An Introduction to the Rotor-Router Mechanism

Evaluating the Efficiency of Rotor-Router Network Patrol

Julian M. Rice
University of California, Los Angeles
UID: 204793996

Masuzawa Laboratory
FrontierLab@OsakaU

August 2018

Table of Contents

Word Count: 9614 | **Page Count**: 27

Abstract

This paper goes over the fundamental ideas and concepts behind the rotor-router model, discussing the value and potential problems with the utilization of both single and multi-agent models. It includes a short overview on mobile agents and the problem with using random walks to visit each and every node and edge. There is a visualization of the rotor router algorithm using an undirected graph after explaining the algorithm and its time evaluation for an Euler tour. Simulations are performed on two different graphs to investigate how long it takes for an agent to *lock into* a forever ending *Euler cycle* that allows for efficient network patrol. The simulations provide insight on how stabilization periods are on average, compared to worst and best case stabilization periods. Finally, various ways to improve the rotor-router mechanism discussed in recent papers are summarized and evaluated; this involves changing the initialization of a graph before exploration takes place, adding a single bit of memory to an agent, and adding multiple agents into a graph to increase visit frequency of edges and vertices. This paper is aimed at computer scientists who want to learn about network patrolling and obtain a strong understanding of the rotor-router mechanism, its efficiency and importance through simulation data, and improvements that have been made to it.

1 | Introduction

Distributed systems have become a necessary technology in our lives over the course of the last few decades, particularly as a result of the increased use of personal computers and electronic devices today. Rather than having a large central server that receives, sends, and manages all of the data it is responsible for, a decentralized system of servers that each communicate with each other provide numerous improvements to a network. For example, having multiple servers located around the world for a popular online computer game allows for players from all over the world to, rather than connect to a centralized server that could otherwise be on the opposite side of the planet, connect to one of many servers; this server would operate as the primary server for the region, and provide player and system information to other servers with reasonably fair latency. Taking this example and broadening it out to different industries and systems, like e-commerce, the banking industry, and bioinformatics, reveal that further development of distributed systems will take us a long way.

These decentralized systems are sophisticated and complex to design and program. Start up cost is higher than having a singular central system; there is potential latency and data cost between the transfer of data from computer to computer and the problem of making the system fail-safe. A fail-safe system is able to continue operating even if one or multiple components running the system stops operating or fails to continue to function. Potential problems like a single point of failure (SPOF) may render an entire network unusable. Using the aforementioned online game server example, if the administrator server that relays system and player information to the rest of the individual servers located from region to region stops functioning, then it would become difficult to relay all of the individual information coming from the region servers without the entire system falling apart. The question that then remains is: what can we use to prevent these problems from arising, and how can we reduce latency times within a network?

Patrolling a Network with a Mobile Agent

One solution for some of the aforementioned problems found in distributed systems is the development and utilization of the **mobile agent**. A mobile agent is a bot that contains data and migrates from node to node within a network. It can transport its state throughout the network without incurring any changes to the data within the agent. The agent saves its own state as a process image before it moves, then after moving, resumes execution of the code within it from the previously saved state. This allows the connecting time and bandwidth consumption of a network to be reduced, because rather than processing and exchanging data to and from the client, the mobile agent processes the data at the source and only sends relevant details to the client.

The term *mobile agent* stems from two separate concepts: mobility and agency[1]. Mobility refers to a mobile agent's ability to move within a network, whereas agency (agent) refers to a program that gathers information and processes tasks. Mobility is an essential aspect that separates mobile agents from other types of agents. Within this aspect, **migration** is the most crucial element; migration allows an agent to move from

[1] V.A. Pham, A. Karmouch, *Mobile software agents: An overview, IEEE Commun*. Mag., 36(7):26-37, July 1998

one location to another. For a mobile agent to be able to migrate from place to place, the agent system must include support for execution stopping, state collection, data serialization and transfer, data deserialization, and execution resuming[2].

Another problem that arises from the use of mobile agents in a network is the challenge of making sure that when the mobile agent is tasked with visiting every node in a network, it efficiently visits each node the same amount of times. It is difficult and unrealistic to expect an equal amount of visits per node with random movement, but developing an algorithm for the mobile agent that ensures a more balanced series of traversals is not a trivial matter either. There has been research on how to produce these preferred results; one of the proposed models, the rotor router mechanism, is a mechanism that allows for a near equal amount of visits per node by both single agent network systems and multi agent network systems.

2 | Rotor Router Mechanism

The rotor-router mechanism, also known as the Propp Machine[3] and Edge Ant Algorithm, consists of quasirandom analogue of a **deterministic** walk through a network. The focus lies behind the problem of **patrolling**, which is the ongoing exploration of a network by a decentralized group of simple memoryless robotic agents[4]. This can range from using a single agent to traverse a network to using multiple agents to traverse a network. The rotor-router mechanism works because by storing when the last visit time for each edge connected to the node is, the mobile agent is able to visit the edge connected to a vertex that was traversed the longest time ago. There are claims and proofs[5] that show that after a *stabilization period*, one or multiple mobile agents are able to complete an **extended Eulerian cycle**, a cycle where every edge is traversed the same amount of times. This revolutionary algorithm has played a deep role in the development of mobile agents today; I will demonstrate how the algorithm works, run simulations to obtain stabilization period data, explain some of the claims behind it, and discuss potential problems that may arise from using the rotor-router mechanism.

Overview of the Algorithm[6]

```
Initialization: for every vertex v, set next exit pointer in v to point to
                first edge emanating from v.
Function:
def RotorRouterAlgorithm(Vertex u):
    (1) Let k be the location of the next exit pointer in u
    (2) e = u → v         #e is the corresponding edge
    (3) Move next exit pointer in u to the edge (k+1) mod degree(u)
    (4) u := v;
    (5) Go to (1)
end RotorRouterAlgorithm
```

[2] J. Cao and S. K. Das, *Mobile Agents in Networking and Distributed Computing, John Wiley & Sons.*, 4-16, 2012
[3] J. Spencer and J. Cooper, *Deterministic Random Walks on the Integers, Renyi Institute*, 2-3
[4] V. Yanovski, I. Wagner, *A Distributed Ant Algorithm for Efficiently Patrolling a Network*, *Algorithmica*, 2003
[5] Ibid
[6] Modified version of the algorithm seen in *A Distributed Ant Algorithm for Efficiently Patrolling a Network.*

The algorithm initializes itself by setting *next exit pointer*, which tells a vertex *v* to point to the first edge *emanating* from *v* (*connected to v*); another way to phrase this would be that the edge that the mobile agent traversed earliest becomes the *next exit pointer*, and becomes the next edge that the mobile agent traverses through. A popular analogy refers to how ants forage for food via a network-like system of pathways and nodes. By leaving a pheromone that loses its traces over time, ants can tell what paths have been visited recently and what paths have a weak trace. They can use this to dictate what direction to head, leaving their own pheromone that tells the other ants that an ant has recently taken this path. We assume that an ant, or our mobile a(ge)nt, will always choose the path that has the weakest trace, or the edge in a closed undirected graph G that has been traversed the least recently. Each step within the function RotorRouter Algorithm(Vertex *u*) is explained in more detail below:

(1) **Let k be the location of the next exit pointer in u**

During this first step, *k* is assigned the location of the *next exit pointer*, which means that *k* will point to a certain edge. Utilizing our ant analogy from above, this location would be the edge with the weakest pheromone level.

(2) $e = u \rightarrow v$

During this second step, the edge *e* is set equal to the edge between vertex *u* and vertex *v*. The arrow represents the edge, and this is what the *next exit pointer* is set to.

(3) **Move *next exit pointer* in *u* to the edge (k+1) mod degree(u)**

During this third step, the *next exit pointer* in vertex *u* changes from what it was set to before to **(k+1) mod degree(u)**. The edge that *k* is assigned to has its value increased by one, then that value is modded by the degree of that vertex. No matter how large k becomes, the modulo will always return a value between **0** and **degree(u)-1**.

(4) $u := v$;

During this fourth step, the vertex *u* has already had its *next exit pointer* set; the next step is to therefore set *v*, the vertex that *u* is currently connected to by edge *e*, as the *new* vertex *u*. Within this algorithm, the variable *u* is not locked into any certain vertex. Rather, it acts as a variable that always some vertex that is part of the graph, or node within a network.

(5) **Go to (1) and end RotorRouterAlgorithm**

During this final step, we repeat the first few steps mentioned above and is presumably perpetual. The reason for continuous movement throughout the nodes and edges is connected to the idea that a mobile agent should always be checking individual nodes within a network for problems of any sort.

Unlike **random walk**, which consists of the mobile agent uniformly choosing a random edge to traverse, this deterministic algorithm is significantly more efficient in regards to cover time and blanket time, which will be discussed in 2.3.

