A Fuzzy Architecture for Robotics Sensor Information Integration

A THESIS PRESENTED
BY
JOTHAM C. LENTZ
TO
THE DEPARTMENT OF MECHANICAL ENGINEERING

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF MECHANICAL ENGINEERING
IN THE SUBJECT OF
MECHANICAL ENGINEERING

SAINT MARTIN'S UNIVERSITY
LACEY, WASHINGTON
MAY 2018

Jotham C. Lentz, M.M.E. Candidate	Date
Rico A.R. Picone, Ph.D., Thesis Committee Chair	Date
Frank M. Washko, Ph.D., J.D., Examiner	Date
Stephen Parker, Ph.D., Examiner	Date

Thesis advisor: Professor Rico Picone, PhD. Jotham C. Lentz

A Fuzzy Architecture for Robotics Sensor Information Integration

Abstract

The aim of this paper is to present an information system to enhance a humans ability to interpret large amounts of qualitative and quantitative data. These sorts of data have traditionally been difficult to correlate. By leveraging the information architecture introduced in the author's *The fuzzification of an information architecture for information integration*,[15] it is possible to enable a human confronted with large amounts of data to identify hidden and unexpected relationships much more easily.

The information system proposed leverages the dialectical method to minimize visible navigable information, mitigating information overload. Quantitative data is processed through fuzzy membership functions into fuzzy sets. Assigning these qualitative representations to the quantitative data synthesizes the disparate data types into a single representation. The relations among the sets can then be analyzed via fuzzy set theory and the underlying structure modeled for presentation to a human partner in a human-robot team.

By observing this organic structure, a human can rapidly recognize previously unknown relations among disparate types of information. Computational scope can then be focused for other software tools, resulting in an enhancement of human-computer interaction.

Contents

For my father who established the value of ceaseless diligent learning. And for my mother who demonstrated the virtues of temperance and humility in her dealings with others and patience when dealing with me.

Acknowledgments

I'M HUMBLED, by the magnitude of the knowledge generously gifted to me by the teachers and coaches I've been privileged to learn under throughout my learning career. In particular I would like to express my deep gratitude to my advisor, Professor Rico Picone who's valuable technical support, enthusiastic encouragement and useful critiques catalyzed most of the effectual activity on this project. The following was written with full cognizance of the enormous debt owed to all of my instructors and mentors.

Listing of figures

0

Introduction

The proliferation of mechatronic systems has led to an unprecedented ease in acquisition of empirical data about the world. Swarms of autonomous or tele-operated robots can be deployed with arrays of sensors, each relaying back streams of data on a multitude of environmental variables. However, despite advances in computing power and speed, this ubiquitous availability of raw data has not been coupled with a commensurate increase in the human teammates ability to process it. Sorting through vast quantities of new and prexisting data to recognize underlying patterns within is largely the purview of the nascent field of artificial intelligence.

However, frequently it is the case that certain types of information are better suited to review by a human intelligence. This is particularly the case with more qualitative data. Our ability to draw on past experiences to recognize patterns and draw conclusions and inferences is still largely unmatched. For an unaided human operator the sheer volume of data available can easily lead to "information overload" and a subsequent bottleneck in the bandwidth of the information system. A computer augmented method to create a human-navigable structure from unstructured data could partially mitigate this risk.

This paper proposes an information system for augmenting human intelligence through which large quantities of sensor data can be associated with sets, then relationships among the sets identified and passed to a human. By integrating both quantitative and qualitative data into a structure that can be displayed by a computer, the possibility is opened for humans to make connections among data and other forms of information, like a paragraph in a document, or an image. This structure is not imposed upon the data, it is constructed by defining an "organic hierarchy"[13] which visualizes the underlying flow of the intersecting data streams by calculating the intersections between data sets.

The framework for this system was laid out in *Human Interface and the Management of Information. Information and Knowledge Design*[14] (referred to as the crisp organic hierarchy) The organic heirarchy allows atoms to be categorized with any number of categories. It calculates which categories can be considered subcategorys of others, for example category *Y* in Figure 1 is a subcate-

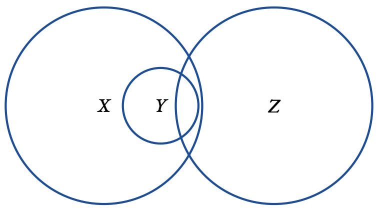


Figure 1: Venn diagram for the categories of example categories.

gory of category X. Nodes are defined as recursive intersections of categories. Levels $\mathcal{L}$ of the heirarchy are defined where the level number is the number of categories in the intersection. For example $\mathcal{L}_2$ in Figure 2 contains all intersections of two sets. $\mathcal{L}_0$ is the Union node which contains the union of all categories.

A key feature of the organic heirarchy definition of visible and hidden relationships between nodes to be displayed in a user interface. Any category that is wholly a subcategory of another is not displayed at the level above. In Figure 1 Y is a subcategory of X. Y is therefore not displayed at $\mathcal{L}_1$ in Figure 2. The only way to access Y is by first browsing X, then $X \cap Y$, or through Z, $Z \cap X$, $Z \cap X \cap Y$. This definition of "visible" and "hidden" relations minimizes the amount of information displayed to a human teammate at each level in the organic hierarchy.

The organic heirarchy as laid out above was expanded to incorporate fuzzy set information in *The Fuzzification of an Information Architecture for Information Integration: Human Interface and the Management of Information: Information, Knowledge and Interaction Design*[15] (referred to as the fuzzy organic hierarchy). This enhancement as it relates to the particular problem of robotic sensor integration will be expanded on.

The generation and transmission of set relationship information through the information system

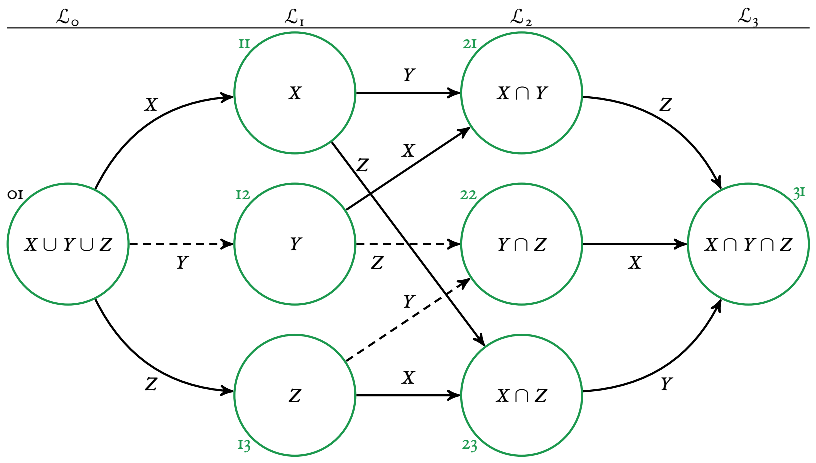


Figure 2: graph of categories for Example ??. Dashed lines are **hs**-relations and solid lines are **vs**-relations.

can be broken into several discrete tasks: acquire the data stream, categorize the data, store and retrieve, and visualize/interface the structure. Figure 3 illustrates the interoperability between layers. Organizing these tasks into layers permits variances in individual layers in order to customize for individual applications, without requiring a redesign of the entire architecture. Data can be gathered in any manner, and simply plugged into the categorization engine with an API. The categorization engine stores the data and category information in a database for later retrieval by the presentation engine. The organic hierarchy is then presented to a human partner in a browse-able web interface.

This paper will lay out one of many possible instantiations of this innovative information system. Each processing step will be defined and tools used to achieve individual processing steps documented. Core concepts used in the development of the fuzzy organic hierarchy will be expanded on. Finally the information system will be tested through simulation and the user interface investigated.

