IONIAN UNIVERSITY
DEPARTMENT OF INFORMATICS

Thesis

Random Walker simulations in Sensor Networks using OMNET++

Κωνσταντίνος Γαρείος

Supervisor: Ελένη Χριστοπούλου

Corfu, 2025

ΙΟΝΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Πτυχιακή Εργασία

Προσωμοιώσεις Τυχαίων Περιπατητών στην OMNET++

Κωνσταντίνος Γαρείος

Επιβλέπουσα: Ελένη Χριστοπούλου

Τριμελούς Επιτροπή: Ελένη Χριστοπούλου,
Κωνσταντίνος Σκιαδόπουλος,
Κωνσταντίνος Οικονόμου

Κέρκυρα, 2025

Περίληψη

Η παρούσα εργασία ερευνά το μοντέλο κίνησης του τυχαίου περιπατητή (Random Walker) στο περιζήτητο θέμα των Δικτύων αισθητήρων.
Τα Ασύρματα Δίκτυα Αισθητήρων (WSN) είναι τομέας μεγάλου ενδιαφέροντος με πολυάριθμες εφαρμογές στην ασφάλεια του πολίτη από φυσικές καταστροφές (πυρκαγιές, σεισμοί, ρωγμές σε έργα υποδομής), στρατιωτικής ασφάλειας και ευρύτερης παρακολούθησης του περιβάλλοντος (smart city / home, WSSN).

Συνεπώς, το μοντέλο διάχυσης πληροφορίας δύναται να καθίσταται κύριος παράγοντας για να είναι το δίκτυο σε θέση να αντιμετωπίσει προληπτικά και αποτρεπτικά αυτές τις καταλυτικές απειλές.

Η μελέτη αξιοποιεί την πλατφόρμα προγραμματισμού και προσομοιωτή δικτύων Omnet++ για την συναγωγή μετρικών των μοντέλων, καθώς και αναπαράγονται σχήματα για όλα τα αποτελεσμάτα. Τα μοντέλα που ερευνώνται είναι ο μονός τυχαίος περιπατητής, οι πολλαπλοί τυχαίοι περιπατητές καθώς και τρείς παραλλαγές του μοντέλου. Οι παραλλαγές του τυχαίου περιπατητή διακρίνονται: α) στον αλγόριθμο κλωνοποίησης, που αξιοποιεί διάστημα διπλασιασμού του πράκτορα, β) στον αλγόριθμο απαγόρευσης επιστροφής (no-backtracking), που απαγορεύει την άμεσα επανειλημμένη επίσκεψη του πράκτορα στον προηγούμενο κόμβο του δικτύου, και γ) στο υβριδικό μοντέλο που συνδυάζει αυτές τις δύο παραλλαγές του αλγορίθμου.

Ταυτόχρονα, εστιάζεται στα πλεονεκτήματα του τυχαίου περιπατητή έναντι άλλων μοντέλων διάχυσης πληροφορίας όπως η χαμηλή κατανάλωση ενέργειας, ευελιξία, και μη ντετερμινιστική συμπεριφορά που διεξάγει. Η συμπεριφορά του πράκτορα αναλύεται σε διάγραμμα ροής και βίντεο κάλυψης ενός συμμετρικού αμερόληπτου δικτύου "πλέγματος" με ίσες και παράλληλες αποστάσεις συνδέσεων.

Στην ενότητα ανασκόπησης αναφέρονται τα μοντέλα μετρικής για την ενεργειακή κατανάλωση. Στο μοντέλα της κλωνοποίησης και υβριδικής διεξαγωγής ορίζονται οι αντίστοιχες ευρετικές συναρτήσεις, καθώς και εγκαθίσταται μία πρωτότυπη μεταβλητή "rc relative" rc_r "σχετική ακτίνα εμβέρλειας", η οποία επιτρέπει την σύγκριση μετρικών της ακτίνας εμβέλειας για δίκτυα διαφορετικού εμβαδού, ορίζοντας την ως ποσοστό της γεωμετρικής διαγωνίου του πεδίου.

'Ολοι οι αλγόριθμοι και παραλλαγές εφαρμόστηκαν σε δίκτυα RGG τοπολογίας τυχαίου γεωμετρικού γράφου, όπως και σε πραγματικά δεδομένα του ΟΑΣΑ για δικτύο από σημεία στάθμευσης ταξί στην Αθήνα.

Τα πειράματα αποτελούνται από συγκρίσεις ελεγχόμενων δοκιμών μέσου όρου 500 προσομοιώσεων. Στο αποτέλεσμα σύγκρισης των 3 παραλλαγών και του μονού τυχαίου περιπατητή, παρατηρείται ότι το υβριδικό μοντέλο

είχε την καλύτερη απόδοση χρόνου κάλυψης και κατανάλωσης ενέργειας. Το μοντέλο κλωνοποίησης ήταν δεύτερο στον χρόνο κάλυψης καθώς και στην εξοικονόμηση ενέργειας, με εξαίρεση τις περιπτώσεις που είχε ατελέσφορο διάστημα διπλασιασμού. Στην τρίτη θέση καταγράφεται ο αλγόριθμος no-backtracking, ο οποίος είχε και επιπρόσθετο πείραμα σύγκρισης με τον μονό περιπατητή, δεν είχε παρουσίασε κανένα μειονέκτημα, και πιθανότατα ανεξαρτήτως των ευρετικών που αξιοποιεί η εν λόγω μελέτη. Ο αργότερος αλγόριθμος ήταν ο μονός τυχαίος περιπατητής, με εξαίρεση του κλωνοποιημένου, ύπο ορισμένες προϋποθέσεις.

Σε επιπλέον πείραμα, το μοντέλο κλωνοποίησης τυχαίου περιπατητή παρήγαγε καλύτερα αποτελέσματα χρόνου και ενέργειας συγκριτικά με τους δύο τρέχοντες τυχαίους περιπατητές.

Συγκρίνοντας τα δύο δίκτυα, βάσει αρχικής υπόθεσης, οι παραλλαγές είχαν βελτιωμένα αποτελέσματα ενέργειας για τα πειράματα του δικτύου ταξί, πιθανότατα λόγω μεγαλύτερης κλίμακας.

Αρχική υπόθεση της έρευνας ισχυρίζοταν πως οι πολλαπλοί περιπατητές θα δημιουργούσαν απώλεια ενέργειας λόγω επιπλέον βημάτων και όξυνσης γειτονικών προβλημάτων. Τα αποτελέσματα έκριναν πως ο χρόνος κάλυψης υποχώρισε γραμμικά, με αποτέλεσμα η κατανάλωση ενέργειας να είναι παρόμοια για κάθε διακριτή προσομοίωση.

Τέλος, για διαφορετική υπόθεση στην επιρροή της ακτίνας εμβέλειας, έγινε σύγκριση στα δύο δίκτυα αξιοποιώντας προσομοιώσεις στην Python. Τα αποτελέσματα ενέκριναν την θεωρία και στα δύο δίκτυα.

Contents

1 Introduction

1.1 Main

Wireless Sensor Networks (WSNs) have been widely adopted in various fields, monitoring the environment and cleverly presenting relevant information to the end-user. It's essential to focus on which models achieve this objective best, as this work compares, finds and simulates different information dissemination algorithms, focusing on several adaptations that feature the Random Walker mobility model.

1.2 Purpose

This paper presents the results of experiments that test different adaptations of the random walker. This includes multiple concurrent random walkers, the cloned random walker, the no-backtracking random walker, and a hybrid model combining the former two, all simulated in the Omnet++ programming environment. Additionally, the said algorithms were analyzed and compared on real-world data of a Greek Athens' taxi stand network.

This study offers groundwork, functional open source code of Omnet++ experiments, and a methodology for quantifying energy consumption in random walker WSNs, including some of its adaptations. Among several performance metrics, this work focuses primarily on energy consumption and coverage time, both critical for evaluating the efficiency of information dissemination in networks.

1.3 System components

1.3.1 Sensor networks

Sensor networks function to measure the environment. Generally, in the form of wireless sensor networks (WSNs), sensors transmit their data to a powerful centralized Base Station/Gateway/Sink device, which will export all of the network quantification data of interest to a cloud system that can then utilize it (This works similarly to the definition described by Hu & Evans in 2003).[1]

1.3.2 The sensors

The sensors in a Wireless Sensor Network are low-cost devices that typically consist of an antenna, a microcontroller, a battery, and an electronic circuit. They have little computational ability and memory. For the purpose of preserving the battery -and by extension, the overall network lifetime- they should trasmit data

sparingly using small messages. If any computation is run, it should aim to either preserve energy or facilitate transmission. Sensors communicate wirelessly using their antenna of limited range. Its reach determines not only the topology, but also the energy consumption of the network, which translates to the longetivity of the system.

1.3.3 The data

Sensors usually monitor temperature, sound, motion, distance, pressure, humidity, wind, and pollution. The message is typically a 32-bit floating-point value, accompanied by metadata including sensor location, time stamps, sensor ID, and protocol-specific overhead [2]. While such measurements represent the core information of interest, the experiments conducted herein also involve auxiliary data—both cached and functional—maintained by the random walker agents and the sensors themselves to support network operations.

1.4 Significance of WSNs

Wireless Sensor Networks (WSNs) have become an indispensable technology across many domains due to their cost-effectiveness and efficiency in sensing and data communication. Their strategic importance is especially evident in military applications, where they provide a rapid, low-cost means for real-time environmental monitoring and battlefield awareness. Writers Raj et al. argue that WSNs represent a critical investment in modern military technology, in response to the current climate of rising military expenditures, thereby underscoring the relevance of research on this topic. [3].

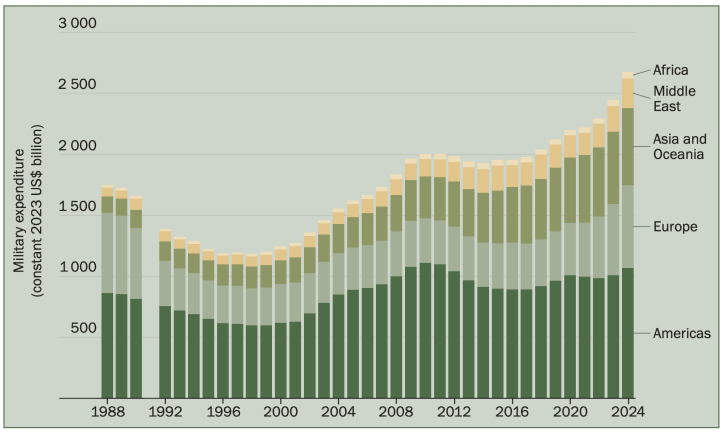

Figure 1: Military expenditures - constant 2023 US$ billion

A major project employed in the 1990s named 'SOSUS' leveraged WSN technology during the Cold War. It deployed wireless sensor systems in the Pacific Ocean that would monitor U.S. interests by intercepting USSR transit. Beyond military applications, the WSN industry is experiencing rapid growth, WSNs stand within a booming period having been integrated with Wireless Smart Sensor Networks (WSSN). "Research and Markets" forecast a compound annual growth rate (CAGR) of 14.82% between 2024 and 2032. [4]

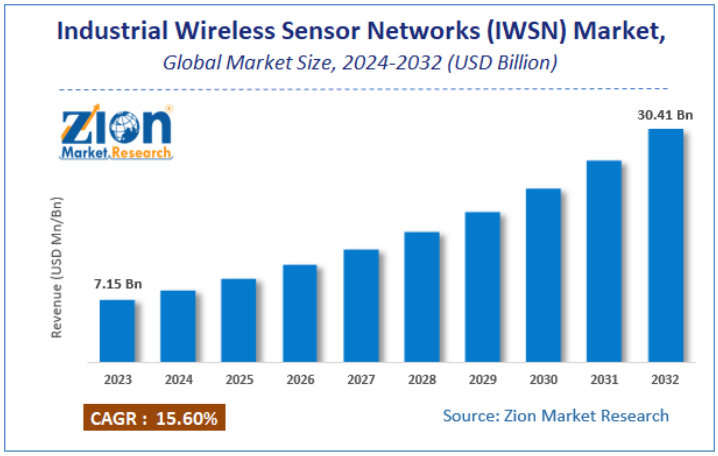

Figure 2: Market projection for the WSN industry 2024-2032

In infrastructure, WSNs are instrumental in structural health monitoring. Bridges utilize sensors to detect early anomalies, damages, bendings, cracks. [5]

Other WSNs are employed in tunnels, to detect earthquakes, and floods. While in places closer to the equator, the focus shifts to wildfire detection. [6]

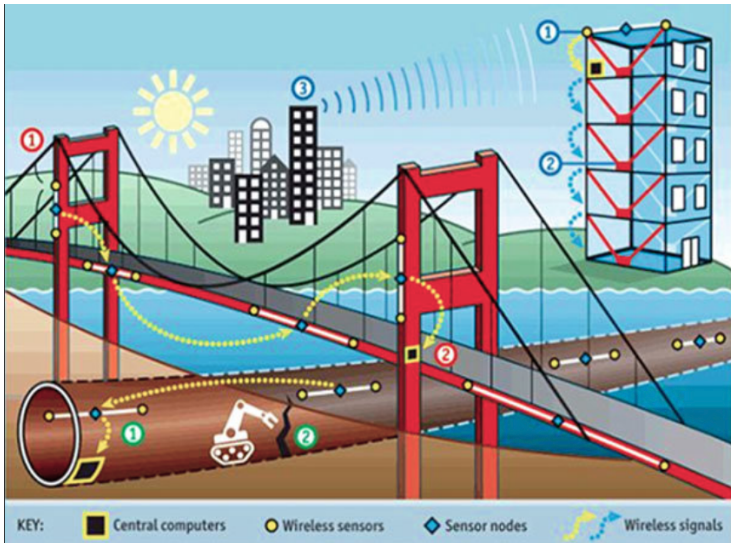

Figure 3: WSN sensors on a bridge | Illustration made by Haque et al.[7]

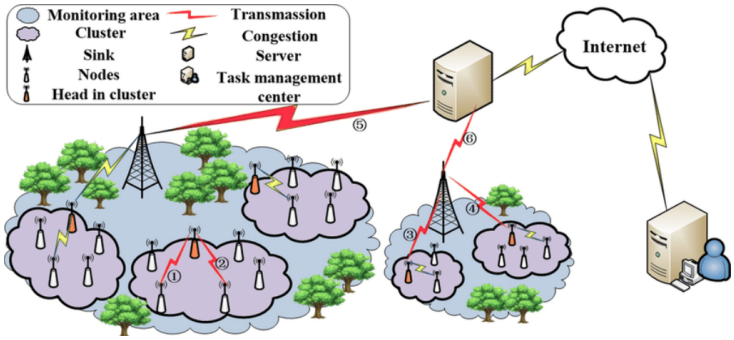

Figure 4: WSN sensors for wildfire detection | Illustration made by Suoping et al.[6]

By and large, these applications collectively demonstrate the versatile and vital role this type of research can have in enhancing civilian safety, security, and day-to-day efficiency.

1.5 The Random Walker

Definition

A Random walker is an agent that simulates the *Random Walk* process, a fundamental stochastic model first described by Karl Pearson. [8]

In the context of WSNs, random walkers are viewed as an edge case of *Markov chains* as explained by Randal et al.'s *"Markov Chains"*. [9]

At each step, a random walker selects its next move uniformly at random from the set of available options, reflecting a process of maximum uncertainty/entropy. This behavior is often illustrated by the concept of Brownian motion.

Why use Random Walkers in a network?

Random walkers have several attractive features that make them particularly useful in wireless sensor networks:

- **Low complexity and adaptable:**

  Random walkers require minimal memory and computational resources, making them ideal for resource-constrained devices. Their simplicity also allows easy integration into hybrid or dynamic network mobility models. If the WSN system requires for an inexpensive solution random walkers are amongst the most effective choice. Its low complexity allows for low-powered devices such as Raspberry Pis to run this model with assurance and consistency.

- **Energy:**

  Random walkers necessitate little energy usage due to low overhead and low functional requirements. Since the host devices run on little battery, which by extent translates to the network's lifetime, it's essential to neglect extravagant models that are wasteful. Contemporary algorithms adjacent to "AI", that require whole batches of machine learning data, are too excessive for WSNs. A simple model like the Random Walker is ideal for this environment. Curtailing costs, while enabling long-term network function.

- **Dynamic:**

  Unlike deterministic models, where a change in the network's topology -caused by a node joining or leaving- breaks the functionality, Random walkers allow the network to subsist. Consequently, processes that restructure the algorithm to adapt to various instances or new networks,

which are typical in traditional deterministic models, don't constitute a desideratum for this model since the next movement will advance as is.

- **Apt for distributed systems:**

  Ad-hoc systems like WSNs function in a distributed procedure, without centralized control. Random walkers, which require no global information, offer load balanced dissemination of data [11].

- **Scalable:**

  Although a single random walker can be slow in covering large networks, various adaptations—such as the "Jump," "Clone," and "No Backtracking" methods—aim to reduce coverage time by avoiding bottlenecks and sparse regions. For detailed analysis on scalability in random geometric graphs, see this work by Tzevelekas et al. [12]

2 Literature Overview

2.1 Network Model

This methodology follows common standards for quantifying WSN metrics, as they refer to graph theory. Wireless Sensor Networks are modeled as Graphs composed of Vertices (commonly mentioned as "nodes") and Edges (indicating the radius and neighbors of a vertex). For graph G, vertex V, and edge E, a network is denoted as:

$$G(V, E)$$

All nodes in the network, can only communicate through their wireless antenna. This is referred to as the "communication radius" or rc value which represents the maximum range of each sensor's wireless antenna. It is a critical network parameter, as it governs the network's topology, density, and energy consumption. Any sophisticated review of the network's performance should justify a particular rc to be used.

If the network is bi-directional, the graph is undirected as well. To express the "neighborhood" of a node, we will be using the standard of **Dominating Sets** [13]. According to which, a vertex v dominates itself, and all its neighbors N(v), that it shares an edge with. This creates a subset of the network, the dominating set of V ($dom(v)$), where:

$$dom(v) = \{v\} \cup N(v) \tag{1}$$

A similar and more widespread metric is the connectivity degree $\delta(v)$, which instead refers to the neighborhood of the node, without accounting for itself. Thus: $\delta(v) = dom(v) - 1$.

A common point of contention is how the density of the network affects its coverage. For this reason we define **Cardinality**, as it has proven useful in this work of random walker city density by Natapov et al. [14]. Cardinality conveys the length of a Dominating Set, which extends to the density of its neighborhood:

$$C(u) = |dom(u)| \tag{2}$$

The network has a Cardinality range, ranging from the minimum number of neighbors a vertex incorporates, to the maximum amount of neighbors a vertex can afford in the network. Supposing that v* is the vertex of the most amount of neighbors in the graph:

$$|C(v*)| = |max(N(v)|, u \in V \tag{3}$$

Random walkers will follow a uniform function to latch on to the next node. Hence, within the G(V,E) network, assuming that the agent's current node is denoted as $v \in V$, the agent will choose the following node u with probability:

$p_u = 1/(\delta(v)), (v, u) \in E..$ (Tzevelekas et. al establish a similar discrete probability formula: [12])

2.2 Random walker behavior

The position of a random walker after multiple steps approximately follows a normal (Gaussian) distribution, owing to the Central Limit Theorem. In a one-dimensional random walk, the distribution of the walker's end-point after many steps resembles a bell curve, as illustrated in Figure 5.

