How Many Oracle Calls Does It Take to Locate a Lightbulb?

MSc Thesis *(Afstudeerscriptie)*

written by

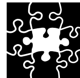
Dominique Marie Danco
(born November 17, 1994 in Burlington, Vermont, USA)

under the supervision of **Dr. Malvin Gattinger** and **Dr. Luca Corolli**, and submitted to the Examinations Board in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam.*

Date of the public defense:	**Members of the Thesis Committee:**
August 21, 2023	Dr. Benno van den Berg (chair)
	Dr. Luca Corolli (supervisor)
	Dr. Malvin Gattinger (supervisor)
	Dr. Ronald de Haan
	Dr. Rebecca Reiffenhäuser



Institute for Logic, Language and Computation

Contents

1 Introduction **6**
1.1 Related work . . . 9

2 Problem description **11**
2.1 Key definitions . . . 11
2.2 Additional definitions . . . 13
2.3 Challenges & pitfalls . . . 14
2.3.1 Symmetry . . . 14
2.3.2 RSSI reliability . . . 15
2.4 Computational complexity . . . 18
2.4.1 Oracle calls . . . 18
2.4.2 Search complexity . . . 18

3 SAT solver approach **20**
3.1 First-order logic representation . . . 20
3.2 ASP & Clingo . . . 23
3.3 Fuzzy constraint representation . . . 24
3.4 Issues . . . 25

4 Algorithmic approaches **26**
4.1 Basic greedy algorithms . . . 26
4.1.1 Numerical estimation . . . 26
4.1.2 Comparison algorithm . . . 30
4.2 Speed optimizations . . . 31
4.2.1 Witness pairs . . . 31
4.2.2 Iterative approach . . . 34
4.2.3 Final pseudocode . . . 36
4.2.4 solve . . . 36

4.2.5 chooseSolution 36
4.2.6 solutionCorrect 39
4.3 Termination and correctness 39

5 Data & evaluation 42
5.1 Simulated data 42
5.2 Real-world data 42
5.2.1 Averaging over time 43
5.2.2 Data spread 44
5.2.3 RSSI symmetry 44
5.3 Self-healing when a node has inaccurate output or input . . . 45
5.4 Determining node issues 46
5.5 In-depth example 49
5.6 Implementation 50
5.7 Performance 50

6 Conclusion 53
6.1 Future work 54
6.1.1 3-Dimensional spaces 54
6.1.2 Clustering 54
6.1.3 Oracle implementation 55

Abstract

Wireless nodes in smart buildings have many benefits, but one of their drawbacks is the added complication of mapping the nodes on a floor plan after installation. Currently, it is common practice to manually record the location of each placed node. This is labor intensive and prone to error. In this project, we attempt to create a system that will reduce this manual labor by locating the nodes using the Received Signal Strength Indicator (RSSI) data that each node receives from its neighbors. By comparing the possible locations of each node with the strength of these signals, we can craft a constraint solving problem to determine the most likely mappings of the nodes. RSSI data is notoriously inaccurate and inconsistent, so we have built a system that combines constraint solving and human input, and has a high tolerance for error. Our system has been shown to suggest the correct solution after asking a small number of questions to the user, to clarify between symmetric placement options or ambiguous node placement due to inconsistent signal strength. While the system will need improvements to scale out to scenarios with hundreds or thousands of nodes, it is a successful proof of concept that demonstrates that a balance can be struck between automatic and manual strategies, given a constraint problem with unreliable data.

Acknowledgements

To my parents, for their unwavering love and support whenever I am in the world and in my life.

To my supervisors, Malvin and Luca. Luca, though we never got to meet in person, our calls were always interesting and insightful. And Malvin, every time I left one of our meetings, I had regained any of the optimism and enthusiasm about the project that I might have lost since the last one. Thank you both for helping to keep the journey enjoyable.

To all my friends at the MoL, for making me feel more at home than I have with any other group of classmates. I hope we find ways to see each other for years to come. And to Jasper, for everything.

Chapter 1

Introduction

This thesis concerns a real world problem, which is relevant for professionals working with wireless node systems. The project was proposed by a company[1] which specializes in smart building technology. The core technology of its installations are the lighting systems. The company installs wireless lights throughout the building, each of which can send and receive messages to each other and to other systems in place. Lights will henceforth be referred to as nodes. “Wireless” here means that the nodes are not connected to each other. (Nodes are still wired for electricity.) We need the nodes to be mapped on a floor plan so that the correct nodes can send and receive relevant messages relative to their positions.

Because these nodes are relatively low-cost, and located throughout the building, they are a useful resource for asset tracking. This is the process by which people in the building can locate items – for example, an ultrasound machine in a hospital building – by checking which node(s) it is close to, and deducing where in the building it must be.

This process depends on a key detail: that we know where in space each of the nodes is. However, this is not trivial. As the nodes are wireless, regardless of what complicated or simple installation directions are given to the installation crew, it is possible (and in practice, probable) that mistakes arise and nodes are misplaced. When this happens, it can lead to a loss of trust in the system, as the technology cannot be properly used. Even in situations where the installation and node mapping is done perfectly, this is a time- and labor-intensive process.

The goal of this project is to create an semi-automated process (with a

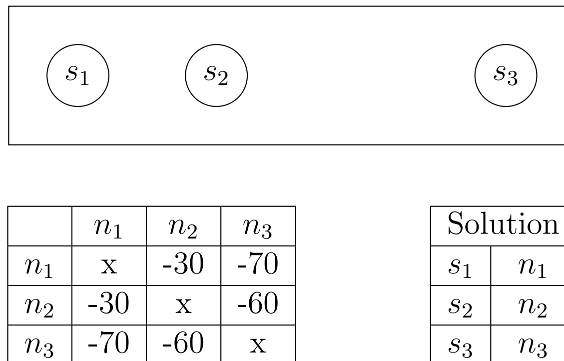
[1] Company name omitted by request.

human-in-the-loop) which determines where each node is located, based on the strength of the signals that each node receives and sends to its neighbors.

Received Signal Strength Indicator (RSSI) is a measurement of how well one device can hear another. We can think of it as a proxy for distance (further apart nodes will hear each other less well), though as we will elaborate upon later, there are many reasons why this might not always be an accurate estimator.

Consider the following example in Figure 1.1. We have three sockets arranged in a line, and three nodes: n_1, n_2, and n_3. RSSI values can range between -255 dBm (decibel milliwatts) and 0 dBm, with 0 as the strongest signal and -255 as the weakest. In practice they are always measured as negative values–even a very strong signal is generally not close to 0. Throughout this thesis, RSSI values should be assumed to be in dBm, and distance values in meters, unless otherwise specified.

From the table of RSSI values, we can quickly see that n_1 and n_3 are the farthest from one another (because -70 dBm is the weakest signal in the table), and n_2 must be in the middle. If we can trust the RSSI values to be relatively accurate, there is only one possible solution to where the nodes may be placed, as n_2 should be closer to n_1 than to n_3, based on the RSSI values.



	n_1	n_2	n_3
n_1	x	-30	-70
n_2	-30	x	-60
n_3	-70	-60	x

Solution	
s_1	n_1
s_2	n_2
s_3	n_3

Figure 1.1: Toy example with unique solution.[2]

Effectively, this becomes a graph isomorphism problem with a twist. The first graph represents the physical space – vertices are socket locations with coordinates, edges are weighted by Euclidean distance. The second graph represents the RSSI values – vertices are nodes and edges between nodes are

[2]ChatGPT was used to generate the TikZ code for figures.

weighted by the RSSI values between those two nodes. Figure 1.2 represents the graph isomorphism visualization of the same example from Figure 1.1. We are not looking for a strict isomorphism, as our graphs use different types and the values of weights are not equal – rather, we are looking for the mapping which best relates the relationships (spatial and signal-wise) between the vertices of the graphs.

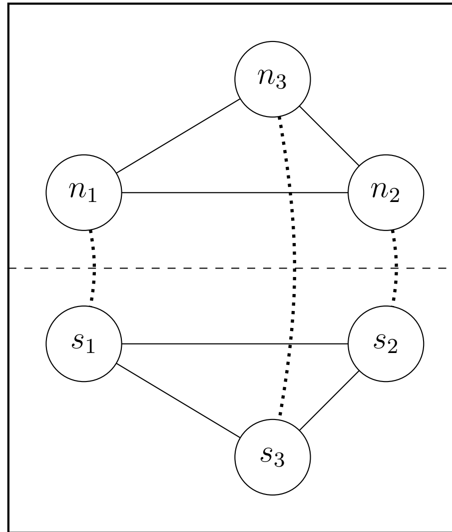


Figure 1.2: Isomorphism between socket graph and node graph.

The special cases that we have for this problem are as follows. First, we cannot fully trust the edge weights of the RSSI graph, due to the signal noise involved and the unreliability of RSSI values (as will be discussed in Section 1.1). Second, we effectively have access to an oracle which can tell us node locations on demand. This oracle (representing the human-in-the-loop) should be used as effectively and sparingly as possible, as the goal of this project is to reduce the human effort needed for this task.

The thesis is structured as follows: in Chapters 1 and 2, we discuss the theoretical setup, background, issues, and complexity of the problem. In Chapters 3 and 4, we discuss possible algorithmic approaches. In Chapter 5, we evaluate our algorithms on data, both simulated and real-world. Finally, in Chapter 6 we conclude and discuss further remaining work.

1.1 Related work

RSSI has historically been used with mixed success for various localization problems. Generally, RSSI signals are used to track objects as they are moving through space, not stationary as they will be in our scenarios.

For example, Shams and Haratizadeh [2018] created a system to use RSSI and graph-based methods for indoor subarea localization – the ability to detect in which subarea an object or person is using the RSSI signals between various moving devices (for example, a person's cell phone) and stationary devices located throughout the space, such as WiFi routers.

The fact that RSSI signals can be captured from a variety of pre-existing devices makes them an easy and affordable option for this type of task, but these methods are not necessarily accurate [Patwari et al., 2003].

Bulten et al. [2016] attempted to locate nodes using a combination of both node-to-node RSSI data (as we will be using) as well as motion data of users in the space. Their attempt was partially successful, but still suffered from inaccuracy due to noise. In our system, we will attempt to use only the node-to-node data, and not the human motion data.

Benkic et al. [2008] warned that "RSSI is, in fact, a poor distance estimator when using wireless sensor networks in buildings. Reflection, scattering and other physical properties have an extreme impact on RSSI measurement and so [they] can conclude: RSSI is a bad distance estimator."

However, there are findings that gave some level of hope for this type of task. For example, one paper titled "RSSI is Under-Appreciated" [Srinivasan and Levis, 2006] argued as such; the authors claim that RSSI is more accurate and consistent than it has been given credit for. Other findings give mixed results. Ramirez et al. found that "the positioning accuracy could reach 10 cm when the beacons and scanners were at the same horizontal plane in a less-noisy environment. Nevertheless, the positioning accuracy dropped to a meter-scale accuracy when the measurements were executed in a three-dimensional configuration and complex environment" [Ramirez et al., 2021].

This last result is interesting in the context of our problem. Various buildings in which this system might be useful might be wildly varying with regards to noise and layout. A sparsely-filled warehouse with a large 2-D grid of lights might be subject to less noise than a 3-D hospital, filled with moving parts and differing levels. It is very possible that our solution might work well in some of these environments, but not be a feasible solution in others.

A previous initial attempt was made by a student based at the same company that proposed our current project. That attempt was ultimately unsuccessful except for the simplest, least-complex cases [Said, 2022]. In this thesis we will attempt to build a system that is usable for a range of scenarios and can tolerate the amount of messiness that is to be expected in RSSI data.

The common trend across previous RSSI localization research is that it can be successful and has potential, but is plagued by issues of reliability. In situations where the correctness of our solution is of high importance, it is important to create a system where such guarantees can be made. It is for this reason that we will build a semi-automated system which uses both automated logical reasoning and manual human input to come to a reliable conclusion.

Chapter 2

Problem description

In this chapter, we will formally define the problem and all of the relevant types and functions. We will then discuss the particular difficulties of this problem, namely, why it is that for many cases (in fact, all cases when we don't have guarantees of the accuracy of our data), there will not only be one possible solution, but many.