Visualization of the Algorithm

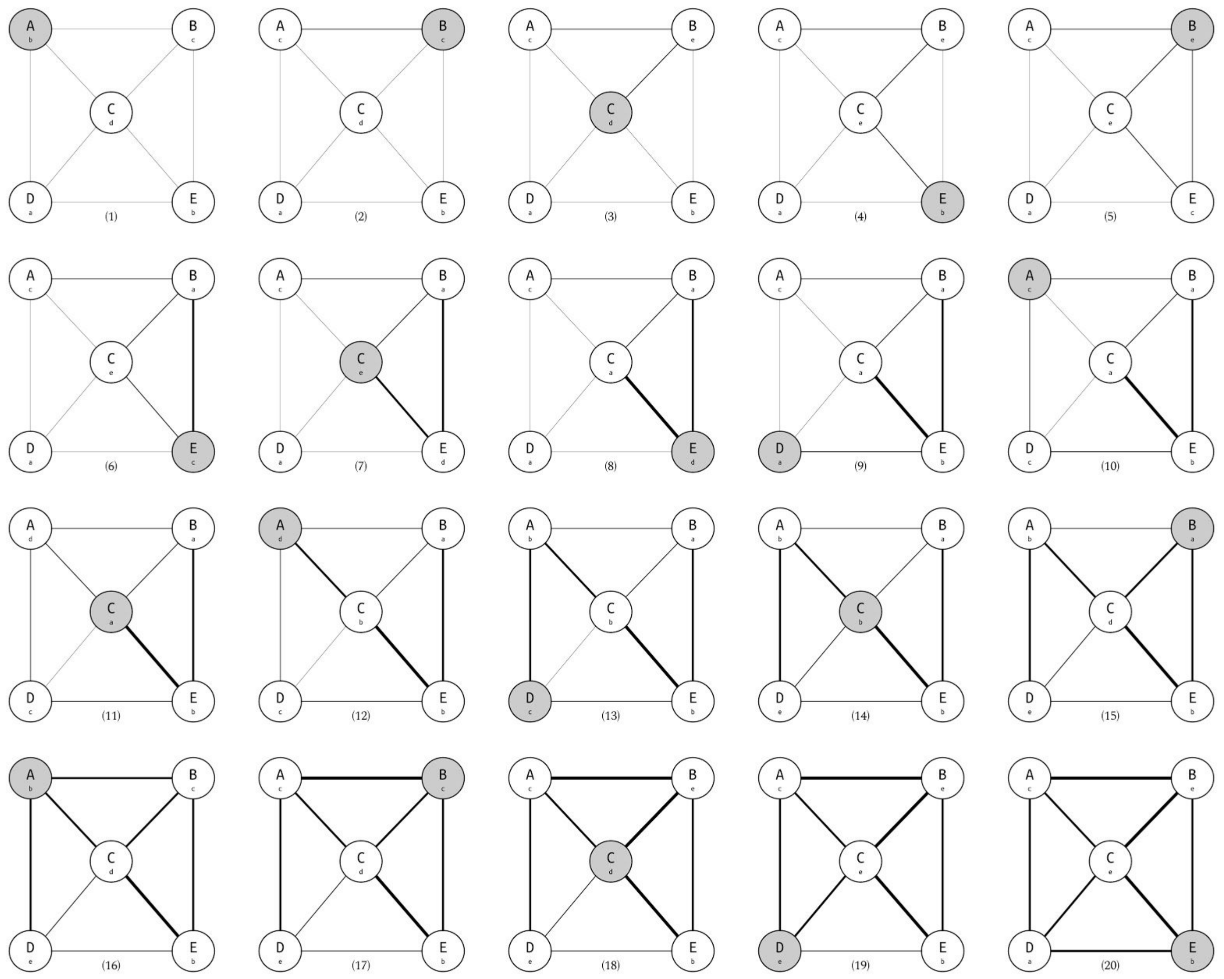


Figure 1: Example Walkthrough of RotorRouterAlgorithm

In **Figure 1** (1), the mobile agent starts at vertex *A*. The *next exit value* is pointing to the edge between vertex *A* and *B*, which means that the mobile agent will cross this edge and its destination vertex will be vertex *B*. Because the destination has already been set, vertex *A*'s *next exit value* will change (like a *rotor*) using the equation (**k+1**) **mod degree**(**u**). In this first case, k is equal to the current value [**0**]. Incrementing *k* and then taking the modulo of the degree of A [**3**], returns us with the value [**1**]. Now, as *one step* occurs and the mobile agent arrives at vertex *B*, its previous destination vertex *A*'s *next exit pointer* will now be pointing to the edge that leads to vertex *C*. The single mobile agent will, without doubt, travel to vertex *C* the next time it arrives at vertex *A*. This process is repeated from (2) to (20).

The edges proceed to become thicker and thicker as the number of visits increase. This is done to show the visit frequency and how it balances over 20 iterations, or *steps*. All edges are visited within 20 steps and the frequency is similar despite this not having a Euler cycle. As seen in steps (6) and (20), the edge between vertex *B* and *E* is not visited for a good amount of iterations. This is an example of above-average **blanket time**, which, in this context, is the amount of time a path or computer is not visited by the single mobile agent. On the other, the other edges are visited more often and the blanket time is low. Examples include the edge between vertex *D* and *E*, seen in steps (9) and (20), and the edge between vertex *A* and *D*, seen in steps (10) and (13).

Time Evaluation of the Algorithm

Many theorems and lemmas were shown after the publication of Yanovski's intricate take on the rotor-router algorithm. The **cover time**, or time it takes for a mobile agent to traverse all edges in a graph, is O($m \bullet D$)[7], where m is the number of edges and D is the **diameter** of the graph. Furthermore, the **lock-in time**, or time it takes for a mobile agent to traverse the graph as an Eulerian cycle (directed symmetric version) in a sequence that will be repeated forever[8], has the same number of steps as an upper bound compared to the cover time, O($m \bullet D$).

Yanovski also shows that increasing the number of agents does not *increase* the cover time and can sometimes help decrease the cover time of a graph. Furthermore, the blanket time is also significantly decreased as the number of agents increase. Finally, *both* the cover time and blanket time was shown to be significantly lower than the results of the same simulation using a random walk.

	Number of Agents									
Edges	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***
150	27	53	63	84	72	71	130	106	99	71
300	80	76	88	92	95	97	135	116	128	96
600	164	139	136	139	137	136	160	142	165	134
1200	454	284	242	227	222	219	230	214	225	212

Table 0: Average Lock-in Time (over 1000 runs)[9]

Various simulations tested the cover time and blanket time of the rotor-router algorithm compared to a standard random walk. **Figure A** consists of the difference between cover time for single and multi-agents using both types of walks in a randomly generated graph with 50 vertices and 1200 edges. The rotor-router algorithm (labeled as EAW, or *Edge Ant Algorithm*) had an average cover time that was 5.63 times that of the random walk. The increased amount of agents reduced the cover time for both the random walk and rotor-router algorithm, and the difference between both types of walks decreased, however the rotor-router algorithm continued taking less time than the random walk.

In **Figure B**, the blanket time for edges depending on the number of vertices is compared between one, two, and three mobile agents with only the rotor-router algorithm using a logarithmic scale. The graph shows that the blanket time behaves

[7] V.Yanovski, I.A. Wagner, *A Distributed Ant Algorithm for Efficiently Patrolling a Network*, *Algorithmica* 2003
[8] J. Chalopin, S. Das et al, *Lock-in Problem for Parallel Rotor-Router Walks*, *Researchgate*, July 2014
[9] V.Yanovski, *A Distributed Ant Algorithm for Efficiently Patrolling a Network*

similarly to n^2, meaning that the growth in blanket time relative to the increase in number of vertices is exponential.

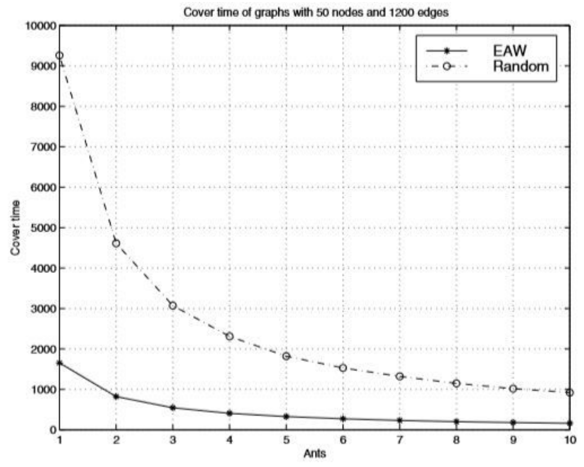


Figure A: Cover Time Comparison[10]
[**1**] EAW/Rotor Router vs Random Walk

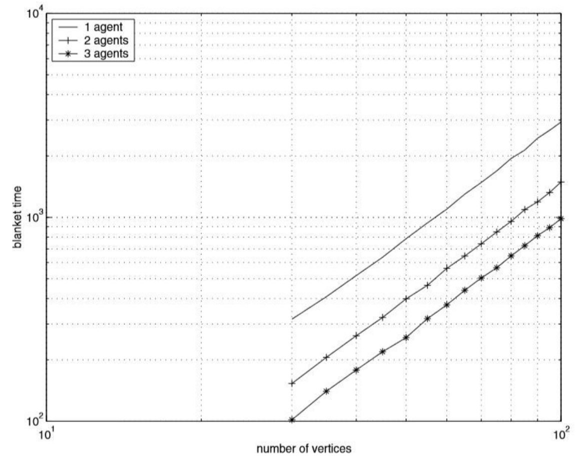


Figure B: Blanket Time Comparison
[**1**] Single Agent vs Multiple Agents

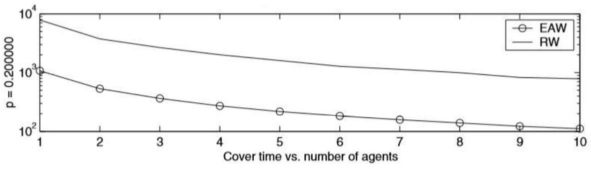


Figure C: Cover Time Comparison (*Small World*)
[**1**] EAW/Rotor Router vs Random Walk
[**2**] Single Agent vs Multiple Agent

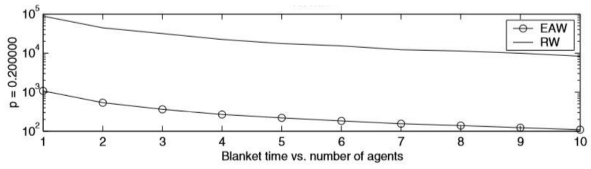


Figure D: Blanket Time Comparison (*Small World*)
[**1**] EAW/Rotor Router vs Random Walk
[**2**] Single Agent vs Multiple Agent

In **Figure C** and **D**, randomly generated **small world graphs** are used in the simulation. Because of their similar nature to real-life and electronic communication networks[11], using small world graphs instead of randomly generated general graphs allow us to more accurately see the impact the rotor-router algorithm has on traversal efficiency and decreased cover time and blanket time. **Figure C** specifically looks at cover time for single and multiple agents following random walk and the rotor-router algorithm. The difference here compared to the results in **Figure C** is significant; the rotor-router algorithm becomes up to 100 times more efficient than random walk, which shows how revolutionary this model could be if properly implemented. **Figure D** looks at blanket time, which is also roughly 100 times more efficient and faster than random walk. Covering all units necessary in quick manner and making sure the blanket time for each edge on average is as low as possible in numerous *types* of graphs demonstrate the weight of significance Yanovski's rotor-router algorithm plays in the world of networks.