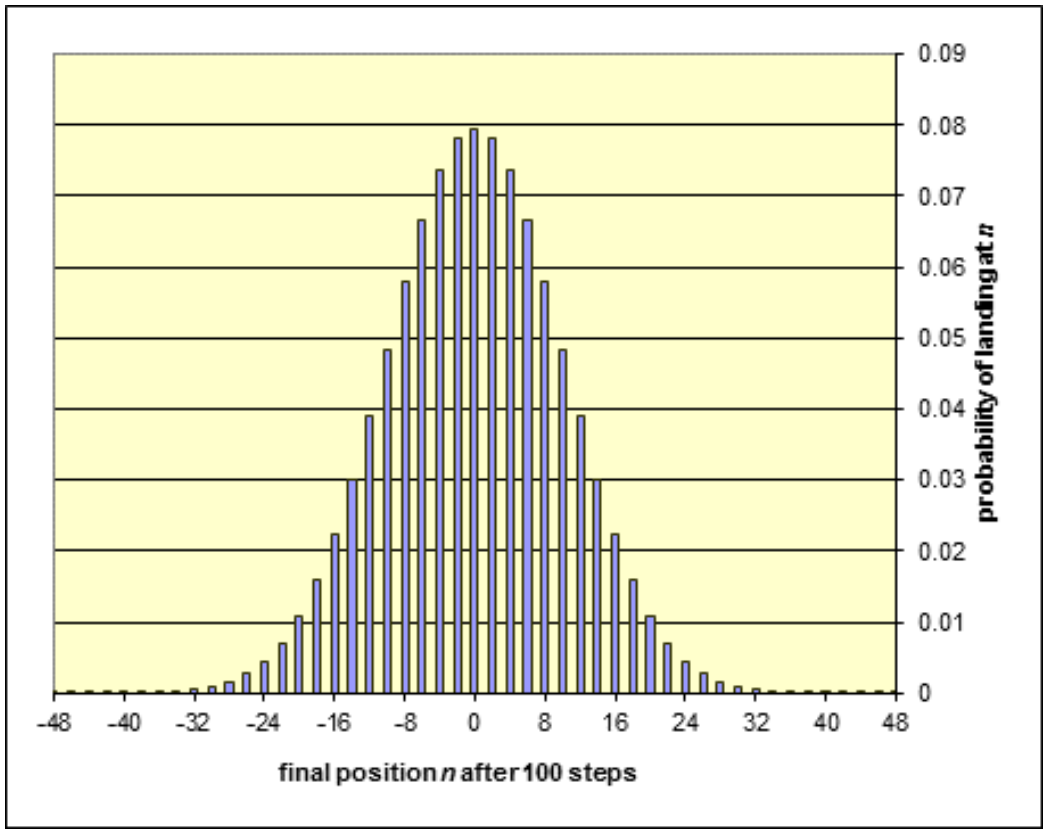

Figure 5: Distribution of end-point position of a random walker in One-Dimensional space [15]

In higher dimensions—two or three—the random walk behavior is more complex. Each coordinate axis (x, y, and possibly z) is treated as an independent random variable, each following a normal distribution. The overall distance from the starting point in two dimensions follows a Rayleigh distribution, which K.L. Besser references. [16]. Other works like Codling's et al. describe it as multiple

independent normal distributions for each axis, with the assumption that they are **isotropic** (directionally symmetric).[17]

Thus, the starting position of the walker within the network greatly influences coverage efficiency. Initiating the random walker near the dense core of the network increases the likelihood of faster coverage since the walker has more neighboring nodes to explore evenly. Conversely, starting near the network's edge or diagonal results in biased mobility toward that region, leaving opposite areas less covered. This spatial bias can exacerbate coverage time, especially in networks with bottlenecks.

The walker's network coverage is expressed with the function $C(T)$, ranging from 0 to 1.

The random walker this study engages with is a discrete-time agent whose behavior is illustrated in the flowchart of figure 6.

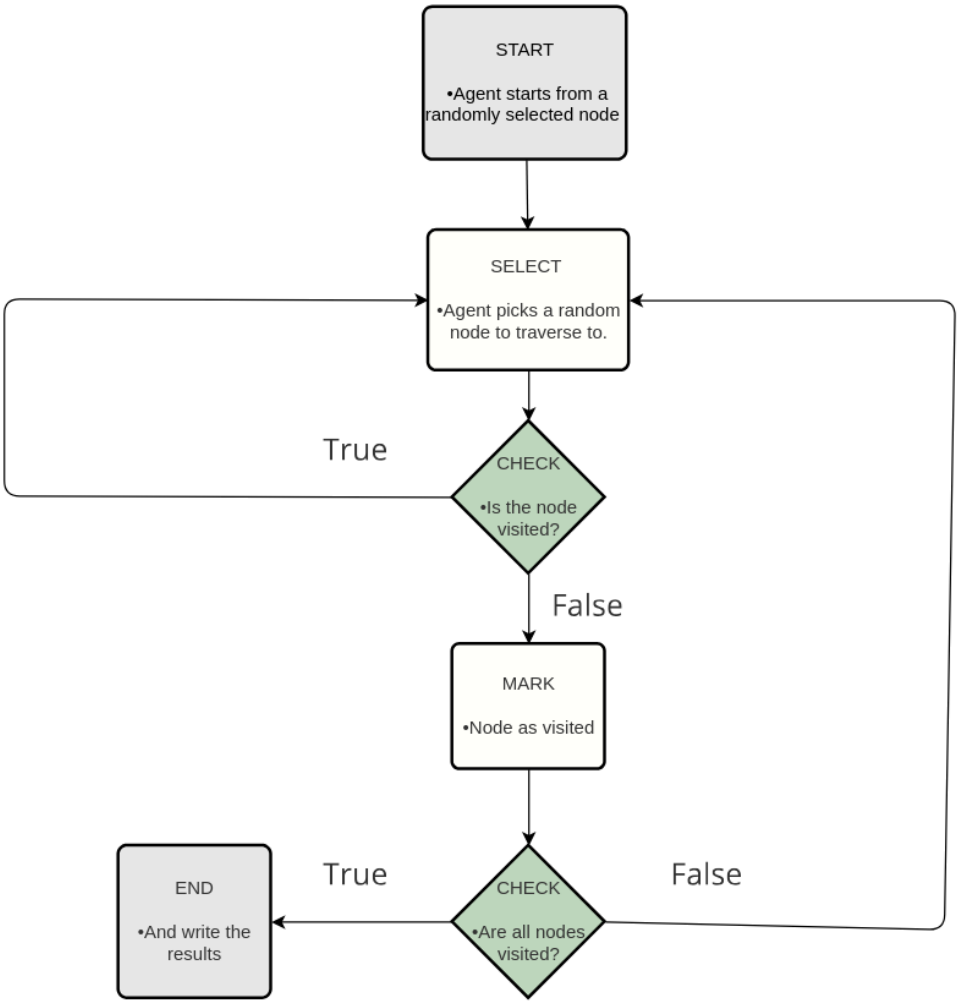

Figure 6: Flowchart of the random walk model

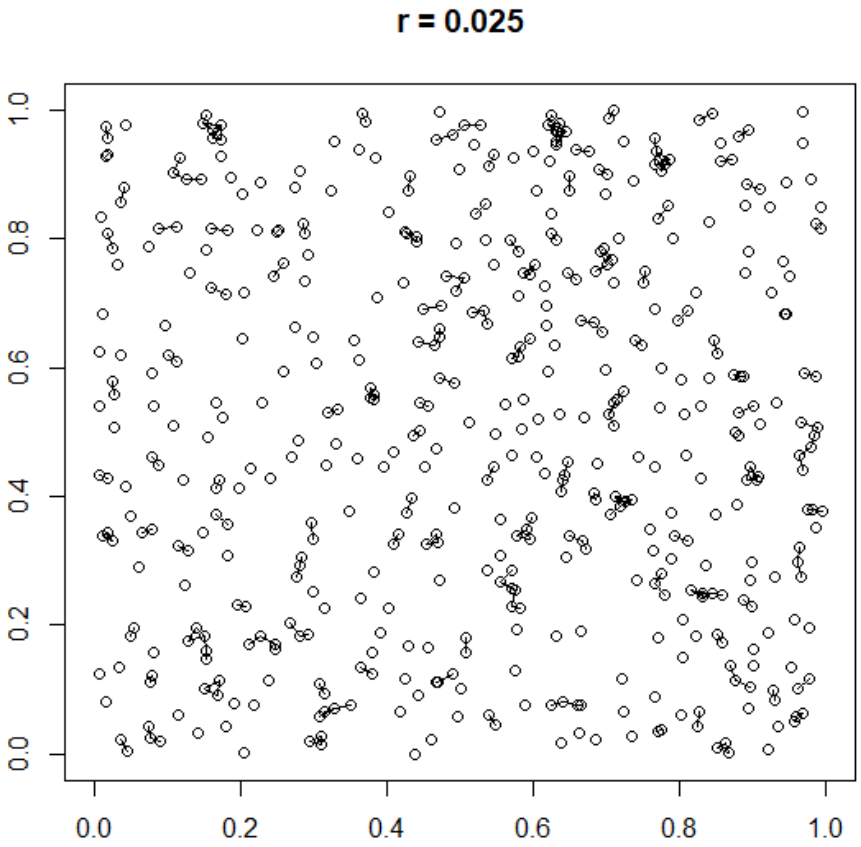

Figure 7: Random Geometric Graph's topology as the Connection Radius increases | Work by Morris Kurz

2.3 Random Geometric Graphs

Since WSNs are typically ad-hoc networks, Random Geometric Graphs are the most useful representative topology to make qualitative assessments. Nodes are randomly placed in a 2-dimensional Euclidean space and are connected by an edge if their Euclidean distance is less than or equal to a fixed connection radius rc. This work by Morris Kurz on figure 2.3 illustrates how the topology in an RGG network transforms as the rc increases (gif mirror available at: `https://github.com/Gkarios/Random-Walks-Omnet/blob/main/REPORT-GIFS/Random_Geometric_Graph.gif`). All 2D RGG networks need to

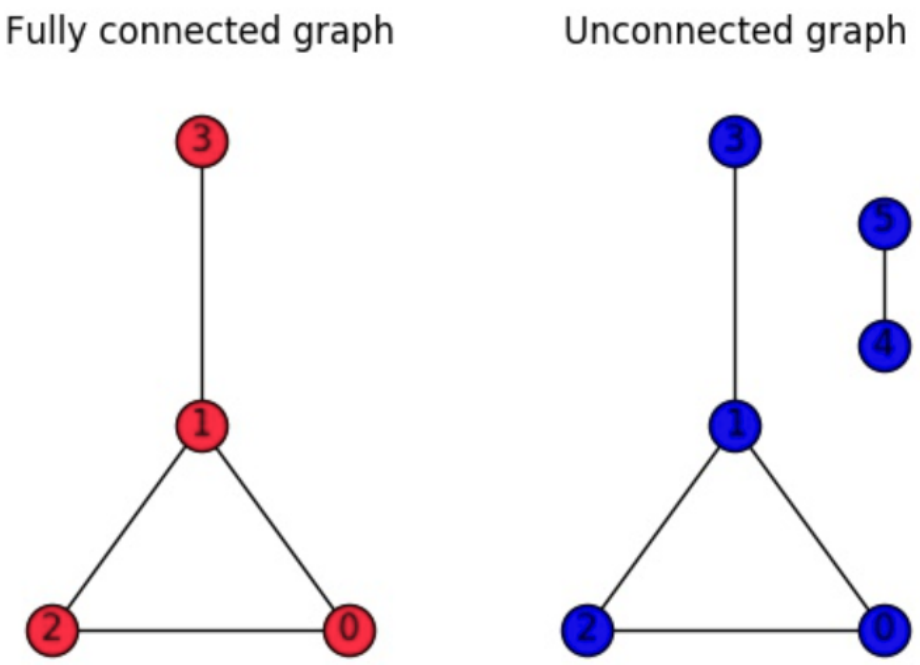

Figure 8: Connected vs Unconnected Graphs | Work by Jessica Walkenhorst

check for connectivity using Euclidean distance:

$$d(u, v) = \sqrt{(x_u - x_v)^2 + (y_u - y_v)^2} \tag{4}$$

All networks possess the concept of being connected or unconnected. This simply refers to whether or not a node in the network is desolate, meaning it has no neighbors. Figure 8 illustrates this difference effectively.

Quantifying the energy costs

Accounting for the fact that the radius signal will not always find a neighbor, in which case node $v, v \in G(V, E)$ is considered desolate, we express the random walker's behavior with this piecewise function:

$$p_{u,v} = \begin{cases} \frac{1}{|\delta(v)|}, & \text{where } (u, v) \in E \\ 0, \end{cases} \tag{5}$$

Our objective would be to avoid the second case, which is achieved by providing a sufficient signal for each node V∈G

Since my algorithms involve networks of different sizes, I define a variable rc_r "rc relative". This variable controls for network size expressing the connection radius as a fraction of the network's area diagonal length (with an rc numerator to the diagonal of the network denominator). Thus, I propose that for a network G(V,E) with a connectivity radius rc:

$$rc_r = \frac{rc}{diagonal(G_{area})} \tag{6}$$

As a means to account for the amounts of energy costs of the increased transmission radii across the network, I chose to use the most widespread energy model for WSNs, taken from energy-cost calculations of the LEACH algorithm, particularly the one defined in Heinzelman et al.'s *first-order radio energy model.* [18]

In this model, the energy consumed by transmitting a k-bit packet over a distance d is:

$$E_{\text{tx}}(k,d) = \begin{cases} kE_{\text{elec}} + k\varepsilon_{\text{fs}}d^2, & d < d_0 \\ kE_{\text{elec}} + k\varepsilon_{\text{mp}}d^4, & d \geq d_0 \end{cases} \tag{7}$$

where :

- E_{elec} is the energy per bit consumed by the transceiver electronics (nJ/bit); this will be assumed a common/static value, across all my sensor devices,
- ε_{fs} is the amplifier coefficient in the free-space channel model (pJ/bit/m^2); the amplifier coefficient generally calculates the packet creation-costs, which essentially correspond to hop amounts in the type of simulations I'm using,
- ε_{mp} is the amplifier coefficient in the multipath channel model (pJ/bit/m^4); also corresponding to hop amounts, if the model consists of a multipath channel,
- d is the transmission distance between sender and receiver,
- $d_0 = \sqrt{\varepsilon_{\text{fs}}/\varepsilon_{\text{mp}}}$ is the threshold distance distinguishing the free-space from the multipath propagation regime. [18]

It's unequivocal that the parameters for ε_{fs} and ε_{mp} are the main prerequisites in understanding which propagation regime our network uses. Based on Heinzelman's model, Cromeau et al. and various similar works have defined standard parameters that are appropriate for WSN energy-based calculations. [19]

By following Cromeau's standard parameter values on Figure 9, standard values should be:

- $E_{elec} = 50nJ/bit$
- $\varepsilon_{fs} = 10pJ/bit/m^2$ (rounded from 10,255)
- ε_{mp} = 0.0013 pJ/bit/m^4

Parameter	Value
Sensor deployment area	100 x 100 m
Base station location	(50, 150) m
Number of nodes	100
Data packet size	100 bytes
Control packet size	25 bytes
Initial energy of sensor	0.5 J
Aggregated packet size from cluster head	500 bytes
Electronics energy	50 nJ/bit
Free space factor	10, 255 pJ/bit/m^2
Multipath factor	0.0013, 0.0050, 0.0063 pJ/bit/m^4

Figure 9: LEACH standard parameters according to Cromeau et al. on page 936 Table 1 [19]

The model assumes that every transmission incurs a fixed electronic cost kE_{elec}, plus a distance-dependent amplifier cost. For distances shorter than d_0, the network ought to follow the free-space model with a quadratic cost (d^2). For longer distances, multipath fading dominates and energy scales quartically (d^4).

In our setting, the transmission range of each node is determined by the chosen relative communication radius r_c. Since all nodes share the same r_c, we approximate the average hop length d as a fraction of this radius (i.e., $d \approx \alpha r_c$, with $\alpha \in [0.5, 0.7]$ depending on the spatial distribution of neighbors).

Thus, the total network energy consumption over the course of a simulation can be estimated as:

$$E_{\text{network}} = \sum_{t=1}^{T} E_{\text{tx}}(k, d_t) \tag{8}$$

where T is the total number of transmissions during the coverage process, and d_t is the hop distance of the t-th transmission.

And so, by following the convention of the LEACH model we'll be first seeking the value of d_0 to determine whether to use the free-space channel ($n = 2$) when $r_c < d_0$, or the multipath channel ($n = 4$) likewise.

The cutoff diagonal length

We can reliably solve for the particular diagonal length needed for this cutoff (which determines the channel type). By using common values we solve for d_0:

$$\begin{aligned} d_0 &= \sqrt{\frac{\varepsilon_{fs}}{\varepsilon_{mp}}} \\ &= \sqrt{\frac{10 \text{ pJ/bit}/m^2}{0.0013 \text{ pJ/bit}/m^4}} \\ &= \sqrt{\frac{1.0 * 10^{-11} \ J/\text{bit}/m^2}{1.3 * 10^{-15} \ J/\text{bit}/m^4}} \\ &= \sqrt{7.69 * 10^3 \, m^2} \\ &= 87.7 \, m \end{aligned} \tag{9}$$

Energy cost of the cloned random walker

The cloned random walker adaptation duplicates each running agent at a preset duplication interval. Since the running agents exponentially duplicate at said interval, the energy cost should follow an exponential summation formula.

I propose the heuristic function for the energy cost of the cloned random walker to be the following:

$$E_k = k(\sum_{i=0}^{n-1} 2^i * I + 2^n * r) \tag{10}$$

Where:

- k is the packet size in bits,
- I is the duplication interval in time slots,
- T is the total amount of time slots
- n is the amount of **full** intervals: $\lfloor \frac{T}{I} \rfloor$
- r is the leftover time: $r = T - n * I$

2.4 Advantages of Random Walkers

As aforementioned, the main advantage of random walker agents is the energy efficiency they constitute. We can observe research supporting this across a multitude of network type of systems. This paper, by Quin et al. utilizes a linear algorithm with the aim of accessing circuit nodes inside a power grid. [20] This work written by Ramenzapour et al. focuses on budget walking in different sites of a city, pertaining a low-cost algorithm to disseminate movement. [21] Both works underline the simplicity and accessibility of the algorithm.

As Natapov et al. note, in visual search simulations, random walkers provide the huge advantage of revisiting previous nodes. This helps collect or distribute up-to-date information during the revisit, without requiring additional energy. The authors model this metric, of time spent to revisit the same node, as the Recurring Time (RT), in complement to the more widespread metric of First Passage Time (FPT),[14] also commonly referred to as "First Hitting Time", with examples like R. Patel et al. [22]

2.5 Models of interest

There is scholar work on both normal and hybrid models that utilize Random Walkers. Some of these models, I found to be workable in the Omnet++ environment and will be used for the simulations. Each adaptation model offers its own advantages. For instance, one paper by Tzevelekas et al. elaborates on the "jump" method, which is appropriate for speeding up mobility in large-scale networks and can handle bottleneck issues. [12] This adaptation I chose to not use in this paper, for the reason that there is uncertainty on the energy cost or function of the long-jumps.

One adaptation which this paper uses is the clone method, discussed in this paper by Egger et. al.[23] This adaptation of the agent is used to speed up coverage time. However, unlike other methods, this one precisely accelerates the process when it is needed. If the network does end early, it won't need to duplicate, as the cloning/duplication interval can potentially never be reached. If the coverage is quicker than the set interval, it will not require excessive energy to hasten coverage. The amount of walkers by the moment of full-coverage entirely depends on the current coverage status, aligning with the dynamic environment that random walks excel in.

Tzevelekas et al. [12] state a very straight-forward and inexpensive tinker with the random walker agent. They posit and model the random walker with no backtracking. This model adds a simple feature that prohibits the agent from returning to the previous node it last stayed in, preventing it from backtracking. They claim that in a network G(V,E), the probability that the agent on a node

v has to visit a node u, where u node wasn't the origin node from its last hop, would be equal to $p_u = \frac{1}{\delta(v)-1}$, $u, v \in E$. This is simple, low-cost and unbiased.