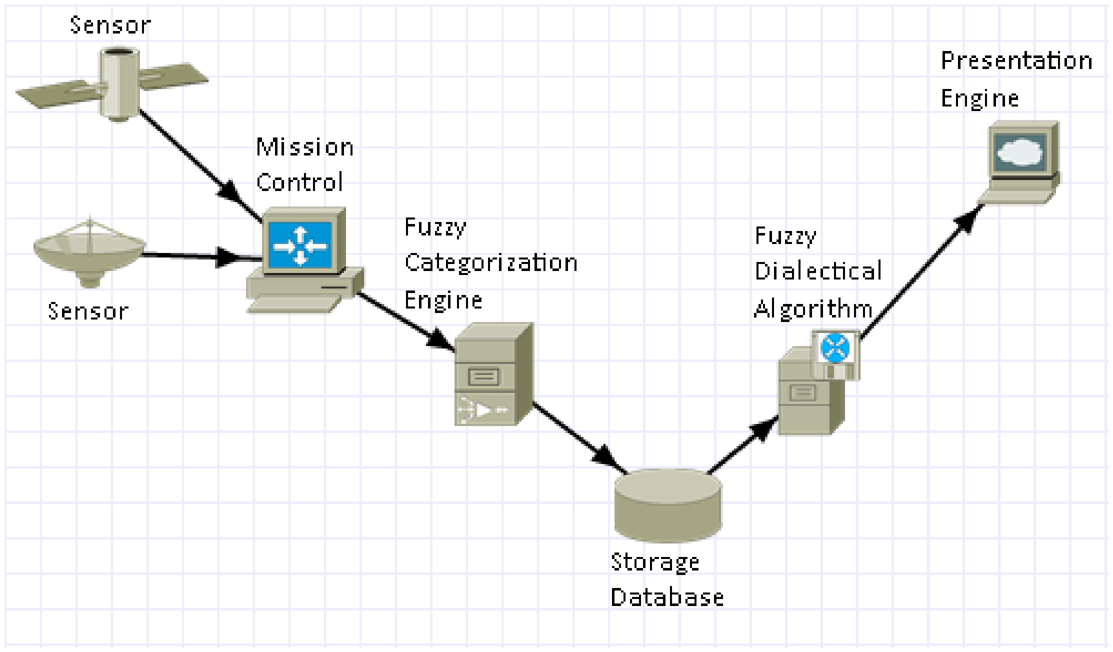


Figure 3: Processing Steps from Sensor to Display

We can be absolutely certain only about things we do not understand.

Eric Hoffer

1 Fuzzifying the Data Stream

Some data naturally falls into discrete categories. This tends to be data of a more qualitative nature. However, much of the numerical data gathered from sensor streams is much more quantitative in nature and does not fall into such discrete categories. Take, for example, temperature: we could assign categories of hot or cold but these are subjective. For example hot to an oceanic Remotely Operated Vehicle is orders of magnitude different from hot when taking measurements from an active volcano. The categorization system should be able to accommodate contextual definitions that are dependent on the context of the experiment. Furthermore, when it comes to these subjective forms of information, categorical boundaries cannot be clearly delineated. Refer back to the temperature example. If the boundary for hot and cold were set at a value of 25 degrees Celcius, then that would mean that 24.9 deg C would be in the cold category and 25.1 deg C hot. But we know intuitively 24.9 deg C is much closer to being hot than a value of say 0 deg C would be. In order for the system to give useful information, some accommodation must be made for this sort of nuance in the definitions of categories, allowing for *degrees* of membership in each category.

1.1 Fuzzy logic overview

The perfect tool for this is fuzzy logic. It allows the definition of categories using natural language. Individual data points can then possess membership to varying degrees in each category. Fuzzy logic allows the modeling of vagueness in our categorical definitions. It does this by assigning a membership value for each category between zero and one. Fuzzy systems excel in two contexts: (1) highly complex systems whose behaviors are not very well understood, and (2) situations requiring an approximate, but fast solution.[17] Both of these situations would apply to the problem of analyzing sensor data. Mobile sensors such as on an ROV could be employed to gather detailed data about a poorly understood but complex system. Also, due to the degree of uncertainty extant in certain types of semi-automated information gathering such as real-time image recognition in the field, a slow exact solution is infeasible. This situation lends itself much more readily to an imprecise, rapid solution which a human partner can then examine.

1.2 Fuzzy sets

Membership in a fuzzy set is defined by a fuzzy membership function. An element x in a crisp value range known as the universe is evaluated against the membership function $\mu(x)$ and given a membership value in the interval $[0, 1]$, with zero meaning "no" membership and unity meaning "full" membership.[27] Membership functions can take various shapes in order to conform more closely to natural language. Figure 1.1 shows one membership function based on a Gaussian distribution and another that is a triangular membership function. In the figure, data values x range from -1 to 4 while membership in a set $\mu(x)$ ranges from 0–1. Set-theoretic operations such as union $\cup$ and intersection $\cap$ are also defined as they would with more traditional crisp sets.[27] This fuzzy extension of set theory is critical to the implementation of the fuzzy dialectic architecture for this purpose.[15]

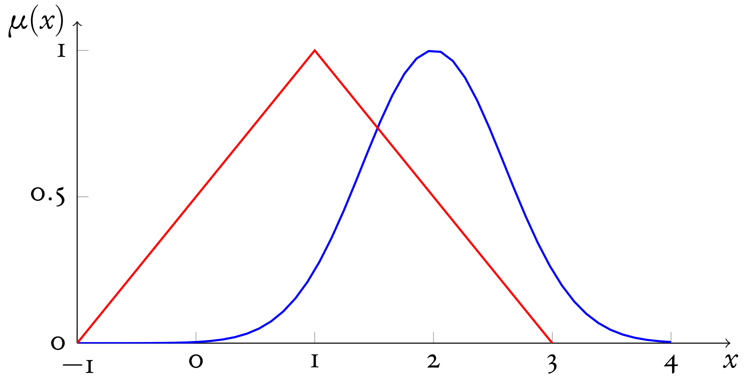


Figure 1.1: Gaussian and Triangular Membership Functions

1.3 Fuzzy logic engine

In order to pass fuzzy set membership values to the dialectic algorithm a fuzzy logic software package is required. This package needs to accept pre-defined membership functions, and process large streams of sensor data in near real time. SciKit-Fuzzy[24] is a fuzzy logic toolbox for the well known scientific software suite SciPy. Scipy is an extremely popular Python-based[18] ecosystem of open-source software for mathematics, science, and engineering[7]. As such it is likely to be used in some fashion in real world scientific robotics and data analysis. It therefore makes sense to remain interoperable with it as much as possible. SciKit-Fuzzy contains fuzzy logic algorithms for defining fuzzy sets and for performing set theory calculations upon them.

1.4 Implementation

Typically, fuzzy logic implementations extend into using fuzzy set theory to calculate results using tools such as control system simulations[24]. Those fuzzy results can then be de-fuzzified and returned as crisp results. In this implementation however, the fuzzy membership values will be retained and passed to the display algorithm for human interface. For the purpose of establishing the organic hierarchy, fuzzy membership values will be calculated and retained for display. It is left up to the human partner to decide which tool is most appropriate for further analysis depending on the type of and relationship among the data.

The problems are solved, not by giving new information,

but by arranging what we have always known.

Ludwig Wittgenstein

2 The Fuzzy Dialectic structure

Analyzing large amounts of stored data to identify internal structural relationships is a nontrivial task. The algorithm developed for this purpose needed to be able to read data in various forms, as well as the categorical information associated with each data point. Many data sets also are not strictly hierarchical in nature. There's no way of knowing before analysis, what structure will intrinsic to the data. The structure must be discovered, rather than pre-defined and filled in.

The algorithm presented here excels at this. With the addition of quantitative data via the mechanism of fuzzy set theory it is agnostic as to data type and discovers the hidden structure within the raw data. It relates this structure in a hierarchical format which is easily browse-able by a human partner.