However, there are certainly ways to improve these results, and this ranges from the customization of initialization states, use of memory, and observing the value and potential problems that using multiple agents in traversals may bring upon.

[10] All graphs were pulled from Yanovski's *A Distributed Ant Algorithm for Efficiently Patrolling a Network*
[11] D. J Watts, *Small Worlds-The Dynamics of Networks Between Order and Randomness*, Princeton University Press, Princeton, NJ, 1999.

Customized Simulations of the Algorithm

To test the efficiency of the rotor-router mechanism and find more unseen patterns that previous papers do not mention in great detail, I created a simulation program that takes an predefined undirected graph (**Graph 1**) and provides (line by line) the step-by-step movement of a single mobile agent. The program takes **Graph 1** as input, produces a **Graph 2** version of the input, and consists of *three* primary functions that allow the simulation to work. The RotorRouterAlgorithm asks for two inputs:

Input 1: Step Number *S* (*Number of steps the mobile agent takes in one walk*)
A certain step during an individual test is labeled as *s*.

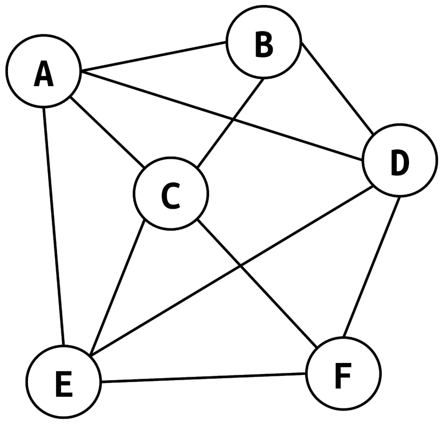
Input 2: Test Number *T* (*Number of individual walks the mobile agent undergoes*)
A certain test case within the set of tests is labeled as *t*.

The vertex where the mobile agent starts its exploration is set randomly for each test. After all tests in *T* have been run (all of which are *S* steps long), the results are outputted as an average. The results include the following:

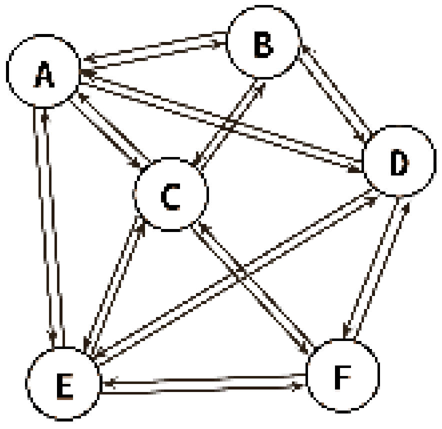
Average visit count for each vertex: This is the amount of visits that the mobile agent made to each vertex averaged among *T* tests. For **Graph 1**, a list of six different vertices and their average visit count will be present.

Average visit count for each edge: This is the amount of visits that the mobile agent made when traversing edges (or **arcs**) averaged among *T* cases. For **Graph 1**, a list of 11 different edges and their average visit count will be present. Note that instead of using (*directed*) arcs seen in **Graph 2** and Yanovski's claims with his rotor-router mechanism, this program finds the average visits for each undirected edge.

Stabilization period and Euler cycle: For each runthrough, the number of steps it takes for the agent to lock into a Euler cycle and the exact Euler cycle are also outputted by the program after *T* tests have been run.



Graph 1: Graph Used for Simulation (Undirected)



Graph 2: Graph Used for Simulation (Directed)

The rotor-router algorithm was run under numerous test cases that included various amounts of steps (*S*), run under *T* = 10,000 (*number of test cases*). The results of the simulation are summarized in the table below. The visit average has been rounded to the nearest ten-thousandth or thousandth, depending on allocated space per column.

T = 10000			**Vertices**			
Steps (S)	*A*	*B*	*C*	*D*	*E*	*F*
50	9.0191	6.8108	9.0975	9.0975	9.0905	6.8118
100	18.1829	13.6278	18.1900	18.1899	18.1805	13.6289
200	36.3609	27.2686	36.3677	36.3677	36.3679	27.2672
500	90.9084	68.1737	90.9162	90.9163	90.9103	68.1751
1000	181.819	136.359	181.823	181.822	181.820	136.358
2203	400.545	300.400	400.553	400.553	400.546	300.402

Table 1: Average Visit Count (*Vertices*)

T =10000						**Edges**					
Steps (**S**)	*AB*	*AC*	*AD*	*AE*	*BC*	*BD*	*CE*	*CF*	*DE*	*DF*	*EF*
50	4.5485	4.5396	4.5394	4.5378	4.5514	4.5512	4.5407	4.551	4.5405	4.5508	4.5491
100	9.0951	9.0844	9.0846	9.0832	9.0973	9.0976	9.0853	9.096	9.0855	9.0962	9.0948
200	18.182	18.180	18.177	18.175	18.185	18.183	18.178	18.191	18.176	18.188	18.186
500	45.456	45.449	45.447	45.446	45.460	45.458	45.450	45.463	45.448	45.461	45.460
1000	90.912	90.904	90.906	90.901	90.915	90.916	90.903	90.914	90.904	90.915	90.911
2203	200.274	200.267	200.264	200.264	200.278	200.276	200.269	200.282	200.267	200.280	200.279

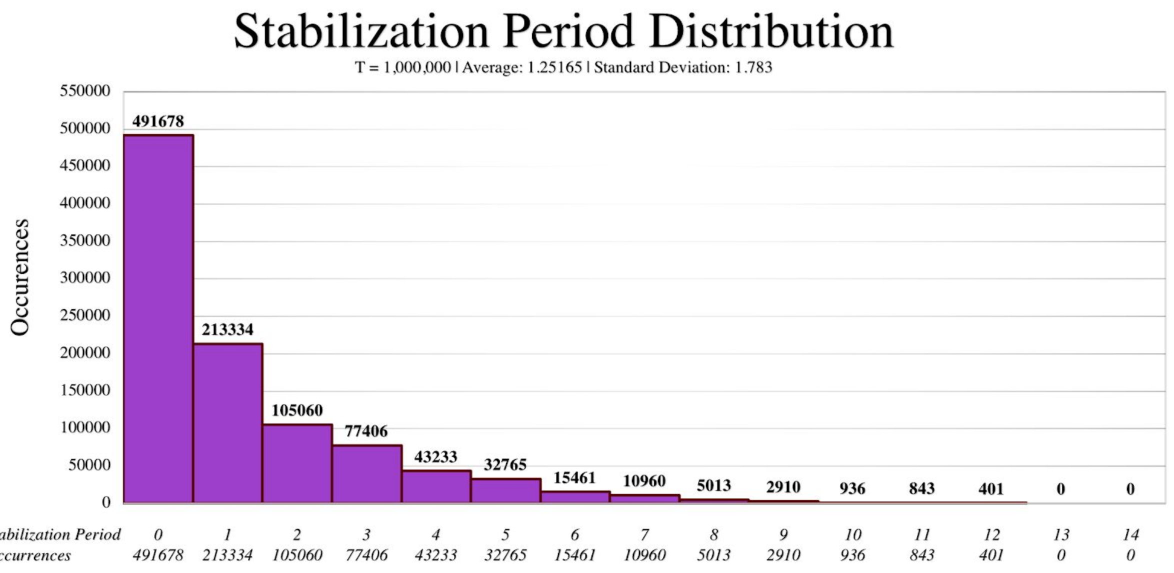
Table 2: Average Visit Count (*Edges*)

Exploring the edges and vertices in **Graph 1** over different amounts of steps (S) show a pattern of edge visit frequency uniformity. In **Table 2**, when *S* is equal to 50, a perfect tour (see **perfect average**) of the graph with a **stabilization period** *P* equal to 0 would return an average edge visit frequency of 4.545 (50 steps divided by 11 undirected edges). By simulating the rotor router algorithm 10000 times, the data shows that each edge's visit frequency is within the range 4.5378 (0.1584% *lower* than average) and 4.5514 (0.1408% *higher* than average), which is significantly small. Furthermore, a vertex visit count is also within a very tight range: vertices with a degree of 3 have a perfect average of 6.8175 visits over 50 steps and vertices with a degree of 4 have a perfect average of 9.090 visits over 50 steps; the range for degree 3 vertices is 6.8113 (0.0909% *lower* than average) and degree 4 vertices is 9.0762 (0.1518% *lower* than average). The visit frequency is dependant on the degree of the vertex (*see difference in visit frequency for vertices ACDE and BF*). Therefore, the rotor-router algorithm allows a mobile agent to visit each and every edge and vertex (*by degree*) in a graph with the same frequency as the average frequency found by dividing number of steps *s* by the number of undirected edges.

This is further reinforced when looking at a longer lasting traversal; when *S* is equal to 2203, the average edge visit frequency is 200.272 an individual edge's visit frequency ranges from 200.264 (0.0043% *lower* than average) and 200.282 (0.0046% *higher* than average). The individual edge visit frequency is even more accurate and closer to the average frequency than when *S* is equal to 50. There is one core reason why the range decreases and closes in on the average edge visit frequency.

Why? *The stabilization period* **does not** *necessarily increase when the step count is increased*: The stabilization period has been defined as the number of steps it takes for a mobile agent to lock into a forever-repeating Euler cycle, which is bound to the number of (*directed*) edges multiplied by the diameter of the graph[12]. In **Graph 2**, the graph has 22 total directed edges and a diameter of 2, making the maximum stabilization time **44**.