2.6 AI usage in WSNs

As AI has become a focal point for IT innovation and competitive advantage development, it is worth noting that it is not particularly common in WSNs. The Machine Learning (ML) algorithms that are worth looking into consider the austere energy constraints WSNs function under. Among some, k-NN, linear regression, decision trees and neural networks can be used to find an efficient solution in a static network [24]. Among reinforcement learning methods, Q-learning has attracted particular attention for its ability to optimize routines dynamically. This algorithm learns low-cost paths, as it provides live-data feedback to network routes that attatch a reward value to each path based on the given weight parameters. It then finds the optimal solution through the Q-table function, as introduced in Equation 10 by Yun et al. [25]

$$Q(s,a) = (1-\alpha)\,Q(s,a) + \alpha\left\{R + \gamma \cdot Q\left(s',a\right)\right\} \quad (10)$$

Figure 10: Q-learning formula | Equation 10 by Yun et al.[25]

2.7 Historical uses of the random walk

While IOT and WSN systems find widespread growth, the random walk has a history beyond WSNs. Pearson's original definition for the random walk was paved by previous works, botanist Robert Brown first observed *the randomized movement of pollen grains in the water* in 1827. [26]. Brown's concept was later applied by French mathematician Louis Bachelier to model the fluctuations of the *stock market* in 1900. Albert Einstein in his 1905 paper described the random walk as the result of random collisions between the observed particles and the surrounding molecules. It was the work and cooperation of a shared scholar framework that led to the formalization of this current model, and that should not be a given.

3 Methodology

3.1 Software tools & credits

- My work in computing and plotting random walker simulations materializes in **Omnet++**, a C++ framework specialized for network simulations. Since Omnet++'s IDE provides an interface that allows for multiprocessing and CPU segmentation, as it regards everything a discrete-event to optimize for computational complexity, I took notice that in comparison to a controlled trial in Python, the compilation time of multiple simulations is noticeably reduced in Omnet++.

- Even though I originally used Gnuplot for plotting the figures, I found that it didn't meet the demand of computing average values across a host of files effectively; hence, I selected **Python** to nimbly iterate through different files, and perform math on multi-type variables, in addition to Python's Numpy, Glob and Matplotlib's libraries, I deem this method suitable for plotting Omnet's data.

- I designed the sequence diagram for figure 6 using **Visual Paradigm**. I annotated on graphs such as on Figure 13 , to embolden the visual of interest using **GIMP**. This report was written in **LaTeX**, in a local environment using **TexLive-LuaLatex**. The gifs on figures 5.1 and 2.3 were segmented using **FFMPEG** and then processed through LaTeX's 'animate' library.

- For the illustration and annotations on Figures 19 and 16 I used **Google Maps** to import the linked dataset of the Athens Mass Transit System: `https://data.nap.gov.gr/dataset?organization=oasa-s-a`.

- For the Python simulations on the Jump adaptation and Hypothesis $H1$, I used **Google Colab**, using Python's random, math, numpy, and matpotlib libraries.

- This project's code repository is uploaded on **Github** on url: `https://github.com/Gkarios/Random-Walks-Omnet`. All algorithms can be easily replicated by changing the parameters in Omnetpp.ini. Datasets used are available and can be replicated using my Python scripts at folder /simulation/prints/data. In this directory, if the user wants to collect Omnet data, they need to keep an empty /qtcmd folder. Datasets and code are available in the /plotting subfolder. To import the workspace project, the user has to select File->"Import" using the git wizard on the Omnet++ IDE (current version: 6.1). In the git wizard, select to clone the URI in the

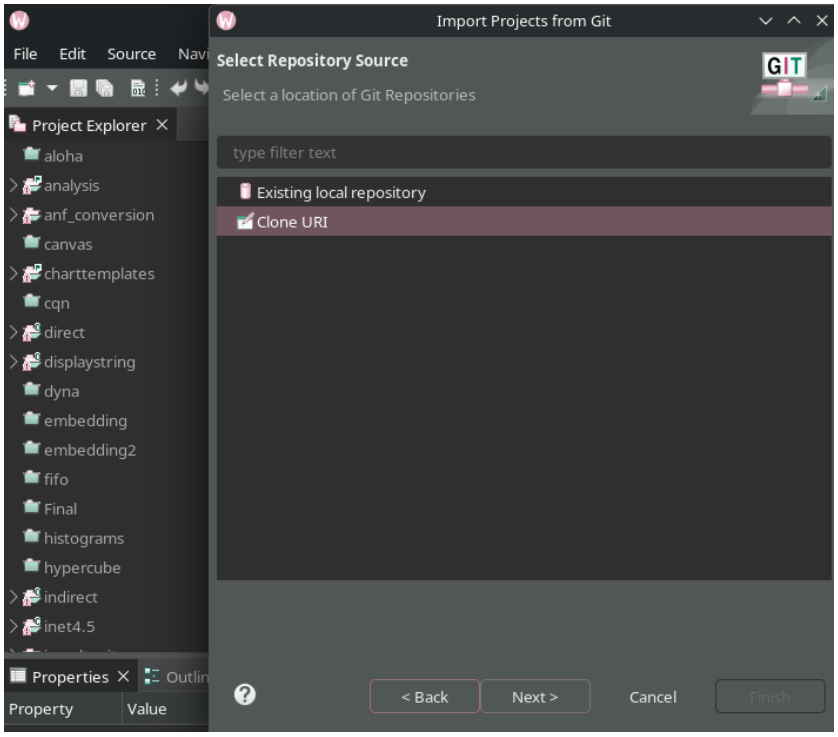

Figure 11: Importing the git repository on Omnet++ IDE

pop-up window shown on figure 11, in which case there will be an option to also add a locally cloned git repository.

3.2 The requirement of a coverage target

After I initially ran the model for complete network coverage, I took notice of the massive fluctuations of the time-slots required, that would translate to a high margin of error. Some simulations would end at $t_{max} > 10,000$ time slots. This would cause the metric average of the other 499 simulations to skew much higher, distorting the graph of the result as observed in Figure 12.

Zooming in on the plot as seen in Figure 13, we can see that C(T)=0.9999 or 99.99% coverage time is achieved at roughly $t_{99} = 1100$ time slots, while pure 100% coverage is attained at time slot $t_{100} = 2785$.

As the problem scales, the graph will visually become even more ambiguous, and so, I consider a preferable coverage fraction of the network to be about at 99.5% coverage, which would be commensurable with 1 node skipped for every 200 nodes. since my graphs on figures 12 and 13 would show the rate of change

(2nd derivative) to be near 0. For this cause, my experiments will use a ceiling of a ”Preferable Coverage”: $C_P(T) = 0.995$.

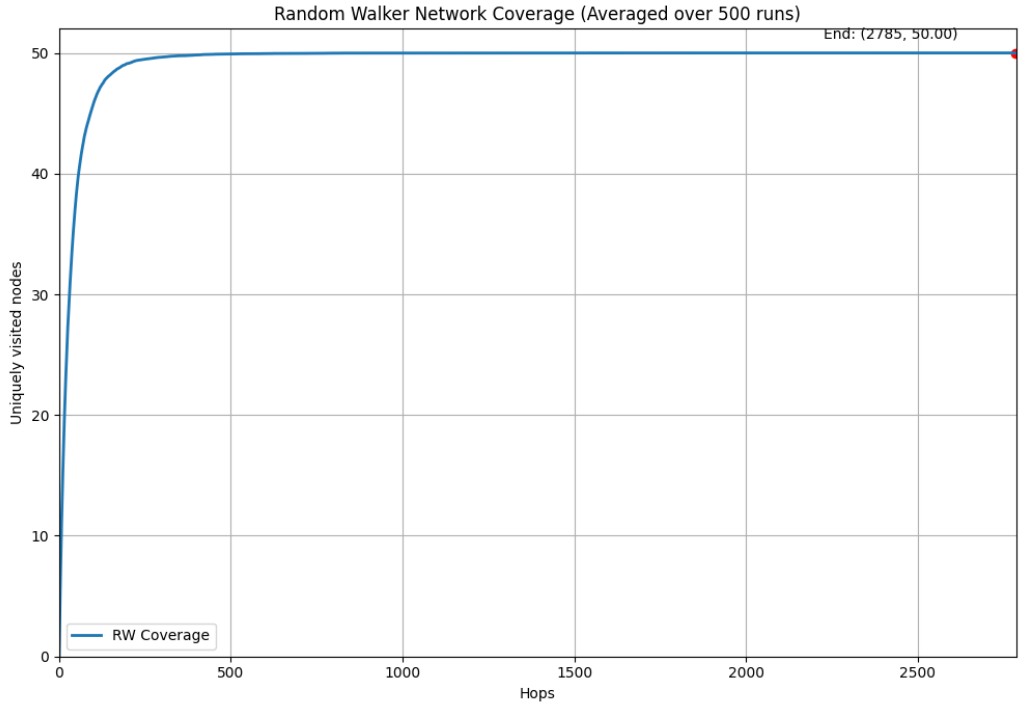

Figure 12: Omnet RGG Network coverage for V=50 nodes, rw=2 random walkers from one shared random placement, radius range $rc = 0.33$, Ends at time slot $T = 2785$. (Note that y-Axis refers to |V|)

The results made in this paper will stop at preferable coverage. I have marked this on figure 13 with a red dot for the purpose of reducing the margin of error, as well as increasing visual detail on the graph.

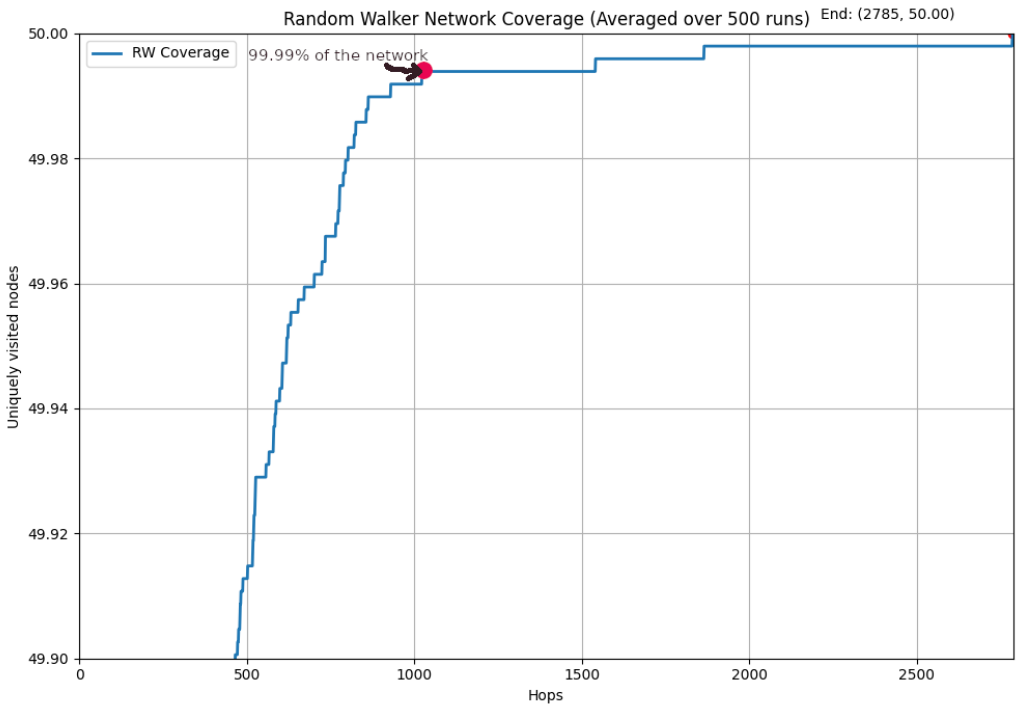

Figure 13: Omnet RGG Network coverage for V=50 nodes, rw=2 random walkers from one shared random placement, radius range $rc = 0.33$, Ends at time slot $T = 2785$. Figure zoomed in for coverage range = [49.90-50.00]

3.3 Real world data - Taxi stand experiment

All algorithms were applied to real-world taxi data, as publicly shared by the Greek National Accept Point administrator on `https://data.nap.gov.gr/dataset/taxi-staseis`. This dataset pinpoints all taxi stands (Piazza) stops of the OASA Attica network in Greece. Once I saved the file, I took notice of its syntax on figure 14. The label of interest is the `COORDINATES` field.

```
<tr style="background-color:#DDDDFF;"><td>gid</td><td>11</td></tr>
<tr><td>perioxi</td><td>ΑΘΗΝΑ-ΝΟΣΟΚ. ΑΛΕΞΑΝΔΡΑ</td></tr>
</table>]]></description>
            <ExtendedData>
                <Data name="gid">
                    <displayName>gid</displayName>
                    <value>11</value>
                </Data>
                <Data name="perioxi">
                    <displayName>perioxi</displayName>
                    <value>ΑΘΗΝΑ-ΝΟΣΟΚ. ΑΛΕΞΑΝΔΡΑ</value>
                </Data>
            </ExtendedData>
            <Point id="11">
                <coordinates>23.75552825435481,37.98033945980961,0</coordinates>
            </Point>
        </Placemark>
        <Placemark id="17">
            <name>Μεγ.Αλεξάνδρου</name>
            <description><![CDATA[<table>
```

Figure 14: A snippet from the original KML file, as downloaded from the OASA data repository

I ran a script to iterate through the file's elements, and map the KML labels onto Omnet++ syntax. On figure 15 I pull the latitude and longitude values under the respective label in the KML file, and convert those to float x, y axis coordinates.

```python
# Read the KML file
kml_file = 'taxi_231002-1.kml'
tree = ET.parse(kml_file)
root = tree.getroot()

# The KML namespace needs to be handled to find the elements correctly
ns = {'kml': 'http://www.opengis.net/kml/2.2'}

# Find all Placemark elements
placemarks = root.findall('.//kml:Placemark', ns)

print(f"Coordinates formatted for omnet.ini with a {canvas_width}x{canvas_height} canvas:")
print("--------------------------------------------------------------------------")

# Iterate through each placemark with an index to format the output correctly
for i, placemark in enumerate(placemarks):
    # Find the <Point> and then <coordinates> element
    coordinates_tag = placemark.find('.//kml:Point/kml:coordinates', ns)
    if coordinates_tag is not None:
        # The coordinates are a single string "longitude,latitude,altitude"
        lon_str, lat_str, _ = coordinates_tag.text.strip().split(',')

        # Convert to float
        lon = float(lon_str)
        lat = float(lat_str)

        # Normalize and scale the coordinates
        scaled_x, scaled_y = normalize_and_scale_coordinates(lon, lat, min_lon, max_lon, min_lat, max_lat, canvas_width, canvas_height)

        # Print the output in the desired format
        print(f"**.node[{i}].xCor = {int(scaled_x)}")
        print(f"**.node[{i}].yCor = {int(scaled_y)}")
```

Figure 15: Defining the KML namespace, and converting syntax

It can be observed that I have also normalized all the values under certain boundaries. I opened the KML dataset on Google Maps, and saw the locations for myself to ensure validity. I then manually drew a rectangle surrounding the area of interest, which will define the network I'll be running simulations on. I placed 4 markers North-East, South-East, South-West, North-West, as portrayed on Figure 16.

I corroborated the values with an offset to keep the latitudes/longitudes parallel when importing it to the script.

North East	38°08'34"N	24°01'46"E
South East	37°48'08"N	24°01'46"E
South West	37°48'08"N	23°29'51" E
North West	38°08'34"N	23°29'51" E

Table 1: Physical real-world borders for my Omnet++ network

I then had to convert these borders to a decimal degree format, that omnet will easily use. And so the script converts the Degrees, Minutes, Seconds (DMS °'") format to decimal degrees (0.000...). This will bound all of the taxi stands in my area, which can map onto a 6000X6000 Omnet++ network.

Once I simulated the output nodes in Omnet++, I noticed that the network was not very similar, in-fact it was upside down, as Omnet reverses the y-axis to increment as the y-axis decreases. Hence, I ensured to reverse the y-axis during

Figure 16: Border lines of the dataset on the physical map in Attica, Greece

```
35   # Manually converted to decimal degrees for the bounding box
36   min_lat = dms_to_dd(37, 48, 8, 'N')
37   max_lat = dms_to_dd(38, 8, 34, 'N')
38   min_lon = dms_to_dd(23, 29, 51, 'E')
39   max_lon = dms_to_dd(24, 1, 46, 'E')
40
41   # Define the size of the Omnet++ simulation canvas
42   canvas_width = 6000
43   canvas_height = 6000
```

Figure 17: The formatted network bounds

```
18      # Normalize coordinates to a 0-1 range
19      normalized_x = (lon - min_lon) / (max_lon - min_lon)
20      normalized_y = (max_lat - lat) / (max_lat - min_lat)
21
22      # Scale the normalized coordinates to the canvas size
23      scaled_x = normalized_x * canvas_width
24      scaled_y = normalized_y * canvas_height
25
```

Figure 18: The processing of the y-axis in the dataset as opposed to x-axis

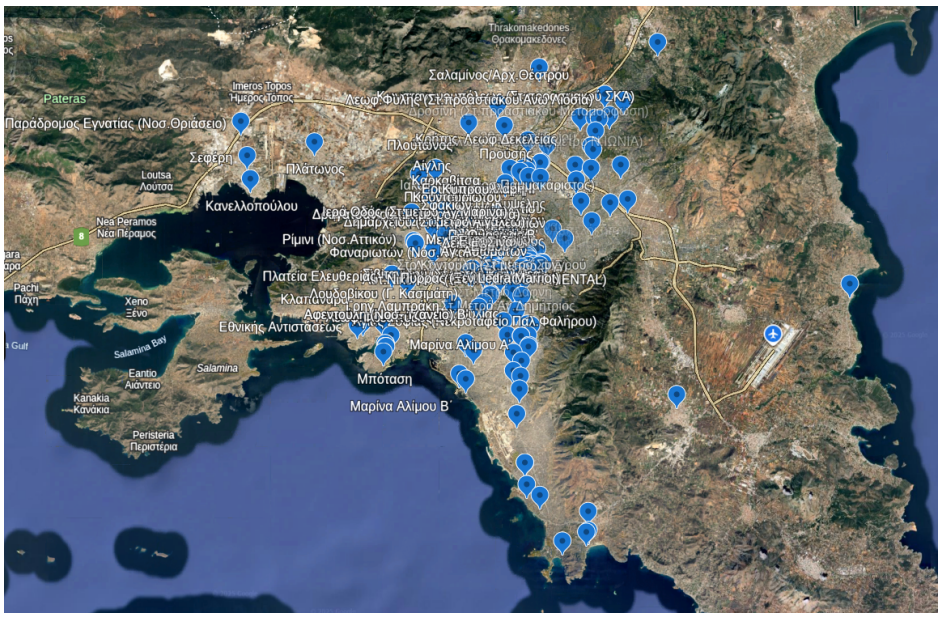

Figure 19: Original network, as imported from the dataset on Google Maps

the normalization when processing the data. For this reason, it's observed on Figure 18 that y-axis normalization is the reverse of the one x-axis uses.

Once I resolved that issue, I came across the correct result (as illustrated on Figures 19 and 20, which does reflect directly the network that I previously generated on Google Maps.