2.1 Key definitions

Definition 1 (Socket)**.** *A* ***socket*** *is a tuple* $s_i = (i, c)$ *where* $i \in \mathbb{N}$ *is the ID of the socket and* $c = (x_i, y_i) \in \mathbb{R}^2$ *represents the coordinates of that socket in 2-dimensional space*[1]*. We assume distinct sockets to have distinct IDs and distinct coordinate locations.*

Definition 2. *The* ***distance*** *between two sockets* $s_i = (i, (x_i, y_i))$ *and* $s_j = (j, (x_j, y_j))$ *is the Euclidean distance between the sockets:*

$$\mathsf{dist}(s_i, s_j) = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}.$$

Definition 3 (Node)**.** *A* ***node*** n_i*,* $i \in \mathbb{N}$*, represents a wireless light.*

Definition 4 (Problem)**.** *A* ***problem*** *is a tuple* $P = (\mathcal{N}, \mathcal{S}, \mathsf{RSSI})$*, where* $\mathcal{N}$ *is a set of nodes,* $\mathcal{S}$ *is a set of sockets,* $|\mathcal{N}| = |\mathcal{S}|$*, and* RSSI *is a function with type* $\mathcal{N} \rightarrow \mathcal{N} \rightarrow [-255, 0]$.

[1] The problem can easily be generalized to n-dimensional space, as discussed in Section 6.1.1. For this thesis, we will limit the problem to 2-dimensional space.

Example 1. *For a problem* $P = (\mathcal{N}, \mathcal{S}, \mathsf{RSSI})$*,* $\mathsf{RSSI}(n_1, n_2) = -100$ *indicates that the RSSI value strength received by* n_1 *and sent from* n_2 *is* -100*.*

A larger RSSI value (closer to 0) indicates a stronger signal. RSSI values between two nodes are not necessarily symmetric (due to environmental factors, manufacturing defects, etc.), so the order of node arguments to the RSSI function matters when we are dealing with the raw data. We can preprocess this data to create symmetry with regards to argument order, as will be discussed further in Section 5.2.3.

Single RSSI values are always integers, but as we will be averaging values over time, as discussed in Section 5.2.1, we will treat them as type $\mathbb{R}$.

Definition 5 (Oracle)**.** *An* ***oracle*** *is a function with type* $\mathcal{N} \rightarrow \mathcal{S}$*, and indicates in which socket a given node is placed.*

Our oracle represents a human operator who can determine the location of any given node. In practice, this would involve the operator causing the node to blink, and then reporting which socket that node is in. We will assume that the oracle is always correct (does not provide faulty information) and for the purposes of our algorithm complexity, that it responds in time $O(1)$. Obviously, in real life it might take a fair amount of time for the operator to make these checks, so we would like to minimize the number of oracle calls.

There are other potential oracle types that we could use. For example, an oracle of type $\mathcal{S} \rightarrow \mathcal{N}$ can indicate which node is in a given socket. In real life, the naive approach to this would correspond to (potentially getting on a ladder and) reading an identification sticker on a node in a socket. For our lighting set-up, this would be more taxing than the $\mathcal{N} \rightarrow \mathcal{S}$ oracle. There is also the possibility of automating this process, as is discussed in Section 6.1.3. However, as this is not yet an option, we will stick with the $\mathcal{N} \rightarrow \mathcal{S}$ oracle, which is easier to operate.

An oracle of type $\mathcal{N} \rightarrow \mathcal{S} \rightarrow \textit{Boolean}$ could give us a yes/no answer of if a given node is in a given socket. This would correspond to flashing a node and asking the operator "is this node in socket X"? Oracles with *Boolean* output will be useful for verification steps, as detailed in Section 4.2.6.

Unless stated otherwise, when we speak of an oracle in this paper, it will be referring to that of the type in Definition 5.

Definition 6 (Mapping)**.** *A* ***mapping*** *for some problem* $P = (\mathcal{N}, \mathcal{S}, f)$ *is a partial injection* $m : \mathcal{S} \rightarrow \mathcal{N}$*. If* $m(s) = n$ *for some socket* s *and some node*

n, that means that n has been assigned to s.[2]

Definition 7 (Solution). *A* ***solution*** *is a mapping m that is a bijection.*

Our goal is to define an algorithm of type *problem* $\rightarrow$ *solution*.

2.2 Additional definitions

The following definitions are less foundational than those above, but are important elements for the design and implementation of our algorithms.

Definition 8 (Answer key). *An* ***answer key*** $k : \mathcal{S} \rightarrow \mathcal{N}$ *for problem* $(\mathcal{N}, \mathcal{S}, f)$ *is a solution which has been created from the real life (or simulated) lighting setup, where all nodes are paired with the socket in which they are installed. A solution m is* correct *iff it is equal to the answer key k. That is,* $\forall n \in \mathcal{N}, \forall s \in \mathcal{S},\ m(s) = n \Leftrightarrow k(s) = n$.

Definition 9 (estDist). *This function has type:* $\mathcal{N} \rightarrow \mathcal{N} \rightarrow \mathbb{R}$ *(and must be called with* RSSI *in the context) and will give an estimated distance between two nodes, based on the RSSI signal strength between them.*

$$\mathsf{estDist}(n_i, n_j) = 10^{(c_M - \mathsf{RSSI}(n_i,n_j))/(10 \cdot c_N)}$$

The constant c_M is also known as Measured Power, and is the measured RSSI value at one meter from a node. This might vary between types of devices, and if we are using the same type of nodes (which in our case we always do), we will consider this to be the same value for all nodes. (As discussed in Section 5.4, this is not the case in practice, but we generalize by node type when calculating these estimates.) The constant c_N depends on the environment, ranges between 2 and 4, and can be measured for any given building [Shah, 2021].

Definition 10 (estRSSI). *This function has type:* $\mathcal{S} \rightarrow \mathcal{S} \rightarrow \mathbb{R}$. *is effectively the inverse of* estDist*, as* estDist *translates RSSI values to distance, while this function translates distance to RSSI.*

$$\mathsf{estRSSI}(s_i, s_j) = c_M - 10 \cdot c_N \cdot log_{10}(\mathsf{dist}(s_i, s_j))$$

with c_M *and* c_N *chosen based on the equipment and environment, as in Definition 9.*

[2] By abuse of notation, we will sometimes treat a mapping as a list of pairs.

Definition 11 (Perfectly correlated)**.** RSSI *is* ***perfectly correlated*** *with respect to some answer key* k *when*

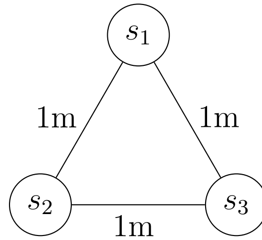
$$\forall s_i, s_j \in \mathcal{S}.\ \mathsf{RSSI}(k(s_i), k(s_j)) = \mathsf{estRSSI}(s_i, s_j).$$

Note: When we claim perfect correlation, we assume that the values that we are using for c_M *and* c_N *have been measured correctly. The dependency on these constants is a downside of this definition, which will motivate us to create a looser, related concept in Definition 13.*

2.3 Challenges & pitfalls

2.3.1 Symmetry

Even when our RSSI data is perfectly correlated with the distance between nodes, there can still be situations in which it is impossible to determine the socket-to-node solution without querying our oracle. This is due to possible symmetry in the socket setup. Consider the problem in Figure 2.1.



	n_1	n_2	n_3
n_1	x	-30	-30
n_2	-30	x	-30
n_3	-30	-30	x

Figure 2.1: A symmetric problem.[3]

Initially, any solution could be correct, as all distances and RSSI values are equal. Now imagine that we ask the oracle for the location of node n_1,

[3]If units are omitted in a figure, negative values indicate that they are RSSI values (in dBm), and positive values indicate socket distances (in m). If a label is s_i for some i, it represents a socket, and n_i labels represent nodes.

and it returns the top socket, s_1. Now, we still do not have a unique possible solution, as n_2 and n_3 could each be in either of the remaining sockets. (We have two remaining possible solutions, as mappings are injective.) Therefore, we will have to ask the oracle for the placement of a second node, making two queries in total before we have a unique remaining solution.

Thus, for this problem, if we initially choose any fitting solution, we have only a $\frac{1}{3}$ chance of being correct, and to be certain of our solution we must make 2 oracle queries.

So far we have assumed that our RSSI data is perfect: that it is exactly correlated to the distance between nodes. Unfortunately, we also have a bigger issue than symmetry – the unreliability of RSSI signals. With symmetry problems, we can at least detect such issues and utilize our oracle to remove uncertainty. With reliability issues we cannot make such guarantees.

2.3.2 RSSI reliability

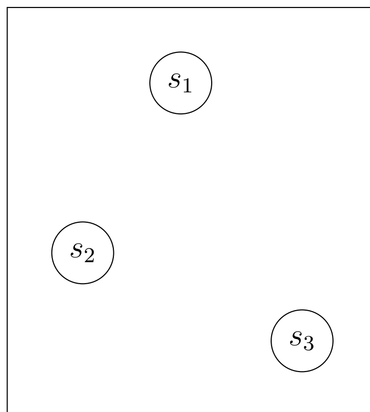
As discussed in Section 1.1, RSSI signals have been shown to be unreliable and not a good standalone resource for localization problems [Patwari et al., 2003]. RSSI signals are not always correlated to distance, as signal strength can be distorted due to a number of factors: walls, objects, manufacturing defects, installation errors, and more. They are not always consistent over time, and two nodes will not be guaranteed to give the same RSSI result with respect to each other.

In a worst case scenario, we could have two nodes that report RSSI values as if they are the other, effectively imitating each other, such that without further investigation, we have no reason to believe that their locations are not switched.

Example 2. Consider the socket and node data depicted in Figure 2.2.

The left table in Figure 2.2 represents RSSI_p, the function for RSSI values under perfect conditions. The right table represents RSSI_w, the RSSI function for a scenario where, due to some environmental factors, the values have been warped. In this case, $\mathsf{RSSI}_p(n_1, n_2) = \mathsf{RSSI}_w(n_1, n_3)$ and $\mathsf{RSSI}_p(n_1, n_3) = \mathsf{RSSI}_w(n_1, n_2)$. The RSSI values suggest the solution where n_2 and n_3 switched from their actual socket positions.

[4]When distance and RSSI labels are omitted from diagrams, values can be assumed to be proportional to the distances portrayed in the diagrams – nodes that are closer to each other have stronger signals between them.



Perfect RSSI			
	n_1	n_2	n_3
n_1	x	-30	-40
n_2	-30	x	-35
n_3	-40	-35	x

Warped RSSI			
	n_1	n_2	n_3
n_1	x	**-40**	**-30**
n_2	**-40**	x	-35
n_3	**-30**	-35	x

Figure 2.2: An example of inaccurate RSSI data.[4]

Example 2 can be generalized to any number of nodes, which leads to the following unfortunate truth in Theorem 1.

Theorem 1. *For any problem with* $|\mathcal{N}| > 1$*, if we do not have any guarantees on the reliability of the RSSI sensors, under the worst case we will need to make* $|\mathcal{N}| - 1$ *oracle queries if we want to guarantee the correctness of our solution.*

Proof. Suppose we have some algorithm A, which outputs solution m for problem $P_1 = (\mathcal{N}, \mathcal{S}, \mathsf{RSSI}_1)$, while making less than $n - 1$ oracle calls, and we can guarantee this to be correct (thus, $m = k_1$, the answer key for P_1). Let A also not use any hard-coded information to solve the problem[5]. Fix any $s_x, s_y \in \mathcal{S}, s_x \neq s_y$. Let $n_x = k_1(s_x), n_y = k_1(s_y)$.