Most importantly, the step count plays *no role* in affecting the stabilization period. However, because the stabilization period ranges from 0 to 44, there will be possible cases where a high stabilization period causes the repeating Euler cycle to start repeating at a later step, which then shifts the edge visit frequency by a miniscule amount. This miniscule amount does *not* change as the cycle continues to loop; the difference in visit frequency for each individual edge does not change once the stabilization period has ended and lock-in has taken place.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
▨ *Stabilization Period*	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
■ *Occurrences*	*491678*	*213334*	*105060*	*77406*	*43233*	*32765*	*15461*	*10960*	*5013*	*2910*	*936*	*843*	*401*	*0*	*0*

Figure E: Stabilization Period Distribution Results from Graph 2
Note: The occurrence count for s = 13 to 44 was 0 | **Average Stabilization Period**: 1.25165 steps

[12] Yanovski, ibid

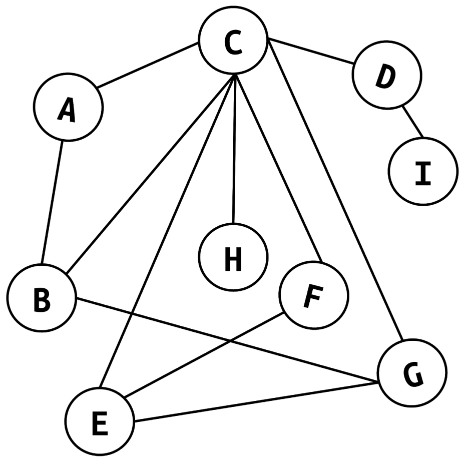
Sample Euler Cycles (*T = 1,000,000*)
C1: *CB->BA->AB->BC->CE->EA->AC->CF->FD->DF->FE->EC->CA->AD->DA->AE->ED->DB->BD->DE->EF->FC*
C2: *FC->CA->AD->DA->AE->ED->DB->BD->DE->EF->FD->DF->FE->EA->AB->BA->AC->CB->BC->CE->EC->CF*
C3: *DA->AC->CE->EA->AD->DB->BA->AE->EC->CF->FE->ED->DE->EF->FC->CA->AB->BC->CB->BD->DF->FD*
C4: *AE->ED->DE->EF->FC->CB->BC->CE->EA->AB->BD->DF->FD->DA->AC->CF->FE->EC->CA->AD->DB->BA*
C5:*FD->DA->AB->BD->DB->BA->AC->CF->FE->EA->AD->DE->EC->CA->AE->ED->DF->FC->CB->BC->CE->EF*
C6: *AE->EC->CB->BC->CE->ED->DE->EF->FC->CF->FD->DF->FE->EA->AB->BD->DA->AC->CA->AD->DB->BA*
C7: *DF->FE->EA->AC->CA->AD->DA->AE->EC->CB->BA->AB->BC->CE->ED->DB->BD->DE->EF->FC->CF->FD*
C8: *BA->AB->BC->CE->ED->DB->BD->DE->EF->FC->CF->FD->DF->FE->EA->AC->CA->AD->DA->AE->EC->CB*
C9: *DF->FE->EA->AE->EC->CB->BA->AB->BC->CE->ED->DA->AC->CF->FC->CA->AD->DB->BD->DE->EF->FD*

Table 3A: First 9 Euler Cycles from Figure E

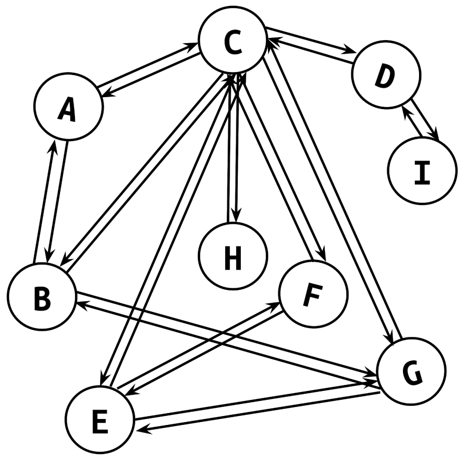
Figure E is a sizable distribution of stabilization periods (*S*, steps) over *T* = 1,000,000 for **Graph 2**, or one million traversals of the rotor-router algorithm-equipped mobile agent. The average period is 1.25165 steps and the worst case is 12 steps. Despite the maximum stabilization period being **44** steps, there were *no* occurrences where the stabilization period was over 12 steps. I theorize that the random initialization settings are one of the key elements that prevent the worst case stabilization time to occur, despite there being 1,000,000 tests. Another note is that the number of occurrences gradually decreases with no longer periods having a greater amount of occurrences than shorter periods. This follows a smooth exponential trendline with the equation $\boldsymbol{y = 446232e^{-0.581x}}$ and each stabilization period on the *x-axis* decreases at a reasonable negative exponential rate.

Table 3A showcases a small portion of the output from the million test simulation, and shows the full Euler cycle that the agent locked-in on after the stabilization period ended. Each cycle displayed in the table repeat endlessly until the specified number of steps *s* has been executed. Over the course of 22 traversals, each arc is visited once, and a never-ending Euler cycle will guarantee that each arc is only visited a second time after every other arc in the graph has been visited. In addition to the nature of the mobile agent's to-be repetitive movement, the stabilization period in **Figure E** has a range from 0 to 12, with the majority being a mere 0, 1, or 2 steps. This value, alongside with the total number of steps *S* for a single test case, are what bring variation to the visit frequency for each arc (*and hence, edge*).

After testing the stabilization period for **Graph 2** a million times (*literally*), I decided that a graph with a lower vertex to edge ratio would potentially result in an average period that is higher than the data provided in **Figure E**. **Graph 3** is the next graph that I run 100,000 single agent network patrolling simulations, and the data that follows contains similarities and key differences between what is found in **Figure E**.

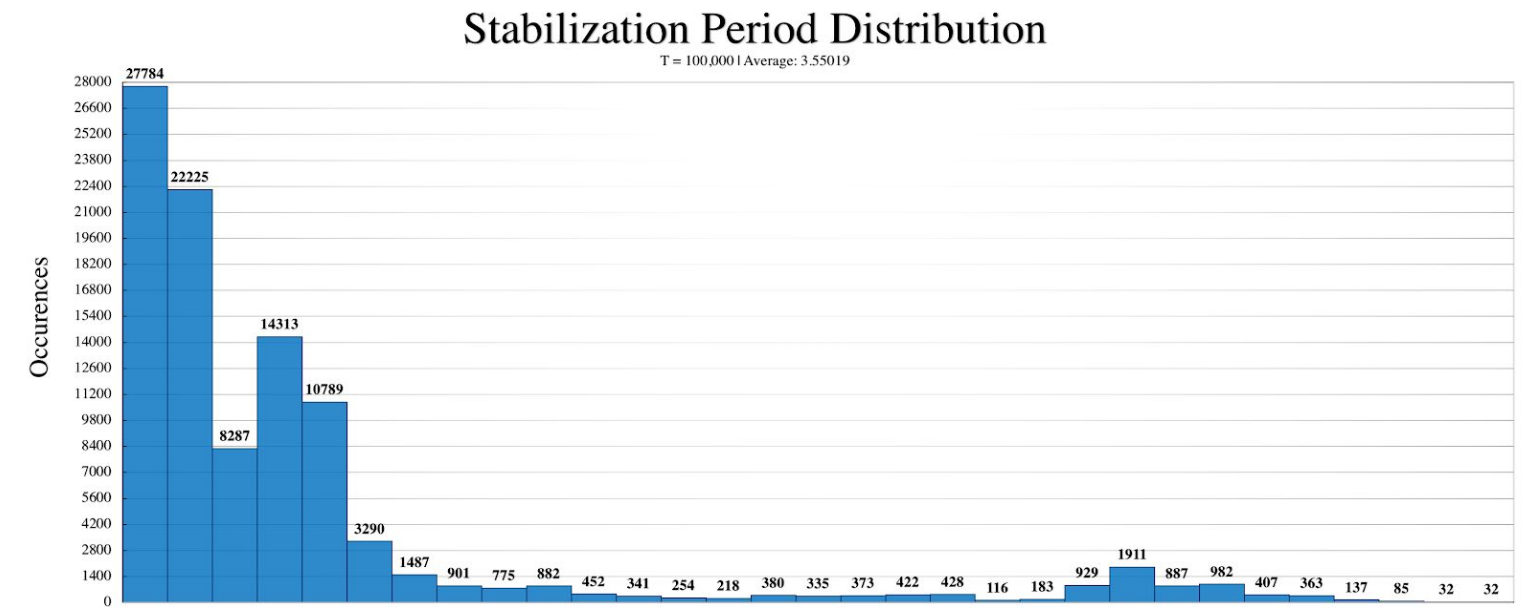


Graph 3: Another Graph Used for Simulation



Graph 4: Another Graph Used for Simulation

Figure F is another distribution of simulation periods (*S*, steps) over *T = 100,000* for **Graph 4**. Compared to **Graph 2**, there are three more vertices but only one more undirected edge (*two* arcs), meaning that the ratio of vertices and edges has been decreased. I hypothesize that the lower the ratio, the greater the stabilization period will be on average, and this is proven as the average stabilization period in **Figure F** is 3.55, compared to **Figure E**'s 1.25. Vertex *C* has a great amount of edges connected to it with a degree of 7, and acts somewhat similar to a main server or hub in a distributed system, whereas there are points like vertex *I* and *H* that only have one vertex connected to it with a degree of 1. This also means that vertex *I* and *H* do not have a cyclic order, because both are fixed to point to the only edge it can point to (*C* and *D* respectively).



🟩 *Stabilization Period*	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*	*28*	*29*	*30*
🟦 *Occurrences*	*27784*	*22225*	*8287*	*14313*	*10789*	*3290*	*1487*	*901*	*775*	*882*	*452*	*341*	*254*	*218*	*380*	*335*	*373*	*422*	*428*	*116*	*183*	*929*	*1911*	*887*	*982*	*407*	*363*	*137*	*85*	*32*	*32*

Figure F: Stabilization Period Distribution Results from Graph 4
Note*: The occurrence count for s = 31 to 144 was 0 | **Average Stabilization Period**: 3.55019 steps

Sample Euler Cycles (*T = 100,000*)
***C1**: EC->CF->FC->CG->GB->BG->GC->CH->HC->CA->AB->BA->AC->CB->BC->CD->DI->ID->DC->CE->EF->FE->EG->GE*
***C2**: DI->ID->DC->CE->EF->FE->EG->GE->EC->CF->FC->CG->GB->BG->GC->CH->HC->CA->AB->BA->AC->CB->BC->CD*
***C3**: CF->FE->EF->FC->CG->GE->EG->GB->BG->GC->CH->HC->CA->AB->BA->AC->CB->BC->CD->DI->ID->DC->CE->EC*
***C4**: EG->GB->BA->AC->CG->GC->CH->HC->CA->AB->BC->CB->BG->GE->EC->CD->DI->ID->DC->CE->EF->FC->CF->FE*
***C5**: GB->BA->AC->CH->HC->CA->AB->BC->CB->BG->GC->CD->DI->ID->DC->CE->EC->CF->FC->CG->GE->EF->FE->EG*
***C6**: CH->HC->CA->AB->BC->CB->BG->GC->CD->DI->ID->DC->CE->EC->CF->FC->CG->GE->EF->FE->EG->GB->BA->AC*
***C7**: CA->AC->CB->BG->GC->CD->DI->ID->DC->CE->EC->CF->FC->CG->GE->EF->FE->EG->GB->BA->AB->BC->CH->HC*
***C8**: CB->BG->GC->CD->DI->ID->DC->CE->EC->CF->FC->CG->GE->EF->FE->EG->GB->BA->AB->BC->CH->HC->CA->AC*
***C9**: CD->DI->ID->DC->CE->EC->CF->FC->CG->GE->EF->FE->EG->GB->BG->GC->CH->HC->CA->AB->BA->AC->CB->BC*

Table 3B: First 9 Euler Cycles from Figure F

Table 3B showcases nine Euler cycles generated from the 100,000 test simulation that the mobile agent will lock into after the stabilization period has passed. Similar to **Table 3A**, each starting point is different per test, which verifies the randomness of agent starting positions, different fixed cyclic orders per vertex, and the next exit value for each vertex at its initialization state (*before* the mobile agent has started patrolling).