Once I define an rc for this network, I observed that the edge-nodes needed significantly longer radii signals as compared to the rest, with the farthest desolate node being the easternmost "node[97]". Through brute force, I found the minimal amount of radius units required for the signal to be $rc_{min} = 1945$, com-

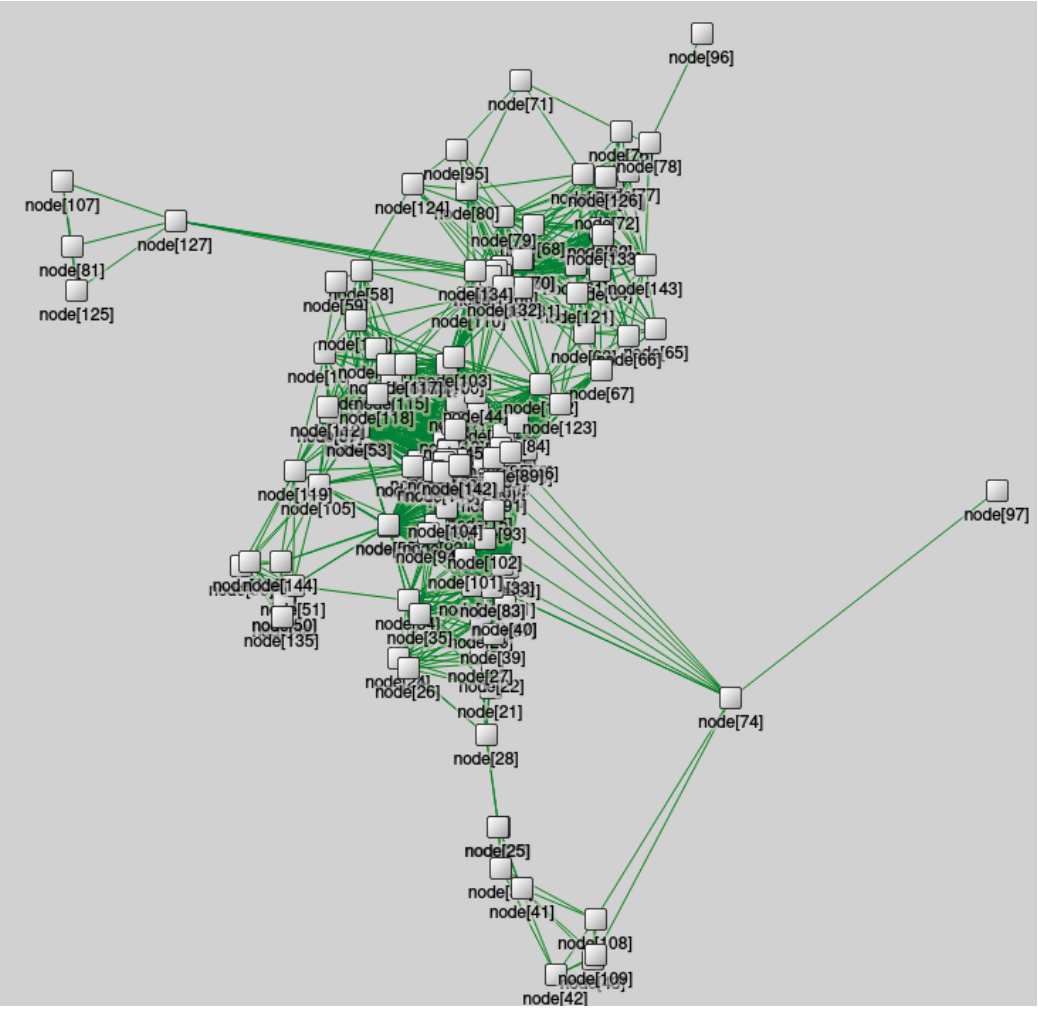

Figure 20: The taxi stands illustrated in an Omnet++ 6000X6000 network

pared to the diagonal, where for diagonal d:

$$d = \sqrt{h^2 + l^2} \tag{11}$$
$$= \sqrt{36000000 + 36000000} \tag{12}$$
$$= 8485.28 \tag{13}$$

Diagonal d is roughly 4 times bigger than the minimum possible rc in the network, which is very inefficient, considering that it will leads to hundreds of links, making the network intensely bulky and dense; as it was an obvious hypothesis to make, I verified the occurence in Figure 21, where indeed, the network would be excessively bulky.

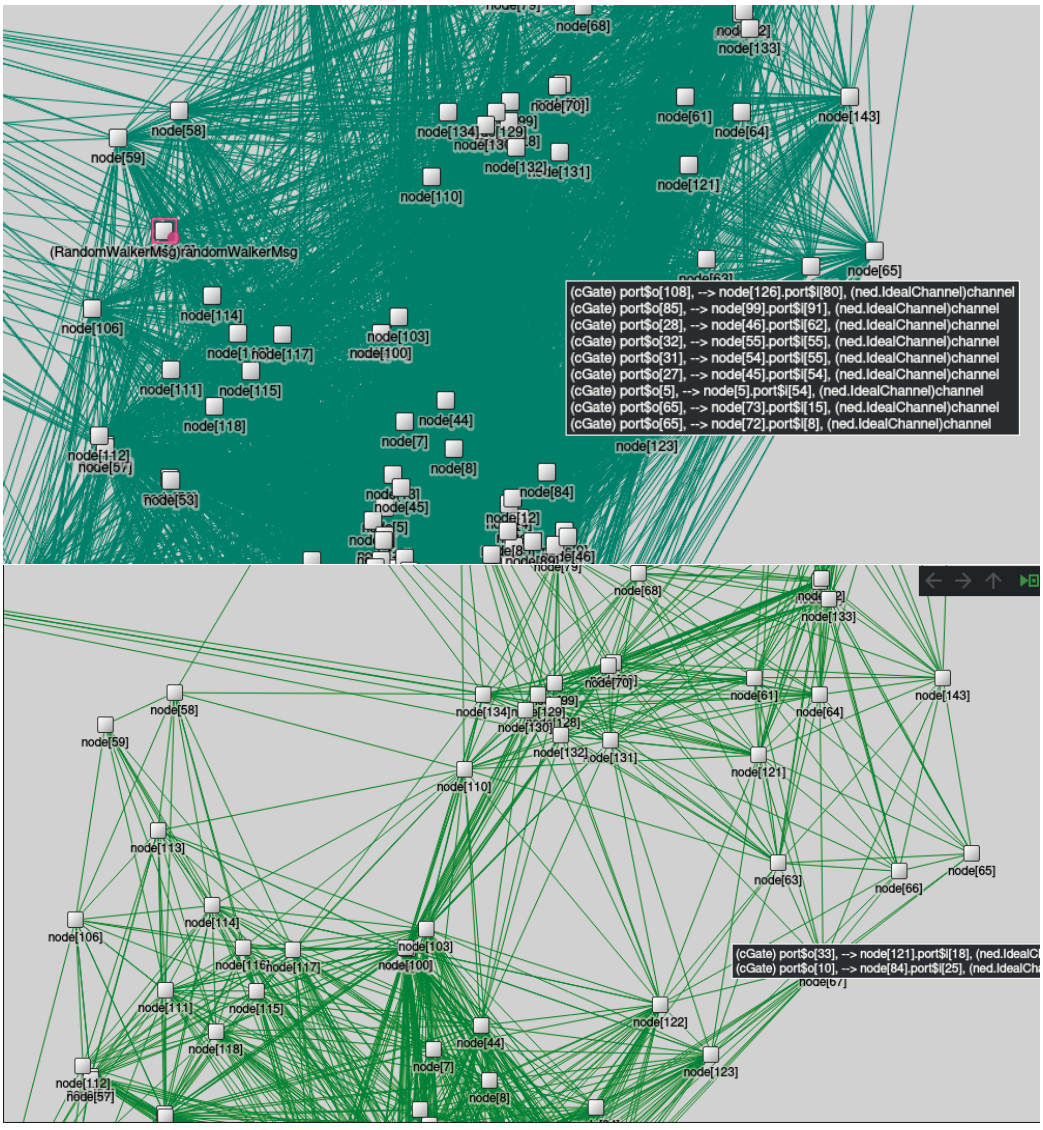

Figure 21: COMPARISON: Single vs Double rc on the taxi network

3.4 Hypotheses

This occasion of my densely connected network in Figure 21 raises the question of how inefficient an rc that dense could be. For this reason, I propose the following hypothesis:

> **H1:** Increasing the communication radius rc improves coverage time across the network, but ultimately requires higher energy usage.

One of the comparisons I intend to make is how multiple random walkers can improve performance. I suggest that as random walkers scale, issues with the walker getting stuck on a visited neighborhood either by unlucky movement or bad topology luck will be significantly more noticeable with multiple random walkers. Additionally, multiple random walkers are all forced to spend energy during a time slot, which likely increases energy by a little.

For these reasons I propose the following hypothesis:

> **H2:** Multiple random walkers will underperform linearly in terms of coverage time, since bottlenecks will punish the agents linearly, and multiple walkers will perform excess movements.

Another doubt that arises is whether or not taxi stand network results will be aligned with the RGG results. Considering that 100% coverage alone has found huge fluctuations in past comparisons with 99% coverage, I assume that the taxi stand comparison will find a similar issue. In addition to that, the topology differences and double rc usage will likely produce potentially unexpected results.

Therefore, I propose the following hypothesis:

> **H3:** The taxi stand network will behave differently to the RGG network. Performance comparisons will be skewed; though I don't expect the ranking placement of each algorithm adaptation to change.

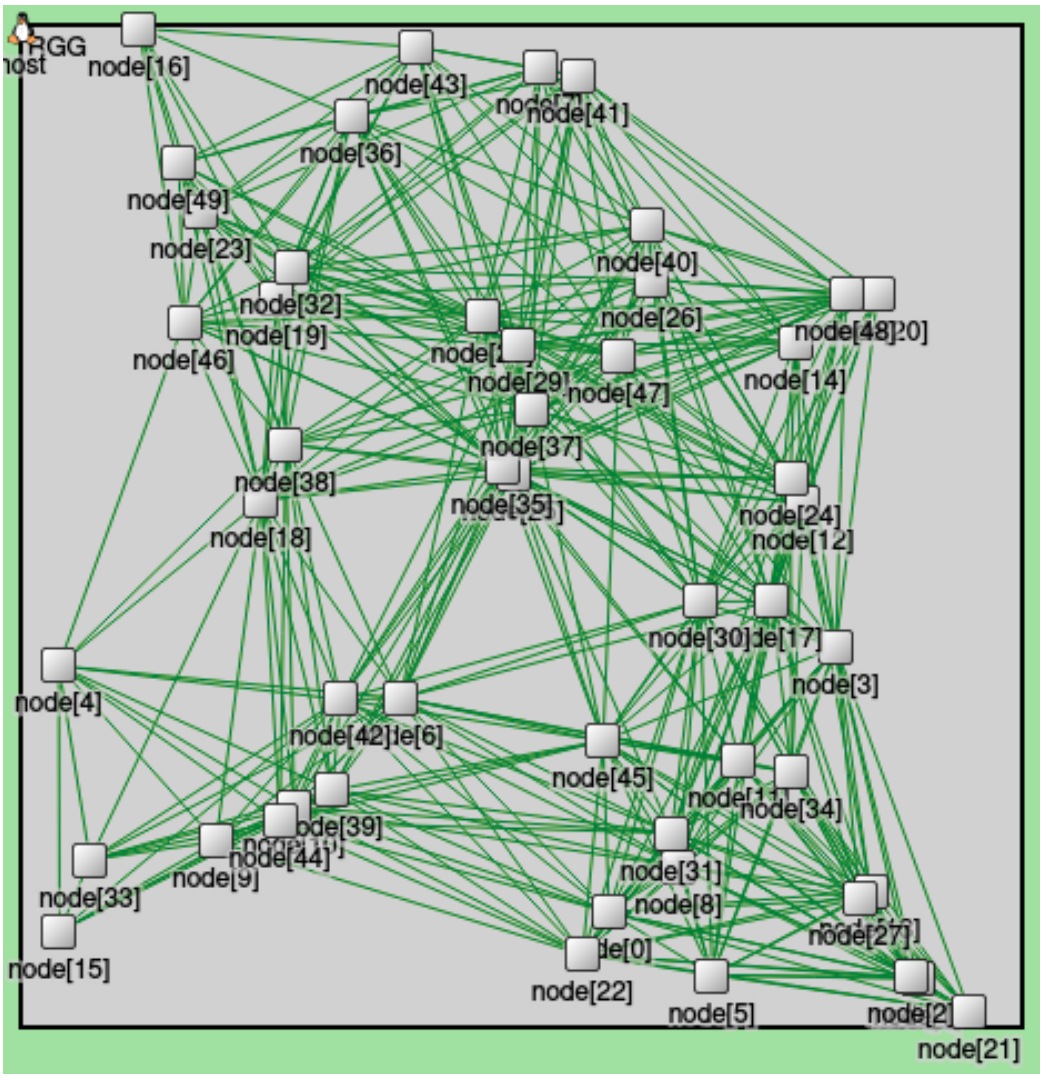

Figure 22: Depiction of a 1X1 Random Geometric Graph in Omnet++ (V=100, rc=0.25)

4 Programming overview

4.1 Python's rc program

I used a Python program to measure rc's influence on the energy costs of a RGG WSN. The fundamental network setup functions were developed by K. Skiadopoulos [10]. Building upon these, I integrated my own functions to execute multiple controlled experiment runs, and averaged, formatted, plotted, and visualized the resulting data.

```
class network_rgg:
    def __init__(self,population,r):
        self.pop = population
        self.r = r
        self.nodes = []
        self.xmax = 1
        self.ymax = 1
        self.construct()

    def construct(self):
        for i in range(self.pop):
            self.nodes.append(node(i,self,random.random()*self.xmax,random.random()*self.ymax,self.r))
        for i in self.nodes:
            self.find_neighbors(i)

    def find_neighbors(self,node):
        for i in self.nodes:
            ro = self.distance(node,i)
            if ro <= self.r and node.id < i.id and i not in node.neighbors:
                node.neighbors.append(i)
                i.neighbors.append(node)

    def distance(self,i,j):
        return math.sqrt((i.x-j.x)**2+(i.y-j.y)**2)
```

Figure 23: The constructor of my network in Python

To begin with, on Figure 23 the program defines a Random Geometric Graph network, which takes two arguments in its constructor, the number of total nodes, and the connection radius value. Once it registers the arguments, it runs a secondary constructor `CONSTRUCT()`, which loops for each node and assigns them random x, y coordinates. This function then calls `FIND_NEIGHBORS()`, which attaches neighbors for every node within its connection radius range; this additionally uses the distance(self,i,j) function, to calculate the Euclidean distance between the two nodes that will determine the edge between the two nodes.

```
def metric(self):
    max = 0
    min = 10**6
    distr = distribution()
    edges = 0
    for i in range(self.pop):
        ei = len(self.nodes[i].neighbors)
        distr.add(ei)
        if ei > max :
            max = ei
        if ei < min :
            min = ei
        edges += ei
    edges /= 2


def is_connected(self):
    informed = set()
    start = random.randint(0,self.pop-1)
    to_inform = {self.nodes[start].id}
    while len(to_inform) > 0 and len(informed) < self.pop:
        to_inform_next = set()
        for i in to_inform:
            if i not in informed:
                informed.add(self.nodes[i].id)
                for j in self.nodes[i].neighbors:
                    if j.id not in informed:
                        to_inform_next.add(j.id)
        to_inform = to_inform_next
    return len(informed) == self.pop
```

Figure 24: Checking for network connection and providing metrics for cardinality and distribution.

On figure 24, THE METRIC() function is used to measure and give me feedback on the cardinality range and edges of the network, the average number of neighbors. It calls IS_CONNECTED(), which warns whether or not the network is connected, in a similar fashion to the omnet++ function, it will set every node as unvisited, and perform Breadth-First-Search (BFS) on the network, until the amount of visited nodes equates the amount of total nodes in the network (pop-

ulation).

Lastly, DISTRIBUTION() prints the degree distribution of the network. This will establish an incrementing y-axis of the length of the neighbors set, and an x-axis with the amount of nodes that currently have that length of neighbor-set.

```
class random_walker:
    def __init__(self,net,start):
        self.net = net
        self.path = [start.id]
        self.visited = {start.id}

    def next_step(self):
        now = net.nodes[self.path[-1]]
        self.path.append(random.choice(now.neighbors).id)
        self.visited.add(self.path[-1])
```

Figure 25: The random walker model in Python

Figure 25 is the random walker model, it possesses a buffer of the random walker's path. It's movement is repetitively called using the NEXT_STEP() function, unlike the Omnet++ model.

For the second taxi stand aligned large-scale network that I'll reference in the RESULTS section, I utilized IS_CONNECTED() to determine the minimum rc required for the network to be connected. By brute forcing values, I reached a reliable range of $rc_{min} = 860$.

Following the logic of figure 26, I then incremented this amount up to $rc_{max} = 1360$, in intervals of 100. This would provide 5 points of comparison.

After initializing the rc array and the time slot array which will be the two axes, I create 6 different networks for each rc value. If the rc happens to not be adequate (which is possible considering that the topology is a Random Geometric Graph), it will skip that rc value, which can be seen on the result on figure 59, in which case the random network for $rc = 1160$ was not connected.

Once it's determined that the network is connected, the program will run 50 simulations of the random walker being deployed at a random node. The agent will keep a list of visited nodes, and update it during visits, until it reaches 100% coverage. During each step, with the proviso that the network has not been fully covered by the agent, the time slot array is incremented.

From that point on, once the loop finishes iterating, it will average each data point and matplotlib will plot the results straightforwardly.

```
num_trials = 6  # Number of trials for different rc values
rc_values = np.linspace(860, 1360, num_trials)
average_steps = []

for rc in rc_values:
    net = network_rgg(net.pop, rc, xmax=net.xmax, ymax=net.ymax) # Creating network with the current rc value
    if not net.is_connected():
        print(f"Error: Network with rc={rc} is not connected. Skipping simulations for this value.")
        average_steps.append(np.nan) # Append NaN for disconnected networks
    else:
        steps_for_rc = [] # List to store steps for each simulation with current rc
        for _ in range(50): # 50 simulations for averaging
            start = random.randint(0, net.pop - 1)
            end = random.randint(0, net.pop - 1)
            while end == start:
              end = random.choice([i for i in range(net.pop) if i != start])

            rw = random_walker(net, net.nodes[start])
            steps = 0
            while end not in rw.visited:
                rw.next_step()
                steps += 1
            steps_for_rc.append(steps + 1)
        average_steps.append(np.mean(steps_for_rc))  # Calculate and store the average steps
```

Figure 26: Simulating and formatting the data for plotting

4.2 Omnet++ program overview

Project structure

Omnet++ and its default template strictly follow modular programming disciplines. Each task and object is segmented and inherited if need be. My program respects this direction.

Everything is under a common project workspace, which is segmented in multiple modules. I section the models in two categories. The first being the fundamental and widely inherited models for all networks. This includes the common node, and the centralized Controller/export node (also referred to as the ”Host” node, considered to have a long-range reach or connection to an external device for the purpose of exporting large amounts of data that the agents have collected). This category is denoted in .ned files, which lay the groundwork for their own associated .cc and .h modules, that implement the program’s low-level functionality. The other category is consisted of the auxiliary modules that support the functionality. For these experiments, it would include the Randomwalker.msg module, which is an object that holds cache variables for the agent. Additionally, the other main module of interest in this category wouldb e the ”omnetpp.ini”, which provides the user an interface that configures variables belonging to the top module as well as the submodules.

In essence, the .cc modules establish the functionality of the C++ code, as the .ini module hastily configures the simulation and directly can set and change attributes values that the simulation will then utilize.

Every node module is defined as an object. To acquire access for a node’s

information Omnet++ provides the ”getParentModule()” function, which parses all associated parameters, gates and associated objects. The values of interest are returned in whichever format is specified.

For easy access to properties of the network, I use the ”cTopology” library, which helps aggregate the nodes and distance between the node modules.

To debug the code, I commonly model the program to print many types of update notices, whether proactively or reactive to an error. I observe the produced simulation within Omnet's QT environment, which offers its own EV console that prints debug messages. The side window during the simulation presents all attributes by node (for example it can detail the ”neighborhood” of the node). The primary purpose of this environment is to visually simulate and depict in detail the behavior of the network in action. To choose the simulation environment, the user has to navigate the Main screen of ”Run configurations” listed on the IDE's options, as depicted in figure 27. This offers the selection of visual simulations, or command line simulations (which are ideal for high performance repeat runs).