2.1 The Fuzzy Dialectic Architecture

The structure of the fuzzy architecture will be defined from fuzzy set theoretic relations from unstructured (fuzzily) categorized information[*].

Consider a collection of data, each member of which we call an *atom*. Each atom is associated to a certain degree with a collection of *categories*[†] which are represented as fuzzy sets. Crisp set operations union $\cup$ and intersection $\cap$ are analogous to the fuzzy set operations union and intersection.[17,27]

The fuzzy dialectical structure is a *directed graph* of nodes and edges.[22,1] Other than the "universal" *union node*, which contains all atoms, every node in the graph represents the fuzzy intersection of a collection of categories (fuzzy sets). Just as an atom can belong to a given category to some degree, so an atom can belong to a given node to a certain degree (membership value). This degree is computed from the fuzzy intersection operation, which returns the minimum membership value for a given atom shared between two nodes; i.e. let the element x in the universe X have membership $\mu_A(x)$ in fuzzy set A, where μ_A is the membership function for set A, let x have membership $\mu_B(x)$

[*] Portions of this section are reprinted from the author's *The fuzzification of an information architecture for information integration.*[15]

[†] where used, the term category is meant in a general fashion to refer to a logical grouping of data points, not the algebraic structure known as a category

in fuzzy set B with membership function μ_B, and let $\wedge$ be the operator that takes the minimum of its two arguments—then the membership of x in the fuzzy intersection $A \cap B$ is[17]

$$\mu_{A \cap B}(x) = \mu_A(x) \wedge \mu_B(x). \tag{2.1}$$

Directed edges connect the nodes to generate a natural hierarchy. All edges are defined by *has a priori subcategory* relations or s-relations; for instance, the node $A \cap B$ is an *a priori* subcategory (fuzzy subset) of fuzzy sets A and B. This generates a natural hierarchy with graph *levels* defined by the number of categories that intersect to define the node; e.g. node $A \cap B$ has level two.

Two types of s-relation are defined: the suggestively named (1) *has visible a priori subcategory* or vs-relation and (2) *has hidden a priori subcategory* or hs-relation. The definition of the hs-relation first requires the concept of a *metacategory*. A metacategory for a given node is a collection of subcategories that contain as a subset all atoms associated with the node. A node's vs-relations are those that have tails connected to the node and heads connected to subcategory nodes contained in a minimal metacategory. By minimal, we mean containing the minimum number of subcategories to fully contain all atoms. An hs-relation is defined as any s-relation that is not a vs-relation.[‡]

Finally, atoms themselves can be either "visible" or "hidden," names suggestive of how the user interface in later sections will be defined. An atom is visible at a given node if and only if it has nonzero membership in all categories intersected to define the node and zero membership in all

[‡]These definitions have strong parallels in[14], where more mathematically oriented definitions are presented. We favor a narrative approach here. The interested reader may find the explicit mathematical definitions of the previous work elucidating.

others. This definition requires that an atom be visible in one and only one node in the structure.

2.1.1 Visibility and hiddenness

The names given to the two types of atoms and s-relations—“visible” and “hidden”—are a crucial aspect of the structure’s advantage for intelligence amplification in human-computer interaction. In a user interface these signifiers will be taken literally: at a given node, hidden atoms and hidden s-relations (edges) will not be presented to the user. The definition of each guarantees that a hidden atom will be visible if one navigates via visible s-relations to a lower level. The primary advantage of this from a usability standpoint is that the user is not inundated with as much information, one of the key aspects of a hierarchy, while remaining in a logically categorized structure—the other key aspect of a hierarchy.

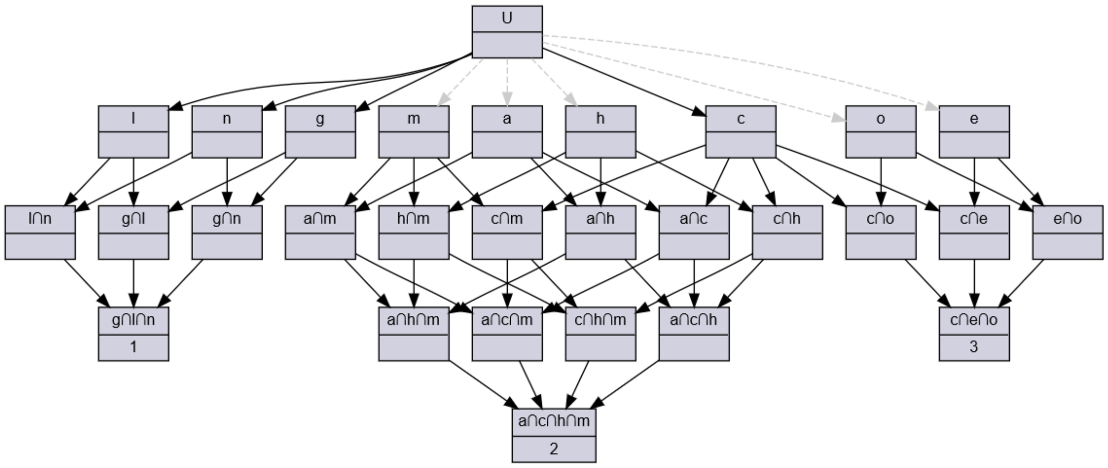


Figure 2.1: Example structure showing visible and hidden relations

2.1.2 Structure as estimation

Let us consider what type of structure this graph has. It is constructed from a collection of fuzzily categorized atoms (in the case of quantitative information, these atoms are data points with membership values in each category). For a given variable, say temperature, the subset relationships are pre-defined by the membership function of the data; e.g. "luke-warm" will be a subset of "warm." However, the inter-variable relationships are typically not so; for instance, "cold" might be a subset of "high-pressure." The structure defined here can be understood as an estimation process for these relationships, one of several applications to be discussed in later sections.

2.1.3 Organic hierarchy

The hierarchy is "organic" in the sense that it evolves with each new atom's introduction to the structure. Unlike a traditional static hierarchy that requires insertion into the structure at a specific node, an organic hierarchy evolves with the information, and a user need not explicitly define the hierarchy, which is implicit in the user's categorization of each atom.

2.1.4 Invariance of path

Another aspect of the architecture is that of the invariance of path—that is, the fact that navigation of the structure is invariant to the order in which one navigates. Let us represent each navigation along a vs-relation as the "selection" of the additional category for the intersection that defines the edge's head node. Let each selection add that category to the path, similar to a traditional file system path (e.g. /A/B/C). For the dialectical architecture, the order of the selection is inconsequential; for instance, /A/B/C, /B/A/C, and /C/A/B all point to the same node, due to the invariance of the fuzzy intersection operation.

2.1.5 Fuzzy flows

The concept of a *flow* was introduced in the context of the crisp dialectical architecture[14]. It's definition—a flow is a series of atoms—applies directly to the fuzzy dialectical architecture, but unique implications emerge. Previously, flows have been used to represent the sequential aspect of several types of information, such as narrative, audio, and video. In a fuzzy dialectical architecture representing quantitative information, each data point is an atom and a data stream is a flow. Thus each atom should not be presented to a human teammate as an isolated data point at each node, but should be displayed in a plot (more on plotting in section 2.3) with a trace representative of a flow. This yields an additional method of navigation, as well. A flow may intersect a node and continue on another node; the user should be able to "follow the flow" to the other node in addition to navigating the categorical structure directly, via edges.

2.1.6 Fuzzy dialectic

The Fichtean *dialectic* is the evolution of understanding. It is often represented as a position taken, a thesis; an alternative position taken, an antithesis (not necessarily in conflict with the thesis); and a sublation of the two to form a synthesis[8].[§] Fichte goes so far as to claim that every act of thinking is a

[§]See Ref.[8] for a discussion of the similarities and differences between the Fichtean and Hegelian dialectics.

synthesis,[9] and so it is natural for an information architecture designed to enhance human thinking to express this model.