The stabilization period distribution in **Figure F** provides very different and potentially unexpected details about the nature of stabilization periods and how, unlike in **Figure E**, there are tendencies for certain stabilization periods to be more common when running the simulation multiple times. To be more specific, sudden increases in the number of occurrences for tests that returned a stabilization period of 3, 21, and 22 steps are key results that have been returned from simulating **Graph 4**. The reason behind these increases in occurrences likely lies behind the structure of the graph used in the simulation. Conversely, note that 2, 13, and 19 step long periods occurred much less frequently than the stabilization period before it. Finally, a period of 0 steps remains the most frequently returned stabilization period, at roughly 27.78% of the 100,000 tests made. This raises the question whether there is some sort of graph that could force the most commonly returned stabilization period to *not* be 0 steps, and what kind of period distribution that graph would return when run through my simulation.

Simulating only two graphs means that there are many unexplored possibilities for stabilization period distribution and average visit frequency for both edges (*arcs as well*) and vertices. A **star graph** with the agent starting at the vertex with the highest degree would have a stabilization period of 0 steps, because every other vertex would point back to such vector. However, starting the agent at any other vertex would return a period of 1 step, because the initial step is what locks the agent into a Euler cycle. Running these simulations with small world graphs also seems like a reasonable next step, as these graphs are purposely designed to be akin to real networks, and may provide better insight on how to effectively patrol the networks that run our world.

3 | Enhancing the Rotor Router Mechanism

There are a few problems with the original rotor router model proposed in Yanovski[13], and the majority of these problems rely on inefficiency. For example, it has been proven that without considering the initial configuration of the rotor-router mechanism (assigning port numbers and initial *next exit pointer* values), the mobile agent's lock-in time is bound at the worst case O(m • D). Including constants, the number of steps it takes to cover all edges and settle into an Eulerian cycle (lock-in) is at most $4 \cdot m \cdot D$. This lock-in time also applies to labeled graphs and memoryless mobile agents. However, there have been strides to increase the efficiency of the rotor-router algorithm through various measures, most of which include taking deeper looks at the modification of the initial configuration and adding memory to mobile agents and nodes.

Euler Tour Lock-in Problem

There have been two recently published papers that cover the robustness of the rotor-router mechanism and how modifying the initial configuration details can affect both the lock-in times for both worst-case and best-case graphs. Ilcinkas[14] et al uses a two player game analogy where the player, ***P***, has the goal of making the singular mobile agent lock in as quickly as possible, and the adversary, ***A***, has the goal of preventing such a lock-in from occurring. There is an emphasis on using **anonymous graphs**, which are graphs where all nodes have no identifier, and it makes it impossible for the agent(s) to know where they are. This requires memory of some sort in either in the node, the agent, or both, to be able to store information about whereabouts and location. **Table 4** demonstrates different possibilities when it comes to modifying the initial configuration of the graph.

	Player's Choice		**Adversary's Choice**	
	Chooses First	*Chooses Last*	*Chooses First*	*Chooses Last*
Port Assignment (**↻**)	P(↻) ---	--- P(↻)	A(↻) ---	--- A(↻)
Next Exit Pointer (**π**)	P(**π**) ---	--- P(**π**)	A(**π**) ---	--- A(**π**)
Both Selected (**All**)	P(**All**)	P(**All**)	A(**All**)	A(**All**)

Table 4: Modifiable Parameters (initial configuration)

All Potential Configurations (**1/2**)	P(↻)A(**π**)	Player chooses port assignment ***then*** Adversary chooses next exit pointer
	P(**π**)A(↻)	Player chooses next exit pointer ***then*** Adversary chooses port numbers

[13] Ibid

[14] **Paper 1**: D. Ilcinkas et al, *Euler Tour Lock-in Problem in the Rotor-Router Model*, 2009
Paper 2: D. Ilcinkas et al, *Robustness of the Rotor-Router Mechanism*, *Algorithmica*, 78:869-895, 2017

	P(**All**)	Player assigns port number and the next exit pointer. No order specified.
***All Potential Configurations* (2/2)**	A(↻)P(**π**)	Adversary chooses port assignment ***then*** Player chooses next exit pointer
	A(**π**)P(↻)	Adversary chooses next exit pointer ***then*** Player chooses port numbers
	A(**All**)	Adversary assigns port number and the next exit pointer. No order specified.

Table 5: Possible Initialization Scenarios

The six potential configurations, or scenarios, are set before the mobile agent starts its walk and traversal. For scenarios that have either the player or adversary choosing only one of the two modifiable parameters, the entity making the second choice is able to view the choices made by the entity that made the first choice. For example, in **Table 5**, P(↻)A(**π**) means that ***P*** will choose the port numbering, and ***A*** will have access to this information and can use it to the best of ***A***'s ability to choose values for each node's *next exit pointer* to increase the lock-in time as much as possible. This works the opposite way as well; by selecting A(**π**)P(↻), ***A*** will choose the port numbering and ***P*** will choose the *next exit pointer* values in such a way that the lock-in time is reduced by as much as possible.

The lock-in times for worst-case and best-case graphs are demonstrated in **Table 6**, where *m* represents the number of edges and *D* represents the graph diameter.

Scenario	*Worst-case graph lock-in time*	*Best-case graph lock-in time*
P(**All**)	$\Theta(m)$	$\Theta(m)$
A(↻)P(**π**)	$\Theta(m)$	$\Theta(m)$
P(**π**)A(↻)	$\Theta(m \bullet \min\{\log m, D\})$	$\Theta(m)$
A(**π**)P(↻)	$\Theta(m \bullet D)$	$\Theta(m)$
P(↻)A(**π**)	$\Theta(m \bullet D)$	$\Theta(m)$ if $D \leq m^{1/3}$
A(**All**)	$\Theta(m \bullet D)$	$\Theta(m + D^2)$ [*best starting node*] $\Theta(m \bullet D)$ [*worst starting node*]

Table 6: Worst & Best Case Lock-in Times[15]

The results demonstrated in **Table 6** show the importance of the state the graph is in during initialization, and how much more efficient lock-in time can be achieved. The worst case lock-in time for a graph that has its *next exit pointer* values chosen by ***P*** after the port numbering has been assigned by either ***P*** or ***A*** is $\Theta(m)$, which is

[15] D. Ilcinkas et al, *Robustness of the Rotor-Router Mechanism*, *Algorithmica*, 78:873, 2017

substantially different from Yanovski's less efficient $\Theta(4 \bullet m \bullet D)$. The four remaining scenarios have a worst-case graph lock-in time of either $\Theta(m \bullet \log m)$ or $\Theta(m \bullet D)$. These scenarios consist of ***A*** choosing the *next exit pointer* values or giving ***A*** the second choice. Despite increased control and access to making the initialization state of the graph as convoluted as possible, the lock-in time is reduced to $\Theta(m \bullet D)$ at least efficiency, which is still two times faster than Yanovski's $\Theta(4 \bullet m \bullet D)$. Finally, Ilcinkas proves that a mobile agent traversing a **Hamiltonian graph** reaches lock-in time within $O(m)$. These results give the rotor-router mechanism a significant efficiency increase and shows that the rotor-router algorithm is not the only crucial element in regards to efficiency; the initial configuration of the graph and data stored in each of the individual nodes also plays an important role in the efficiency of the rotor-router mechanism and its lock-in time.

Finite State Machines and One Bit of Memory

A very recent paper[16] by Menc et al looks at the potential of one bit, and how allocating a small amount of memory to both a single mobile agent and each node in an undirected graph and using finite state machines can allow for increased robustness within the rotor-router mechanism. The basis behind Yanovski's rotor-router mechanism revolves around the usage of memoryless mobile agents. However, Menc uses a mobile agent that has only *one bit* of memory, and nodes that each contain a certain amount of memory, dependent on the node degree.

An anonymous graph ***G*** with *n* nodes that contain no identifiers and *m* edges is explored by a single agent. The challenge comes with exploring the graph without the agent knowing the incoming port numbering of the node it will arrive at, which means that it cannot backtrack its moves. The unique agent used for this exploration is written in the following format: **(M_a, M_w)-agent**. $\mathbf{M_a}$ represents the amount of memory, in bits, that the agent has (*a constant*). $\mathbf{M_w}$ represents the amount of memory, also in bits, that each individual node contains (*also a constant*). The information stored inside $\mathbf{M_w}$ is modified by the (M_a, M_w)-agent when the agent approaches a node.

The (M_a, M_w)-agent is modelled as a **finite state machine** with two different states, and each of the nodes in ***G*** have markers as to whether it has been visited or not. The two states that the (M_a, M_w)-agent can be in are outlined below:

EXPLORATION MODE *State 0, Default State*	The agent explores the graph regularly, but if it lands on a vertex *v* that has had all of its neighbors visited, then the agent switches into REVISIT MODE.
REVISIT MODE *State 1, Speedup State*	The agent follows the same sequence of vertices $v_0 = v$, v_1, … $v_k = v$ performed during EXPLORATION MODE, and traverses each visited arc for the second (third, etc.) time. Landing on a new border vertex v_i enables a subsequent phase of EXPLORATION MODE, which returns to REVISIT MODE when the agent has finished exploring and respectively returns to v_i.