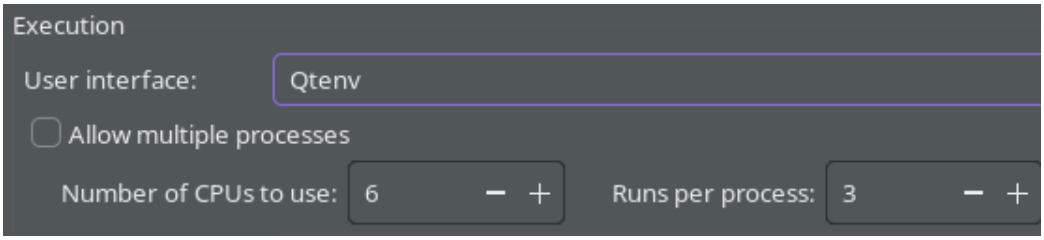

Figure 27: Choosing the QT Environment in Omnet++, options: QTenv, cmdenv

Each of my simulations respect some common boundaries. They are controlled for common attributes (node count, rc, etc.), ran in 500 separate simulations of different seed-set random values. The results of each simulation are averaged out, for the purpose of eliminating uncertainty. Afterward, they are plotted on figures, with predetermined verbose checkpoint values, at different Coverage amounts.

Methodology of the Omnet++ program

4.2.1 Preset components

All nodes in the network share the node module, which Node.ned defines on figure 28. Its basic functionality is to carry data for xCor and yCor coordinates, which are set on Omnetpp.ini. The Node also consists of gates, which are set on the network.ned file. This module defines how they will be displayed on the QTEnv simulation, as well as their gate links.

Other variables used during the simulation are defined in Node.cc, which imports most of its parameters from Node.h. The libraries that are used on `Node.h` are as follows: `<algorithm>`, `<iostream>`, `<limits>`, `<map>`, `<omnetpp.h>`, `<vector>`, and the "RandomWalkerMsg_m.h" agent file that I elaborate on later.

```
simple Node
{
    parameters:
        double xCor = default(0.0);
        double yCor = default(0.0);

        @display("p=$xCor,$yCor;i=block/process;is=vs;");
        @node;
    gates:
        inout port[];
        input radioIn @directIn;
}
```

Figure 28: The Node.ned file, which defines the node module

Host.ned on figure 29 doesn't have any links with other neighbors, it just needs to read the transmission it acquires at the end of the simulation, for that reason we only define a DirectIn input gate to read the coverage data.

I utilize three networks. For random walker simulations the one of most interest is the RGG network as aforementioned in section 2.3.

In Omnet, it is common practice to define user-input variables in the network .ned section, which will later be overwritten by the user on Omnetpp.ini. For that reason all algorithm adaptations and fundamental parameters for the network topology are initialized here with mutable and default() tags. Vertex module types, are considered submodules of the Top module (top module being the network and omnetpp.ini modules).

```
simple Host
{
    parameters:
        int startNodeIndex = default(-1) @mutable;
        int startNode = default(11);
        @display("i=abstract/penguin;is=vs;p=0,0");
        @name("Host");
    gates:
        input directIn; // Direct input gate for receiving messages from nodes
}
```

Figure 29: The Host.ned file, which defines the host module

```
network RGG
{
    parameters:
        int numNodes = default(25);
        double rc = default(0.2);
        @display("bgb=1,1");
        bool enableDuplication = default(false) @mutable;
        bool noBacktracking = default(false) @mutable;
        int numWalkers = default(1);
    submodules:
        node[numNodes]: Node;
        host: Host;
    connections allowunconnected:
        for i=0..numNodes-1, for j=i+1..numNodes-1 {
            node[i].port++ <--> {@display("ls=green,1"); } <-->
            node[j].port++ if i!=j && sqrt((node[i].xCor - node[j].xCor)^2+(node[i].yCor - node[j].yCor)^2)<rc;
        }
}
```

Figure 30: The RGG.ned file, which defines the RGG network module

In the CONNECTIONS section, every node engages in a Euclidean distance check with every other node in the network. This is essentially the function expressed in equation {4}.

The main difference the Grid network has on figure 31, are the coordinate positions. All nodes have a preset index ranging from 0 to numNodes-1. By setting one axis to perform a modulo operation and the other to keep the integer's floor division result, all nodes will be evenly distributed in the area with parallel links.

The taxi network build is managed on figure 32. This sets new dimensions and parameters that I derived from calculations. Since this network uses two different rc values, whenever a pair of nodes performs the distance check, it will check the node's index as well. If it matches the index of a node I've declared desolate, it will check the connection using the longRc instead.

Initialization

The program holds discrete "stages" during the simulation. The first stage initializes the simulation, as it defines and intializes every required variable. The

```
submodules:
    node[5*5]: Node {
        parameters:
            xCor = int(index % 5);
            yCor = int(index / 5);
    }
    host: Host;
```

Figure 31: The Grid.ned's node placements

```
network TaxiNetwork
{
        parameters:
        int numNodes = default(145); // Placemarks
        double rc = default(0.2);
        double rcLong = default(1946); //min longRc
        int startNode = default(11);
           @display("bgb=6000,6000");
        bool enableDuplication = default(false) @mutable;
        bool noBacktracking = default(false) @mutable;
        int numWalkers = default(1);
    submodules:
        node[numNodes]: Node;
        host: Host;

    connections allowunconnected:
        for i=0..numNodes-1, for j=i+1..numNodes-1 {
            node[i].port++ <--> {@display("ls=green,1"); } <--> node[j].port++
                if i!=j &&
                   sqrt((node[i].xCor - node[j].xCor)^2 + (node[i].yCor - node[j].yCor)^2)
                   < ((i == 74 || i == 97 || i == 127) ? rcLong : rc);
        }

}
```

Figure 32: The Taxi.ned file, which defines the taxi network module

Host module on figure 34 defines each walker agent's starting node. This includes a feature that selects a random node as a starting vertex. For this cause, we previously ensure to fetch the value of the total amount of nodes in the network from the "Parent Module".

In Omnet++, the parent module is the "Omnetpp.ini" file, which selects the network that will be added to the module during the simulation, using a line such as "network = RGG" for instance. Omnet will then look for RGG.ned and parse its "numNodes" parameter. For convenience and proper reusability, we assign all fetched values as local pointers.

The cTopology library provides easy access to the network's information. Therefore, I parse the information with by using its `extractbyProperty()` function. All the "EV" lines that follow, allow for convenient debugging from the QTEnvironment's console. After selecting an arbitrary node from the network and saving that to "randomNodeIndex", I update the int value of the .ned file, so that all nodes will be able to fetch the right current value.

The more interactive part comes with Node.cc's code. My approach to this algorithm was to use static shared variables, for all nodes. It is important to note that this fails on multiple network simulations at the same time, we should restrict each simulation to one network at a time, as the variables and vectors would be otherwise distorting each other. This is executed in a distributed manner as each node decides for the walker's movement when it intercepts a visit, and discretely declares the next node the agent will traverse to.

Figure 27 shows the global variable initialization that each Node shared before the simulation has yet started. There is a separate category of variables on Figure 35. These global variables are instead fetched from the Omnetpp.ini module, to provide user-input functionality. In this case they consist of mostly boolean values that determine whether or not a random walker adaptation should be used at stage 0 of the simulation. Once they fetch "NumNodes", "enableDuplication" and "disableBacktracking", they begin initializing the static/shared vectors (from C++'s standard library). There will be a "visited" boolean vector, which keeps track of the ID key of the vector's index(), as a key, and a true/false value, for when it has been visited by any random walker agent in the past. This serves the purpose of coloring the visited nodes and tracking the coverage amount. "globalAllVisited" is a boolean used to end the simulation when all nodes have been visited. Additionally during this stage the vector int "visitedPerTimeStep" is the actual node coverage vector we'll be using to plot the graphs, with one more column for the respective time slot. We intitialize this vector with clear(), and push a value "0" for simulation start, before random walkers are deployed. The other column be filled with the simple int vector "timestep", which will simply increment per random walker hop.

```
1   #include "Node.h"
2   #include "Host.h"
3
4   Define_Module(Node);
5
6   // Initialization of static members
7   std::vector<bool> Node::visited;
8   std::vector<int> Node::visitedPerTimestep;
9   bool Node::enableDuplication = true;
10  bool Node::noBacktracking = false;
11  bool Node::globalAllVisited = false;
12  int Node::numWalkers = 0;
13  int Node::walkersMovedThisStep = 0;
14  int Node::duplicationInterval = 500;
15  int Node::walkerIdCounter = 100000;
16  int Node::timestep = 0;
```

Figure 33: Node.cc static Global variable initialization, before simulation start

```
if (stage == 0) {
  topo.extractByProperty("node");
  int numNodes = getParentModule()->par("numNodes").intValue();

  int randomNodeIndex = intuniform(0, numNodes - 1); //or a fixed value for comparisons
  cTopology::Node *randomNode = topo.getNode(randomNodeIndex);
  if (randomNode) {
    // write value into the Host's parameter table
    par("startNodeIndex").setIntValue(randomNodeIndex);

    EV << "Randomly selected start node: " << randomNode->getModule()->getFullName()
       << " with index: " << par("startNodeIndex").intValue() << endl;
  } else {
    throw cRuntimeError("No node found for random index: %d", randomNodeIndex);
  }
}
```

Figure 34: Stage 0 of the Controller node

```
void Node::initialize(int stage) {
    if (stage == 0) {
        //fetch values
        int numNodes = getParentModule()->par("numNodes").intValue();
        enableDuplication = getParentModule()->par("enableDuplication").boolValue();
        disableBacktracking = getParentModule()->par("disableBacktracking").boolValue();
        //initialize the static data
        visited.assign(numNodes, false);
        globalAllVisited = false;
        visitedPerTimestep.clear();
        visitedPerTimestep.push_back(0); // To start with [timestep, nodes] =  [0, 0]
    }
```

Figure 35: Stage 0 of every node module

Stage 0 - Deployment of the agent

Since Omnet's design fundamentally consists of nodes sending data to one another, the random walker is implemented as a simple cMessage object, as designed, according to Omnet's official manual: `https://doc.omnetpp.org/omnetpp/api/classomnetpp_1_1cMessage.html` . It involves the creation of a separate .msg file. For this agent, I define a few variables that will hold necessary cache data for functionality. Its cache memory is showed on figure 36. It consists of a multitude of variables that are used to pull information during the program, compatible with single and multiple active random walkers. On simulation run, Omnet automatically generates accessor and mutator functions (getters and setters) for all the variables in figure 36, depending on whether the type is scalar (walkerId, hopCountr, lastDuplicationRound) or array fields (path[], visitedNodes[], visitedPerHop[]). Thus, this will automatically create getVisitedNodesArraySize() for visitedNodes[] and so on.

Finishing the simulation

The node then checks if the total amount of nodes it fetched from the network module, is equal or less to the amount of visited nodes, to direct whether or not the simulation should finish. If it does, it updates the random walker's final metrics and exports the two data arrays of interest to the Host node (which could follow a Dijkstra's algorithm, as cTopology readily provides it. I have not implemented this, as it's not necessary for the present experiment). The node disposes of the walker, and calls Omnet's `FINISH()` function.

While Host also calls its `FINISH()` function, as shown on figure 38, it acquires the two arrays, looks for a /qtmcd folder inside Omnet's /prints/data subfolders, and creates a .txt file, with the two columns formatted as the axes of the graph.

In figure 37, the module first retrieves the amount of random walkers the user sets in the configuration file from omnetpp.ini. Then it assigns a firstNodeIndex,

src > RandomWalkerMsg.msg

```
message RandomWalkerMsg{
    int hopCountr = 0;
    int path[];
    int visitedNodes[];
    int visitedPerHop[];
    int walkerId = -1;
    int lastDuplicationRound = -1;
}
```

Figure 36: Random walker's structure

```
if (stage == 1) {
    numWalkers = getParentModule()->par("numWalkers").intValue();
    cModule *hostModule = getParentModule()->getSubmodule("host");
    if (!hostModule)
        throw cRuntimeError("Host module not found!");

    int startNodeIndex = 7; // The node from which the RW starts from
    if (getIndex() == startNodeIndex) {
        for (int i = 0; i < numWalkers; ++i) {
            startRandomWalker(i);
        }
    }
}
```

Figure 37: Deployment of the random walker

```
void Host::finish() {
    if (lastWalkerMsg) {
        int runNumber = getSimulation()->getActiveEnvir()->getConfigEx()->getActiveRunNumber();
        std::string filename = "prints/data/qtcmd/unique_visited_nodes_run" + std::to_string(runNumber) + ".txt";
        std::ofstream out(filename, std::ios::out | std::ios::trunc);
        if (out.is_open()) {
            int m = lastWalkerMsg->getVisitedPerHopArraySize();
            for (int i = 0; i < m; ++i)
                out << i << "," << lastWalkerMsg->getVisitedPerHop(i) << std::endl;
            out.close();
        } else {
            EV << "Could not open file for writing unique visited nodes.\n";
        }
        delete lastWalkerMsg;
        lastWalkerMsg = nullptr;
    }
}
```

Figure 38: Host module exports the data

which reads the random value that Host calculated during stage 0. All nodes check if their index (which they learn from getIndex()) corresponds to the firstNodeIndex. If so, they create as many random walker agents as the user specified by calling the `startRandomWalker()` function. This creates a new cMessage object, and starts initializing all variables we have defined in the .msg file. Each random walker calls this function with a sole argument of its ID, which is determined from a for loop that iterates for the amount of random walkers the user defined. At the end of the function, it triggers `sendToRandomNeighbor()`, which will flow for the rest of the program until all nodes will be visited.

The random walker's movement

The movement function is shown on figure 39. The input argument named (msg) in this instance, is the random walker pointer object. The node which is now carrying the random walker, calculates its connectivity degree δ(u) using the gateSize("port") function. If we're using the classical model and neighbors exist, the node simply selects a random neighbor with max equal to its δ(u) and sends the random walker to the output gate in communication with that neighbor denoted by the "port$o" value. It performs this using Omnet's `send()` function, which takes three arguments, the message type, the gate type, and the gate index. This will also be the major value printed in the QTEnv console during the simulation for clarity. As the agent reaches the other end of the gate, it will trigger Omnet's default `handleMessage()` function.

`handleMessage()` is depicted on figure 40. As the agent reaches a node, this function is automaically called. I want to focus on the first line; the program at runtime casts the incoming random walker to a `self_and_cast` function which checks if it's a RandomWalkerMsg type (the one of interest). self_and_cast is provided by Omnet's library, as it's salient to discriminate by message type, we don't want this to conflict with possible future adaptations.

Thus, the node's index is added to the walker's path cache with the setter functions. If the node has not yet been visited, there's a call to color the node in the graphical interface with red, from the default (gray color), for visual clarity with the use of Omnet's function: `setDisplayString.setTagArg("i", 1, "red")`, where i is the current unvisited node's index. The node fetches random walker's uniquely visited nodes and checks each one to see if it has been visited before, by using the ID (index) of the node, and if not, is set to "Already visited" and its ID is added to the uniquely visited array: `visitedNodes[]`. This is where the timestep increments and `visitedPerTimestep[]` updates the uniquely visited nodes to the current timestep. Both of these arrays represent the X and Y axis of the final graph for coverage.

```
114 void Node::sendToRandomNeighbor(RandomWalkerMsg *msg) {
115     int n = gateSize("port");
116     if (n > 0) {
117         int neighborGateIdx;
118         if (disableBacktracking && msg->getPathArraySize() > 1) {
119             int prevNode = msg->getPath(msg->getPathArraySize() - 2);
120             std::vector<int> candidates;
121             for (int i = 0; i < n; ++i) {
122                 cGate *outGate = gate("port$o", i);
123                 cGate *inGate = outGate->getNextGate();
124                 if (inGate) {
125                     cModule *neighborMod = inGate->getOwnerModule();
126                     int neighborNodeIdx = neighborMod->getIndex();
127                     if (neighborNodeIdx != prevNode)
128                         candidates.push_back(i);
129                 }
130             }
131             if (!candidates.empty()) {
132                 neighborGateIdx = candidates[intuniform(0, candidates.size() - 1)];
133             } else {
134                 neighborGateIdx = intuniform(0, n - 1); // fallback: no choice but to backtrack
135             }
136         } else {
137             neighborGateIdx = intuniform(0, n - 1);
138         }
139         send(msg, "port$o", neighborGateIdx);
140         EV << "Sent randomWalkerMsg to neighbor at gate index " << neighborGateIdx << endl;
141     } else {
142         EV << "No neighbors to send to!" << endl;
143         delete msg;
144     }
145 }
```

Figure 39: Sending the random walker to a random neighbor

```
46  void Node::handleMessage(cMessage *msg) {
47      RandomWalkerMsg *rwMsg = check_and_cast<RandomWalkerMsg *>(msg);
48
49      int nodeId = getIndex();
50      if (!visited[nodeId]) {
51          visited[nodeId] = true;
52          getDisplayString().setTagArg("i", 1, "red");
53      }
54
55      // Add to path (per walker)
56      int pathLen = rwMsg->getPathArraySize();
57      rwMsg->setPathArraySize(pathLen + 1);
58      rwMsg->setPath(pathLen, nodeId);
59
60      // Add to visitedNodes (per walker, for their own record)
61      int visitedLen = rwMsg->getVisitedNodesArraySize();
62      bool alreadyVisited = false;
63      for (int i = 0; i < visitedLen; i++) {
64          if (rwMsg->getVisitedNodes(i) == nodeId) {
65              alreadyVisited = true;
66              break;
67          }
68      }
69      if (!alreadyVisited) {
70          rwMsg->setVisitedNodesArraySize(visitedLen + 1);
71          rwMsg->setVisitedNodes(visitedLen, nodeId);
72      }
73
74      // Increment walkersMovedThisStep
75      walkersMovedThisStep++;
```

Figure 40: Omnet's `handleMessage()` function

Omnetpp.ini notes:

Note that in an ad hoc network, when doubling the Node amount V‘=2V, the rc signal should adapt to typically a slightly higher than half the amount of the former signal range: $rc' = 2rc$. Considering that the random placement of the vertices has a higher likelihood to cause gaps than before, half the range is not adequate, as I've observed from thousands of simulations that can be replicated in my code by halving the *.numnodes variable in Omnetpp.ini.

4.2.2 Random walker adaptations

Cloning method

Every node in the network uses a static variable for the duplication interval.
After the timestep increases in `HANDLEMESSAGE()`,
there's a call to `DUPLICATEWALKER(RWMSG)`. This function will duplicate the current agent in the node it's currently visiting, setting off two random walkers in its neighborhood. The function is shown on figure 41. It initially checks if the adaptation is enabled on Omnetpp.ini. If it is, there will be a currentDupRound variable defined, which will calculate the current round of duplications $currentDupRound = \frac{timestep}{duplicationInterval}$, provided that duplicationInterval is a positive integer.

The current duplication round is subject to the following condition: a) there exists at least one unvisited node; b) the duplication interval is positive; c) the timestep is positive; d) the timestep divided by the duplication interval, leaves a modulo equal to 0; e) the duplication round is bigger than the last value stored in the random walker.

When that condition is met, the random walker updates its duplication round to the current one, and the `TRIGGERWALKERDUPLICATION(CURRENTDUPROUND)` function is called.

`TRIGGERWALKERDUPLICATION()` is depicted on figure 42. To avoid redundant calculations, I designed this function to receive the currentDupRound variable that was created in (duplicateWalker()) as an argument. This function initializes a new random walker agent in the current node and sends it to a random neighbor using `SENDTORANDOMNEIGHBOR()`.

Multiple random walkers

If the user defines multiple amounts of random walkers in Omnetpp.ini, the simulation will release that amount of agents from the starting node in the same fashion. Stage 0, checks to call `STARTRANDOMWALKER()` enough times to satisfy the user's request. `HANDLEMESSAGE()` ensures to only increment the timestep

once enough random walker agents have incremented the static variable ”walkersMovedThisStep”. Once that variable has incremented enough times as the amount of walkers, the program increments time (since their movement is supposed to be simultaneous) and push the current status to the arrays that will be graphed.

