not in current path} \end{cases}$
- λ is the vulnerability score of the vertex in question

Neighbors of v_{cur}, will be ranked in order of largest to smallest likelihood, $L\left(v_{next} \in A_{v_{cur}}\right)$, and then popped from that list in order for consideration by the feasibility function.

Feasibility Function

Whether a particular neighboring vertex, v_{next} is a feasible next hop will be determined by:

$$\text{bool}\left(L\left(v_{next}\right) > 0\right)$$

If true, then v_{next} will be appended to current path P. P will then be passed to the Objective Function for further evaluation

Objective Function

Whether a path P is feasible will be determined by:

$$\text{bool}\left(P_{last} \in N_T\right)$$

Where P_{last} is the final vertex in path P. If true, then P is fulfills the objective function, P is then evaluated according to:

$$\Lambda_P = \frac{\sum_{i=1}^{P} \lambda_i}{||P||^3}$$

Where:

- Λ_P is the path score of some path P
- $\sum_{i=1}^{P} \lambda_i$ is the sum of vulnerability scores of all vertices $v \in P$
- $||P||$ is the magnitude of (number of vertices in) P. This value is cubed to add increased emphasis on shorter paths.

Finally, each edge $e \in P$ is added to the solution set F according to:

$$(\forall e \in P) \rightarrow \begin{cases} \Delta_e + \Lambda_P & \text{if } e \in F \\ F = F \cup e, \Delta_e = \Lambda_P & \text{if } e \notin F \end{cases}$$

Where:

- Δ_e is the final worthiness of an edge

F is then sorted based on Δ_e, with the larger values being placed at the front of the list.

Delay Termination

Once all paths have been explored, return the first n edges from F to D_O:

$$\left(F_{0 \ldots n}\right) = S \rightarrow D_O$$

Functional Algorithm Specification

For this paper, three algorithm implementations are designed and implemented. All code and pseudocode are included with this document.

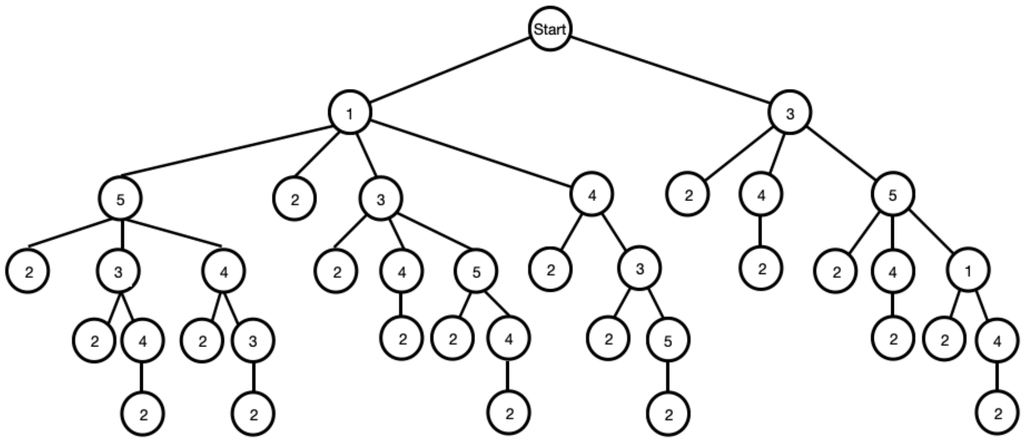
Deterministic Algorithm Specification

What follows is the pseudocode specification for the deterministic implementation of this algorithm.

```
###SET OF CANDIDATES
Input G : Graph ; Verts : set of objects ; NumSens : number ;
VertsEdge : set objects ; VertsTarget : set of objects ;
###OTHER INTERMEDIATE AND OUTPUT SETS
P : set of objects ; F : set of objects ; Output S : set of objects
|    ###INITIALIZATION
|    F = empty; S = empty
|    Main Loop
|    |   P = empty
|    |   ###DELAY TERMINATION <- until all v from vertsEdge explored
|    |   Choose object v from VertsEdge, append to P
|    |   Find_Paths(P, Verts, VertsTarget, F)
|    Endloop
|    For i in range (1..NumSens)
|    |    S.append(F[i])
|    Endfor
|    Return S
|
|    Func Find_Paths(P, Verts, VertsTarget, F)
|    |    v_cur = P.last
|    |    ###OBJECTIVE FUNCTION
|    |    If v_cur in VertsTarget
|    |    |    evaluate P
|    |    |    add edges in P to F
|    |    Else
|    |    |    ###SELECTION FUNCTION
|    |    |    find likelihood of neighbors of v_cur
|    |    |    sort neighbors by likelihood
|    |    |    For v in neighbors
|    |    |    |    ###FEASIBILITY FUNCTION
|    |    |    |    If likelihood(v) > 0
|    |    |    |    |    ###NEXT STATE GENERATOR
|    |    |    |    |    Find_Paths(P+v, Verts, VertsTarget, F)
|    |    |    |    Endif
|    |    |    Endfor
|    |    Endif Endif
|    Endfunc
```