The fuzzy dialectical architecture includes a special type of flow to express the dialectic called the *thesis flow*. A thesis flow is defined for each node and can be considered to be a user's description of the intersection of the categories defining the node. When another flow intersects a thesis flow, it is considered an antithesis flow to the thesis. A user would then be prompted to resolve these to form a newly informed thesis. But flow intersections are in fact not limited to thesis flows, so each intersecting flow is an antithesis to a given flow. This dialectical manner can have many instantiations; for instance, consider a thesis flow for the node $A \cap B$ (the relationship between A and B). Perhaps a user has written a document comprising this thesis flow, and then brings in a new quantitative data set such that the flow it defines intersects $A \cap B$. The thesis flow would then require the sublation of the thesis and the antithesis (data). In this way, when newly connected information is introduced to the information system, those flows that are affected can be immediately identified.

2.2 Algorithmic instantiation of the structure

A naïve approach to writing an algorithm to instantiate the fuzzy dialectical architecture would yield exponential computation time. In this section, we discuss some salient ideas to consider when instantiating the architecture. A highly efficient algorithm for the structure remains an open problem, but progress has been made.

A key insight is that the entire structure need *not* be recomputed when a new atom is inserted or removed. This allows us to incrementally build a structure, which should, of course, be invariant to the order in which atoms are inserted. This is especially important for real-time applications such as robotics.

What requires recomputation when an atom is inserted? Only the relations originating at those nodes that are constructed by categories in associated with the new node need be recomputed. That is, (typically) most of the structure is untouched by the insertion of a new atom. Furthermore, the visibility or hiddenness of an atom never needs to be computed because an atom is visible in only one node, that which is defined by the intersection of all categories associated with it.

Moreover, any node that is *new* to the structure requires no structural computation, since all its relations must be vs-relations because no relation can possibly contain more than the others, since only one atom (the new one) is at the "bottom" of those paths. This allows extremely quick insertions for new categories and combinations of categories.

The unavoidably most computationally intensive aspect of the computation is the re-computation of metacategories for those nodes affected by the insertion of a new node. It is important to note that once a minimal metacategory has been found at a given level, no more levels are required.

It is also of note that memory resources can become an issue if the structure is maintained in memory (especially if metacategories are stored). It is advisable to use a graph database to persist and access the structure.

2.3 Human interfacing for the fuzzy architecture

This information architecture can have innumerable instantiations.[13] Guidelines for these instantiations are presented below¶.

2.3.1 general guidelines for dialectic interface architecture

The user should be able to browse nodes like a traditional hierarchy.

The nodes represent the intersection of categories, as they typically do in a hierarchy or in tag-based browsing. The hierarchy has a long and illustrious history of value to human thinking[4]. Although the structure is, in fact, a graph, it will be natural to most users to experience it as a hierarchy. The "hierarchy" the user interacts with will be *organic* in the sense that it may change when new information is added to the system. All the spatial metaphors so valuable to hierarchies will be applicable, like "up" and "down," "in" and "out." At each node, the visible edges should be represented as single categories—the category that would be intersected with the current node to yield the lower node.

The user should be presented only visible edges.

"Information overload" has been identified as a significant challenge to our information age.[19,26] One of the primary advantages of the dialectical architecture is that it minimizes the amount of information a user is presented at each node, much like a traditional hierarchy,

¶Portions of this section are reprinted from *The fuzzification of an information architecture for information integration.*[15]

which "tucks" the information that is further-categorized into lower levels. This means "hidden" atoms and edges should not be presented, explicitly (although exceptions can be made, of course). In some instances, hidden atoms, as defined above, might also be hidden from the user's view; however, caution is advisable here, since in some instances, the interface might call for their visibility.

The user should be able to browse "up" to any parent node.

The property of the architecture that the path order is invariant can be exploited to allow browsing the structure in a manner analogous to the hierarchical "up-one-level," but with multiple possibilities. The user can traverse "up" to any parent node, of which there may be several, unlike in the hierarchy, which allows each node to have only a single parent. This can be visualized by allowing the user to de-select any selected category along the path, and not merely the last-selected.

The user should be able to browse by following edges or flows.

Following edges is the *structural* method of navigating and is isomorphic to browsing traditional hierarchies. The dialectical architecture adds the ability to browse along *flows* as well. A flow can intersect a node for one or more consecutive atoms, then move to another node. For instance, an article may be discussing the intersection of several topics, then drill deeper into it with an additional categorization, which would lead it to a child node. This could be navigated by "going with the flow," such that the user continues to see the series of atoms that comprise the flow.

The user should be able to synthesize newly intersecting flows.

The dialectical aspect of the architecture requires the thesis–antithesis–synthesis structure of information development. An information attempting to enhance human thinking should certainly capture the development of that thinking, which this feature accomplishes. A flow can be "intersected" when another flow is coincident with a node the flow traverses, and this intersection may provide a new perspective to the original flow (antithesis). A user should be able to synthesize the two perspectives such that their information system remains well-curated.

The user should be able to view quantitative data in graphs.

With the inclusion of quantitative information, the fuzzy dialectical architecture should have a user interface that presents quantitative information in a concomitant manner, typically a graph. A data point (atom) that is visible at a given node may belong to a multivariate data set and belongs to the node with some membership value in the range $[0, 1]$. A two-dimensional graph of given data set intersecting a node is often the best option; the user's ability to change which variables are plotted on the abscissa and ordinate axes is important. Data series should be connected and multiple series on the same graph should appear with different line properties or colors.[‖]

The user should be presented the membership of an atom in a node.

[‖]We suggest a designer make liberal use of the advice given by Tufte[23] for the visual display of quantitative information.

The fuzziness of the architecture yields an interesting aspect of the information: the degree to which each atom belongs to a given node. For quantitative information, the membership value of each point in the node should be presented; we suggest opacity of the data point. For other types of information, several techniques are possible, including sorting, iconic differentiation, color, and opacity.

Everything that happens happens as it should, and if you observe carefully, you will find this to be so.

Marcus Aurelius

3 An Application to be Simulated

WITH THE INTENT OF demonstrating how the information system presented here can be applied for identifying previously unrecognized relationships among data, a simulation environment was developed. The environment is meant to represent a system of underwater caves or rooms, each room having different environmental qualities. The entire cave system contains five distinct fictional items of food and five separate fictional species of fauna that consume the food: Shrieking Eels, Decapodians, Jaguar sharks, Furry trout, and Battletoads. Each species has proclivities toward environmental conditions temperature and illumination as well as food types. A scientific Robot is to navigate the environment, gathering data which is to be fed to the fuzzy dialectical algorithm for analysis.

Some relationships among species and habitats are well known, others are not. It is the role of the dialectical algorithm to organize the known relationships as well as to discover unknown relationships. To that end, some "known", as well as some "unknown" relationships were embedded in the environmental definitions. The method by which these were embedded will be shown in the discussion of the simulation itself. In this hypothetical scenario the important previously unknown discoverable is as follows: it is thought to be the case that Shrieking Eels and Battletoads never coexist due to competitive exclusion. The scenario environment was structured such that they *do coexist*, in an as-yet unknown circumstance. Analysis of the dialectical structure should reveal this previously unknown relationship.

3.1 Guidelines

The hidden factor unknown to the operator is that, in the presence of *two* food species, Shrieking Eels focus on one food, which they prefer, leaving the other available for Battletoads. This very specific condition, which hinges on two variables, is the unknown discoverable which allows the two species to exist without competitive exclusion. Since there are 16 rooms and a multitude of seed variables per room, it was important to lay out the rules of the simulation in a methodical manner.

These guidelines were used as an aid to inform seed values which were loaded into the simulation software itself. The room colors are only present to the define the simulation. It gives the simulator a convenient, repeatable way to define the map in any custom configuration, and different properties for each room of the map.

3.1.1 Room guidelines

The colors below map to the 16 colors available in a standard 4 bits per pixel bitmap file.[12] Two are reserved leaving 14 colors available for room definitions. The color of each room determines its physical properties in the simulation. This separates the tasks of laying out the topology and defining conditions.