Prohibition of backtracking

Once the user enables this variable on Omnetpp.ini, there will be an interim condition met on the `sendToRandomNeighbor(*msg)` which can be viewed on Figure 39. This according to this, if the random walker has visited more than 1 node already, there will be a previousNode variable based on their `getIndex()`. This is accessed through its getPath(), where the index is the second last element of the array (`getPath(msg->getPathArraySize() - 2)`). Since that node is found, the function defines a new ”candidates” vector, which will hold all the IDs of the neighbors (accesed through gateSize(”port”)).

In Omnet++, each node has a set of gates that start from index 0, which are bidirectional in this instance. The fixed indexing, allows for a simple loop to find all the respective index values that lead to each neighbor. Using the function gate(”port$o”, i), we can assign a cGate type of pointer to hold the neighbor’s outGate during the present iteration of the loop. The end of the outerGate can be fetched using Omnet’s `getNextGate()`, which we can assign to a cGate inGate pointer of the current neighbor. Using `getOwnerModule()`, we now get access to the neighbor module and its Index, If that neighbor index is equal to the previous node in the random walker’s path, it will be excluded from the candidates vector. This ensures that the random walker will not backtrack to the node it just came from.

```
void Node::duplicateWalker(RandomWalkerMsg *rwMsg) {
    if (!enableDuplication) {
        return;
    }

    int currentDupRound = (duplicationInterval > 0) ? (timestep / duplicationInterval) : -1;
    if (!globalAllVisited && duplicationInterval > 0 && timestep > 0 && timestep % duplicationInterval == 0 &&
        rwMsg->getLastDuplicationRound() < currentDupRound) {
        rwMsg->setLastDuplicationRound(currentDupRound);
        triggerWalkerDuplication(currentDupRound);
        EV << "Duplicated walkerId=" << rwMsg->getWalkerId() << " creating new walker. Total now=" << numWalkers << endl;
    }
}
```

Figure 41: The duplication method

```
void Node::triggerWalkerDuplication() {
    // Only the node(s) currently holding a walker will execute this
    // Each walker duplicates itself

    int currentDupRound = (duplicationInterval > 0) ? (timestep / duplicationInterval) : -1;

    RandomWalkerMsg *dupMsg = new RandomWalkerMsg("randomWalkerMsg");
    dupMsg->setPathArraySize(1);
    dupMsg->setPath(0, getIndex());
    dupMsg->setVisitedNodesArraySize(1);
    dupMsg->setVisitedNodes(0, getIndex());
    dupMsg->setHopCountr(0);
    dupMsg->setVisitedPerHopArraySize(1);
    dupMsg->setVisitedPerHop(0, 1);
    dupMsg->setWalkerId(walkerIdCounter++);
    dupMsg->setLastDuplicationRound(currentDupRound); // <-- Set to current round!

    sendToRandomNeighbor(dupMsg);

    numWalkers++;
}
```

Figure 42: Creation of the RW cloned object

5 Results

5.1 Random Walker illustration

To create an illustration of the random movement, I designed a discrete 5x5 grid network with node links parallel to one another without any diagonal links. This symmetrical topology eliminates bias.

Figure 5.1 demonstrates Random Walker behavior in a simple 5x5 grid network (Mirror at `https://github.com/Gkarios/Random-Walks-Omnet/blob/main/REPORT-GIFS/grid%20coverage.gif`).

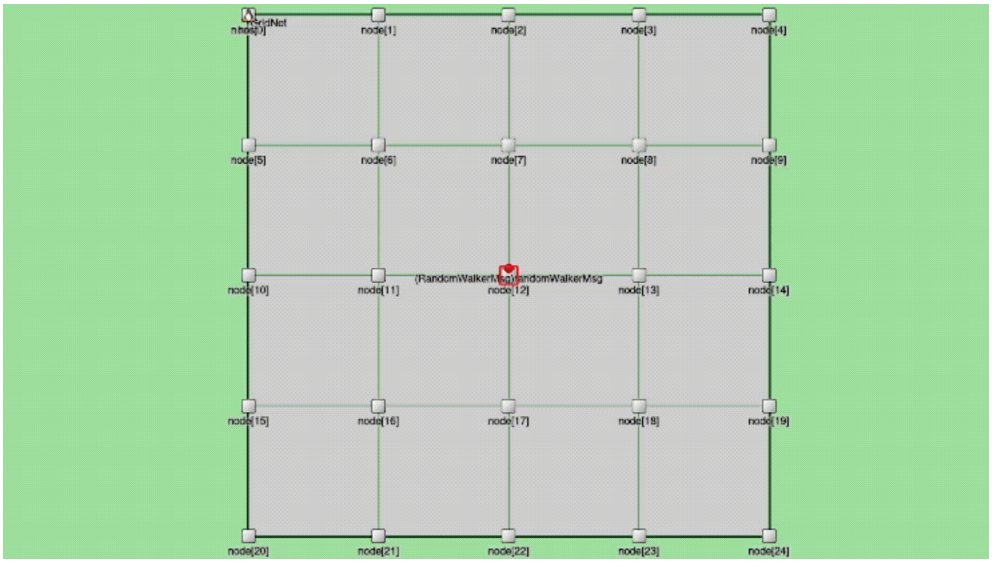

Figure 43: Coverage of a 5x5 grid by a Random Walker planted in the middle point of the diagonal | Simulated in Omnet++

We can see that the random walker efficiently covers the first upper part of the network. It then proceeds to revisit some nodes, and eventually reaches the lower half, where it once again finds a burst of unvisited nodes, until it ultimately covers it entirely. The departure being from the middle point generally gives the agent the best chance to cover the network in the least amount of time as aforementioned by K.L. Besser's work on 2D simulations.

5.2 Multiple random walks

To compare the impact of multiple random walkers, I conducted a controlled RGG simulation for 500 runs.

Simulating for a network of V=50 nodes, rc = 0.4 radius units, with limit of 99.5% average coverage, when comparing rw= 1, 2, 5 random walk agent simulations respectively we are met with the following result on Figure 44:

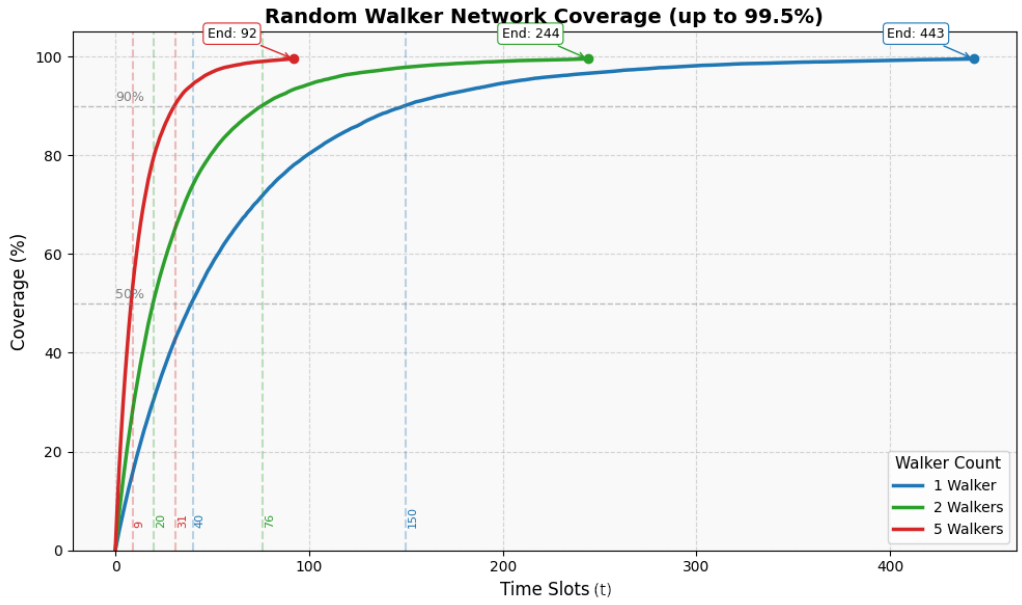

Figure 44: $rw = 1, 2, 5$, $rc = 0.4$, $V = 50$.

Evidently, the simulation of $V = 50$ with 5 random walkers finishes at $T_5 = 92$ time slots. Additionally $T_2 = 244$, and $T_1 = 443$; Thus

$$\begin{aligned} \frac{T_1}{T_2} &= 1.81 \\ \frac{T_2}{T_5} &= 2.65 \\ \frac{T_1}{T_5} &= 4.81 \end{aligned} \tag{14}$$

In a controlled trial of V=100, rc=0.25 radius units we get a similar outcome on Figure 45. For $rw = 1$ it is noticeable that as nodes have doubled, time slots have increased more than doubled as well. Since $T_1 = 1201$, $T_2 = 589$, and

$T_2 = 244$. The rest of the ratios are as follows:

$$\begin{aligned} \frac{T_1}{T_2} &= 2.03 \\ \frac{T_2}{T_5} &= 2.41 \\ \frac{T_1}{T_5} &= 4.92 \end{aligned} \tag{15}$$

Both metrics are roughly close to the expected linear reduction of time slots of 2.5x and 5x, whether the nodes are 50 or 100.

Since the experiments are controlled for shared rc and network components, energy consumption scales linearly with the amount of random walkers. Since time coverage is approximately reciprocal to the amount of walkers, energy consumption should also stay approximately equal in all instances if we suppose no energy-cost during the random walker creation.

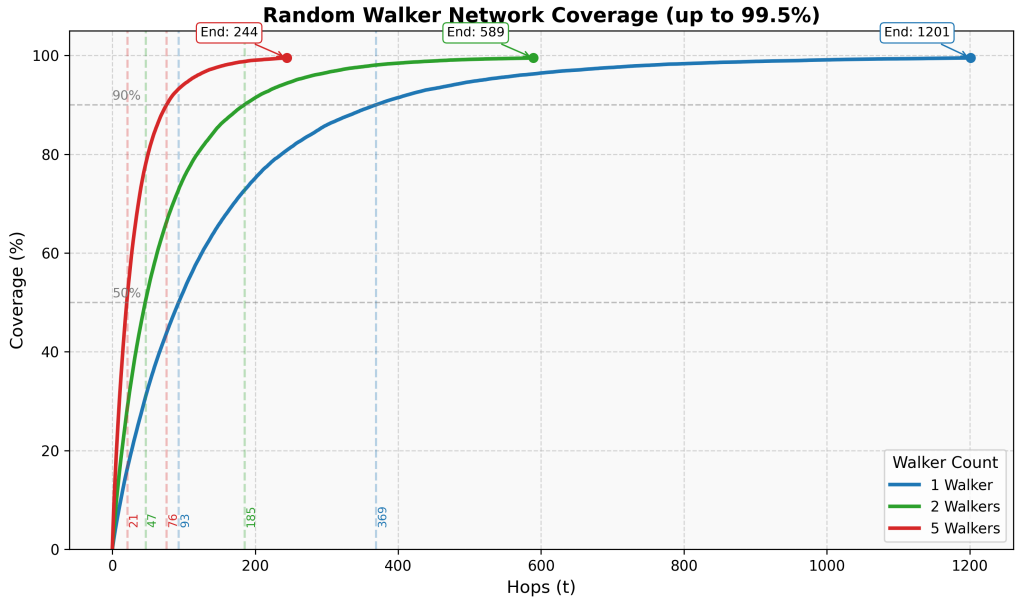

Figure 45: $rw = 1, 2, 5, rc = 0.25, V = 100$

5.3 Cloned Random Walker

Using the clone model, with a 50-timeslot Duplication Interval, results indicate that the simulation ends at $2^4 = 16$ active random walkers, at which point 4 duplications have occurred. Note that 4 random walkers at simulation finish guarantee coverage time to be between $t_p = [200, 250]$ because of the pre-set interval.

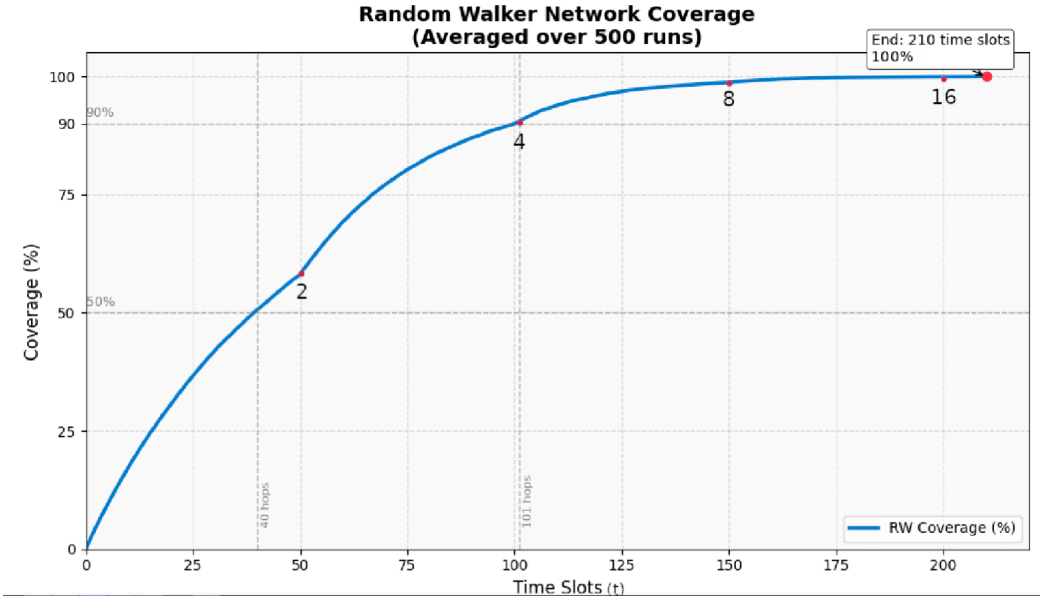

Figure 46: $rw = 1, V = 50, rc = 0.4, DuplicationInterval = 50t$ The annotated red spots demonstrate the duplication time slot, with the current amount of active random walkers in the network.

Figure 46, shows that the duplication begins after half the network has already been covered, substantiating the energy-focused practicality of this adaptation. 4 random walkers start spending energy once 90% coverage has been reached. Whenever duplication occurs, we can notice the rate of coverage on the graph noticeably pick up on the softly marked bullet points. The coverage time of this algorithm is $T_{Clone} = 210$ time slots, lower than the controlled trial of $rw = 2$.

Figure 47 compares the coverage of 2 random walkers vs the clone method. We can notice a significant performance advantage in the clone method, which is about twice as fast as the double random walker coverage, ending with 16 random walkers.

To compare energy costs, we make the presupposition that for V=50 nodes, and the network connected within bounds for the diagonal to be smaller than

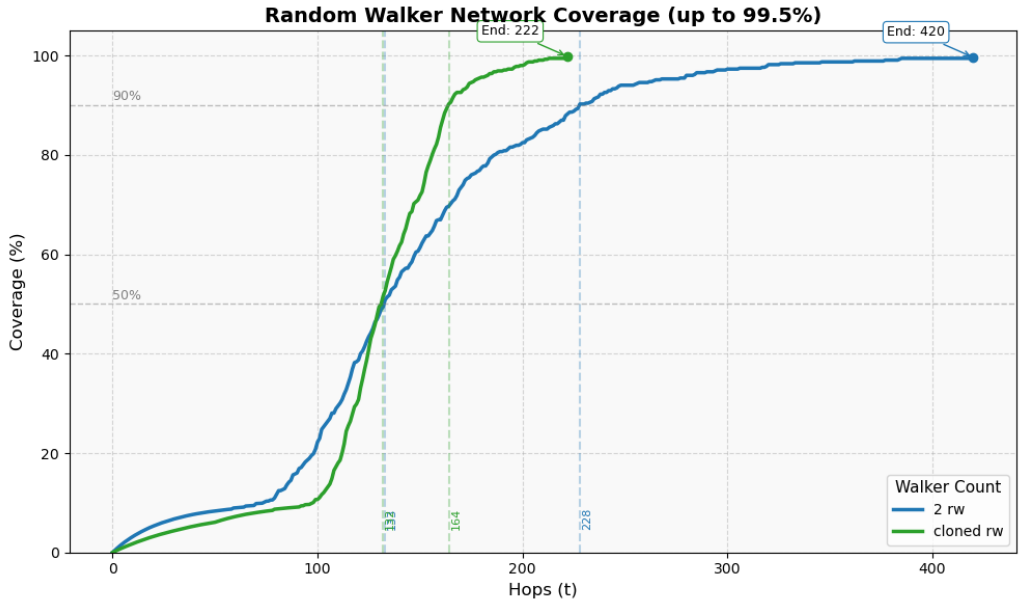

Figure 47: $rw = 2$ vs Clone method, $V = 50, rc = 0.4, DuplicationInterval = 50t, C_T = 0.995$

the threshold $d_0 = 87.7m$.

Thus, we can use the appropriate formula: $E = k(50nj + 10pj)d^2$, where k is the amount of transmissions, and 2 random walkers spend 2k units per time slot, yielding:

$$E_{2rw} = 2 * 240k * (50nj + 10pj) * d^2$$
$$E_{2rw} = 480k * (50nj + 10pj) * d^2$$

E_{clone} utilizes the heurstic function from equation 10.
Resulting in:

$$\begin{aligned} E_{\text{clone-k}} &= k\left(\sum_{i=1}^{3} 2^i\right) * 50 * 2^4 * 10 \\ &= 1102k \\ E_{\text{clone}} &= 1102k * (50nj + 10pj) * d^2 \end{aligned}$$

Comparing these yields an energy efficiency ratio:

$$\frac{E_{2rw}}{E_{clone}} = \frac{480k}{1102k} = 0.436$$

In conclusion, the two random walkers are 56.4% more energy efficient than the clone method, despite being twice as slow.

5.4 Prohibition of backtracking

The backtracking method involves a single random walker. Thus, the energy heuristic function used will be directly reciprocal to the time slots, with the presupposition that the bonus cache memory of this adaptation is negligible.

A simulation of prohibited backtracking for 500 runs, results in figure 48.

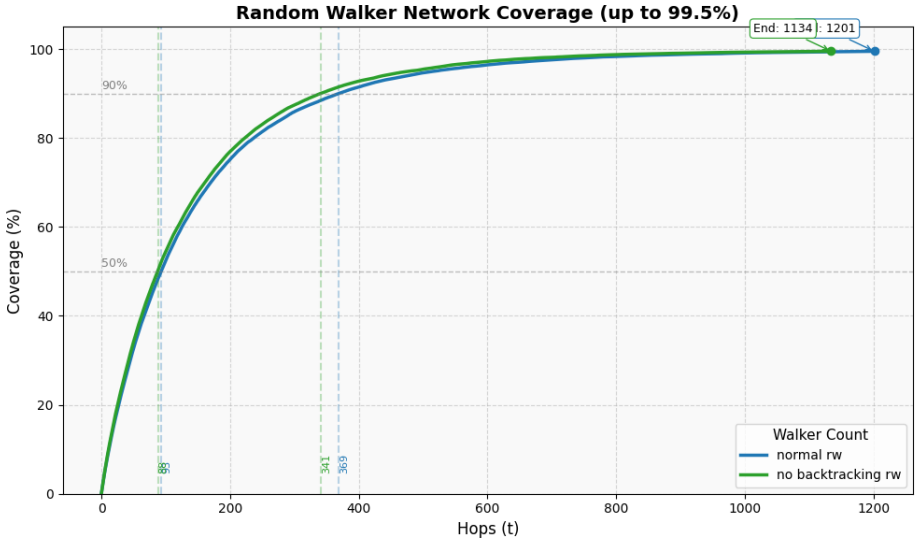

Figure 48: $rw = 1, V = 100, rc = 0.25, C_T = 0.995$

Considering how easy and inexpensive the implementation of the prohibition of backtracking is, the small reduction of coverage time should not be neglected. Figure 48 finds that for $C_P(T_P) = 0.995$, T_P is reduced by 67 time slots, while $C(T_5) = 0.5$ reduces time slots by 1, and $C(T_9) = 0.9$ by 28 time slots.