Now, consider a problem $P_2 = P_1$: the set of sockets and nodes and the RSSI data is identical. However, the answer key for P_2 is k_2, where the

[5]Of course, if we were to program in a hard-coded way to answer the solution, knowing information about the contents or structure of the answer key, we could potentially guarantee correctness. Algorithm A should only have access to the problem and the oracle, and not know any information about the answer key that would help guide its decisions.

location of nodes n_x and n_y have been switched as follows:

$$k_2(s) = \begin{cases} n_y & \text{if } s = s_x \\ n_x & \text{if } s = s_y \\ k_1(s) & \text{otherwise} \end{cases}$$

While it might seem strange that two problems which are indistinguishable have different answer keys, this can happen in the following way. Suppose RSSI_2 is the perfect RSSI data for problem P_2 relative to k_2. We can transform RSSI_2 back to RSSI_1 by having n_x and n_y impersonate each other in the data – in this way, P_2's RSSI data is identical to that of P_1, despite the node switch.

$$\mathsf{RSSI}_1(n_i, n_j) = \begin{cases} \mathsf{RSSI}_2(n_y, n_x) & \text{if } i = x,\ j = y \\ \mathsf{RSSI}_2(n_x, n_y) & \text{if } i = y,\ j = x \\ \mathsf{RSSI}_2(n_y, n_j) & \text{if } i = x \\ \mathsf{RSSI}_2(n_x, n_j) & \text{if } i = y \\ \mathsf{RSSI}_2(n_i, n_y) & \text{if } j = x \\ \mathsf{RSSI}_2(n_i, n_x) & \text{if } j = y \\ \mathsf{RSSI}_2(n_i, n_j) & \text{otherwise} \end{cases}$$

This is an example of (highly) unreliable RSSI data. Our algorithm A cannot distinguish between P_1 and P_2 as input. The only way to be able to determine that the solutions should not be identical would be, when solving P_2, to query the oracle for n_x or n_y.

However, as n_x and n_y could be any two nodes, as long as there remain any two nodes which have not been queried, it is possible that both n_x and n_y remain unqueried, in which case we have no reason to believe their socket assignments should be switched, compared to P_1.

We claimed that we can guarantee A to be correct for some run with input P_1 and output m. However, neither we nor the algorithm can distinguish this run from a run with input P_2 and output m, so our claimed-to-be-correct run might have actually have been using P_2. If this were the case, our solution would not be correct, as k_2 is the answer key for P_2, and $m = k_1 \neq k_2$. Therefore, as this outcome is a possibility, it is impossible that we are able to guarantee the run of A with input P_1, and we have a contradiction.

Thus, the only way to guarantee the correctness of a solution is to query at least $n - 1$ nodes. Once we have queried $n - 1$ nodes, there is only one possible solution, as we know the position of $n - 1$ nodes, and the one remaining unqueried node must go in the remaining unassigned socket. $\square$

This is much worse than the issues presented by symmetry, as we have no way of knowing which nodes are reliable and which are not.

2.4 Computational complexity

2.4.1 Oracle calls

As discussed in Section 2.3.2, under the worst case we will need to make $n - 1$ oracle calls to be sure of the accuracy of our solution. This is clearly not a desired outcome; if that situation arises we have saved almost no time compared to the fully-manual strategy.

Theorem 2. *If we have no limit to the number of oracle calls, the time and space complexity of our problem is* $\mathcal{O}(|\mathcal{N}|)$.

Proof. Ask the oracle where every single node is located. Each call takes time $\mathcal{O}(1)$ and space $\mathcal{O}(1)$ to record the answer, so we have a solution in n steps[6]. □

It therefore makes sense, when reasoning about the complexity of the problem, to consider that we will not go above some threshold of oracle calls.

2.4.2 Search complexity

Definition 12 (Fitting)**.** *A solution m is **fitting** for a problem $P = (\mathcal{N}, \mathcal{S}, \mathsf{RSSI})$ when*

$$\forall s_i, s_j, s_k \in \mathcal{S}.\ \mathsf{dist}(s_i, s_j) < \mathsf{dist}(s_i, s_k) \leftrightarrow \mathsf{RSSI}(m(s_i), m(s_j)) > \mathsf{RSSI}(m(s_i), m(s_k)).$$

Due to the symmetry possibilities discussed in Section 2.3.1, there might be more than one fitting solution for a problem.

Definition 13 (Effectively accurate)**.** *The RSSI data in a problem P is **effectively accurate** with regards to an answer key k when k is a fitting solution for P.*

[6] We can also achieve this with $n - 1$ steps, but both options will lead to a complexity of $\mathcal{O}(n)$.

A problem P having effectively accurate data therefore means that, when we look at the correct placements (those in the answer key), the socket relationships and the node relationship are in agreement with regards to distance and strength, respectively. A benefit of this definition compared to Definition 11, perfect correlation, is that here we do not depend on $\mathsf{estRSSI}$, and therefore do not have to worry about measuring c_M and c_N.

Data can be effectively accurate without being perfectly correlated, but perfectly correlated data will always be effectively accurate.

Theorem 3. *Finding a fitting solution to a problem with effectively accurate data is in NP.*

Proof. To show that our problem is in NP (nondeterministic polynomial time), we must show that there exists a polynomial time deterministic Turing machine that takes as input a problem P and a mapping m, where the size of m is polynomial in the size of P, and can output True iff m is a fitting solution for P.

Consider that our input is problem $P = (\mathcal{N}, \mathcal{S}, \mathsf{RSSI})$. Let o be the size of this input. If $|\mathcal{N}| = |\mathcal{S}| = n$, then o has size $\mathcal{O}(n^2)$, due to the fact that we have RSSI values for all pairs of nodes. If we are given a possible solution m, to verify that it is a fitting solution, we need to check n^3 constraints, as stated in Definition 12. Thus, the verification of m can be done in time $\mathcal{O}(o \cdot n)$, which is polynomial in the input size o (as o is $\mathcal{O}(n^2)$). The size of m is $\mathcal{O}(n)$, which is polynomial with respect to o, as it is simply a mapping from sockets to nodes.

Therefore, our problem is in NP. □

Chapter 3

SAT solver approach

We can view our problem as a set of constraints that must be satisfied, therefore, we can view it as a satisfiability (SAT) problem. As such, we can use some of the many existing SAT-solving techniques that are at our disposal. There are constraint programming languages which are well-suited for problems such as these – we can use one such language to express the facts and constraints which define the requirements needed by a solution to a problem, and then run a constraint solver to find the solution(s).

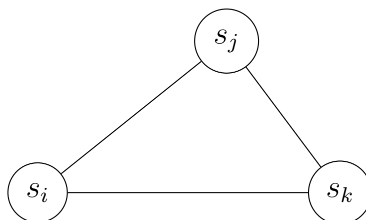
3.1 First-order logic representation

We will first discuss how to use first-order logic to express the requirements of a fitting solution to a problem.

Suppose we have a problem $P = (\mathcal{N}, \mathcal{S}, \mathsf{RSSI})$.

First, let us define our predicates:

- S: $S(s_i, s_j, s_k)$ indicates that the distance from socket s_i to socket s_j is less than the distance from socket s_i to socket s_k.



- N: $N(n_i, n_j, n_k)$ indicates that node n_i receives a stronger signal from node n_j than it does from node n_k.
- in: $\mathsf{in}(n, s)$ indicates that node n is placed in socket s. This is equivalent to saying that, for a mapping m, $m(s) = n$.

We use the following constraints:

- $\forall n \in \mathcal{N}.\ \exists s \in \mathcal{S}.\ \mathsf{in}(n, s)$
- $\forall n \in \mathcal{N}.\ \forall s_i, s_j \in \mathcal{S}.\ \mathsf{in}(n, s_i) \wedge \mathsf{in}(n, s_j) \rightarrow s_i = s_j$
- $\forall n_i, n_j \in \mathcal{N}.\ \forall s \in \mathcal{S}.\ \mathsf{in}(n_i, s) \wedge \mathsf{in}(n_j, s) \rightarrow n_i = n_j$
- The *correlation constraint*, as defined below.

Definition 14 (Correlation constraint)**.** *The* ***correlation constraint*** *is the following, and reflects the desired relationship between RSSI signals and distance.*

$$\forall n_i, n_j, n_k.\ \forall s_i, s_j, s_k.\ S(s_i, s_j, s_k) \wedge \mathit{in}(n_i, s_i) \wedge \mathit{in}(n_j, s_j) \wedge \mathit{in}(n_k, s_k) \rightarrow N(n_i, n_j, n_k)$$

Given our problem P we include the following sets of facts as true statements about P:

- $\{N(n_i, n_j, n_k) \mid n_i, n_j, n_k \in \mathcal{N} \wedge \mathsf{RSSI}(n_i, n_j) > \mathsf{RSSI}(n_i, n_k)\}$
- $\{S(s_i, s_j, s_k) \mid s_i, s_j, s_k \in \mathcal{S} \wedge \mathsf{dist}(s_i, s_j) < \mathsf{dist}(s_i, s_k)\}$

Note: it is also helpful (though not strictly necessary) to begin with at least one seed placement – $\mathsf{in}(n, s)$ *for some node* n *and socket* s *– to aid the solver and increase its speed of finding the solution. We can find this using an oracle call.*

By enforcing the correlation constraint, we ensure that any solution that we output will be a fitting solution. Thus, for effectively accurate input data, if we output all fitting solutions, it is guaranteed that one of them is the correct solution.

In the correlation constraint, we only look in one direction: socket facts imply node facts. However, this is equivalent to the bidirectional definition (as seen in the definition of fitting).

Claim 1. *For a given solution m and problem P, if the correlation constraint passes for m (considering $\mathsf{in}(n, s)$ to be equivalent to $m(s) = n$), then m is a fitting solution.*

Proof. Consider that we have some problem P and solution m, such that the correlation constraint passes, but m is not a fitting solution.

Let us consider the cases in which m could be not fitting.

Case 1: We have $S(s_a, s_b, s_c)$, and not $N(n_i, n_j, n_k)$[1] for some nodes n_i, n_j, n_k placed in sockets s_a, s_b, s_c, respectively. This is not possible, by definition of the correlation constraint.

Case 2: We have $N(n_i, n_j, n_k)$, and not $S(s_a, s_b, s_c)$ for some nodes n_i, n_j, n_k placed in sockets s_a, s_b, s_c, respectively.

$N(n_i, n_j, n_k) \equiv \neg N(n_i, n_k, n_j)$, and $S(s_a, s_b, s_c) \equiv \neg S(s_a, s_c, s_b)$, by definition of N and S. Due to the fact that we quantify over all sockets and nodes when creating our sets of facts, we consider all S and N facts which are true.

By the contrapositive, the correlation constraint says that $\neg N(n_i, n_k, n_j)$[2]$\rightarrow \neg S(n_a, n_c, n_b)$. As $N(n_i, n_j, n_k)$, and equivalently, $\neg N(n_i, n_k, n_j)$ is true, then $\neg S(n_a, n_c, n_b)$, and equivalently, $S(s_a, s_b, s_c)$, must be true. This contradicts our original assumption.

There are therefore no possible cases in which the correlation constraint passes and m is not a fitting solution. This proves our claim. □

We could use an analagous argument to claim that we could have designed the correlation constraint in the node-to-socket direction. However, we chose the current direction so that we will be able to adjust the strictness of socket constraints, as will be discussed later in Section 3.3.

---

[1] We will use the S and N predicates even when talking about fitting, as S and N are defined to be equivalent to the properties we test for in Definition 12.

[2] We omit the in statements, for brevity.

3.2 ASP & Clingo

The previous statements were expressed in first-order logic, but since our domain is finite, they can also be translated to propositional logic so that we can utilize the power of SAT solvers. Many constraint languages allow us to keep the expressibility of predicates such as these, while making the required translations under the hood.

Answer Set Programming (ASP) [Brewka et al., 2011] is one such method. Our toy example from Figure 1.1 can be seen translated to ASP in Listing 3.1. (The predicates N and S now have names n_stronger and s_closer, respectively.)

```
node(n1).
node(n2).
node(n3).
socket(s1).
socket(s2).
socket(s3).

n_stronger(n1, n2, n3).
n_stronger(n2, n1, n3).
n_stronger(n3, n2, n1).
s_closer(s1, s2, s3).
s_closer(s2, s1, s3).
s_closer(s3, s2, s1).

1 { in(N, X) : node(N) } :- socket(X).
:- in(N, S1), in(N, S2), S1 != S2.
:- in(N1, S), in(N2, S), N1 != N2.

s_closer(X, Y, Z) :-
    n_stronger(N1, N2, N3), in(N1, X), in(N2, Y), in(N3, Z).
:- s_closer(X, Y, Z), s_closer(X, Z, Y).
```

Listing 3.1: Problem formulation in ASP.