[16] A. Menc, *On the power of one bit: How to explore a graph you cannot backtrack?*, *Researchgate*, 2015

Figure 2: Overview of (M_a, M_w)-Agent States

The anonymous graphs observed are undirected, but is looked at by using two directed edges (**arcs**) per undirected edge. From this, a **border vertex** is defined as a vertex that has *at least one* already traversed **outgoing arc** and *at least one* not-traversed outgoing arc. Whether a vertex *v* is or is not a border vertex plays an essential role in the action that the mobile agent takes during its exploration. The following example demonstrates the change between states and how an anonymous graph can be efficiently explored.

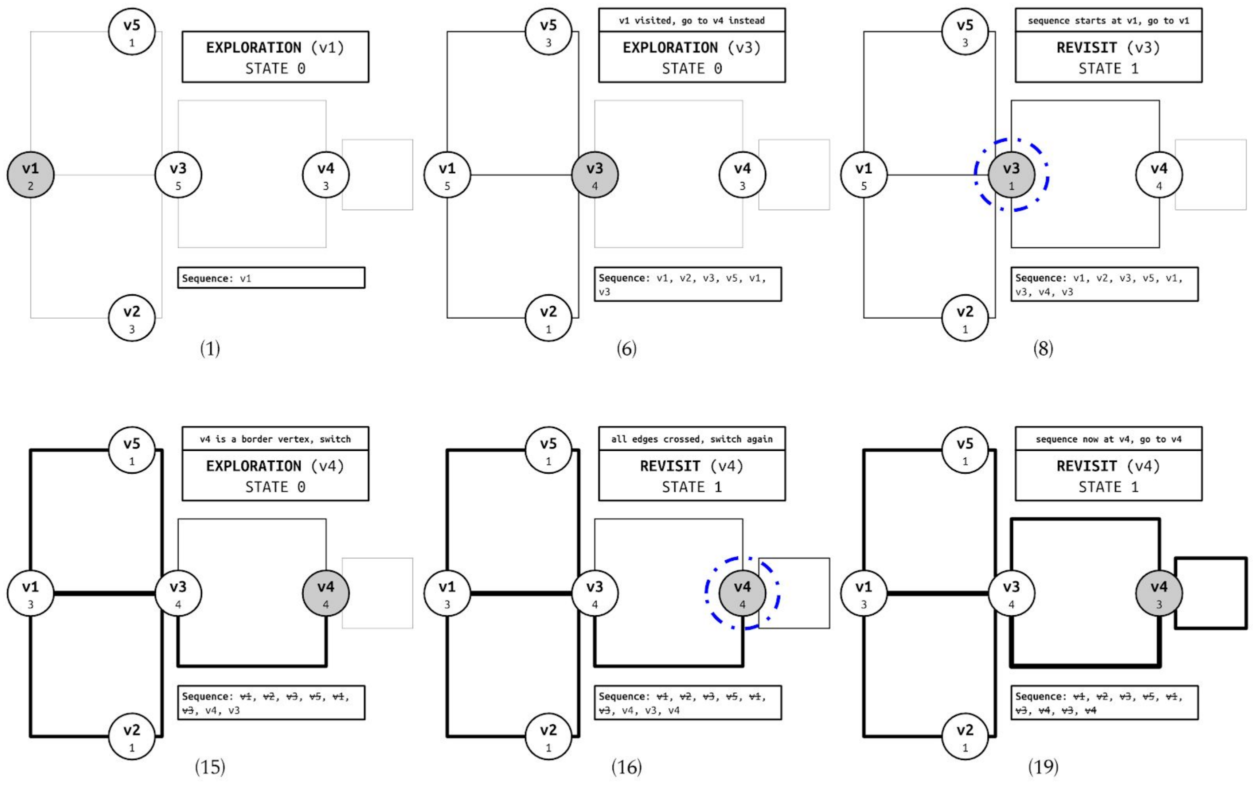

EXPLORATION (v1)
STATE 0
Sequence: v1
(1)
v1 visited, go to v4 instead
EXPLORATION (v3)
STATE 0
Sequence: v1, v2, v3, v5, v1, v3
(6)
sequence starts at v1, go to v1
REVISIT (v3)
STATE 1
Sequence: v1, v2, v3, v5, v1, v3, v4, v3
(8)
v4 is a border vertex, switch
EXPLORATION (v4)
STATE 0
Sequence: ~~v1~~, ~~v2~~, ~~v3~~, ~~v5~~, ~~v1~~, ~~v3~~, v4, v3
(15)
all edges crossed, switch again
REVISIT (v4)
STATE 1
Sequence: ~~v1~~, ~~v2~~, ~~v3~~, ~~v5~~, ~~v1~~, ~~v3~~, v4, v3, v4
(16)
sequence now at v4, go to v4
REVISIT (v4)
STATE 1
Sequence: ~~v1~~, ~~v2~~, ~~v3~~, ~~v5~~, ~~v1~~, ~~v3~~, ~~v4~~, ~~v3~~, ~~v4~~
(19)

Figure 3: Shortened Example of Enhanced Rotor-Router Algorithm (*OneBit*)[17]

As the mobile agent traverses through the anonymous graph in the initial EXPLORATION MODE, the sequence of vertices the mobile agent travels through is saved (as seen in each sub image in **Figure 3**). The mobile agent initially starts in the first state, EXPLORATION MODE in (1), and numerous steps, lands in *v3*, which is no longer a border vertex. This is demonstrated in (8); all of the neighboring vertices have been visited, and each edge has also been walked through once as well. The result of this is a state change of the mobile agent to the second state, REVISIT MODE. At this point, the sequence, or order that the mobile agent traversed the graph in is repeated until the agent lands on a border vertex. This is shown in (15); *v4* is a border vertex because one of its edges (*outgoing arcs*) is unvisited, and another one of its edges (*outgoing arcs*) has been visited. This changes the agent state into back to EXPLORATION MODE as seen in (16), and concludes with a return to REVISIT MODE

[17] The full walkthrough of the *OneBit* function can be seen in the Appendix (**Figure 3 Full**)

after the agent has traversed the last unvisited edge. Finally, the sequence is continued until it has been finished in (19).

Graph Type	*(1, O(log D))-Agent*	*(∞, 0)-Agent*	*(0, ∞)-Agent*
2-Regular Graph	$O(n)$	$\Omega(n^{1.51})$ [18]	$\Omega(n^2)$
General Graph	$O(m)$ $\Theta(4m)$	$\Omega(n^4)$ [19]	$\Omega(n^3)$

Table 7: Exploration Time Using Various (M_a, M_w)-Agents

Table 7 shows that the optimal exploration time using an (**1, O**(**log D**))-**agent** is the number of nodes for a **2-regular graph** and the number of edges for a general graph. More specifically, Menc proves that a mobile agent following the *OneBit* algorithm will explore with return in time $\Theta(4m)$. Each edge is traversed four times, or in other words, every node's outgoing and incoming arc(s) will be crossed *twice*. Menc also shows that using a memoryless agent allows for a lower bound of $\Omega(n^2)$ for a 2-regular graph and $\Omega(n^3)$ for a general graph. Notice that despite being a lower bound for exploration time efficiency, the addition of one bit of memory to the agent while maintaining a certain amount of memory on the nodes allows a tremendous speedup and is certainly an enhanced version of the rotor-router mechanism.

Advantages and Disadvantages of Multi-Agent Walks

Improving the rotor-router mechanism does not only apply to modifications to the algorithm, but also the use of multiple agents in network patrolling. Rather than a single agent exploring each edge and vertex and having potentially lengthy cover and blanket times for expansive graphs, utilizing multiple agents allow for a potential reduction in cover and blanket time. However, a mystery arises with the bounds associated with the lock-in time, or stabilization period, for the system. Chalopin investigates the stabilization period upper bound and proved that the polynomial upper bound $O(m^4 \bullet D^2 + m \bullet D \bullet \log k)$ is the number of steps it takes for a system to stabilize[20], with m being the number of edges, D being the diameter of the graph, and k being the amount of agents that are patrolling the system. This disproves Yanovski's initial conjecture that a system with multiple concurrent walks occurring also stabilizes in a period of at most $4 \bullet m$ steps. The reason why the period can be so long comes from the fact that the value is derived from many small parts, each of which exhibits a small and different period length. **Table 8** describes both the upper and lower bounds for a two agent parallel rotor-router walk.

[18] H. dai and K. Flannery. *Improved length lower bounds for reflecting sequences.* In *Computing and Combinatorics*, volume 1090 of *Lecture Notes in Computer Science*, 56-67. Springer Berlin Heidelberg, 1996.
[19] A. Borodin, W.L Ruzzo et al, *Lower bounds on the length of universal traversal sequences*, *Journal of Computer and System Sciences*, 45(2):180-203, 1992
[20] J. Chalopin, S. Das et al, *Lock-in Problem for Parallel Rotor-Router Walks*, *Researchgate*, July 2014

Number of Agents	***Worst Case*** *(Upper Bound)*	***Best Case*** *(Lower Bound)*
$k = 2$	$O(m^4 \cdot D^2 + m \cdot D \cdot \log(2))$	$\Omega(m^2 \cdot \log(n))$

Table 8: Exploration Time for Two Agents in a Parallel Rotor-Router Walk[21]

Despite the greatly increased stabilization period for having two or more parallel agents patrolling a distributed system, Chalopin introduces a certain behavior, **subcycle decomposition**[22], that divides agents into indivisible groups that, "patrol the arcs of the graph, and two groups never traverse the same arc." A graph can be partitioned into arc-disjoint directed Euler cycles, which results in each agent eventually traversing arcs of one of the multiple Euler cycles that the graph has been partitioned into. A unique use for multi-agent parallel rotor-router walks lies in the **team patrolling problem**, which has yet to be deeply researched.