Expanding the analysis to full coverage C(T)=1 in Figure 49:

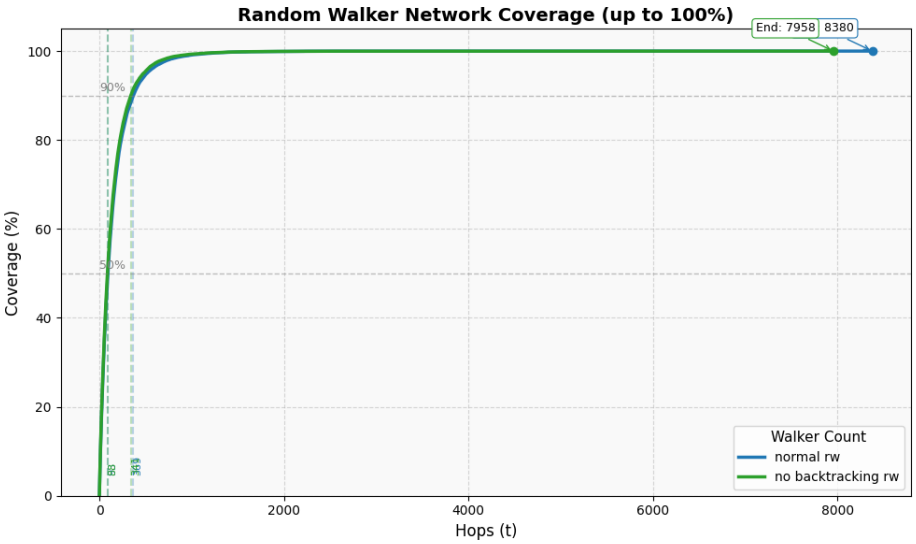

Figure 49: $rw = 1, V = 100, rc = 0.25, C(T) = 1$

The hop reduction is now 422 time slots, reducing complete coverage time T by 5%.

Further tests on a larger network of $V = 500$ nodes with $rc = 0.14$, figure 50 demonstrated a more substantial improvement, achieving a 33.8% decrease in total coverage time. These results indicate that the no-backtracking model can significantly enhance performance, especially in large-scale networks, particularly when complete coverage is required.

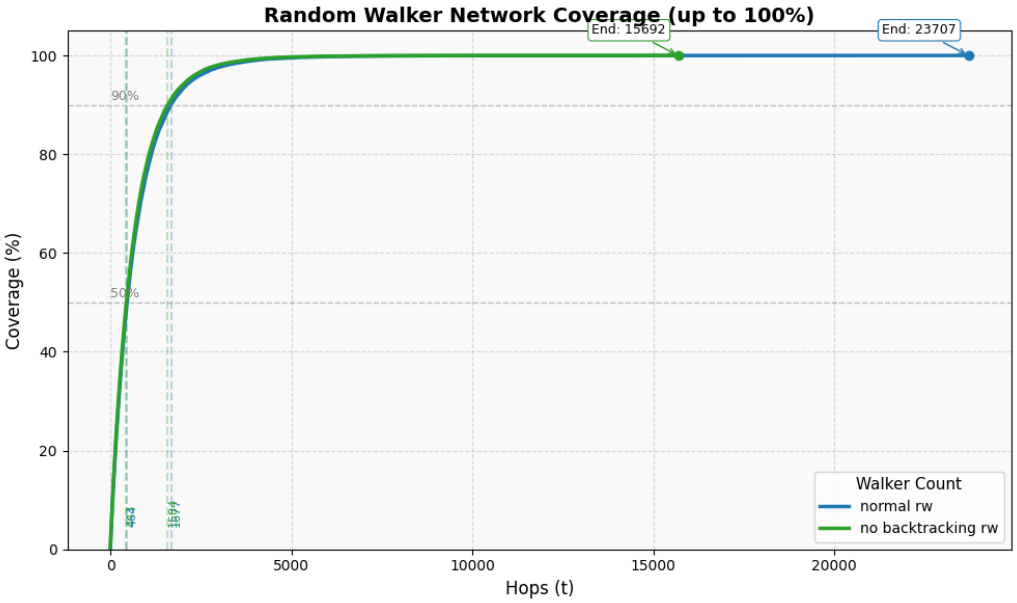

Figure 50: $rw = 1, V = 500, rc = 0.14, C(T) = 1$

5.5 Four algorithm comparison

A comparison of the three adaptations was made with the addition of a hybrid adaptation that combines the cloned random walker with the no backtracking rules.

I ran a controlled simulation for 500 runs, along with the original single-random walker model. Figure 56 compares the performance of the four algorithms together.

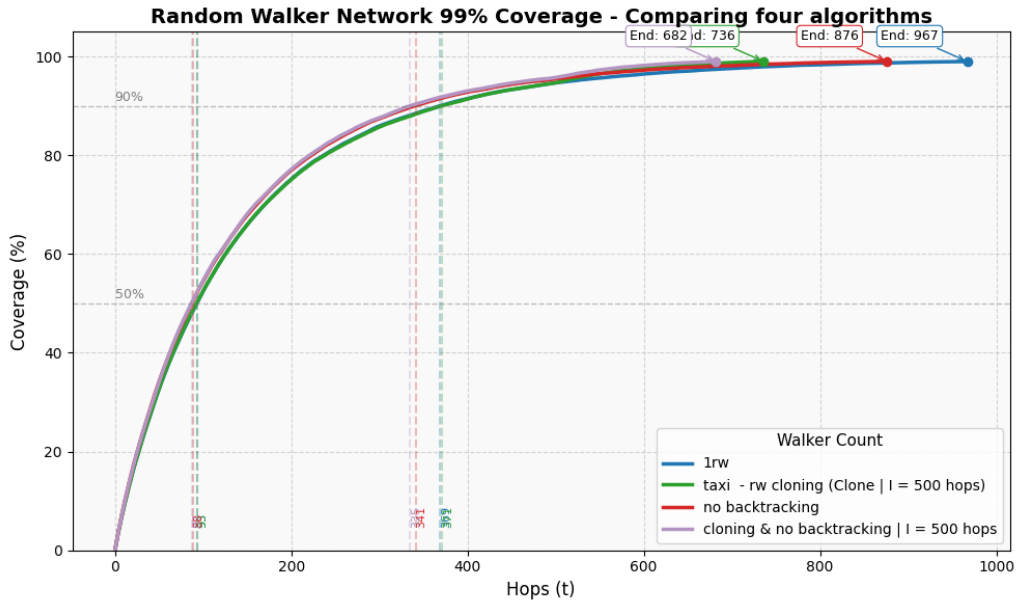

Figure 51: $V = 100, rw = 1, rc = 0.25, C(T) = 0.99, DuplicationInterval = 500t$

Energy comparison

$$
\begin{aligned}
E_{single} &= 967k(50nj + 0.003pj) * (d)^4 \\
E_{NoBT} &= 876k(50nj + 0.003pj) * (d)^4 \\
E_{Clone} &= k * \left(\sum_{i=0}^{0} 2^i \right) * 1000 + 2^1 * 907(50nj + 0.003pj) * (d)^4 \\
&= 972k(50nj + 0.003pj) * (d)^4 \\
E_{Hybrid} &= k * \left(\sum_{i=0}^{0} 2^i \right) * 1000 + 2^1 * 598(50nj + 0.003pj) * (d)^4 \\
&= 864k(50nj + 0.003pj) * (d)^4
\end{aligned}
$$

Accordingly, the hybrid model demonstrates the greatest energy efficiency, achieving an 11% energy saving compared to the single random walker. The no-backtracking model reduces energy use by 9.4%, while the clone method incurs a slight increase of 0.5%.

Time comparison

Regarding coverage time improvements, the no-backtracking model reduces completion time by 10%, the clone method by 31%, and the hybrid model by 42%, when compared to the baseline.

It should be noted that the energy savings estimate for the no-backtracking method does not account for additional cache memory overhead, which may lead to somewhat higher real costs.

Figure 52 compares the same four algorithms but for C(T)=1. The algorithms rank the same, while noticeably adaptations outperform the single-random walker model by a larger margin. This will be further analyzed in Hypothesis 3.

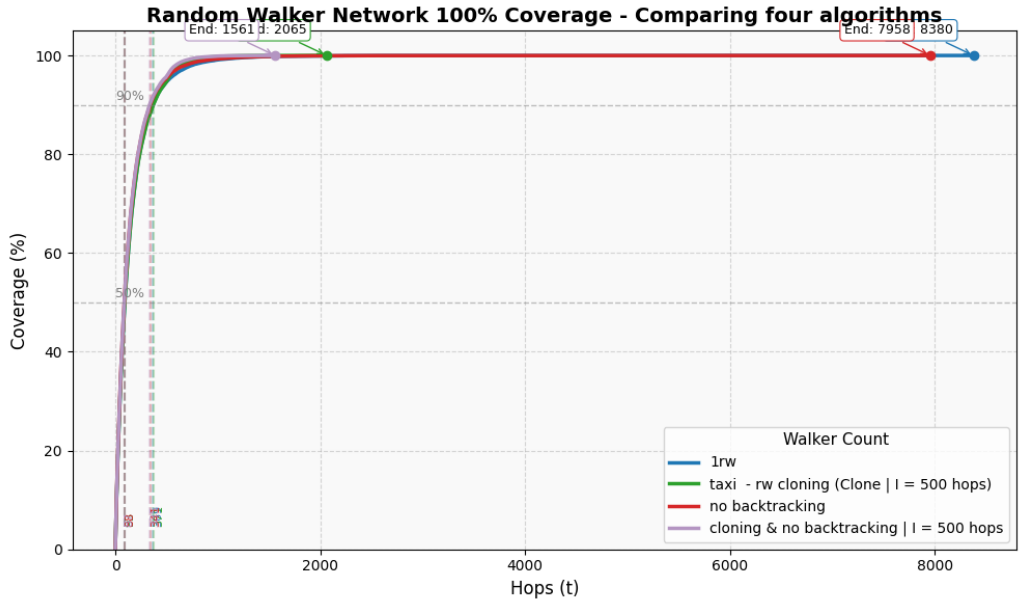

Figure 52: $V = 100, rw = 1, rc = 0.25, C(T) = 1, DuplicationInterval = 500t$

5.6 Taxi stand network

Multiple random walkers and threshold

This section presents results from simulations on a taxi stand network with V=145V=145 nodes, incorporating both a standard communication radius rc=700 and an extended radius longRc=1946, simulating realistic topology and node distribution.

Figure 53 shows that striving for 100% coverage introduces substantial variability. Figures 54 and figure 55 compare multi-random walker simulations for coverage thresholds C(T)=0.99 and C(T)=1.0 respectively.

For C(T)=0.99, coverage time ratios relative to the single-random walker are:

$$\frac{T_1}{T_2} = \frac{6822}{3435} = 1.98$$
$$\frac{T_1}{T_3} = \frac{6822}{2265} = 3.01$$
$$\frac{T_1}{T_5} = \frac{6822}{1311} = 5.20$$

These ratios closely align with the expected linear speedup from increasing the

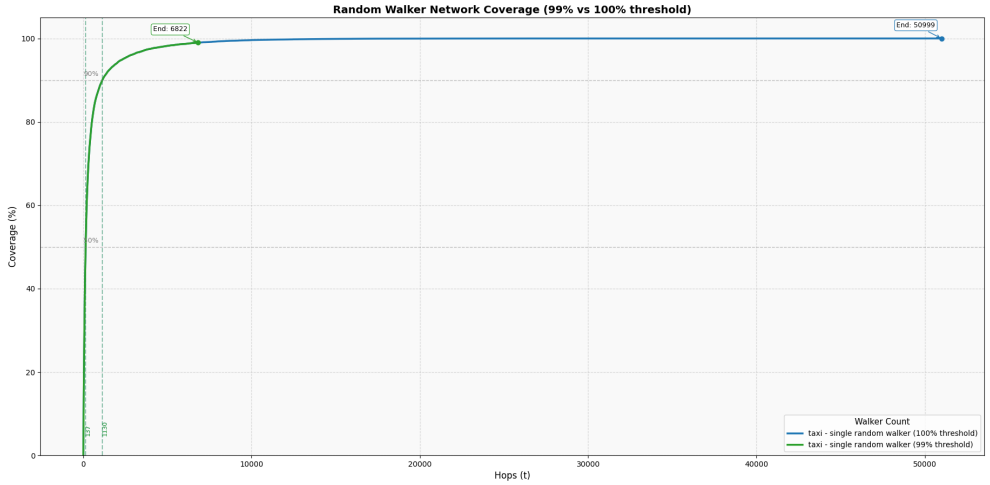

Figure 53: $V = 145, rw = 1, rc = 700, longRc = 1946, T_1 = 6822, T_2 = 50999, C(T_1) = 0.99, C(T_2) = 1$

number of walkers, with five walkers exhibiting slightly better-than-linear improvement.

For full coverage C(T)=1 (figure 55), results display higher variability and uncertainty.Single-walker simulations finished with an average coverage time T1=50999 time slots, approximately 3.27 times slower than two-walker runs. Performance improvements from three to five walkers, however, showed diminishing returns.

Four-algorithm comparison

Simulations were conducted for four algorithms: the original random walker, no-backtracking, clone, and a hybrid model combining clone and no-backtracking behaviors.

Figures 56 and 57 illustrate performance under C(T)=0.99 and C(T)=1 respectively.

At 100 coverage, the hybrid and clone models perform comparably, with negligible benefit observed from the no-backtracking rule. Conversely, at 99% coverage, the hybrid model clearly outperforms the clone model.

Interestingly, at partial coverage (e.g., 50%), the no-backtracking model excels, whereas the clone model barely initiates its first duplication cycle. It's noteworthy that with a more appropriate Duplication Interval results should differ here.

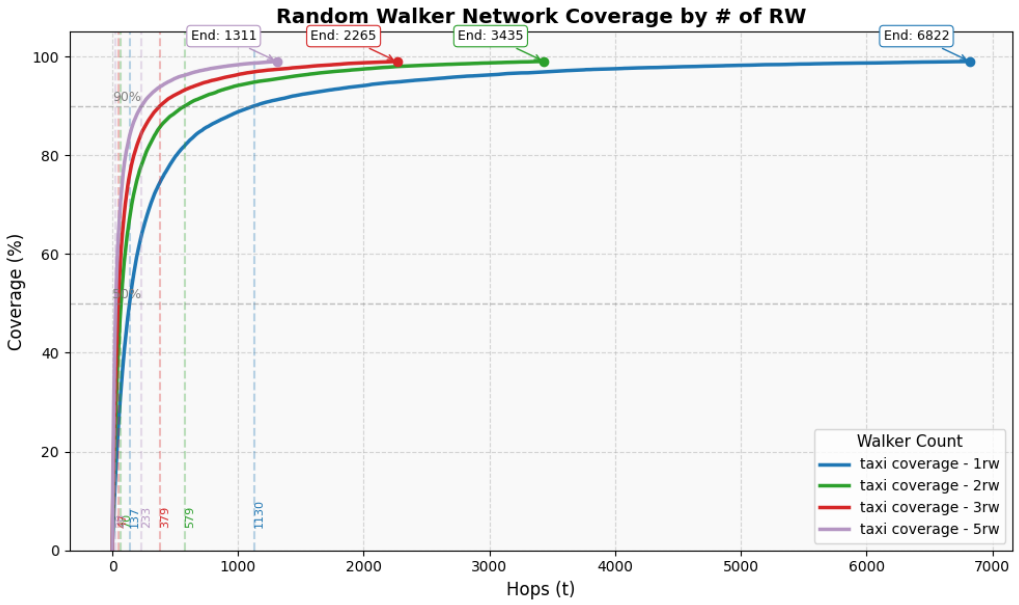

Figure 54: $V = 145, rw = 1, 2, 3, 5, rc = 700, longRc = 1946, C(T) = 0.99$

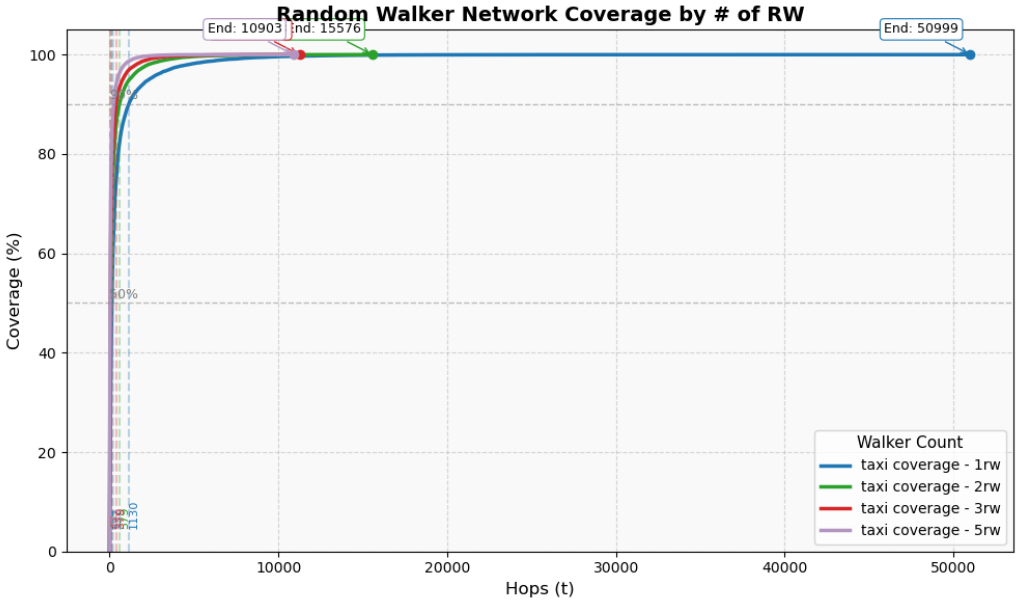

Figure 55: $V = 145, rw = 1, 2, 3, 5, rc = 700, longRc = 1946, C(T) = 1$

Energy and Time at 99% coverage

Based on the single random walker baseline (T1=6822), coverage time reductions for other models are as follows:

$$\frac{T_1}{T_{NoBT}} = \frac{6822}{5275} = 1.29$$
$$\frac{T_1}{T_{Clone}} = \frac{6822}{2907} = 2.34$$
$$\frac{T_1}{T_{Hybrid}} = \frac{6822}{2598} = 2.62$$

The energy costs for each model at C(T)=0.99 is as follows:

$$\begin{aligned}
E_{single} &= 6822k(50nj + 0.003pj) * (d)^4 \\
E_{NoBT} &= 5275k(50nj + 0.003pj) * (d)^4 \\
E_{Clone} &= k * \left(\sum_{i=0}^{1} 2^i\right) * 1000 + 2^1 * 907(50nj + 0.003pj) * (d)^4 \\
&= 4814k(50nj + 0.003pj) * (d)^4 \\
E_{Hybrid} &= k * \left(\sum_{i=0}^{1} 2^i\right) * 1000 + 2^1 * 598(50nj + 0.003pj) * (d)^4 \\
&= 4502k(50nj + 0.003pj) * (d)^4
\end{aligned}$$

Accordingly, the no-backtracking model reduces energy consumption by approximately 29%, the clone model by 42%, and the hybrid model by 51%, relative to the baseline. Note that the no-backtracking estimate may underestimate energy usage due to cache overhead.

As for time management compared to the single-random walker,
the no-backtracking model excels by 29%, the cloned model by 135%, and the hybrid model by 163% respectively.