Deterministic Search Tree Diagram

This is a tree diagram derived from the *verySmallGraph.txt* file.



Stochastic Algorithm Specification

The stochastic implementation for this problem will be based on Karger's Algorithm [1]. In Karger's Algorithm, edges are chosen at random, and the end vertices of that edge are combined. The chosen edge is removed, thereby ensuring that no self edges exist, however, multi-edges are still permitted. This process is repeated until there are only two vertices left in the graph, and the edges connecting those two are considered to be the min cut. The strength of Karger's algorithm comes from the fact that it is statistically likely to produce the minimum edge cut of the graph, it does not, however, provide any guarantees that the resulting set of edges is in fact the true minimum cut of the graph.

The stochastic algorithm for this problem will use this concept of randomly chosen edges. Each candidate will contain a binary gene string representing the edges to cut. Whether a candidate is feasible will be dependent upon whether it successfully divides the graph into two groups. Because it is possible that with the sensor placements, that vertices in N_T could end up on the same side of the cut as vertices in N_E, it will not be required that $\forall v \in N_T$ be completely cut from $\forall v \in N_E$, however, the strength of the divide between these two sets will be considered in the fitness function. The fitness function will also take into consideration the weight of the edges that are cut. Finally, the candidate population will be kept at a steady size, and which members of the candidate population are allowed to remain in the population will be determined by an elitism based selection function.

Initialization

After the graph is ingested into the algorithm, edges will be stored as structs, containing the following information:

- *id*: numerical value
- *source:* source vertex
- *dest*: destination vertex
- *weight*: vulnerability score of destination vertex

These edges will then be placed into set E, where they will be able to be referenced by their ID. Next, the candidate population, C, will be randomly seeded with some number of candidates, c. Each candidate will contain the following information:

- *gene*: binary string, each position in the string corresponding to one of the edges
  - $\text{gene} = \begin{cases} 1 & \text{if } e \in c \\ 0 & \text{if } e \notin c \end{cases}$
- *fitness*: number, the overall fitness score of that candidate
- *feasible*: bool, whether the candidate is a feasible solution
  - for all candidates $c \in C$ this value should be true, it will be used to determine if new candidates are in need of repair
- *reduction*: the reduced graph according the edges to include
  - This will be generated using the same process outlined by Karger's algorithm in [1], however, instead of randomly choosing edges to remove, edges removed will be determined by those edges with a 0 value in the gene string

Evaluation Function

This function will be used to determine the fitness of a particular candidate. It is known that for a particular candidate, $c \in C$, the network graph will be split into two separate halves. Each half of the graph may contain and arbitrary number of $v \in N_E$ as well as an arbitrary number of $v \in N_T$. For one of the two sides of the graph, we find:

$$\delta_E = \frac{v \in N_E}{||N_E||}$$

Which is the percentage of edge vertices on that side of the graph. We also find:

$$\delta_T = \frac{v \in N_T}{||N_T||}$$

Which is the percentage of target vertices on that side of the graph.