4 | Conclusion

Past papers have thoroughly investigated the stabilization period upper bound for a variety of graphs given random and set initialization states. However, predicting the average stabilization period for nearly almost any given graph remains difficult. The data from the simulations run on **Graph 2** have revealed that for certain cases, a graph with a worst case stabilization period of 44 steps has an average stabilization period of 1.25 steps, which is significantly lower than 44, and closer to an instantaneous lock-in. The data from the simulations run on **Graph 4** have revealed that a worst case period of 144 steps returns an average 3.55 steps, which is still relatively close to an instantaneous lock-in. This paper demonstrates that the rotor-router mechanism and algorithm are relative efficient in terms of consistently having a low stabilization period.

Furthermore, the introduction of the *OneBit* algorithm provides an even more excellent lock-in time and is a large leap for the rotor-router mechanism and the field of network algorithms. It would be fascinating to see full simulations done using *OneBit* with many variations of graphs, similar to the simulations that Yanovski did with the first rotor-router algorithm (*Edge ant algorithm*) and the simulations I ran for Yanovski's version of the rotor-router algorithm. More specifically, observing cover time, blanket time, and how these times change depending on whether a randomly generated 2-regular graph, general graph, or small world graph is used. A legitimate walkthrough of *OneBit* being implemented within a network system and an analysis on the results of single and multi agents would be an interesting way to demonstrate the realistic capabilities of *OneBit*, and possibly open up new elements that can be further improved to create an even more efficient variation of the rotor-router algorithm.

Lastly, deeper research and more simulations using parallel walks and multiple agents is a necessity, as more complex load balancing and network patrolling scenarios become more and more complex and take up potentially high amounts of resources.

[21] Ibid
[22] Ibid

5 | References

V.A. Pham and A. Karmouch, *Mobile software agents: An overview, IEEE Commun*. Mag., 36(7):26-37, July 1998

J. Cao and S. K. Das, *Mobile Agents in Networking and Distributed Computing*, *John Wiley & Sons*., 4-16, 2012

J. Spencer and J. Cooper, *Deterministic Random Walks on the Integers*, *Renyi Institute*, 2-3

V. Yanovski and I. A. Wagner, *A Distributed Ant Algorithm for Efficiently Patrolling a Network*, *Algorithmica*, 2003

E. Bampas and L. Gasieniec, *Euler Tour Lock-in Problem in the Rotor-Router Model*, *International Symposium on Distributed Computing*, 423-435, 2009

D. Ilcinkas, E. Bampas et al, *Robustness of the Rotor-Router Mechanism, Algorithmica*, 78:869-895, 2017

A. Menc, On the power of one bit: *How to explore a graph you cannot backtrack?*, *Researchgate*, 2015

H. dai and K. Flannery. *Improved length lower bounds for reflecting sequences*. In *Computing and Combinatorics*, volume 1090 of *Lecture Notes in Computer Science*, pages 56-67. Springer Berlin Heidelberg, 1996.

A. Borodin, W.L Ruzzo et al, *Lower bounds on the length of universal traversal sequences*, *Journal of Computer and System Sciences*, 45(2):180-203, 1992

J. Chalopin, S. Das, P. Gawrychowski, A. Kosowski, A. Labourel, and P. Uznanski, *Lock-in Problem for Parallel Rotor-Router Walks*, *Researchgate*, July 2014

D. J Watts, Small Worlds-*The Dynamics of Networks Between Order and Randomness, Princeton University Press*, Princeton, NJ, 1999

6 | Appendix

Mobile agent: A bot that contains data & migrates from node to node within a network

Migration: Allows bots to move from server to server, node to node.

Deterministic algorithm: A type of algorithm that, given a certain input *x*, will always produce the same output *y*. The same sequence of states will be consistently passed, and make this kind of algorithm reliable.

Patrolling: The ongoing exploration of a network by a decentralized group of simple memoryless robotic agent(s).

Euler cycle: A cycle where each and every edge is passed once in an undirected graph.

Extended Euler cycle: Each and every edge is passed the same number of times.

Degree: The number of edges connected to a vertex in an undirected graph.

Arc: Another term for edge, commonly referred to as an edge in a directed graph.

Blanket time: Amount of time a path is not visited by the mobile agent.

Cover time: Amount of time it takes for a mobile agent to visit every edge in a graph.

Lock-in time: Amount of time it takes for a mobile agent to start an unchanging sequence of movements that result in an overlying eulerian cycle.

Diameter: The largest number of vertices that must be traversed in order to travel from one vertex to another vertex. Looping, backtracking, and detours are ignored.

Small world graph: A relatively sparse graph with a small number (10%) of high degree vertices, which are called **hubs**. Similar to real-world communication networks.

Star graph: A graph where one node has a degree of *n-1*, and the rest of the nodes have a degree of 1.

Anonymous graph: A graph that has vertices that do not have identifiers. Unlike the graph in **Figure 1**, there would be no vertex name (like vertex *A*, *B*, etc.)

Hamiltonian graph: A graph that contains one (or more) Hamiltonian path, which is a traceable path that visits each vertex of a graph exactly once.

Stabilization period: The number of steps it takes for a mobile agent to lock into a forever-repeating Euler cycle.

Perfect average: The average visit frequency for an edge/vertex given that the stabilization period is equal to 0; if the mobile agent started traversal and within 0 steps was locked-in, the average visit frequency would be split evenly (for each complete cycle)

Outgoing arc: Edge in a directed graph that starts at a vertex, points to another vertex.

Finite state machine: Machine (of any size) that can store the status of something at a given time and operate on input to change status.

2-regular graph: A graph where every vertex has a degree of 2. A **regular graph** is a graph where every vertex has the same degree.

Border vertex: A vertex whose set of outgoing arcs contains both traversed and non-traversed arcs. A border vertex must have at least 2 outgoing arcs.

Latency: The interval of time between when data is requested by the system and when the data is provided by system.

Subcycle decomposition: A periodic behavior of parallel rotor-router walks that involves a structural property of different stable states. In such states, agents patrol the arcs of a graph, and two agents in different states do not traverse the same arc.

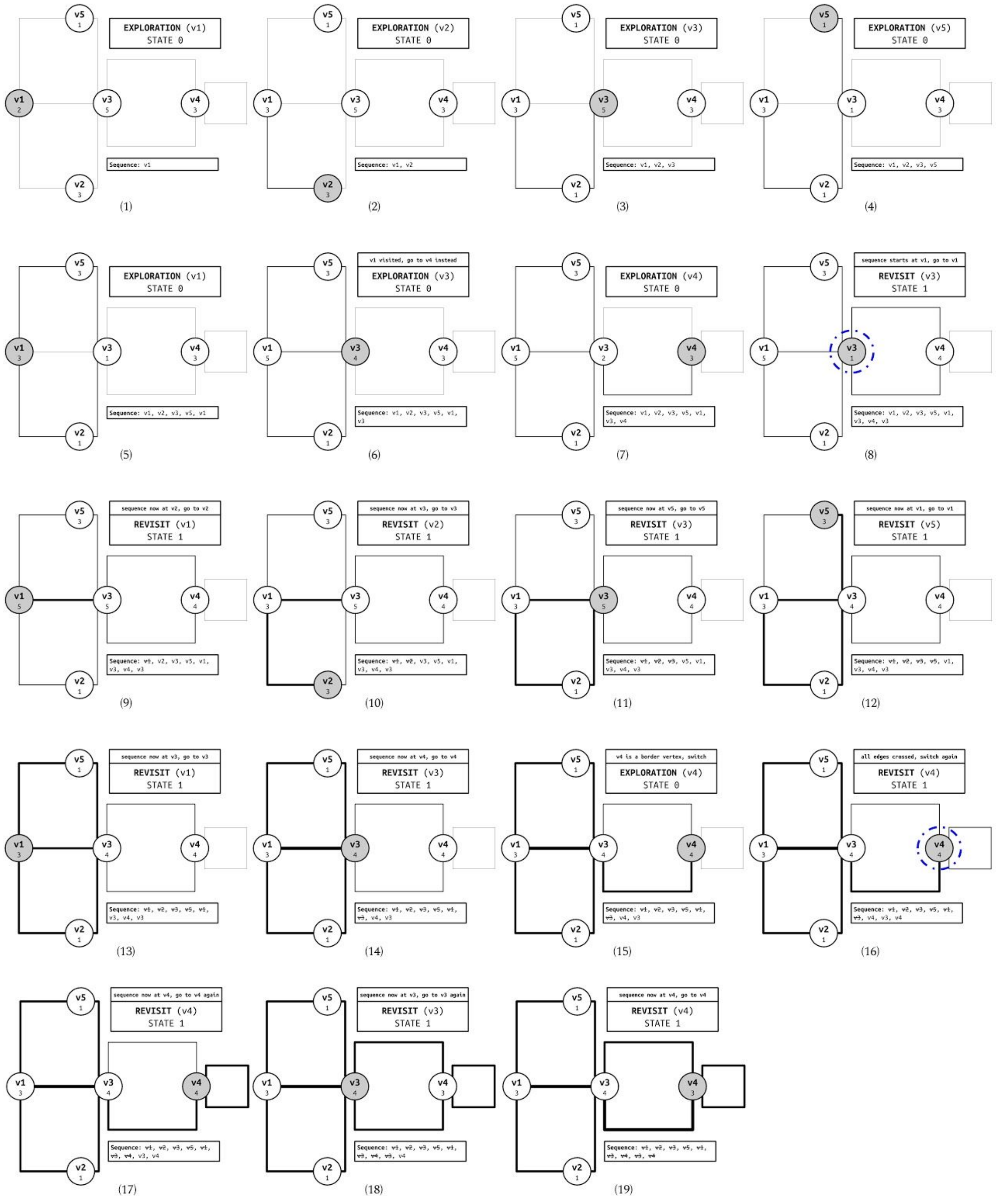
Team patrolling problem: A task in which the goal is to periodically and regularly traverse all edges of a graph with *k* agents.