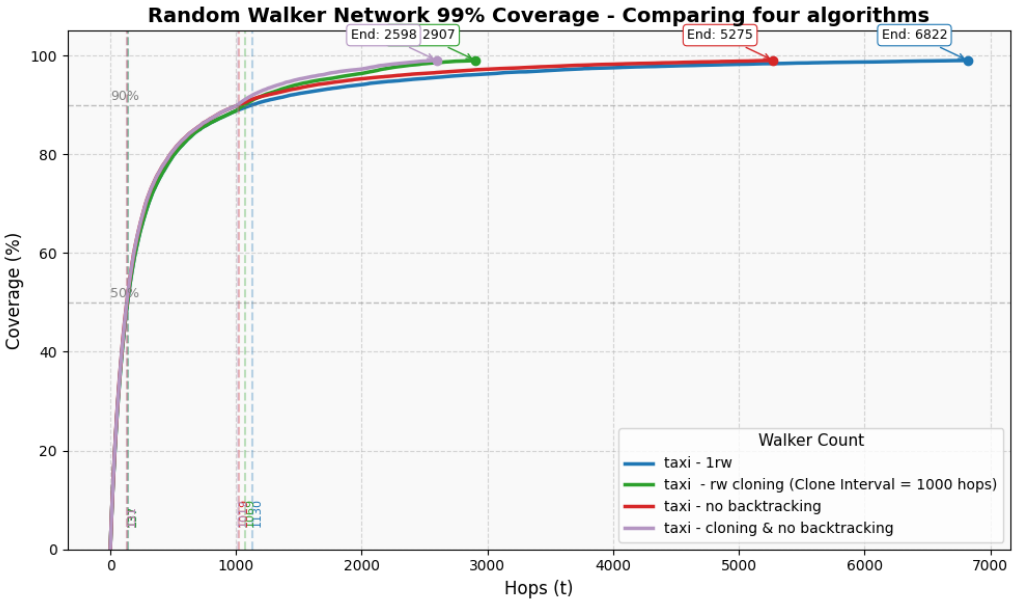

Figure 56: $V = 145, rw = 1, rc = 700, longRc = 1946, C(T) = 0.99$

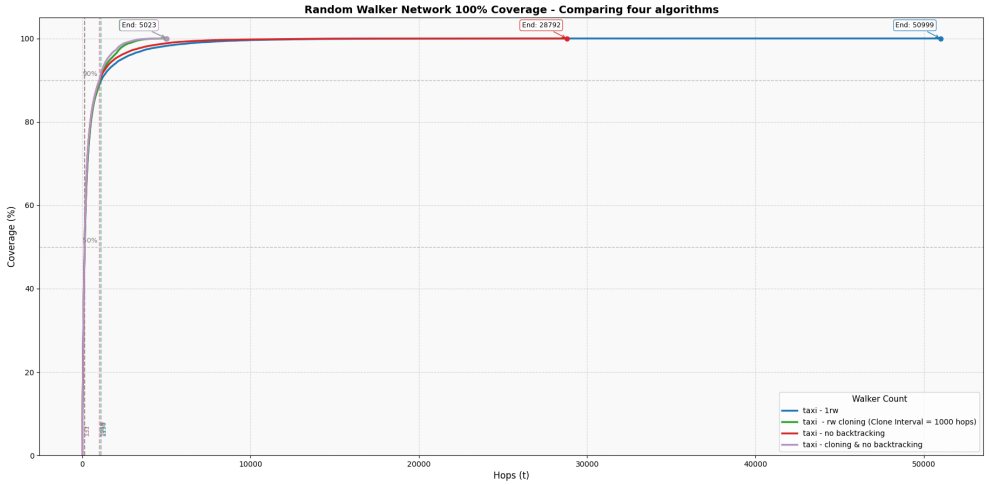

Figure 57: $V = 145, rw = 1, rc = 700, longRc = 1946, C(T) = 1$

5.7 Hypotheses Results

This section evaluates Hypothesis 1 (H1), which states: *Increasing the communication radius rc improves coverage time across the network, but ultimately requires higher energy usage.*

The analysis utilizes the Python rc simulation program, as described in the methodology section.

1X1 network rc analysis:
An initial test was performed on a controlled 1 $\times$1 square network containing V=1000 nodes. For a sparsely connected network with a relative communication radius rc_r defined as

Results in Figure 58 found that for a sparsely-connected network with an $rc_r = \frac{rc}{diagonal(G_{area})}$:
The mininum $rc_r = \frac{0.07}{\sqrt{2}} = 0.05$ with one running random walker, spends half the energy cost compared to the maximum $rc_r = \frac{0.12}{\sqrt{2}} = 0.085$ as expressed on table 2, noting the time per relative radius:

RC-Relative	time-slot
5%	2440
8.5%	1205

Table 2: Notes taken from Figure 58

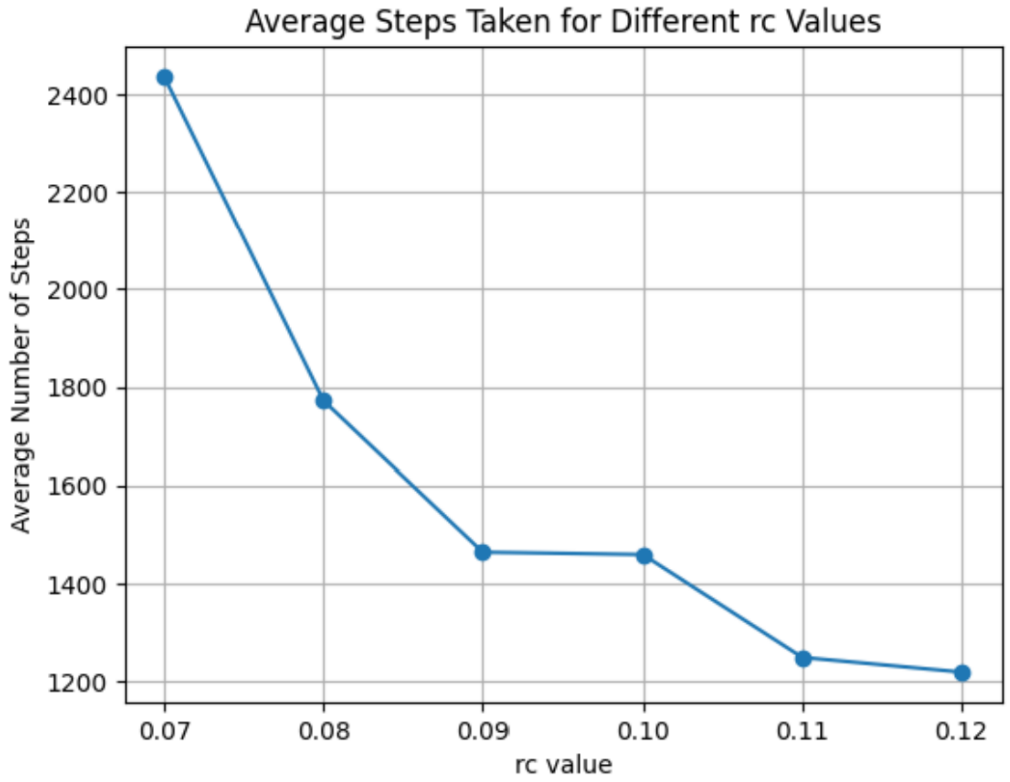

Figure 58: Messages required for network coverage $rw = 1, V = 1000, C(T) = 1$, 1X1 square network. Averaged from 150 Python simulations.

Energy model

Using Heinzelman's first-order energy model, assuming nodes connect within a range roughly half of rc, and following Comaeu's approximation with d=2, the free-space channel model applies since $d < d_0$.

Energy costs for two representative points on Figure 2's network are:

$$E_1 = k(50nj + 10pj) * (0.5d)^2$$
$$E_2 = k/2(50nj + 10pj) * (0.85d)^2$$

With coverage times T1=370 and T2=160 (see table 3 and figure 59).

The ratio of energy costs is approximately:

$$\frac{E_1}{E_2} = \frac{k(50nj + 10pj) * 0.25rc_r^2}{k/2(50nj + 10pj) * 0.72rc_r^2}$$

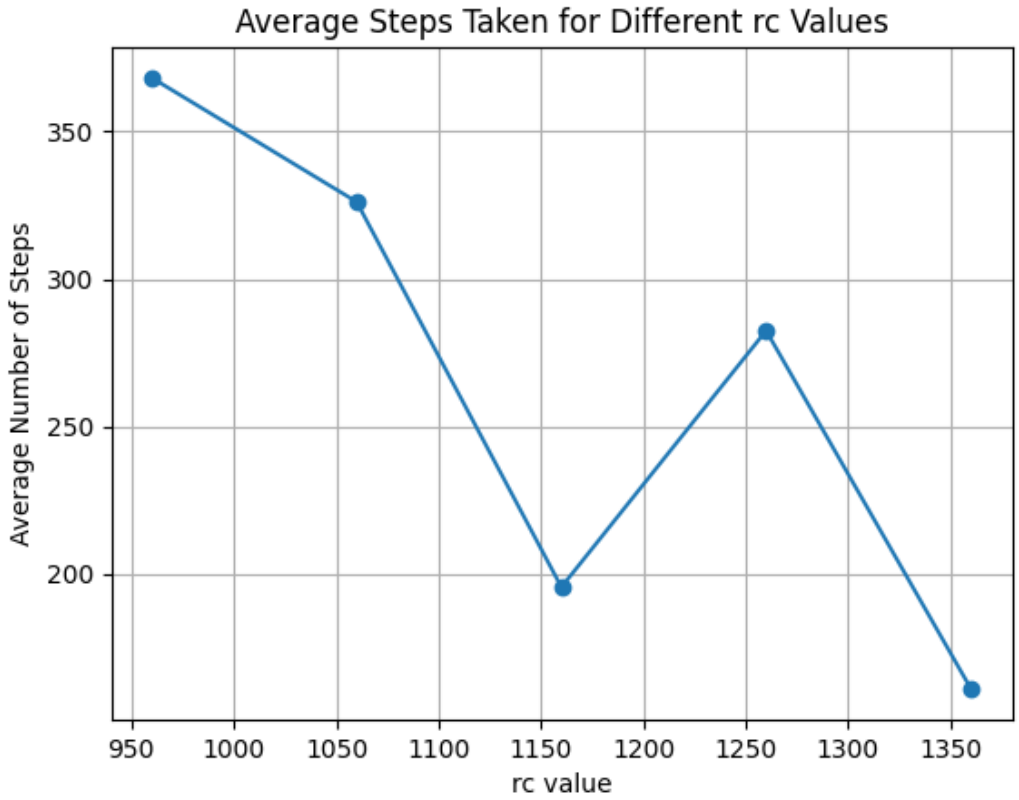

Figure 59: Comparison of different rc values in the 6000X6000 taxi network for an RGG topology. $V = 145, rw = 1, C(T) = 1$. Averaged from 150 Python simulations.

$$\frac{E_1}{E_2} = 0.694$$

Indicating that E_1 approximately spends a 70% of the energy cost that E_2 uses solely due to connection radius differences.

Sample taxi network rc analysis:
The taxi network is characterized by a diagonal distance of 8485.28 units (exceeding the cutoff $d_0 = 87$m). Thus, the free-space channel model does not apply (see equation 9).

This induced a redesign of the rc simulation depicted in Figure 59. The new network built consists of $V = 145$ in a 6000X6000 area. Test rc values range from 860 to 1360 units.

Since, we've established this network to be multipath channel, the energy costs for the two radii values are:

RC-Relative	time-slot
11%	370
16%	160

Table 3: Notes taken from Figure 59 (starting poitn value vs end-point value)

$$E_1 = 370k(50nj + 0.003pj) * (0.11d)^4$$
$$E_2 = 160k(50nj + 0.003pj) * (0.16d)^4$$

yielding a ratio:

$$\frac{E_1}{E_2} = 0.517 \tag{16}$$

which implies that the smaller radius rc, at approximately 0.69% of the length of the higher value, spends about 51.7% of the energy of the larger radius, despite taking over twice as long to complete coverage.

This analysis confirms Hypothesis 1: increasing the communication radius improves coverage time at the cost of increased energy consumption. The trade-off is significant and varies depending on network topology and channel models utilized.

***From H2**: Multiple random walkers will underperform linearly in terms of coverage time, since bottlenecks will punish the agents linearly, and multiple walkers will perform excess movements.*

Hypothesis 2 posited that multiple random walkers would underperform linearly in coverage time. However, analysis of time ratios presented in Equations 14 and 15 contradicts this expectation.

Surprisingly, coverage times scale close to linear expectations without any evident compromise of the suspected constraints. This finding indicates that increasing the number of random walkers leads to roughly proportional improvements in coverage time.

To my surprise, the time ratios kept close to linear expectations and did not underperform, and no type of compromise was observed.

***From H3**: The taxi stand network will behave differently to the RGG network. Performance comparisons will be skewed; though I don't expect the ranking placement of each algorithm adaptation to change.*

Comparing energy and coverage time advantages relative to the single random walker baseline in both network types (of Figures 51 and 56 for C(T)=0.99) reveals the following:

- The no-backtracking model shows a 208.5% higher energy efficiency advantage and a 190% greater time advantage in the taxi stand network compared to the RGG.
- The hybrid model's energy and time advantages are 21.4% and 288% higher, respectively, in the taxi stand network.
- The clone model contrasts the trend, being 0.5% more expensive in the RGG but 42% less expensive in the taxi stand network, while demonstrating a 235.5% lower time advantage in the taxi stand setting.

These results find a significant behavioral differences. One reason attributed is the scalability of the simulations, with one taking significantly more time slots to finish. This is partly caused by the node population difference as well, considering the taxi stand network consists of 45 more nodes. For instance, if the same comparison is made for RGG networks in figures 51 and 52, we will also find alike results. With no-backtracking time comparison being $\frac{10}{5} - 1 = 100\%$ more advantageous, with hybrid at $\frac{436.8}{42} - 1 = 928.6\%$, and clone at 886.4% respectively. It's evident that adaptations become more effective as the run time scales.

Results between RGG and taxi stand network likely diverge more due to the longRc complications of the taxi network, which were not accounted for.

On the whole, Hypothesis 3 is invalid, as the ranking placement of each algorithm adaptation did notably changed for the cloned-random walker. Additionally the reasoning for the divergence seems to be multifaceted, with the network's scalability being the main cause, as well as the longRc implications.

6 Conclusions

While the random walker demonstrates strong effectiveness in small WSN systems, as the environment scales, it becomes more likely that some of its adaptations introduce previously neglected benefits that embolden a covert opportunity cost.

Among these, the Hybrid clone/no backtracking adaptation emerges as the most efficient, offering the best performance in both energy consumption and coverage time. In all instances the no-backtracking adaptation proved to be beneficial to the single-random walker model. Lastly, the cloned-random walker adaptation, while faster, proved very strong when simulations lasted longer or in large-scale networks. Conversely it performed worse than the single-random walker in a 500 run averaged small-scale simulation.

Multiple active random walkers exhibited near-linear improvements in coverage time without noticeable degradation, contrary to my initial assumption.

Moreover, The radius signal was found to consume substuntial amounts of energy over longer distances, emphasizing the importance of minimizing its range to conserve battery.

The taxi network deviated from the RGG results, demonstrating greater efficiency under the random walker adaptations.

Lastly, I'll note that I found my definition of "rc relative" rc_r to be a useful metric for comparing networks of different scale.

6.1 Future work and recommendations

I recommend further investigation into the no-backtracking model, as it seemed a safe choice for any WSN type, with minimal drawbacks. I did not attempt to quantify a heuristic function for it, as I would need much more insight on the energy impact of its additional complexity. My informed intuition suggests that the energy overhead is infinitesimal, and the model may even overperform the cloned random walk model.

Since the cloned random walker model is so dependent on the duplication interval, citing the previous lost performance in comparison to the single-random walker, it would be curious to see the program itself analyze the network and suggest an appropriate interval at simulation start.

Lastly, I suggest future work to examine a model that accounts energy costs for multiple running rc range networks as this current energy model is not apt for this.

References

[1] Lingxuan Hu and David Evans. Secure aggregation for wireless networks. Master's thesis, University of Virginia, 2003.

[2] S. Jaloudi. Communication protocols of an industrial internet of things environment: A comparative study. *Al Quds Open University*, 2019.

[3] S. Deepak Raj and H.S. Ramesh Babu. Identification of intelligence requirements of military surveillance for a wsn framework and design of a situation aware selective resource use algorithm. *IETA*, pages 253–254, 2022.

[4] Laura Wood. (iwsn) a forecasted $29.5 billion industry. *Research and Markets*, 2025.

[5] Kahtan Aziz, Saed Tarapiah, Mohanad Alsaedi, Salah Haj Ismail, and Shadi Atalla. Wireless sensor networks for road traffic monitoring. *International Journal of Advanced Computer Science and Applications*, 6:265–270, 12 2015.

[6] Suoping Li, Qianyu Xu, Jaafar Gaber, Zufang Dou, and Jinshu Chen. Congestion control mechanism based on dual threshold di-red for wsns. *Wireless Personal Communications*, 115, 12 2020.

[7] Ershadul Haque and M. A. Hannan. *Toward Optimum Topology Protocol in Health Monitoring*, pages 81–109. 01 2019.

[8] Karl Pearson. The problem of the random walk. *Nature*, 72:294–294, 1905.

[9] P. Pierre D. Randal, M. Eric. Markov chains. *Springer International Publishing*, 2018.

[10] K. Skiadopoulos. Information dissemination and connected sets in wsns. *Ionian University*, page 21, December 2019.

[11] Chen Avin and Bhaskar Krishnamachari. The power of choice in random walks: An empirical study. *Computer Networks*, 52(1):44–60, 2008. (1) Performance of Wireless Networks (2) Synergy of Telecommunication and Broadcasting Networks.

[12] I. Stavrakis L. Tzevelekas, K. Oikonomou. Random walk with jumps in large-scale random geometric graphs. *Elsevier*, 2010.

[13] Robert B. Allan and Renu Laskar. On domination and independent domination numbers of a graph. *Discrete Mathematics*, 23(2):73–76, 1978.

[14] Asya Natapov, Danny Czamanski, and Dafna Fisher-Gewirtzman. Visuospatial search in urban environment simulated by random walks. *International Journal of Design Creativity and Innovation*, 4:5–7, 12 2015.

[15] Michael Fowler. The one dimensional random walk. *Virginia University - Physics Galileo & Einstein*, 2017.

[16] Karl-Ludwig Besser. On the distribution of a two-dimensional random walk with restricted angles. *IEEE*, page 2, 2025.

[17] Michael J Plank Edward A Codling and Simon Benhamou. Random walk models in biology. *The Royal Society*, page 814, 2008.

[18] W.R. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. *IEE-xplore*, pages 10 pp. vol.2–, 2000.

[19] Frank Comeau and Nauman Aslam. Analysis of leach energy parameters. *Procedia Computer Science*, 5:933–938, 2011. The 2nd International Conference on Ambient Systems, Networks and Technologies (ANT-2011) / The 8th International Conference on Mobile Web Information Systems (MobiWIS 2011).

[20] Haifeng Qian, Sani R. Nassif, and Sachin S. Sapatnekar. Random walks in a supply network. In *Proceedings of the 40th Annual Design Automation Conference*, DAC '03, page 93–98, New York, NY, USA, 2003. Association for Computing Machinery.

[21] Mohsen Ghasemi Nezhadhaghighi and Abolfazl Ramezanpour. Efficiency of energy-consuming random walkers: Variability in energy helps. *Phys. Rev. E*, 111:014301, Jan 2025.

[22] Rushabh Patel, Andrea Carron, and Francesco Bullo. The hitting time of multiple random walks. *SIAM Journal on Matrix Analysis and Applications*, 37(3):933–954, 2016.

[23] Maximilian Egger, Ghadir Ayache, Rawad Bitar, Antonia Wachter-Zeh, and {Salim El} Rouayheb. Self-duplicating random walks for resilient decentralized learning on graphs. *tumfis*, pages 2960–2965, 2024.

[24] Himanshu Sharma, Ahteshamul Haque, and Frede Blaabjerg. Machine learning in wireless sensor networks for smart cities: A survey. *Electronics*, 10(9), 2021.

[25] Wan-Kyu Yun and Sang-Jo Yoo. Q-learning-based data-aggregation-aware energy-efficient routing protocol for wireless sensor networks. *IEEE Access*, 9:10737–10750, 2021.

[26] Marek Biskup. Pcmi-notes: Random walks. *UCLA Mathematics*, page 13, 2007.