To solve for our answer sets (which correspond to all of the solutions which fit our program), we can use Clingo [Gebser et al., 2014], an ASP grounder and solver from the Potassco project for ASP. More information about this solver can be found in Gebser et al. [2011].

The output for Listing 3.1 is as follows[3]:

[3]Facts which were already included in Listing 3.1 (e.g. `node(n1)`) are omitted.

```
SATISFIABLE
Models : 1

in(n1,s1)
in(n3,s3)
in(n2,s2)
```

Listing 3.2: Clingo response output for Listing 3.1.

This answer matches our expected unique solution from Figure 1.1.

3.3 Fuzzy constraint representation

While the most straightforward way to choose our socket facts is simply to include $S(s_i, s_j, s_k)$ for any nodes such that $\mathsf{dist}(s_i, s_j) < \mathsf{dist}(s_i, s_k)$, this is quite brittle and not set up for handling any unreliability or fuzziness.

We run into trouble when sockets are equally or similarly spaced, but node signals are not perfect. For example, Figure 3.1 represents a situation where introducing a small amount of error can cause the valid constraints to change.

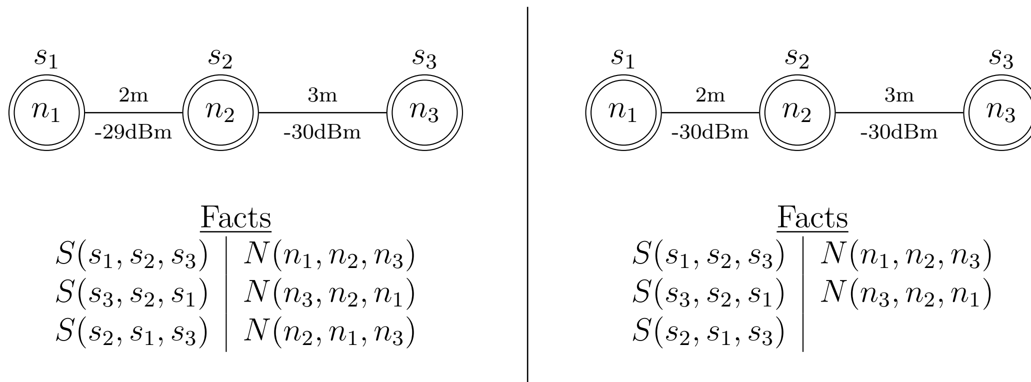


Figure 3.1: Difference in node constraints due to small RSSI error.

The RSSI values in the left diagram in Figure 3.1 represent effectively accurate RSSI data for three nodes which should be placed in an slightly-unevenly-spaced line of sockets. For every S fact, we have the fitting corresponding N fact, and the correlation constraint passes for all three pairs.

However, in the right diagram, because $\mathsf{RSSI}(n_1, n_2) = \mathsf{RSSI}(n_2, n_3)$, we are missing the fact $N(n_2, n_1, n_3)$, and the correlation constraint fails due to the presence of the fact $S(s_2, s_1, s_3)$.

We can mitigate this by adding a parameter to our socket fact requirements.

Definition 15 (ϵ)**.** *This parameter represents the minimum difference (in meters) required between socket distances for* S *fact to be included in our set of facts.*

For example, if we were to set $\epsilon = 2$ for our sockets in Figure 3.1, we no longer have the fact $S(s_2, s_1, s_3)$, and the correlation constraint would not be violated in the rightmost diagram.

The full set of socket facts for a given problem thus becomes:

$$\{S(s_i, s_j, s_k) \mid s_i, s_j, s_k \in \mathcal{S} \ \wedge \ \mathsf{dist}(s_i, s_k) - \mathsf{dist}(s_i, s_j) > \epsilon\}.$$

This can be tuned as is appropriate for a given situation, but using the data discussed in Section 5.2 we have determined $\epsilon = 2$m to be a reasonable value which eliminates many of these issues, while still leaving facts which will trigger the correlation constraint in situations of clear socket distance difference.

3.4 Issues

There are a few issues with the constraint solver approach, which motivate the design of a more customized system. First, exhaustively finding all possible mappings is very time-consuming if we are not able to get inside the constraint solver and optimize in ways that should be reasonable for our problem. We would be able to include problem-specific heuristics more easily were Clingo not a black box solver.

Secondly, and more importantly, the strategy of requiring all constraints to be fulfilled is only going to work for situations in which our data is effectively accurate. While this is achievable in simulations, or in some very specific socket layouts, it is highly unrealistic that this would be something we could count on happening with real life data.

Therefore, it is beneficial for us to take the constraint solving into our own hands and build a similarly-minded solver, which we are able to tune and optimize as we need.

Chapter 4

Algorithmic approaches

In this chapter, we will introduce several algorithms designed to solve our problem. We will start with simpler algorithms that motivate our core design choices, and the move to a more complicated final algorithm which combines the best features of each.

4.1 Basic greedy algorithms

First, we will examine some variations of basic algorithms to solve the problem that choose node placements greedily. These algorithms choose placements based on some heuristic and return the first solution that is found, backtracking when necessary.

4.1.1 Numerical estimation

Our first algorithmic approach is a greedy algorithm that at each step makes the "best" node-to-socket placement. We can define the best placement to be that which minimizes the difference in expected distance and real distance for all currently placed socket-node pairs. To get the estimated distance between two nodes, we use our $\mathsf{estDist}$ function.

Consider a stage in the algorithm when we have placed some nodes and have a partial mapping m.

Definition 16. *Given a mapping problem* $P = (\mathcal{N}, \mathcal{S}, \mathsf{RSSI})$*, the* ***error*** *of a*

potential node placement (n, s) *for* $n \in \mathcal{N}$*,* $s \in \mathcal{S}$ *is*

$$\mathsf{error}(n, s) = \sum_{(n_i, s_i) \in m} |\ \mathsf{estDist}(n, n_i) - \mathsf{dist}(s, s_i)\ |$$

To get a new node placement at any given step of our algorithm, we choose the placement that minimizes this error. Let u be the set of all possible placements, considering what is already in m.

$$u = \{(n, s) \mid \forall (n', s') \in m.\ n \neq n', s \neq s'\}$$

$$\mathsf{nextPlacement} = \arg\min_{(n,s) \in u} \mathsf{error}(n, s)$$

Definition 17. *A* ***seed mapping*** *is a mapping with (at least) one pairing, acquired from the oracle, to use as a starting point for our algorithms.*

We always make sure to start with a seed mapping instead of an empty mapping, as this will greatly aid the speed of the algorithm, and it is not an issue to have a pre-processing step of calling the oracle one time (or more). This will also avoid a situation where m is empty and $\mathsf{error}(n, s) = 0$ for all $n \in \mathcal{N}, s \in \mathcal{S}$.

Definition 18 (isSolution)**.** *This function takes a mapping and returns True iff it is also a solution, i.e. is a total function with domain* $\mathcal{N}$*.*

As a first attempt, the following algorithm, Algorithm 1, when called in a context containing a problem $P = (\mathcal{N}, \mathcal{S}, \mathsf{RSSI})$ and passed a seed mapping m as input, will output a solution.

Algorithm 1 extendMapping: (m: *Mapping*) → *Mapping*

```
1: if isSolution(m) then
2:     return m
3: end if
4: candidates ← {(n, s) | n ∈ 𝒩, s ∈ 𝒮, ∀(n′, s′) ∈ m. n ≠ n′, s ≠ s′}
5: nextPlacement ← arg min_{(n,s)∈candidates} error(n, s)
6: newMapping ← m ∪ {nextPlacement}
7: return extendMapping(newMapping)
```

The main idea of this algorithm is that at each iteration of the function, we place the node chosen by nextPlacement, as defined above.

There are a few main problems with Algorithm 1.

1. This algorithm is very inefficient. Not only is the algorithm already slow by necessity of the problem complexity, but it is also calculating the error for many pairings which clearly do not make sense, as in line 4 we take the full product $\mathcal{N} \times \mathcal{S}$. We can greatly speed up the algorithm by being smarter about which pairings we consider, and which existing pairings we compare them to when calculating the error. This optimization will lead to Algorithm 2 below.

2. The algorithm will never backtrack, as we have not set any threshold for how high the error can go. Just because we take the "best" at any given point does not mean much if, by the end of the algorithm, the only options we have left for placements are terrible ones.

3. Relying on estDist (used by error) is risky, as it has been shown to not be a consistent estimator, as discussed in Section 1.1, and therefore is not ideal to use for such a scoring metric.

Problem 2 raises an important flaw in using this greedy numerical strategy: without an error threshold, the algorithm is not useful in many situations. For example, if we have an equally-spaced grid of sockets (which is very realistic in many building plans), we are often faced with situations where we might have options with equal score.

In the scenario depicted in Figure 4.1, there is an equal chance that the second node should be placed in the socket directly to the right or directly below the corner seed socket. If we make the wrong choice and are not able to later backtrack, we will never get the right solution.

It is clear therefore that we must have some threshold for what is an acceptable pairing. We can introduce a parameter for this, but this will need to be fine-tuned for a given problem. Choosing this value is very difficult, as the error in a given building might vary widely – for example, nodes on one side of a building might have walls between them, and have very large error, while on the other side we might have perfect data.

Example 3. *Consider the situation in Figure 4.2, where three nodes are placed in sockets with walls between them. Imagine that this weakens the*

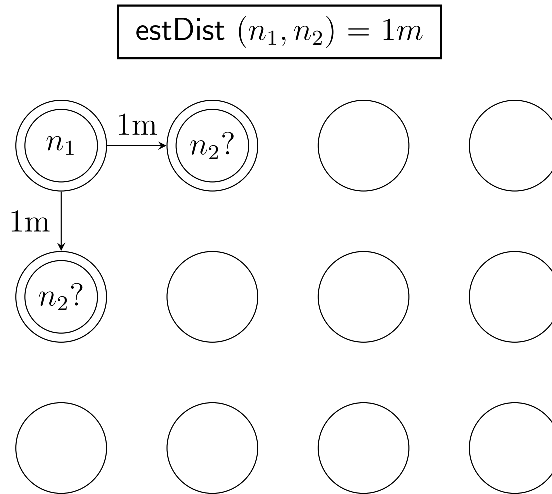


Figure 4.1: Symmetric options often require backtracking.

RSSI values by a factor of 10. It is still the case that n_1 has a stronger signal with n_2 than it does with n_3, which matches the fact that it is closer to n_2 in physical distance.

However, the weakened RSSI values will mean that in a situation where we have a threshold for the amount of error that is allowed, it could easily be the case that no solutions are found, even though the data still paints an informative picture.

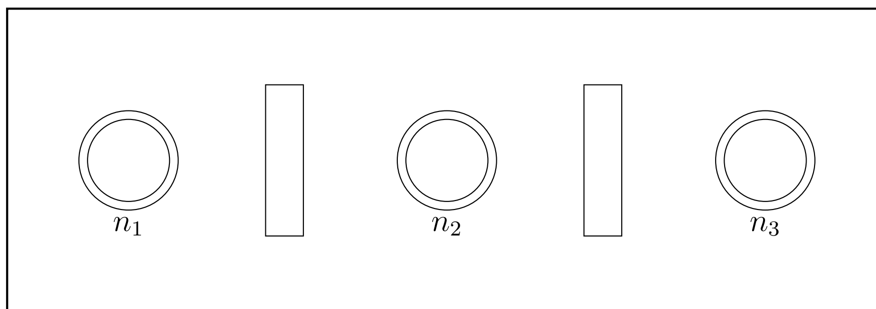


Figure 4.2: Nodes impeded by barriers.

This leads us to another possible similar algorithm, inspired by the constraint solving approach in Chapter 3: we can place nodes in a similar fashion,

but rather than using some numeric score based on estimation, we can simply place nodes according to which nodes fit the current constraints, based on what has already been placed at a point in the algorithm.

4.1.2 Comparison algorithm

Consider Algorithm 2, which is an update to Algorithm 1, where instead of using distance estimation, we generate the same set of facts as in our ASP approach, and then at each step, we only consider placements that would not violate the correlation constraint.