RotorRouterAlgorithm **Implementation Function** (*mobile agent movement*)

```
int mobileAgent(Graph graph, Vertex* start, int iterations, string& temp) {
  Vertex* destination = NULL;
  for (int i = 0; i < iterations; i++) {
    int currentIndex = start->getIndex();
    start->edgeToCross(start, currentIndex)->incrementVisitCount();
    Edge* cross = start->edgeToCross(start, currentIndex);
    destination = cross->getEndVertex();
    cout << "[Step " << i+1 << "] Agent crosses the EDGE " <<
    cross->getStartVertex()->getName() << cross->getEndVertex()->getName()
    << " and is at VERTEX " << cross->getEndVertex()->getName() << endl;
    graph.insertEdgeToOverall(cross);
    destination->incrementVisited();
    start->changeIndex();
    start = destination;
  }
  int stabP = findEulerCycle(graph, temp);
  cout << "  Stabilization Period: " << stabP << endl;
  return stabP;
}
```

Object, Variable Descriptions	***Function Descriptions***
`Graph`: Contains vectors that hold vertices, edges, the overall edge traversal history of an agent, and the Euler cycle that repeats after the stabilization period ends.	`getIndex()`: Return the *next exit value* for a certain vertex (pointer).
`Vertex*`: Contains a vector of edge pointers, a name, index, visit count, and various functions that allow the rotor-router mechanism to work.	`edgeToCross(start, currentIndex)`: Find and return the edge to cross, depending on the *currentIndex* value.
`Edge*`: Contains a starting vector and end vector, a name, visit count, and id, which is used for creating reliable data not only for arcs, but undirected edges as well.	`incrementVisitCount()`: Increase the number of times a vertex has been visited by the mobile agent by one.
`iterations`: *S*, the total number of steps.	`getStartVertex()`: Return the starting vertex that the mobile agent is at.
`temp`: This is a string that will be used in `findEulerCycle(graph, temp)` to find the lock-in Euler cycle.	`getEndVertex()`: Return the vertex that the mobile agent will end up at.
`currentIndex`: This is the current index of the vector that the mobile agent is on, and its value determines what edge the agent will traverse next.	`getName()`: Return the name of the vertex (this is a string variable).
`cross`: Edge that the agent will cross.	`insertEdgeToOverall(cross)`: Insert this edge into an **overall** vector that saves the order the mobile agent traverses in.
`stabP`: The stabilization period, which will be found and returned in the end.	`changeIndex()`: Change the index according to (**k**+**1**) **mod degree**(***v***).
	`findEulerCycle(graph, temp)`: This function finds the Euler cycle that the mobile agent ends up traversing forever.

Figure 3 Full (*Step by Step*)


EXPLORATION (v1) STATE 0
Sequence: v1
(1)
EXPLORATION (v2) STATE 0
Sequence: v1, v2
(2)
EXPLORATION (v3) STATE 0
Sequence: v1, v2, v3
(3)
EXPLORATION (v5) STATE 0
Sequence: v1, v2, v3, v5
(4)
EXPLORATION (v1) STATE 0
Sequence: v1, v2, v3, v5, v1
(5)
v1 visited, go to v4 instead
EXPLORATION (v3) STATE 0
Sequence: v1, v2, v3, v5, v1, v3
(6)
EXPLORATION (v4) STATE 0
Sequence: v1, v2, v3, v5, v1, v3, v4
(7)
sequence starts at v1, go to v1
REVISIT (v3) STATE 1
Sequence: v1, v2, v3, v5, v1, v3, v4, v3
(8)
sequence now at v2, go to v2
REVISIT (v1) STATE 1
Sequence: ~~v1~~, v2, v3, v5, v1, v3, v4, v3
(9)
sequence now at v3, go to v3
REVISIT (v2) STATE 1
Sequence: ~~v1~~, ~~v2~~, v3, v5, v1, v3, v4, v3
(10)
sequence now at v5, go to v5
REVISIT (v3) STATE 1
Sequence: ~~v1~~, ~~v2~~, ~~v3~~, v5, v1, v3, v4, v3
(11)
sequence now at v1, go to v1
REVISIT (v5) STATE 1
Sequence: ~~v1~~, ~~v2~~, ~~v3~~, ~~v5~~, v1, v3, v4, v3
(12)
sequence now at v3, go to v3
REVISIT (v1) STATE 1
Sequence: ~~v1~~, ~~v2~~, ~~v3~~, ~~v5~~, ~~v1~~, v3, v4, v3
(13)
sequence now at v4, go to v4
REVISIT (v3) STATE 1
Sequence: ~~v1~~, ~~v2~~, ~~v3~~, ~~v5~~, ~~v1~~, ~~v3~~, v4, v3
(14)
v4 is a border vertex, switch
EXPLORATION (v4) STATE 0
Sequence: ~~v1~~, ~~v2~~, ~~v3~~, ~~v5~~, ~~v1~~, ~~v3~~, v4, v3
(15)
all edges crossed, switch again
REVISIT (v4) STATE 1
Sequence: ~~v1~~, ~~v2~~, ~~v3~~, ~~v5~~, ~~v1~~, ~~v3~~, v4, v3, v4
(16)
sequence now at v4, go to v4 again
REVISIT (v4) STATE 1
Sequence: ~~v1~~, ~~v2~~, ~~v3~~, ~~v5~~, ~~v1~~, ~~v3~~, ~~v4~~, v3, v4
(17)
sequence now at v3, go to v3 again
REVISIT (v3) STATE 1
Sequence: ~~v1~~, ~~v2~~, ~~v3~~, ~~v5~~, ~~v1~~, ~~v3~~, ~~v4~~, ~~v3~~, v4
(18)
sequence now at v4, go to v4
REVISIT (v4) STATE 1
Sequence: ~~v1~~, ~~v2~~, ~~v3~~, ~~v5~~, ~~v1~~, ~~v3~~, ~~v4~~, ~~v3~~, ~~v4~~
(19)

OneBit **Pseudocode Implementation** (Menc 2015)

```
def ONE-BIT(Vertex v):
      if pointer(v) == 2deg(v) then:
            pointer(v) = 0
            STOP()
      end if

      if (agentmode == 0) && (pointer(v) <= deg(v)) then:
            shift(v) = pointer(v)
      end if
      pointer(v) = pointer(v) + 1
      if (pointer(v) >= 1 && pointer(v) <= deg(v)) then:
            agentmode = 1
            tmp = pointer(v)
      else:
            agentmode = 0
            tmp = pointer(v) + shift(v)
            if ((pointer(v) == 2deg(v)) && (shift(v) > 0)) then:
                  shift(v) = 0
                  pointer(v) = 0
            end if
      end if
      PORT(tmp)
end function
```

Variable, Function Descriptions

`agentmode`: Stores whether the agent is in EXPLORATION or REVISIT MODE.

`tmp`: Stores port number, which is then placed in **`PORT`** to determine the next path.

`shift(v)`: Stores value of `pointer(v)` so that future explorations in REVISIT MODE from vertex *v* will start from the (next) port, rather than port 1.

`pointer(v)`: Stores the number of visits in vertex *v*.

```
int quasiWalk(int steps, Graph g, Vertex* s, double resultsV[], double
      resultsE[], StatList stats, string& temp) {

      int graphVertexCount = g.getVertexCount();
      int currentEdgeID = 0;  //The ID of an edge (for undirected edges)
      int stabilization = 0;  /*The result from mobileAgent gets placed in
                                this variable*/
      Edge* currentEdge = NULL;

      //Calculate the stabilization period
      stabilization = mobileAgent(g, s, steps, temp);

      //Obtain data on visit counts (edge/vertex)
      //and clear all visit counts for the next iteration to occur
      for (int i = 0; i < graphVertexCount; i++) {
      resultsV[i] += g.returnVertex(i)->getVisited();
      for (int j = 0; j < g.returnVertex(i)->getEdgeCount(); j++) {
            currentEdge = g.returnVertex(i)->returnEdge(j);
            currentEdgeID = currentEdge->getID();
            resultsE[currentEdgeID] += currentEdge->getVisitCount();
            currentEdge->resetVisitCountEdge();
            currentEdge->resetCycleCountCheck();
      }
      g.returnVertex(i)->setVisitToZero();
      g.returnVertex(i)->resetAllEdgeVisits();
      }
      return stabilization;
}

int findEulerCycle(Graph graph, string& temp) {
      //This function finds the lock-in Euler cycle for a single test
      int i = 0, currentCheck = 0, newCounter = 0, stabilizationPeriod = 0;
      Edge* startingEdge = NULL;
      int startingIndex = 0, edgesTraversed = 0;
      while (true) {
         //Check if the edge has been visited before (in this function)
         currentCheck = graph.returnEdgeFromOverall(i)->getCheck();
         //Increase the number of edges traversed by one
         edgesTraversed++;

      //If the edge has been visited once & we are at the graph edge count
```

```
        if (currentCheck == 1 && edgesTraversed == graph.getEdgeCount()+1) {
              //At this point, the agent has crossed every edge once
              //We can break out of the loop and record the Euler cycle
              break;
        }
        //An edge from earlier has been met again; we started midway through
        //a cycle and need to restart our search
        else if(currentCheck == 1&&edgesTraversed != graph.getEdgeCount()+1) {
              //Reset all visit counts for each edge (in this function) to 0
              for (int j = 0; j < graph.getOverallCount(); j++) {
                    graph.returnEdgeFromOverall(j)->resetCycleCountCheck();
              }
              //Increase the point where we start counting from
              startingIndex++;
              i = startingIndex-1;
              edgesTraversed = 0;
        }
        //If the edge hasn't been visited, increment by 1; mark it visited
        else if (currentCheck == 0) {
                 graph.returnEdgeFromOverall(i)->incrementCheck();
        }
           i++;
        }
        //Get the edge from the overall history and set it to startingEdge
        startingEdge = graph.returnEdgeFromOverall(startingIndex);
        //The stabilization period is how many steps away from 0 the
        //Euler cycle's starting edge is.
        stabilizationPeriod = startingIndex;
        while (newCounter != graph.getEdgeCount()) { //m edges long
           //From the startingIndex, add each edge to the euler cycle vector
           graph.insertEdgeToCycle(startingEdge);
           startingIndex++;
           startingEdge = graph.returnEdgeFromOverall(startingIndex);
           newCounter++;
        }
        for (int k = 0; k < graph.getEdgeCount(); k++) {
           //Add the edge name to the temp, then an arrow to indicate movement
           temp += graph.returnEdgeFromCycle(k)->getName();
           temp += "->";
        }
        cout << endl;
        return stabilizationPeriod;
}
```