Given that these values are percentages, it can then be said that:

$$\left(\delta_E, \delta_T\right) = \begin{cases} 1 & \text{if all edge/target vertices are on that side} \\ 0 & \text{if no edge/target vertices are on that side} \\ 0 < \delta < 1 & \text{if some other number are on that side} \end{cases}$$

Therefore, we wish to maximize the value:

$$\vartheta = \left|\delta_E - \delta_T\right|$$

As this value will be closer to 1 if the percentages of edge and target vertices present on one side of the graph differ greatly (which is desired), and will be closer to zero if these percentages are closer together. Additionally, we only need to calculate this value for one side of the graph, as the respective δ are percentages, it is true that:

$\delta_{E_L} = 1 - \delta_{E_R}$ and $\delta_{T_L} = 1 - \delta_{T_R}$

Where 'L' and 'R' represent the left and right side of the graph respectively. Through algebra, it is shown that:

$$\left|\delta_{E_R} - \delta_{T_R}\right| = \left|\delta_{T_L} - \delta_{E_L}\right|$$

It is also true that for any value x and y, $\left|x - y\right| = \left|y - x\right|$, so therefore, we need only look at one side of the graph to determine the value of ϑ for that candidate. Finally, this gives the fitness function:

$$\text{fitness}(c \in C) = \frac{\vartheta}{||c||} = \frac{\left|\delta_E - \delta_T\right|}{||c||}$$

Where $||c||$ is the magnitude of the candidate, or in other words, the number of edges included in the candidate. This value is important, because due to the limited number of sensors available on the network, those candidates that effectively split N_E and N_T using a smaller number of edges are more desirable.

<u>Selection Function</u>

The number of candidates, $||C||$, will be maintained at a constant value of 50. Therefore, if bool $\left(||C|| > 50\right)$ = true, then C will be reduced by removing the $||C|| - 50$ least fit candidates from the population.

Crossover Function
In order to generate new members of the population, at each iteration of the algorithm,10 $c \in C$ will be chosen at random to be bred, generating 5 offspring. For the generation of the offspring, it will be assumed that present between the two parents would still be present after the performance of the Karger algorithm in a deterministic approach, therefore, the base gene for the child will be created from a bitwise OR operation over the parent genes:

$c_{child} = c_{p1}$ OR c_{p2}
From here, the child gene will be passed to the mutation function to introduce some amount of randomness into the population.

Mutation Function
Whether a certain allele in a gene is mutated will be up to a probability. This probability, $\mu(a)$, will be based on the difference in the average magnitude of the parent genes, and the magnitude of the child gene. This probability will be given by:

$$\mu(a) = 1 - \frac{||c_{child}||}{0.5 * \left(||c_{p1}|| + ||c_{p2}|| \right)}$$

From here, the gene string will be iterated through, and for each allele, a random number, x, will be generated, and this and the probability of mutation will be passed to the mutation function:

if bool $\left(x < \mu(a)\right)$: flip allele a
After the child's gene has been mutated, it should be passed to the feasibility function for validation.

Feasibility Function
This function is intended to determine whether a certain candidate $c \in C$ successfully splits the graph in half. It is determined with:
bool $\left(||v|| \in G_R = 2 \right)$
Where:
- G_R is the reduction graph for c
- $||v|| \in G_R$ is the number of vertices in the graph

If this value is true, then c is considered feasible and added to the population. If the result of this function is false, it means that c is in need of repair.

Repair Function
If some $c \in C$ is in need of repair, it means that it contains too many edges to effectively split the graph in two. Therefore, for this function we will generate a random number, and if the allele in the position of the gene is a 1, it will be flipped to a 0, otherwise, the number will be incremented until the first non-zero allele is found, and then it will be flipped to a zero.

Once this has been performed, c will again be checked for feasibility.

Solution Function

Once a predetermined number of generations has been created, the fittest $c \in C$ will be chosen as the solution. From there, the edges included in the solution set will be evaluated for their respective likelihoods. This function will be similar to the likelihood function defined in the next state generator of the deterministic approach:

$$L\left(e \in c\right) = \lambda_{v_{dest}} \left(\frac{\sum_{i=1}^{A_{v_{next}}} \lambda_i}{\sum_{i=1}^{A_{v_{next}}} i} \right)$$

From these likelihood values, the n most likely edges will be chosen to have a sensor placed on them.