Algorithm 2 extendMapping: (m: $Mapping$) $\rightarrow$ $Option[Mapping]$

1: **if** isSolution(m) **then**
2:     **return** m
3: **end if**
4: candidates $\leftarrow \{(n, s) \mid n \in \mathcal{N},\ s \in \mathcal{S},\ \forall (n_i, s_i) \in A.\ n \neq n_i, s \neq s_i,$
        $\forall f \in F_s.\ \neg \text{violates}(f,\ (n, s))\}$
5: **for** $(n, s) \in$ candidates **do**
6:     newMapping $\leftarrow$ m $\cup \{(n, s)\}$
7:     **if** extendMapping(newMapping) is not None **then**
8:         **return** newMapping
9:     **end if**
10: **end for**
11: **return** None

In our context, we now also have:

- F_S: The set of all S facts about socket relationships, as defined in Section 3.1. We will also eliminate socket facts according to our choice of ϵ, as mentioned in Section 3.3.
- F_N: The set of all N facts about node relationship, as defined in Section 3.1.
- violates: violates$(f_S, (n, s))$ is True for a socket fact $f_S \in F_S$ iff placing node n in socket s (adding (n, s) to our current partial mapping m) would violate the correlation constraint raised by f_S. That is, the constraint is violated if $f_S = S(s_i, s_j, s_k)$, $m(s_i) = n_i, m(s_j) = n_j, m(s_k) = n_k$ (for some $n_i, n_j, n_k \in \mathcal{N}$), and $N(n_i, n_j, n_k) \notin F_N$

This algorithm will only output a solution if it fulfills all constraints, and will backtrack upon reaching a partial mapping with no further legal placements. We now have an algorithm which can properly output a legal solution (though still not necessarily a correct one, due to symmetry) under the condition that the data is effectively accurate.

4.2 Speed optimizations

One problem in our list of issues with Algorithm 1 in Section 4.1.1 was that it is highly inefficient. We repeatedly test all possible constraints on all possible pairs. Instead of doing this, we can pare down our nodes and sockets to only consider the most likely options at any given iteration. In Algorithm 3, we present the more-complete pseudocode version of Algorithm 2, which has been expanded to include optimizations for efficiency.

4.2.1 Witness pairs

Rather than considering all possible node-socket pairs, we can greatly speed up our process by considering only reasonable pairings, and attempting placement in an order which prioritizes pairings which are closer to already-placed nodes. Figure 4.3 illustrates a situation where we have one placed node, and are looking for our next node to place.

We introduce two new parameters:

- FrontierSize: the maximum number of unassigned nodes we will initially consider for placement at each iteration of our algorithm. These nodes are called our frontier.
- CloudSize: for any given node or socket we are considering in our frontier, we also consider CloudSize-1 other possible nodes or sockets, respectively, if available.

Note: FrontierSize *does not consider the number of nodes included which are accounted for by* CloudSize*. The total number of nodes to consider at each step is at most* (FrontierSize · CloudSize)*. When we say "maximum number", it is referring to the fact that we will sometimes have less then* FrontierSize*-many unassigned nodes available.*

For now, assume we set the parameters to: FrontierSize = 1, CloudSize = 2.

At any given step in the algorithm, let $\mathcal{U}_N$ be the set of all unassigned nodes, and $\mathcal{U}_S$ be the set of all unassigned sockets.

FrontierSize = 1 means that we start by choosing one node, $n \in \mathcal{U}_N$, to consider for placement. The node that we choose is the single node which has the strongest RSSI signal to any of the already assigned nodes[1], which we will call the *witness node*, n_w. Formally, we choose n and the witness node as follows:

$$(n, n_w) = \underset{\{(n_i,n_j)\ |\ n_i \in \mathcal{U}_N, n_j \notin \mathcal{U}_N\}}{\arg\max} \mathsf{RSSI}(n_i, n_j).$$

The pair $(n, (n_w, s_w))$ is called our *witness pair*, where s_w is the socket in which n_w is placed, and is called our *witness socket*.

We can roughly deduce from the RSSI score that n should be in the closest socket to s_w that is unassigned, which we will call s. However, due to the unreliability of the RSSI data, this is not guaranteed to be the case. Therefore, instead of considering only n and s, we will also consider other unassigned nodes close to n_w (with "closeness" for nodes referring to signal strength) and some unassigned sockets close to s_w.

For our example, since we have CloudSize equal to 2, for our nodes we will consider n and n', where n' has the second-strongest RSSI signal with n_w (after n), and for our sockets we will consider s and s', where s' is the second-closest socket to s_w (after s).

In our example, at each step we therefore consider 4 possible assignments, and in general we will consider at most (FrontierSize $\cdot$ CloudSize^2)-many possible node-socket pairings.

We can see this illustrated in Figure 4.3. In our figure, we only have one assigned node, so our witness pair becomes $(n_1, (n_w, s_w))$, as n_1 has the strongest RSSI signal with n_w.[2] The node cloud is therefore $\{n_1, n_2\}$, and the socket cloud is $\{s_1, s_2\}$.

The possible placements that we consider at this iteration of the algorithm are: $\{(n_1, s_1), (n_1, s_2), (n_2, s_1), (n_2, s_2)\}$.

[1] Note that we always start with a seed mapping, so there will always be some pool of assigned nodes.

[2] Closer nodes in drawing have stronger RSSI signals between them.

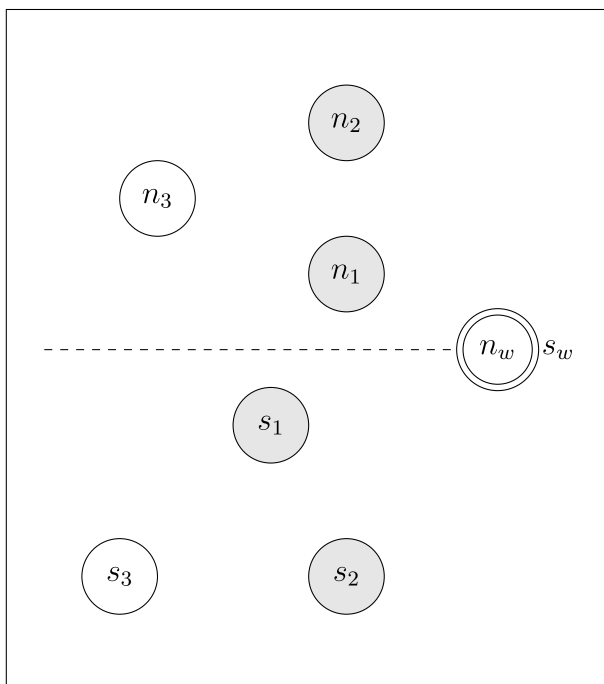


Figure 4.3: Witness pair selection.

In general, we will keep our parameter values quite small to increase the speed of the algorithm. However, as will be discussed later, to guarantee the termination of our algorithm with the correct solution, we will need to consider all possible solutions, so in the worst case we might have to re-run the algorithm with FrontierSize equal to $\mathcal{N} - 1$.

Algorithm 3, presented below, builds on Algorithm 2. Some new elements of the algorithm are explained here.

Elaborations on functions and variables used in Algorithm 3

nodeCloud: takes as input a node n, and returns a set containing CloudSize-many nodes in $\mathcal{U}_N$ with the strongest RSSI signals with n.

socketCloud: takes as input a socket s, and returns a set of CloudSize-many sockets in $\mathcal{U}_S$ closest to s.

percentageConstraintsFulfilled: takes as input a node-socket pair, and returns the decimal percentage of constraints that are fulfilled if that

placement were to be made in the current partial mapping. Only considers constraints where all nodes and sockets occurring in the constraint have already been placed in the mapping. We will then order candidates using this as a score.

threshold: represents the minimum allowable candidate score at each node placement. This score is equal to percentageConstraintsFulfilled. At each step of the algorithm, the only candidates which will be considered are those who score higher than this threshold.

Algorithm 3 extendMapping: (m: $Mapping$) (threshold: $float$) $\rightarrow$ $Option[Mapping]$

1: **if** isSolution(partialMap) **then**
2:     **return** partialMap
3: **end if**
4: witnessPairs $\leftarrow \{(n, (n_w, s_w)) | \neg$assigned$(n)$, assigned$(n_w)$, $m(s_w) = n_w\}$
5: sortedPairs $\leftarrow$ sort witnessPairs by decreasing RSSI between n and n_w
6: topPairs $\leftarrow$ sortedPairs[:FrontierSize]
7: candidates $\leftarrow \{(n_i, s_j) \mid \exists(n, (n_w, s_w)) \in$ topPairs.
        $n_i \in$ nodeCloud$(n_w) \wedge s_j \in$ socketCloud$(s_w)\}$
8: candidateScore $\leftarrow \lambda(n, s) \mapsto$ percentageConstraintsFulfilled(n, s)
9: sortedCandidates $\leftarrow$ sort candidates by decreasing candidateScore
10: validCandidates $\leftarrow \{c \in$ sortedCandidates $\mid$ candidateScore$(c) \geq$
        threshold$\}$
11: **for** $(n, s) \in$ validCandidates **do**
12:     newMap $\leftarrow$ copy(m)$[n] = s$
13:     filledMap $\leftarrow$ extendMapping(newMap)
14:     **if** (filledMap $\neq$ None) **then**
15:         **return** filledMap
16:     **end if**
17: **end for**
18: **return** None

4.2.2 Iterative approach

Algorithm 3 works very well in situations where we have no (overall) symmetry and effectively accurate data, as we will discuss in detail later in Section

5.7. However, if our data is not effectively accurate, the threshold that we pass in will have to be lowered to find our solution, and this could lead to returning a solution which is not the correct one, as we still greedily return the first solution that we find. We also still have the problem that it is not obvious how to choose the value for the threshold. We want to choose it in a way that we maximize the total number of constraints solved.

There are various approaches available to do this. For example, Heras et al. [2007] have built a weighted Max-SAT solver, MiniMaxSat, which is designed to find the satisfying assignment which maximizes the weight of all satisfying clauses. We could then weigh our constraints by how likely they are to be true. For example, if $\mathsf{dist}(s_1, s_2) = 5$ and $\mathsf{dist}(s_1, s_3) = 100$, we can be very confident that the node placed in s_1 should have a stronger signal with the node in s_2 than the node in s_3, as the magnitude of the differences is so large. Therefore, that constraint should have a correspondingly high weight. After weighing our constraints in this way, we could use a solver such as MiniMaxSat to give us the most likely solution.

However, it's dangerous to focus on optimizing for the "best" solution, as this could be an incorrect one. If we rely too heavily on trusting the optimization process to give us a single best solution, we cannot be confident that we are in fact correct.

This brings us to our main update: rather than using a greedy algorithm that returns one possible mapping, we should instead look for a set of the top-scoring mappings, and then use the oracle to determine which, if any, is the correct one, continuing on if no solution is found.

To do this, we will iteratively decrement threshold as we progress. Our iterative algorithm will begin with a threshold of 1 – this will only return mappings which score perfectly. At each iteration of our outer algorithm, we will lower threshold by 0.05. Reducing this score by 0.05 is equivalent to saying that when placing a node, the total percentage of constraints allowed to be violated is increased by 5 percentage points, compared to the previous iteration. When we find a solution that fits our new requirements, rather than returning it immediately, we search exhaustively for all solutions which also pass that iteration's score threshold.

Once we have a set of possible solutions, we are now ready to use our oracle to determine which (if any) is correct. This process will be discussed in Section 4.2.5. If the oracle response indicates that none of these solutions is correct, we continue to lower our threshold. To increase efficiency, at each successive chooseSolution call, we do not need to include any of the possible

solutions that were included in past calls, as they have already been ruled out.

Finally, once the oracle has narrowed down the possible solutions to one (out of our current set of possibilities), we query the verification oracle (discussed in Section 4.2.6) to ensure that we do in fact have the real solution.

4.2.3 Final pseudocode

Algorithm 4, solveIterative is composed of three main interwoven steps: solve, chooseSolution, and solutionCorrect. We will discuss each in greater detail in the following subsections.