Black: represents walls or boundaries between rooms/caverns.

White: represents the position of the robot.

Green: The ”magic” environment where food 5 grows which allows coexistence of species 1,5.

Red: can grow plants, but is too hot for fish life, minimum pressure (at surface).

Yellow: has lots of light to support heavy plant life, and is cool enough for fish life.

Fuschia: has lots of light for plant life, but is too cold for some plants/fish.

Blue-Green: low light with lots of fish.

Brown: dark but warm.

Khaki: nothing lives here and pressure is high.

Navy: most species can live here.

Plum: lots of plants but is cold.

Gray: Maximum Pressure.

Silver: Temperature and pressure high, food high.

Lime: Very similar to green, but lacking the presence of food 5.

Blue: like navy, but lighter.

Cyan: like blue, but even lighter.

3.1.2 Flora Guidelines

Food items such as plants or plankton that live with the simulated cave system dare designated flora. Flora are designated by their color. The following rules define the environmental preferences for each species of aquatic flora.

1. Black food prefers cool dark water (but not too cool).
2. Pink food likes warm bright water.
3. Chartreuse food likes bright cool water.
4. Indigo food prefers hot dark water.
5. Periwinkle food, only occurs in green rooms.

3.1.3 Fauna Guidelines

Aquatic animals that dwell in the cave system and consume simulated flora are designated fauna. Environmental and forage preferences for each fictional aquatic fauna were defined as shown here. This is done for consistency in the simulation and to embed the aforementioned hidden condition.

1. Shrieking Eels and Decapodians frequently coexist.
2. Decapodians like blue rooms.
3. Jaguar sharks are rare and elusive.
4. Furry Trout only like cool dark water.
5. Battletoads and shrieking eels both eat the same food. However shrieking eels are more aggressive and consume the available food first starving out battletoads. This competitive exclusion means that the two species are thought to never coexist.

3.2 Translating guidelines into seed values

Having been established, these guidelines where then used to define mean values for all variables. These mean values will then be loaded into the simulation and used to generate randomized sensor readings.

3.2.1 Environmental variables

Since this scenario is meant to loosely model an underwater cave, pressure was given values ranging from 0 kPa (surface) to 130 kPa (14 meters depth). Temperature varies between 0 degrees Celcius and 37 deg C. Illuminance was given values ranging from 0 lux (darkness) to 130 lux (sunlight at surface). Each color of room was assigned mean values for these variables according to the scenario guidelines outlined above. They were also assigned standard deviations of varying amounts. Mean and standard deviations for all room colors can be seen in Table 4.1. While all of these values are are arbitrary in nature, an attempt was made to keep them within the range of values that would be expected from a real world experiment.

3.2.2 Simulated aquatic Species

In this scenario the simulated robot is using some form of image recognition to identify the presence of aquatic species in each location on the map. Real world image recognition inherently contains a degree of uncertainty, which the simulated image recognition system will incorporate. It will do this by sensing the presence of a species to a certain confidence level. That confidence will be a number from 0 to 1 with 1 being 100% confident. Confidence values will be randomly generated but within ranges dictated by the scenario guidelines. To achieve this a mean confidence value for each species was input per chamber color. This mean confidence value will be stored for use by the simulation. The objects recognized by the robot would be species of flora and fauna that live in the cave system seen in Table 4.2 and Table 4.3. For simplicity, standard deviations were limited to one for all flora and one for all fauna.

We live in a world where there is more and more information, and less and less meaning.

Jean Baudrillard

4 Python Simulation

All of the attributes of a physical robot necessary for the visualization of the dialectic structure can be simulated in software. This provides a a valuable testbed for the information system without requiring that a fully functional ROV with support infrastructure be constructed.

The package used for the fuzzification engine as described in section 1.3, Sci-kit fuzzy,[24] is written in the Python[18] programming language. For ease of interoperability the simulation which generates the data fed into the fuzzification engine is written in Python as well.

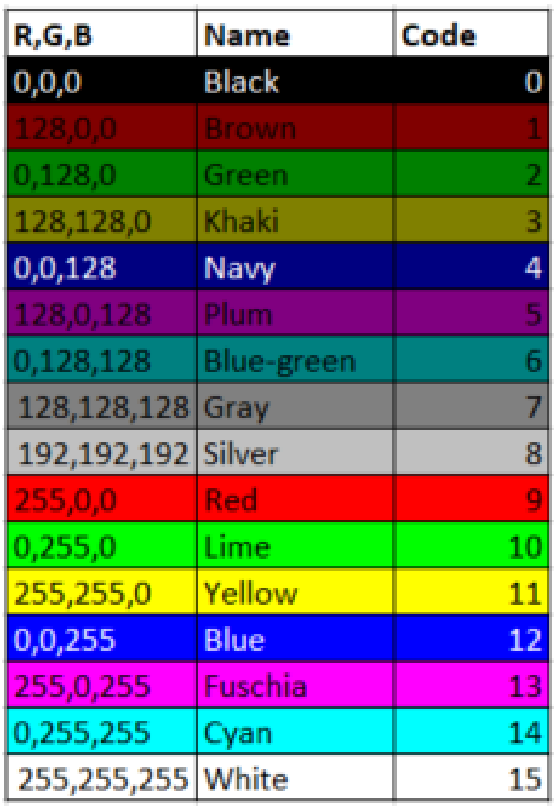
4.1 The map

In the simulation, a single robot will navigate a series of rooms. Each room will have different properties, to be defined in the simulation. A method was required to draw a map topology, defining boundaries, each room, and passageways between. Once drawn, a mechanism is required to programmatically import the map and represent it in a way that can be processed. Python Imaging Library[10] was selected for this purpose in its current implementation, Pillow.[2] By using a combination of Pillow and Numpy[21], bitmap files can be read, and represented as arrays of integers. The rows and columns of the bitmap file will be isomorphic to rows and columns in the array. As implemented, the simulation supports up to 16 colors. When imported into the array colors are represented as integers 0-15. Black (0) is used to represent walls. White(15) represents the robot's position in visualizations. The remaining 14 colors represent rooms or caverns with different properties. Room colors and their corresponding numerical numpy representation are shown in Figure 4.1. One advantage of this method is that it allows the use of existing image editing programs to design the map that the robot will navigate. No further software development is necessary.

4.1.1 Navigation

The simulated robot uses a simple "left wall following" algorithm, the rules for which are as follows:

1. If left is open, then turn left and step.



R,G,B	Name	Code
0,0,0	Black	0
128,0,0	Brown	1
0,128,0	Green	2
128,128,0	Khaki	3
0,0,128	Navy	4
128,0,128	Plum	5
0,128,128	Blue-green	6
128,128,128	Gray	7
192,192,192	Silver	8
255,0,0	Red	9
0,255,0	Lime	10
255,255,0	Yellow	11
0,0,255	Blue	12
255,0,255	Fuschia	13
0,255,255	Cyan	14
255,255,255	White	15

Figure 4.1: Bitmap to Numpy color mappings

2. Otherwise, if forward is open go forward.

3. Otherwise, if right is open turn right and step.

4. Otherwise turn around.

This algorithm is roughly analogous the the simple, but effective, technique a human could use to solve a maze, i.e. place one's left hand on a wall and keep it there until an exit is found. While this algorithm is incapable of solving every possible maze thrown at it, it is sufficient for the purposes of this simulation. The map must be designed in an appropriate manner within the context of gathering sensor data from multiple chambers. For example exits must be at the outside edge, and care must be taken to not make certain rooms unreachable. See A for the programmatic version of this navigation algorithm.

4.2 Example map

The primary map used for the exemplar instantiation is shown in Figure 4.2. This map was drawn to work within the aforementioned limitations of the wall following algorithm. It has 1 room corresponding to each available color. Care was taken when drawing the map to make each room accessible and to avoid loops that may appear when using the left turn first algorithm. The color of each room will be used to seed the random environment measurements simulated by the program.