Stochastic Pseudocode

```
###SET OF CANDIDATES
Input G : Graph ; Input NumSens : number ; Verts : set of objects ;
VertsEdge : set objects ; VertsTarget : set of objects ;
###OTHER INTERMEDIATE AND OUTPUT SETS
C : set of objects ; Output S : set of objects
|    ###INITIALIZATION
|    S = empty; i = 0
|    Initialize C
|    Main Loop, while i < 50
|    |    Find and remove duplicate candidates in C
|    |    ###EVALUATION
|    |    Evaluate all members of C
|    |    ###SELECTION
|    |    Select those members of C that will be allowed to persist
|    |    ###CROSSOVER
|    |    Crossover 10 randomly chosen members of C
|    |    i = i + 1
|    Endloop
|    ###SOLUTION
|    Select fittest c in C
|    Determine likelihood of each edge in c
|    place n most likely edges in c into S
|    return S
|
|    Func Initialize
|    |    Randomly create 50 candidates and add them to C
|    |    For each canditate c in C
|    |    |    ###FEASIBILITY
|    |    |    While c is not feasible
|    |    |    |    ###REPAIR
|    |    |    |    repair c
|    |    |    Endwhile
|    |    |    Evalaute fitness of c
|    |    |    Generate reduction graph of c
|    |    Endfor
|    Endfunc
```

```
|
|    Func Repair
|    |    Randomly choose allele in gene string of c
|    |    If allele = 1
|    |    |    change allele to 0
|    |    Else if allele = 0
|    |    |    move down gene string to first 1 value, change to 0
|    |    Endif Endif
|    Endfunc
|
|    ###CROSSOVER
|    Func Crossover
|    |    create new gene by bitwise OR of parent genes
|    |    ###MUTATION
|    |    mutate gene
|    |    generate reduction graph of c
|    |    ###FEASIBILITY
|    |    While not feasible
|    |    |    ###REPAIR
|    |    |    repair c
|    |    |    recreate reduction graph
|    |    Endwhile
|    |    evaluate fitness of c
|    |    add c to C
|    Endfunc
```

Local Search Algorithm

Tabu search is used for the local search algorithm. Similar to the approach described above, this search method will be based off of Karger's algorithm [1].

Input D_i:

$G\left(V, E, N_E, N_T\right) \rightarrow \left(D_i\right)$

Where:

- V is the set of vertices
- E is these of edges
- N_E is the set of edge vertices
- N_T is the set of target vertices

Additionally, some numerical value n which is the number of sensors will also be provided.

Output D_o:

$\left(E_s \subset E\right) \rightarrow \left(D_o\right)$

The output will be some subset of edges in E such that they are the most likely (based upon the likelihood function defined below), of the edges that connect the vertices remaining in the reduced graph produced when stopping criteria are met.

Initialization

Upon ingestion of the graph file, several data structures are created:

- a hashmap of vertices: this will be used to quickly access and change the vertices present
- a hashmap of edges: this will allow quick access and changes to the edges
- a bit string representing whether an edge is included in the graph

Neighborhood Function $N(S)$:
The neighborhood will be generated by either adding or removing a certain edge from the current graph. Essentially, "flipping the bit" for that edge in the bit string. The action of adding or removing an edge is then assigned a value which considers the following items:
- Does the action merge or add nodes in the graph?
- How does the action affect the value of ϑ for each vertex in the graph

Therefore, the value of a given neighbor is:

$v(n) = \omega + \xi$
Where:
- $\omega = \begin{cases} 1 & \text{if vertex is removed from graph} \\ 0 & \text{if no change in vertexes} \\ -1 & \text{if vertex is added to graph} \end{cases}$
- ξ is the average ϑ for all vertices in the graph

Whether a specific edge has been flipped recently will be stored in short-term memory. Once flipped, that edge will be required to persist in short-term memory for 4 iterations before it is allowed to be considered in the neighborhood again.

Feasibility Function
Check that a given neighbor does not exist in the tabu list

Selection Function
Choose the neighbor with the highest value and set as the current solution

Stopping Criteria:
There will be two stopping criteria for this algorithm:

bool $\left(|| V \in G_R || = 2 \right)$
- The number of vertices in the reduced graph G_R is 2

bool $\left(\xi \text{ has not changed in 10 iterations} \right)$
- This would indicate that despite changing the edges being included in the solution set, we are not achieving a greater amount of separation between N_E and N_T.

If either of these conditions are true, then the algorithm is stopped and the current set of edges remaining in G_R are returned for evaluation.

Evaluation Function
Once a solution set has been produced, the edges contained in that set need to have their respective likelihoods calculated. For this function, the same likelihood function from the stochastic implementation above will be used again:

$$L\left(e \in c\right) = \lambda_{v_{dest}} \left(\frac{\sum_{i=1}^{A_{v_{next}}} \lambda_i}{\sum_{i=1}^{A_{v_{next}}} i} \right)$$

After these likelihoods have been calculated, the n most likely edges are returned to the output domain.