Algorithm 4 solveIterative: (problem: *Problem*) $\rightarrow$ *Option*[*Solution*]

```
1:  m ← seedMapping(problem)
2:  threshold ← 1
3:  while threshold ≥ 0 do
4:      topSolutions ← solve(problem, threshold)
5:      if topSolutions is not None then
6:          bestSolution ← chooseSolution(topSolutions)
7:          if solutionCorrect(bestSolution) then return bestSolution
8:          end if
9:      end if
10:     threshold ← threshold − 0.05
11: end while
12: return None
```

4.2.4 solve

This function has type $Mapping \rightarrow \mathbb{R} \rightarrow Option[List[Solution]]$. The internal logic is identical to Algorithm 3, except for one difference: rather than greedily returning the first solution that we find, we exhaustively return all solutions that are returned (for that threshold).

4.2.5 chooseSolution

Once we have a set of possible solutions from our search algorithm, we then must use our oracle to determine which one is the correct solution.

As our oracle is type $\mathcal{N} \to \mathcal{S}$, we can narrow down our solution the fastest by querying nodes that eliminate the most solutions possible.

Thus, chooseSolution works as follows: at each step, for each node, we calculated the expected solutions remaining if we were to query the oracle for that node. We calculate the expected solutions remaining for a node in the following way. Let $\mathcal{PS}$ be the set of all proposed solutions and A_n be the set of assigned sockets to node n across all solutions in $\mathcal{PS}$. We define

$$\mathsf{expectedRemainingSolutions}(n) = \sum_{s \in A_n} p(s) \cdot \frac{p(s)}{|\mathcal{PS}|}$$

where $p(s) = |\{m \in \mathcal{PS} \mid m(s) = n\}|$.

Example 4. *Consider the following two nodes, when we have 8 solutions left, and this is the breakdown of all socket assignments per node:*

$$n_1 : (s_1, s_1, s_1, s_1, s_2, s_2, s_2, s_2)$$
$$n_2 : (s_2, s_3, s_4, s_4, s_4, s_4, s_4, s_4)$$

$$\mathsf{expectedRemainingSolutions}(n_1) = (4/8) \cdot 4 + (4/8) \cdot 4 = 4$$
$$\mathsf{expectedRemainingSolutions}(n_2) = (6/8) \cdot 6 + (1/8) \cdot 1 + (1/8) \cdot 1 = 4.75$$

As the expected value of n_1 is lower than n_2, we should query the oracle for n_1 first.

At each step of chooseSolution, calculate expectedRemainingSolutions for all nodes which have not been queried yet and query the oracle for the node that minimizes that value. Then, remove solutions for which the oracle's response is incompatible. (For example, if we start with node n_1 and our oracle returns s_1, we then remove all solutions in $\mathcal{PS}$ in which n_1 is not assigned to s_1.) The full pseudocode can be seen in Algorithm 5.

Theorem 4. chooseSolution *will always terminate, calling the oracle a maximum of $n - 1$ times in the worst case, where $n = |\mathcal{N}|$.*

Proof. We start with n nodes which have not been queried to the oracle. In each iteration of the while loop, we query a new node, as the already-queried nodes have been removed from the pool of available nodes. If we do not already have less than two solutions remaining, it must be the case that at

Algorithm 5 chooseSolution: (solutions: $List[Solution]$) $\rightarrow Option[Solution]$

1: nodes $\leftarrow |\mathcal{N}|$
2: **while** len(solutions) > 1 **do**
3: node $\leftarrow \arg\min_{n \in \text{nodes}}$ expectedRemainingSolutions(n)
4: socket $\leftarrow$ oracle(node)
5: nodes $\leftarrow$ nodes $\setminus \{\text{node}\}$
6: solutions $\leftarrow$ filter($\lambda m \mapsto$ m[socket] = node)
7: **end while**
8: **if** len(solutions) = 0 **then return** None
9: **end if**
10: **return** solutions[0]

least one of the possible nodes to query will have expectedRemainingSolutions $<$ |solutions|, as if we have two solutions m_i and m_k, they must differ on at least one node, n, and querying the oracle with input n will eliminate one of m_i or m_j.

Therefore, at each iteration of our while loop, the number of solutions will decrease, and eventually there will be one or fewer solutions remaining and our algorithm will terminate.

As we remove nodes from the node pool after querying them, and we only perform one query per loop, we can loop (and query) a maximum of n times. However, as there is only one possible solution remaining after we know $n-1$ nodes (the unassigned socket must go in the unassigned node), we know that we will never enter the loop again after the $n-1^{th}$ oracle query. $\square$

Our best case scenario is if nodes are spread fairly evenly between many sockets. In this case, querying those nodes will be likely to quickly drop down the expected number of solutions left. For example, if we have 8 solutions left, and one node is assigned to a different socket in each solution, querying the oracle for that node will immediately tell us which solution, if any, is possible among the proposed solutions.

As seen above, chooseSolution will terminate with either a solution m or None. However, it is not guaranteed that m is the *correct* solution. For us to know that our solution is correct, we would have had to make $n-1$ queries, as shown in Theorem 1. solution m has not *yet* contradicted the information from the oracle, and we also know that no other solution from

the list returned from solve is valid. However, it is possible that some non-queried nodes in m were placed incorrectly, and there was in fact no correct solution in our candidate solutions.

It is for this reason that we must check our proposed solution m, and iterate further if necessary. For this, we need another function to be able to verify solutions.

4.2.6 solutionCorrect

We would of course ideally like to not have to do a full manual check of any solution. Indeed, the motivating goal of this project was to semi-automate the process and remove the need for manually checking each socket.

However, as we have seen in Section 2.3.2, there remains a chance that whatever solution we find, regardless of our confidence level, it might be the wrong one (until we ask the oracle $n-1$ queries). We can try to find a middle ground, where our search algorithm and oracle process give us a likely candidate, with a reasonably-small number of queries, and then we can use a different decision oracle to verify the correctness of that solution.

Definition 19 (Verification Oracle)**.** *This oracle has type Solution $\rightarrow$ Boolean, and returns* True *iff the solution is correct.*

The verification oracle should be designed to be an ergonomic process for the human who is tasked with this decision. While the typical $\mathcal{N} \rightarrow \mathcal{S}$ might require the operator to search around for lights, and report specific socket numbers, the design for this oracle would be something like the following: the nodes would turn on in cascading sequence around the current room. (There would be a slightly modified process if the sockets were split into rooms or multiple lines of sight). The operator would verify that they all turned on "in order", according to the prompts on the in-app map. If so, this means that our solution is correct.

The exact design of this process and the corresponding algorithm is more of a question of user experience and user interface design, and has therefore been left out of the scope of the current project.

4.3 Termination and correctness

Theorem 5. solveIterative *will always terminate.*

Proof. The external while loop will loop a finite amount of times, as we begin with our threshold set to 1, decrement it by 0.05 each loop, and stop the loop when it reaches 0. Within the while loop, we might call each of our three functions: solve, chooseSolution, and solutionCorrect.

We consider solutionCorrect to be a single call of an oracle. In reality, this would have to be implemented properly for real life users to ensure that it works as designed in the function description above, but for our purposes it will take time $\mathcal{O}(1)$, and be a reliable source of information.

Without loss of generality, we will consider the case when FrontierSize = $|\mathcal{N}|$, as this is the case that will iterate through the most possibilities, and other values of this parameter will lead to runs that consider a subset of the same possibilities. In this case, it will eventually consider each possible combination of node placements. At each recursive call after a node placement, it will return None once the possibilities of further placements are exhausted. There are no situations where we would loop forever through possibilities.

As seen in Theorem 4, chooseSolution will always terminate. □

Theorem 6. solveIterative *will always either output the correct solution or None.*

Proof. As we only return a solution if it has been verified by solutionCorrect, any non-None output from the algorithm must be correct. As we have seen in Theorem 5 that the algorithm terminates, we will otherwise receive an output of None, due to the return statement after the main while loop in solveIterative has completed. □

Theorem 7. *With the* FrontierSize *parameter set to* $|\mathcal{N}|$, solveIterative *is guaranteed to output the correct answer.*

Proof. By Theorem 6, we know that any output of type solution must be correct. Consider the case where we have reached a threshold value of 0. In this iteration of our solve function, every single possible mapping will be returned, as 0% of the constraints are required to be fulfilled at each node placement, and all node placements are considered, due to the FrontierSize parameter setting. Therefore, the correct solution will be in the set of solutions passed to chooseSolution, and will then be returned (and subsequently verified by solutionCorrect). □

These theorems might appear to not make any impressive claims: yes, the algorithm will always be correct (by definition of the verification oracle),

and always eventually reach the correct solution (because we will eventually consider them all). However, most of the work of this project was to create strategies and heuristics that in most cases would lead us to the right answer long before all options were checked. Furthermore, it is our hope (and so far it has been the case) that the verification step is usually a formality, and the correct solution is the first one that is sent to the verification oracle.

Chapter 5

Data & evaluation

In this chapter, we will discuss the format and possible pre-processing strategies that we can use on our data. We will also evaluate our algorithms on data, both simulated and real-world, and evaluate the results.

5.1 Simulated data

As we had a limited amount of access to real data, it was useful to work with simulated data. To do this, we designed various layouts of sockets, calculated perfectly-correlated RSSI data, using estRSSI, and then perturbed the RSSI data varying amounts to simulate potential data messiness.

We perturbed the perfectly correlated values by distributing them according to various probabilistic distributions. As our focus was on solving this problem for our highly unpredictable real-world data, the simulated data was used more to test the algorithms on well-behaving data, and to test the capabilities of our earlier algorithms, which were not suited for highly-perturbed data. Once access to the real-world data was made available, we were able to focus on the that data for the most insightful look into how these nodes behave in the wild.

5.2 Real-world data

We received data from a large office building, split into three separate floors and rooms (which we handle separately). There was one small office with four nodes, pictured in Figure 5.6, and two identical open-plan floors, pictured in

Figure 5.1, with twelve nodes each. We will refer to them by name as Office, Floor 1, and Floor 2.

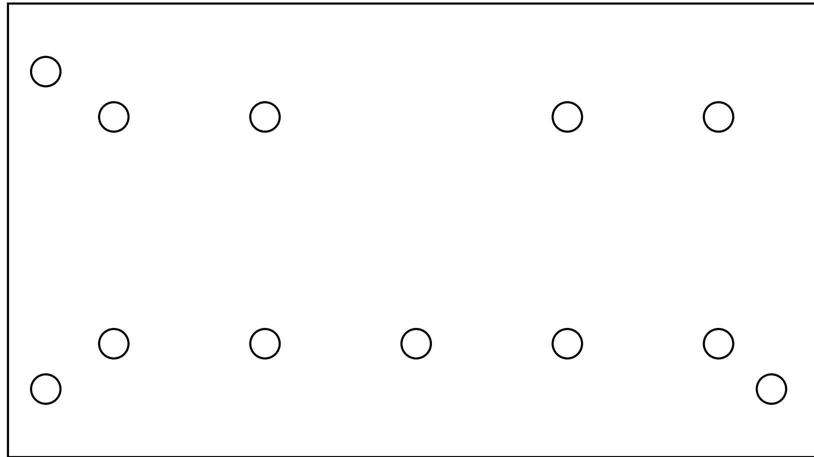


Figure 5.1: Layout of Floors 1 and 2.

We initially receive the raw RSSI data in the following JSON[1] structure, as seen in Listing 5.1.

```
{
    "n_1": {
        "n_2": [{"time": 1, "rssi": -73},
                {"time": 2, "rssi": -72},
                ...],
        "n_3": [{"time": 1, "rssi": -64},
                {"time": 2, "rssi": -65},
                ...],
        ...
    },
    "n_2": {...},
    ...
}
```

Listing 5.1: Raw RSSI input.

5.2.1 Averaging over time

RSSI data can vary widely over time for the same node pair. Therefore, it is preferable to examine data over a longer period of time and then average

[1] JavaScript Object Notation, a popular data format.

it out, so as to lower the effects of bad signals, disturbances, and random outliers. RSSI values are always integers, between 0 and -255. However, since we take the average over many values, we will consider the type to be $\mathbb{R}$ instead of $\mathbb{Z}$, to more accurately represent the mean values.

It is then also possible to do more complex pre-processing, to smooth out the data and try to get the most representative data possible. This is advisable in situations where the data is very messy, or is being collected in an environment that is more prone to signal disturbances.