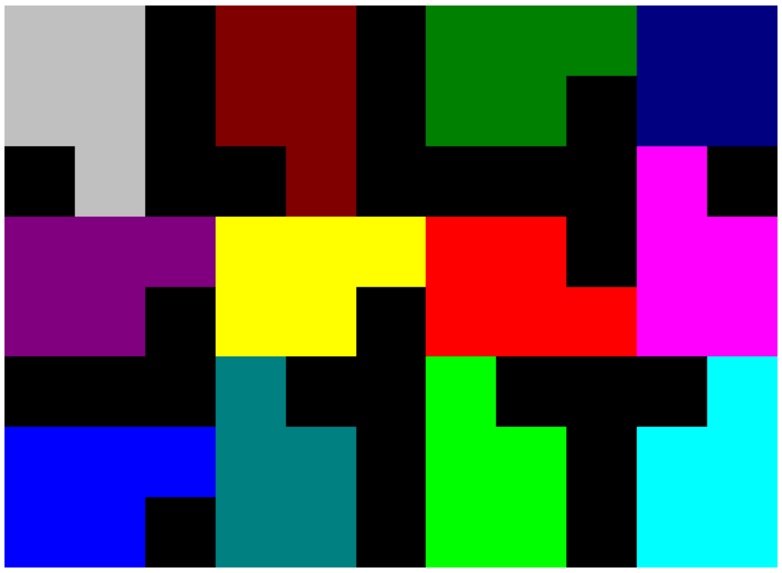


Figure 4.2: Map used in simulations

4.3 Environmental Sensors

Mean μ [*] and standard deviation σ values were selected for temperature, pressure, and illumination variables for each color chamber. Before every step, the simulation first checks the color of the room, then generates a random temperature, pressure, and illumination. The simulated robot then logs these as environmental sensor readings along with its step number and current position. While steps are used in the simulation, a real data logging robot would most likely be using timestamps.

Mean and standard deviation values for each color were stored in a comma separated value file for ease of editing. The Python package pandas[11] was utilized to retrieve the specified values into the simulation from the *.csv file. Environmental sensor seed values for each room are presented in table Table 4.1 The simulated environmental values were then processed using SciKit-Fuzzy[24] to determine membership values in each of three fuzzy sets for each variable: High, Mid, and Low. Visualization of the membership functions used can be seen in Figure 4.3. Figure 4.3

4.4 Simulated Image Recognition

In many scenarios, the robot will be using image recognition. The simulated image recognition system returns a confidence value that it has recognized a given object. This is achieved by assigning a mean confidence value to each color of chamber then generating a random confidence reading based on the mean with a narrow standard deviation range. For the cave exploration scenario the objects recognized by the robot would be species of food items such as plants or plankton designated

[*] The symbol μ was previously used to indicate a membership function, here it represents statistical mean

Table 4.1: Environmental Sensor Seed Values

color code	room color	μ_T	σ_T	μ_P	σ_P	μ_L	σ_L
0	Black	0	0	0	0	0	0
1	Brown	20	1	20	3	20	8
2	Green	15	2	40	3	60	1
3	Khaki	18	5	110	3	100	1
4	Navy	10	7	100	3	70	4
5	Plum	7	4	70	3	90	1
6	Blue-green	10	1.5	50	3	10	2
7	Gray	3	1	130	3	30	6
8	Silver	18	3	120	3	40	9
9	Red	37	3	0	3	120	7
10	Lime	16	2	35	3	50	1
11	Yellow	22	2	10	3	130	10
12	Blue	11	3	90	3	80	6
13	Fuchsia	6	4	60	3	110	4
14	Cyan	12	1	80	3	90	1
15	White	0	0	0	0	0	0

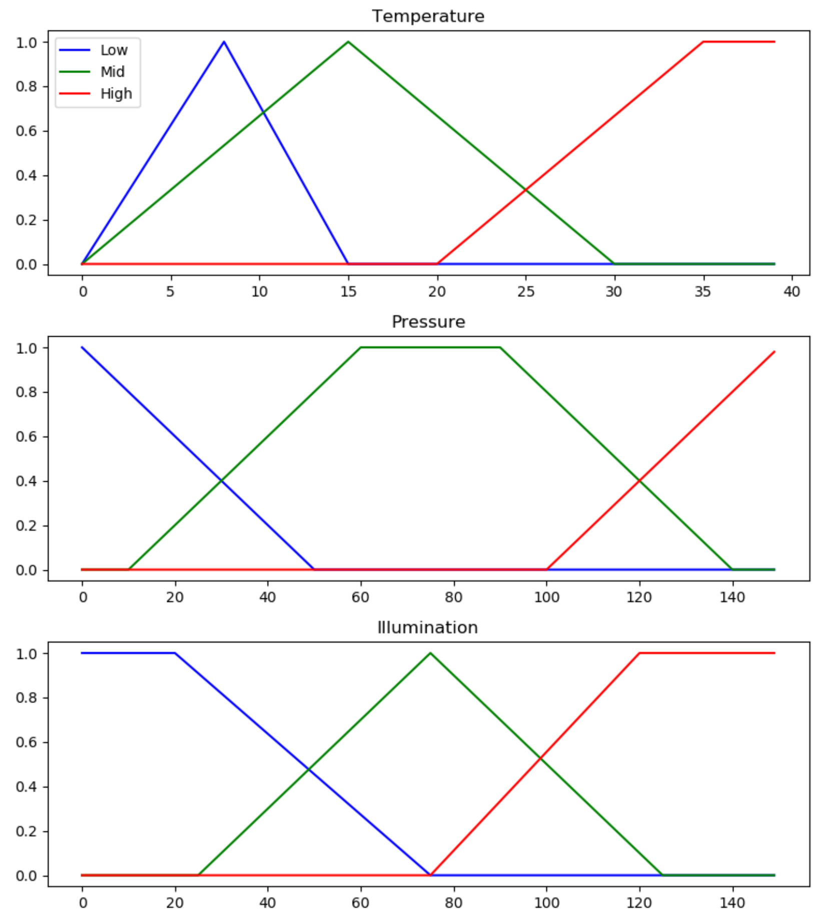


Figure 4.3: Environmental Variable Fuzzy Membership Functions

fauna and species of aquatic animals that consume the food designated fauna. A mean confidence value for each species was assigned to each room and appended to the simulation parameters csv file. Predator species are named as shown in Table 4.2. Food species are identified by their color and their median confidence values by room color can be seen in Table 4.3. For simplicity standard deviations were limited to one for all flora and one for all fauna. Values used were coded into the simulation, as seen in Appendix A.

Table 4.2: Mean detection confidence for aquatic fauna in simulation environment (μ)

Color code	Room color	Shrieking Eel	Decapodian	Jaguar Shark	Furry Trout	Battletoad
0	Black	0	0	0	0	0
1	Brown	0	0	0	0	0
2	Green	0.9	0.6	0	0	0.9
3	Khaki	0	0	0	0	0
4	Navy	0.9	0.9	0	0.7	0
5	Plum	0	0	0	0.7	0
6	Blue-green	0.9	0.8	0	0.9	0
7	Gray	0	0	0	0	0
8	Silver	0	0	0.1	0	0.9
9	Red	0	0	0	0	0
10	Lime	0	0	0	0	0.9
11	Yellow	0	0	0	0	0.9
12	Blue	0.8	0.9	0	0.8	0
13	Fuchsia	0	0	0	0	0
14	Cyan	0.8	0.9	0	0	0
15	White	0	0	0	0	0

Table 4.3: Mean detection confidence for aquatic flora in simulation environment (μ)