Local Search Pseudocode

```
###SET OF CANDIDATES
Input G : Graph ; Verts : set of objects ; NumSens : number ;
VertsEdge : set objects ; VertsTarget : set of objects ;
###OTHER INTERMEDIATE AND OUTPUT SETS
curFittest : tuple(bit string, value) ; Output S : set of objects
###STOPPING CRITERIA
Split : bool ; noImprovement : bool ;
|    ###INITIALIZATION
|    S = empty ; Split = false ; noImprovement = false ;
|    S = initial solution
|    Loop while (Split == false AND noImprovement == false)
|    |    ###NEIGHBORHOOD FUNCTION
|    |    neighborhood = empty set
|    |    evaluate neighbors and add them to neighborhood
|    |    ###SELECTION
|    |    SNew = choose the neighbor with highest value
|    |    ###FEASIBILITY
|    |    If SNew is not in tabu list
|    |    |    If SNew has better value than S
|    |    |    |    update curFittest
|    |    |    Endif
|    |    |    S = SNew
|    |    Else if SNew is in tabu list
|    |    |    return to selection and find new SNew
|    |    Endif Endif
|    |    ###STOPPING CRITERIA
|    |    check stopping criteria
|    Endloop
|    ###EVALUATION
|    Determine likelihood of each edge in curFittest
|    add n most likely edges in curFittest to S
|    return S
```

Mapping to Chosen Computer Language

All algorithms described above will be implemented in Python. If not attached to this document, implementations, along with pseudocode.txt files may be found at the associated GitHub repository: https://github.com/jarnm04/Sensor-Placement. Software was implemented and tested using a 2015 MacBook Pro with the following specifications:

- Model ID: MacBookPro12,1
- Processor: 2.9 GHz Dual-Core Intel Core i5
- Number of cores: 2 (only 1 used for these executions)

- Memory: 8GB
- Operating System: macOS Big Sur, Version 11.4
- Kernel Version: Darwin 20.5.0

Test and Evaluation of SW Execution

Conclusion/Comments

The issue of determining where to optimally place sensor on a network is one of critical importance. The algorithms proposed in the paper are one step towards solving that problem, however there is still quite a bit more work that could be performed in the future. That work might include such actions as including heuristics in the algorithm implementations to better the efficiency of the algorithm, as well as introducing more factors than simply vulnerability scores when determining how an adversary might move through the network. Additionally, simulating different types of machines on the network, as well as the protocols and types of communications coming in an out of those machines would help to better mimic how an attacker might pivot and traverse through the network.

Acknowledgements

Pseudocode format derived from [6]. Standard search elements, algorithm and reporting structure derived from [7].

Resources

[1] Arora, S. (2013). *Karger's Min Cut Algorithm*. https://www.cs.princeton.edu/courses/archive/fall13/cos521/lecnotes/lec2final.pdf

[2] Griffin, T., Shepherd, F. B., & Wilfon, G. (2002). The Stable Paths Problem and Interdomain Routing. *IEEE/ACM Transactions on Networking (TON)*, *10*(2), 232–243. https://dl.acm.org/citation.cfm?id=508332

[3] *Chekuri, C., & Blatti, C. (2009). The Multiway Cut Problem (Vol. 3111). https://doi.org/10.1007/978-3-540-27810-8_24*

[4] Donnelly, K., Kfoury, A., & Lapets, A. (2010). The complexity of restricted variants of the stable paths problem. *Fundamenta Informaticae*, *103*(1–4), 69–87. https://doi.org/10.3233/FI-2010-319

[5] Sabharwal, S. (2020). *Pivoting - Moving Inside a Network (Cyber Security)*. GeeksforGeeks. https://www.geeksforgeeks.org/pivoting-moving-inside-a-network/

[6] Talbi, E.-G. (2009). Metaheuristics, From Design to Implementation. Wiley.

[7] Barr, R. S., Golden, B. L., Kelly, J. P., Resende, M. G. C., & Stewart, W. R. (1995). Designing and reporting on computational experiments with heuristic methods. *Journal of Heuristics*, *1*(1), 9–32. https://doi.org/10.1007/BF02430363