For our experiments, we averaged all values and did not remove any outliers – our goal was to create a system that was robust to messy data, so we chose to test it on the full set of data rather than the cleaned data.

5.2.2 Data spread

Figure 5.2 shows a plot of RSSI compared to distance, where each point in the plot is a pair of nodes from our dataset. We can see that distance is not well correlated in general with RSSI. However, this plot illustrates a common feature of RSSI data: RSSI values are much more likely to error by showing a weaker value (due to interference, etc.), while it is much more difficult and uncommon to have a value be incorrectly stronger. In Figure 5.2, we can see that is a lack of outliers such that the RSSI signal is far "too strong" for the distance between the points. The upper right quadrant, where many of these points would lie, is empty, and we see a clear line where the distance decreases as the maximum RSSI values measured for a given distance rises.

5.2.3 RSSI symmetry

Between any two nodes n_i and n_j, we have two RSSI values: $\mathsf{RSSI}(n_i, n_j)$ which indicates the strength of n_j's signal as received by n_i, and $\mathsf{RSSI}(n_j, n_i)$. With the design of our algorithms, it is generally helpful to consolidate this to one composite value. We therefore have multiple choices of how to choose which final value to use as *the* RSSI value between the two.

If we have any reason to believe that one of the two is more reliable than the other, we could simply use the more reliable value. As discussed in Section 5.2.2, RSSI values are more likely to be incorrectly weak than incorrectly strong. Therefore, when presented with two RSSI values for the same distance, it is generally a safer bet to choose the stronger value as more accurate. Another choice is simply to take the mean between the two values.

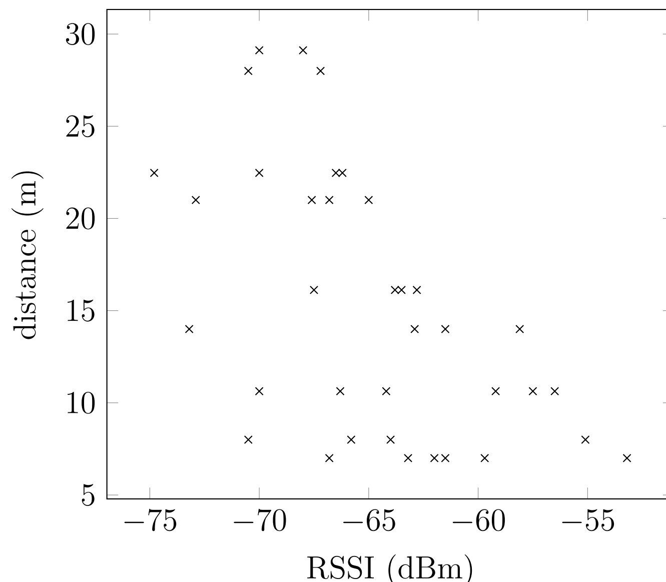


Figure 5.2: $\mathsf{dist}(s_i, s_j)$ vs. $\mathsf{RSSI}(n_i, n_j)$ for pairs $((n_i, s_i), (n_j, s_j))$ in each of the rooms from our dataset.

Before this pre-processing, it is the case that the order of the node arguments to the RSSI function matter; once we have determined our composite value v between the two ($\mathsf{RSSI}(n_i, n_j)$ and $\mathsf{RSSI}(n_j, n_i)$), we set $\mathsf{RSSI}(n_i, n_j) = \mathsf{RSSI}(n_j, n_i) = v$, so that during the algorithm, the direction that we are examining the nodes does not matter. The reason for this is that semantically, it should not matter: there is only one physical distance between the nodes, so we do not want to consider the strength of signal between nodes to have a direction.

For all of our experiments, we use the simplest technique (so as to test our algorithms under the harshest conditions), and choose our composite value by averaging the two values.

5.3 Self-healing when a node has inaccurate output or input

In a case where it can be determined that a node n_i has faulty RSSI output or input (the signals it is sending to its neighbors are not correlated to the

distance to those neighbors), we have an easy fix: we can simply ignore its input, or output signals respectively. For example, if n_i has faulty output signals, for any other node $n_j \in \mathcal{N} \setminus \{n_i\}$, we can ignore the values for $\mathsf{RSSI}(n_j, n_i)$ and only use the value from $\mathsf{RSSI}(n_i, n_j)$. Since each node pairing has two possible values, we still have another value to use for the relationship between those two nodes.

In a concrete example of this technique, in the Floor 1 raw input data, one node has no outgoing RSSI signals. Even though we are missing all outgoing data, we are able to continue with our process by simply using the incoming data exclusively for that node instead of averaging incoming and outgoing.

Obviously, if we have more than two (or more) nodes in the system which we have determined to have an issue, we then have no value to use between those two. In that case, we have two choices: we can still attempt to create a composite value somehow, ignoring the issues (if possible), or we can drop those values, and simply not have a RSSI value between those nodes. As we begin with a complete (multi-)graph, it is more likely allowable to drop many edges and still be able to find our output mapping.

5.4 Determining node issues

We so far have not discussed possible ways to know when a particular node is unreliable. One method of doing this is by quantifying the unreliability of a node as follows.

Definition 20 (symmetryError)**.** *This is a function of type $\mathcal{N} \to \mathcal{N} \to \mathbb{R}$, and is calculated as follows:*

$$\mathsf{symmetryError}(n_i, n_j) = \mathsf{RSSI}(n_i, n_j) - \mathsf{RSSI}(n_j, n_i).$$

A positive symmetry error for (n_i, n_j) therefore indicates that $\mathsf{RSSI}(n_i, n_j)$ is greater than $\mathsf{RSSI}(n_j, n_i)$, which in turn means that n_j is sending stronger signals than n_i, as $\mathsf{RSSI}(n_i, n_j)$ is the strength of the signal that n_i *receives* from n_j.

In our real-world dataset, after RSSI values were averaged over 72 time-points, 81% of node pairs had absolute symmetry error of less than 1 dBm ($|\mathsf{RSSI}(n_i, n_j) - \mathsf{RSSI}(n_j, n_i)| \leq 1$) and 98.4% of node pairs had absolute symmetry error of less than 5 dBm. Figure 5.3 plots each room's pairs of points, comparing their outgoing and incoming values. The fact that most points lie

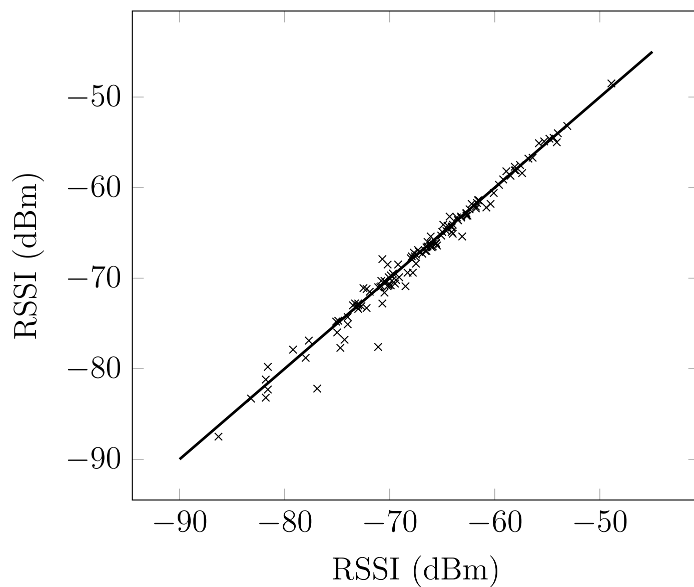


Figure 5.3: $\mathsf{RSSI}(n_i, n_j)$ vs. $\mathsf{RSSI}(n_j, n_i)$ for all nodes $n_i, n_j \in \mathcal{N}$.

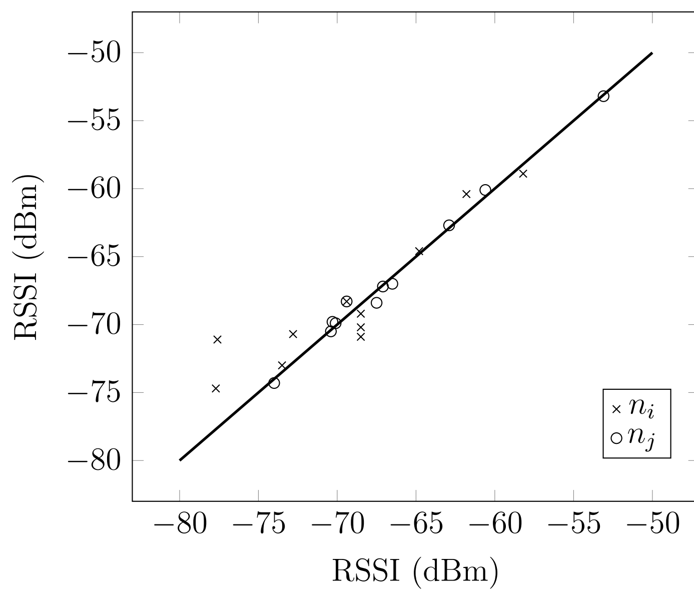


Figure 5.4: An unreliable node (n_i) and a reliable node (n_j).

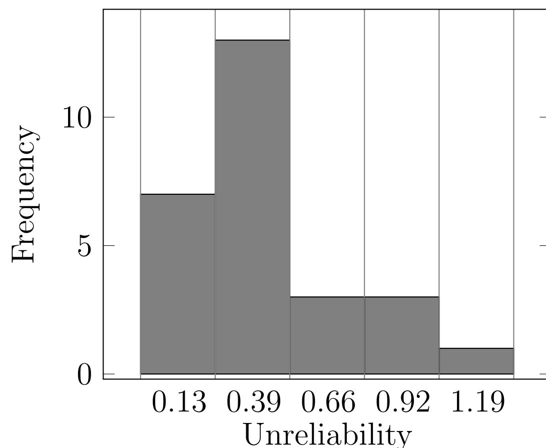


Figure 5.5: Histogram of unreliability of nodes in our dataset

on or near the line $f(x) = x$ demonstrates that in general, nodes had high agreement with their neighbors in terms of RSSI signal.

If we examine the node pairs where there was disagreement of more than 1 dBm, we see some trends of repetition: one node on Floor 2 performs particularly badly, and has absolute symmetry error greater than 1 with 7 out of 11 other nodes in the room. We see this node highlighted as n_i in Figure 5.4 – the figure compares n_i's outgoing and incoming values with each other node in that room. This is contrasted with another node from that room, n_j, which has very low unreliability: the only node with which n_j's symmetry error is above 1 dBm is n_i.

We can quantify the overall unreliability of a node by taking the mean of the absolute symmetry error over all other nodes (provided that both nodes have outcoming and incoming values).

Definition 21 (unreliability)**.** *This is a function with type $\mathcal{N} \to \mathbb{R}$, and is calculated as follows*

$$\mathsf{unreliability}(n) = \sum_{n_i \in \mathcal{N} \setminus \{n\}} \frac{|\mathsf{symmetryError}(n, n_i)|}{|\mathcal{N}| - 1}$$

Examining Figure 5.5, we see that the majority of our nodes in our dataset have low unreliability, however, we do have three nodes with unreliability above 1. By analyzing the data in this way, we can get a sense of which,

if any, nodes are not reliable, so that we can attempt to not use their data when possible, as discussed in Section 5.3.

5.5 In-depth example

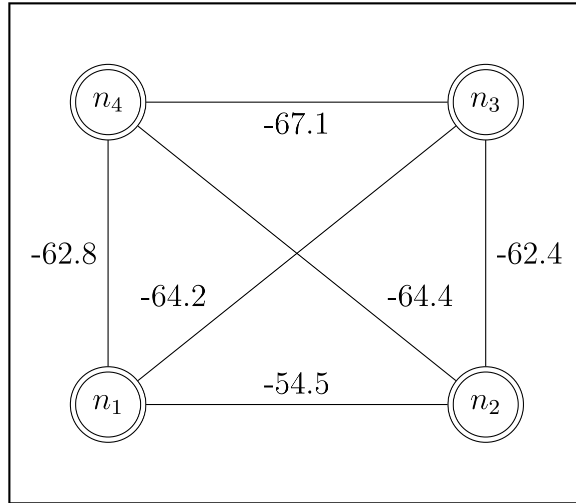


Figure 5.6: Office layout. Nodes are placed in sockets, and socket distances are proportional to diagram.