Color code	Room color	Onyx	Pink	Chartreuse	Indigo	Periwinkle
0	Black	0	0	0	0	0
1	Brown	0	0	0	0	0
2	Green	0	0.9	0	0	0.9
3	Khaki	0	0	0	0	0
4	Navy	0.5	0.9	0.7	0	0
5	Plum	0.5	0.7	0.9	0	0
6	Blue-green	0.9	0	0	0	0
7	Gray	0	0	0	0	0
8	Silver	0	0.5	0	0	0
9	Red	0	0	0	0	0
10	Lime	0	0.6	0	0	0
11	Yellow	0	1	0	0	0
12	Blue	0.7	0.8	0.9	0	0
13	Fuchsia	0	0	0.9	0	0.8
14	Cyan	0.6	0.9	0.9	0	0
15	White	0	0	0	0	0

4.5 Fuzzification

As mentioned, the simulation used the scikit-fuzzy package as the fuzzy logic engine. Food and predator species are randomly generated about median certainty values which range from 0 to 1. The certainties are therefore effectively already a fuzzy membership value which denotes membership in a category that represents the presence of the corresponding species. Environmental sensors however are generating empirical data that must be mapped to membership in each fuzzy category. Three fuzzy categories were defined for each sensor: high, medium and low. Triangular or trapezoidal membership functions were used to define each of the nine categories. At each step, the simulation evaluated the empirical sensor data for each of three sensors against the membership function for each of three fuzzy groups. It then logged the membership results to the sensor output database for later use by the visualization engine.

4.6 Animation

An accompanying program was written to graphically display the navigation of the robot. It functions by reading the simulation data from the database and drawing the map with the current position of the robot at each step. It uses the same packages as the simulation program for the reading of map files and the database. It also adds the open source Python programming library pygame[16] package for the animation of the robot motion. The pygame program creates a graphical window with the map file as a background. It then draws an image representing the robot at the starting position of the robot. It repeats this for each step that to robot took using the x and y coordinates from the sensor data table to locate the robot on the map. It also displays all measurements taken by the simulated robot at each position. Of note is that, due to differences in the way Numpy and pygame read the bitmap file into rows and columns, the x and y values taken from the simulation actually had to be swapped for the animation. This animation proved to be indispensable for visualization of the robot navigation both for debugging the navigation algorithm and for ensuring that logged measurements were within expected values throughout the simulation.

Figure 4.4 shows snapshots of the animation in progress. The robot is shown in its starting position, facing south toward the bottom of the map. A move to the left is available, so according to rule 1, the robot will take it. The next image indicates that the robot has selected this choice. From this position the robot is heading east, or to the right on the map. A move to the left is not available, neither is a move forward. So according to rule 3, the robot will turn to the right and head south. At step 3, the robot evaluates the rules in order again. This time rule 2 is the first to pass so the robot continues straight to the south. The robot continues in this manner until it reaches the total number of steps allocated by the operator.

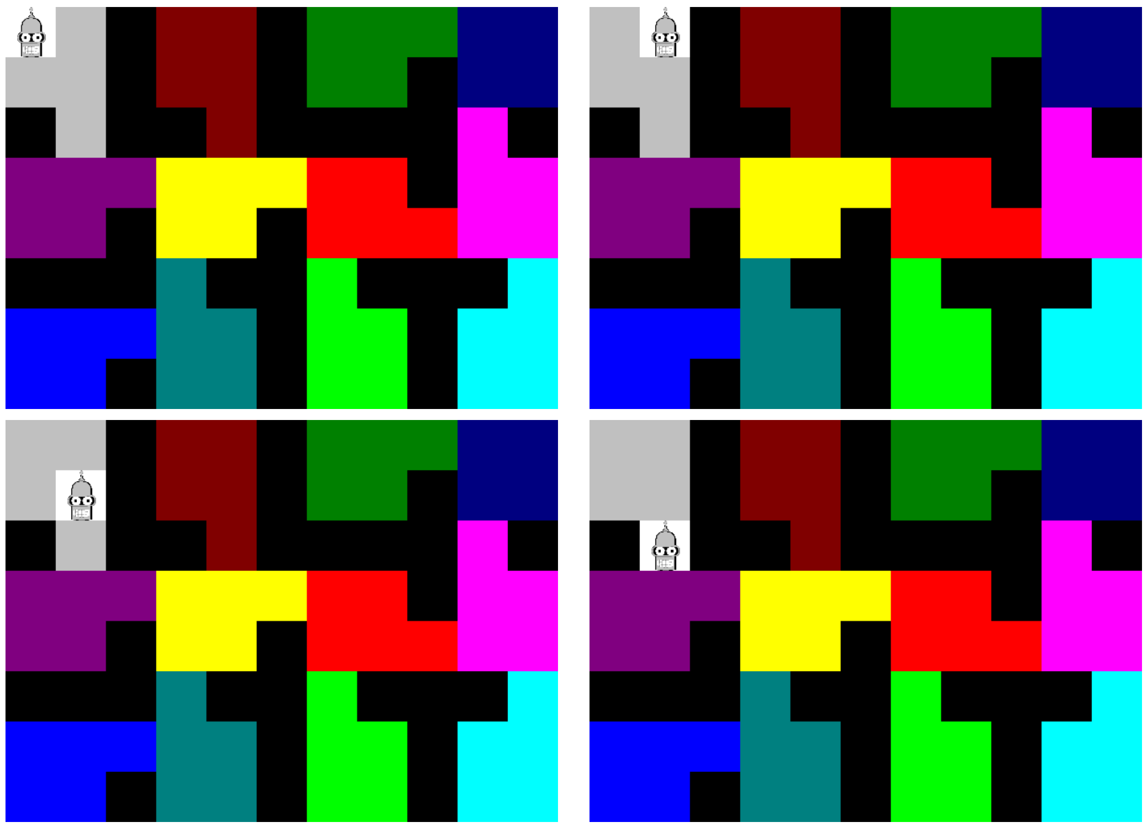


Figure 4.4: Robot Navigation Animation Snapshots

A lot of information isn't a substitute for good information.

Greg Everett

5 Storage and Retrieval

Once the data has been processed, and fuzzy membership values for each category assigned to each data point, the results are stored for later processing by the presentation engine. The final presentation of the data will be a simplified easy to interpret set of graphs, however the original data must be retained. This also enables further processing with other tools, should the operator choose to perform a more in-depth analysis.

5.1 Requirements

The storage and retrieval system has the following requirements. It should be inter-operable with both the categorization and visualization systems, it must retain the original data, as well as all fuzzy membership information, it must scale to accommodate large streams of data, and it should be low-cost and well-documented. It should also easily integrate with existing data collection platforms. Specifically, it should be accessible for writing by the categorization engine and reading by the dialectical retrieval subsystem.

5.2 PostgreSQL

PostgreSQL[20] is a free and open-source database software based on the Structured Query Language[5] (SQL) standard. It has many pre-existing drivers written to support multiple programming languages that are likely to be used in data acquisition and presentation systems such as Python and Ruby. Python provides a pre-written driver for interfacing PostgreSQL databases known as Psycopg2.[3] Psycopg2 is fully DB API 2.0 compliant and provides all routines needed for this project to store the collated and categorized sensor data. As the data streams are processed and categorized, Psycopg2's built in routines can be used to connect to the database and write data. Importing Psycopg2 into the simulation software and coupling it with the PostgreSQL creates a system that meets all of the requirements laid out in section 5.1 Specific code leveraging Psychopg2 for database access can be found in Appendix A.

5.2.1 The experimental PostgreSQL structure

A table dedicated to the experiment was created in order to store sensor and set membership data. One column in the table represents one variable: either location or sensor data, or a corresponding fuzzy set membership value. An example table from the instantiation shown later can be seen in Appendix C. Data is written to the database row by row in near real time as it is gathered and processed. PostgreSQL is robust and scalable enough to handle continuous streams of data from hundreds of robots simultaneously when written in this fashion. Once stored in this manner, the data can be retained indefinitely. PostgreSQL has an ecosystem of support software for optimization, backup and recovery which can be leveraged without the need to develop new tools. Being stored in this fashion, the data can be retrieved by the visualization engine for batch processing at any time. There is no need for the dialectic algorithm to attempt to keep up with the flood of data coming in from robots in the field. It can be stored and only required elements processed on an on demand basis as the operator navigates the dialectic architecture.