The Office, seen in Figure 5.6, shows a succinct example of RSSI data which does not always match the physical distance between nodes. We can see here that $\mathsf{RSSI}(n_1, n_2)$ has a stronger signal than any of the others, even though the distance between n_1 and n_2 is larger than n_1 and n_4, or n_2 and n_3.

Out of the 12 total constraints we have to check for this problem, the following four constraints in Table 5.1 all fail[2].

While the correct solution in this case is $m_1 = \{s_i : n_i \mid i \in \{1, 2, 3, 4\}\}$, the following solution also passes the same percentage of constraints (75%):

$$m_2 = \{s_1 : n_1,\ s_2 : n_4,\ s_3 : n_3,\ s_4 : n_2\}.$$

[2]To simplify the figure, let it be the case that each node n_i is placed in socket s_i. We will therefore also simplify the constraints listed in Table 5.1 by omitting the $\mathsf{in}(n_i, s_i)$ components, for brevity.

$S(s_1, s_4, s_2) \to N(n_1, n_4, n_2)$
$S(s_2, s_3, s_1) \to N(n_2, n_3, n_1)$
$S(s_3, s_4, s_1) \to N(n_3, n_4, n_1)$
$S(s_4, s_3, s_2) \to N(n_4, n_3, n_2)$

Table 5.1: Failing constraints from Figure 5.6

By switching the placements of n_2 and n_4, we score equally well as the correct solution. This demonstrates a real-world scenario where we need our additional procedures to determine the correct placement of nodes. We can also observe that we can get from m_1 to m_2 by reflecting around a line of *almost*-symmetry. This is an example where we have multiple solutions available due to a combination of both rough symmetry and RSSI unreliability, the two challenges discussed in Section 2.3.

As in the Office, the RSSI data of Floor 1 and Floor 2 was also not effectively accurate, and in these two problems, it was also the case that the best scoring solution was not the correct one. However, in all cases, the correct solution was in the first batch of possible solutions returned by solve, and was therefore identified in the first iteration of chooseSolution.

5.6 Implementation

This project was implemented in Python 3, with an additional dependency on Clingo for the ASP version, as discussed in Section 3.2. The code is available upon request on GitHub[3].

5.7 Performance

Tables 5.2 and 5.3 show the performance of solveIterative on our real-world data. The *Solution Correct* and *#Verifications* columns indicate that the first proposed solution by chooseSolution was correct, and further iteration was not needed. This supports our hypothesis that in general, solutionCorrect is more a safety check than anything else, and we should rarely have to look past the first set of possible solutions returned by solve.

[3]Code can be found at `https://github.com/ddanco/nodemapper`. The repository is not public by request of the company; please contact ddanco@gmail.com to be granted access.

For all of these tests, we set ϵ to 2m.

n	Office	Time	# Oracle calls	Solution correct
4	Small office	0.3	2	Yes
12	Floor 1	36.0	4	Yes
12	Floor 2	120.6	3	Yes

Table 5.2: solveIterative overall statistics with Office Data

n	Office	Threshold	# Solutions proposed	# Verifications	Constraint score of solution
4	Small office	1	6	1	1
12	Floor 1	0.8	33	1	0.84
12	Floor 2	0.8	11	1	0.85

Table 5.3: solveIterative run details with Office Data

The reason for the threshold of 1 for the small office (despite the failed constraints as shown in Table 5.1) is that, due to the fact that we have a simple small almost-square configuration of nodes, the ϵ parameter led to the elimination of all constraints. This is not a problem, as this would only happen in a very small space (which should then have correspondingly few nodes), so the algorithm still terminates with the correct answer using 2 oracle calls (which was inevitable for an almost-symmetric setup considering imperfect data, as shown in 2.3.1) and taking only 0.3 seconds.

Table 5.4 gives an overview of the speed and effectiveness of our different algorithms, given varying types of data.

	Algorithm 1	Algorithm 2	ASP	solveIterative
Handles perfect data	Yes	Yes	Yes	Yes
Handles effectively accurate data[4]	Sometimes	Yes	Yes	Yes
Handles non-effectively-accurate data	No	No	No	Yes
Handles symmetry	No	No	Yes	Yes
Speed	Fast	Fast	Slow	Slow

Table 5.4: Pros and cons for each algorithm

As we can see, our greedy algorithms are both quick, but understandably do not do well when there are multiple solutions (they will be right 50% of the time, given perfectly-correlated data for a symmetric grid of sockets) or when the data is not effectively accurate.

Our two exhaustive solutions, the ASP solution and the solveIterative algorithm, are slower (as would be expected), but are able to handle uncertainty much better, by bringing in the oracle to determine the correct solution out of multiple. However, as our ASP approach requires all constraints to be fulfilled, it too cannot handle data that is not effectively accurate.

Out of all of the algorithms, solveIterative is the only one which is able to successfully find the solutions to any of the three rooms in the real-world dataset.

[4] Barring symmetry issues.

Chapter 6

Conclusion

The goal of this project was to create a semi-automated system to determine node positions in space using the RSSI signal strength between them. In that sense, we have succeeded in creating such a system.

Our system, as far as we have seen with the data available, is able to use logical reasoning mixed with limited human (oracle) input to suggest the most likely mapping between nodes and sockets.

We have also shown that it is impossible to be certain of all of the nodes' placements until we have physically checked the location of them all (except for a final node, whose placement is determined automatically by virtue of all other socket placements being taken). Therefore, we also propose a system of verification so that a human operator can easily indicate if a proposed solution is correct or not.

With our available real-world data, the first proposed solution (after requiring an average of 4 oracle queries for the floors with 12 nodes) has been correct 100% of the time. Thus, the verification procedure has, to this point, been able to be the final step of the procedure, with no need to iterate further afterwards to find additional possible solutions.

We have also provided tunable parameters – FrontierSize and CloudSize to adjust the number of placement options considered per iteration, and ϵ, to adjust the number of socket facts included. These parameters can be adjusted to balance speed and exhaustiveness of search. We propose that the parameters be set such that speed is prioritized – so far, with the data we had available, we did not need exhaustive search and the correct solution was always found with the parameters highly favoring speed over thoroughness of search. However, if a case were to arise when the algorithm is not able to find the solution, those parameters can be adjusted such that eventually

every possible mapping is tested and the solution is guaranteed to be found.

The main improvements that will need to be made for this to be a viable business solution are: scalability in terms of speed, and handling of complex layouts and barriers.

The algorithm is currently with acceptable speed for the spaces that we have been provided (rooms with 4 to 12 nodes), but would become prohibitively slow in situations with many more nodes. The focus of this project was to get a prototype designed and working, and there still remain many sections of the code where optimization is possible.

Additionally, as mentioned below in Section 6.1.1, there are further challenges that would have to be addressed before more complex building setups would be supported. It is likely the case that either clustering of rooms and floors, scaling of distance values (e.g. adding artificial distance between sockets to represent walls or ceilings), or both would be required before a building would be able to be handled in its entirety.

6.1 Future work

6.1.1 3-Dimensional spaces

The existing solution has been made for 2-dimensional space. It would be very easy to handle 3-dimensional space, by simply considering the 3-D formula for Euclidean distance measurement rather than the 2-D version.

However, there are practical reasons why 3-D space might lead to more issues. RSSI values will likely be greatly affected by thick ceilings, with pipes and plumbing in between, and the signal strength between two nodes placed directly on top of each other on different floors will likely have weak or non-existent signals between each other. However, as the physical distance between them will be small, the correlation constraint will be violated. In situations such as these, it would be better to consider the floors as separate problem spaces. As we do not initially know which nodes are on which floors, we will have to be able to properly segment the data to allow the algorithm to be run separately. This brings us to our next possible feature: clustering.

6.1.2 Clustering

Clustering the nodes into floors, halls, or other delineated spaces can have many benefits. First, as discussed in the previous subsection, it can help

remove errors introduced by thick walls or floors. Secondly, while our algorithm can run quickly for smaller numbers of nodes, as we venture into larger buildings and warehouses, the number of nodes can lead to prohibitively slow processing.

One of the company's customers is a logistics company with a single warehouse that has 6 different halls, each with 225 lights. 225 nodes is already a large number of nodes to handle, and handling all halls at once (1350 nodes) would be much more difficult. In cases like these it would be especially useful to be able to separate nodes by halls as a pre-processing step.

6.1.3 Oracle implementation

We have so far described an oracle as a function of type $\mathcal{N} \rightarrow \mathcal{S}$, and mentioned that this identification would be carried out by a human operator. There are many ways that this could be done. The most simple would be a system in which the user turns on a given node (the company already has an app with the functionality to do this), and then reports back the location (perhaps by clicking a location on a map on that same app). There is a project being done simultaneously with this project, to make a system in which nodes could be identified by pointing a camera at them and evaluating a unique blinking pattern. While this function would be of type $\mathcal{S} \rightarrow \mathcal{N}$, all of our functions can fairly easily be refactored to work in this direction as well, if this enhanced oracle were to be made available. This node identification functionality could be used for our oracle(s), both for identifying nodes (for creating seed mappings as well as narrowing down options in chooseSolution) and for the verification oracle.

It is our hope that with these future optimizations, our prototype could be made into a viable solution for mapping wireless nodes to their physical locations.

Bibliography

K. Benkic, M. Malajner, P. Planinsic, and Z. Cucej. Using RSSI Value for Distance Estimation in Wireless Sensor Networks Based on ZigBee. In *2008 15th International Conference on Systems, Signals and Image Processing*, pages 303–306, 2008. URL http://doi.org/10.1109/IWSSIP.2008.4604427.

Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer Set Programming at a Glance. *Commun. ACM*, 54(12):92–103, dec 2011. ISSN 0001-0782. URL https://doi.org/10.1145/2043174.2043195.

Wouter Bulten, Anne C. Van Rossum, and Willem F. G. Haselager. Human slam, indoor localisation of devices and users. In *2016 IEEE First International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 211–222, 2016. URL https://doi.org/10.1109/IoTDI.2015.19.

Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The Potsdam Answer Set Solving Collection. *Ai Communications*, 24(2):107–124, 2011. URL http://doi.org/10.3233/AIC-2011-0491.

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + Control: Preliminary Report, 2014. URL http://doi.org/10.48550/arXiv.1405.3694.

Federico Heras, Javier Larrosa, and Albert Oliveras. Minimaxsat: A new weighted max-sat solver. In João Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing – SAT 2007*, pages 41–55, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-72788-0. URL https://doi.org/10.1007/978-3-540-72788-0_8.

Neal Patwari, Alfred Hero, Matt Perkins, Neiyer Correal, and Robert O'Dea. Relative Location Estimation in Wireless Sensor Networks. *IEEE Transactions on Signal Processing*, 51:2137–2148, 08 2003. URL http://doi.org/10.1109/TSP.2003.814469.

Ramiro Ramirez, Chien-Yi Huang, Che-An Liao, Po-Ting Lin, Hsin-Wei Lin, and Shu-Hao Liang. A Practice of BLE RSSI Measurement for Indoor Positioning. *Sensors*, 21(15), 2021. ISSN 1424-8220. doi: 10.3390/s21155181. URL http://doi.org/10.3390/s21155181.

Omar Said. Automated RSSI-based Lights Distribution for Smart Buildings, 2022. URL https://www.cs.vu.nl/~wanf/theses/said-bscthesis.pdf. Bachelor thesis.

Rakshit Shah. Convert RSSI Value of the BLE (Bluetooth Low Energy) Beacons to Meters, Jun 2021. URL https://medium.com/beingcoders/convert-rssi-value-of-the-ble-bluetooth-low-energy-beacons-to-meters-63259f307283.

Bita Shams and Saman Haratizadeh. Graphloc: a graph based approach for automatic detection of significant locations from gps trajectory data. *Journal of Spatial Science*, 63(1):115–134, 2018. doi: 10.1080/14498596.2017.1327374. URL https://doi.org/10.1080/14498596.2017.1327374.

Kannan Srinivasan and Philip Levis. RSSI is Under-Appreciated. In *Proceedings of the third workshop on embedded networked sensors (EmNets)*, volume 2006, pages 1–5, 2006.