5.3 Reading from PostgreSQL with Ruby

Because the fuzzy dialectic algorithm and the presentation engine were written in Ruby, a method to read the stored data into the Ruby language was required. Like psycopg2 for Python, the Ruby[25] language also has a software module written specifically for accessing postgreSQL. In the Ruby language, these modules are known as gems and the gem used to access stored sensor and set membership data is known as Sequel.[6] For processing by the dialectic algorithm the database does not need to be modified in any way, only read.

The highest endeavor of the mind, and the highest virtue,

is to understand things by intuition.

Spinoza

6

Resultant Organic Hierarchy

6.1 Navigation

The aforementioned scenario was simulated and analyzed by the dialectic algorithm. The results of this analysis are presented to the human partner as a web interface. Links corresponding to categories are available at the top of the page. Below in Figure 6.1 is displayed three plots containing each of the three environmental sensor variables plotted against each other (subsection 2.3.1). The human partner can select categories to see their fuzzy group intersection. When this is done, the plots update. Each dot on a plot corresponds to a data point gathered by the simulated robot and its opacity(subsection 2.3.1) indicates its degree of membership in the intersection of all selected fuzzy groups listed at the top of the screen in green. atoms for species identification were also thresholded at a 50% confidence value.

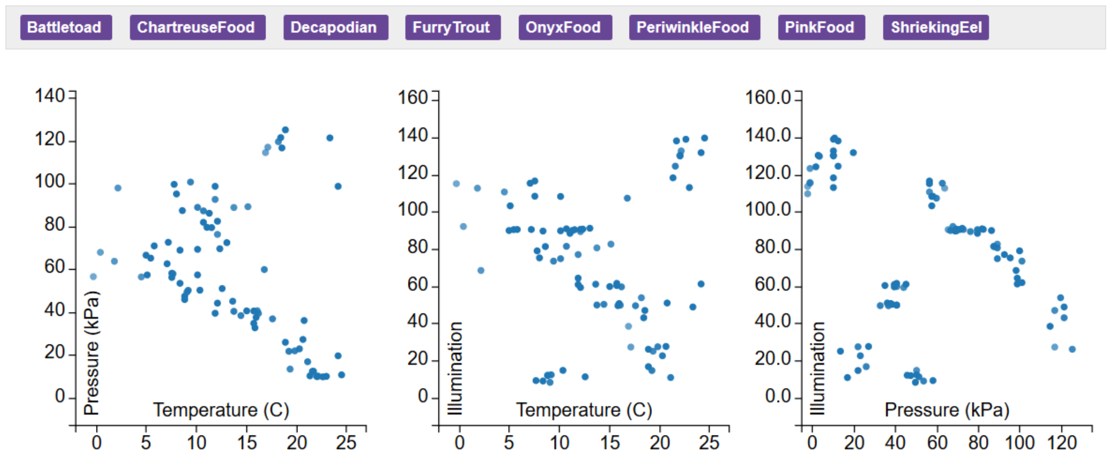


Figure 6.1: Instance of Dialectica Universal Union node

6.1.1 Union Node

The visualization of the dialectic structure begins at the union node in Figure 6.1. The union node shows the top level membership of all data points gathered. The interface displays each of the three quantitative environmental variables plotted against each other. Across the top, all nodes in the structure that are “visible” as described in subsection 2.1.1, from the current node are displayed and selectable. At this point the human partner has the option to browse any of the “visible” nodes (that is, follow visible (vs) relations/edges).

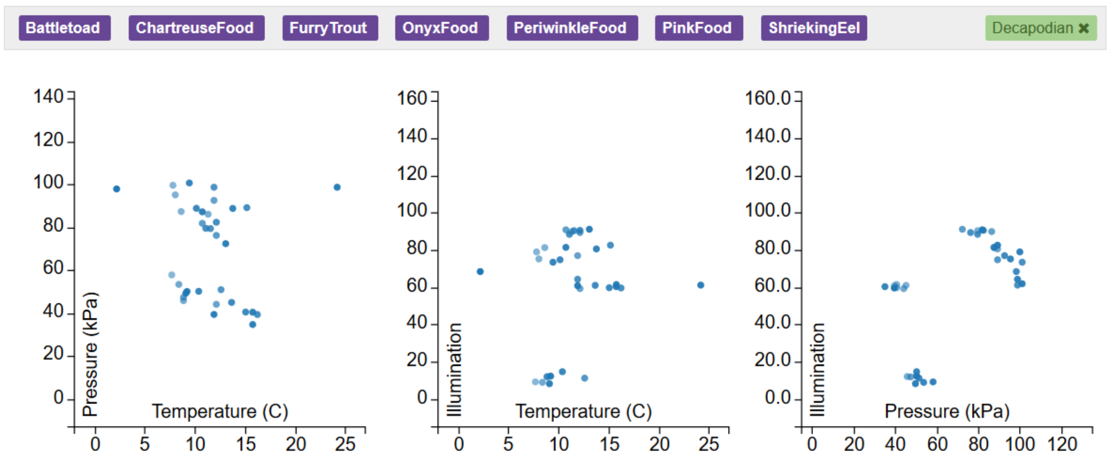


Figure 6.2: *Decapodian*

6.1.2 Level 1 Fauna

Designing this simulation uncovered some new insights into scoping the final display. In some interfaces, such as this one, it is preferable to *graph* quantitative variables and *navigate* qualitative ones. For clarity, the fuzzy sets for pressure, temperature and illuminance weren't read into the dialectic algorithm since they are already plotted. There is no practical reason that navigation cannot start anywhere or jump to anywhere in the structure. However at any node only a limited set is shown consisting of those nodes that are visible one level beneath the current node. (see Figure 2.1 and Figure 6.15) The human partner decides to quickly explore the fauna at level 1 by browsing down to level to and back upsubsection 2.3.1 to level 1 for each aquatic fauna.

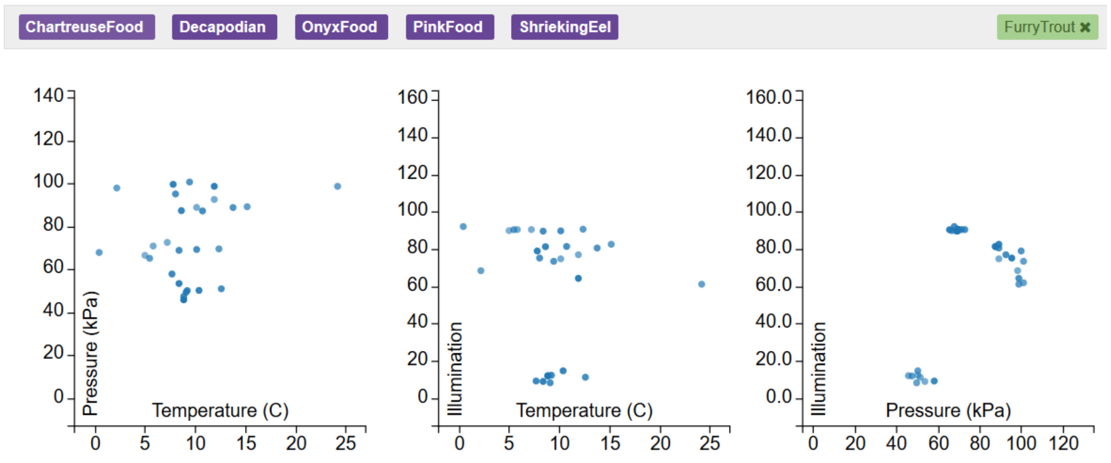


Figure 6.3: *FurryTrout*

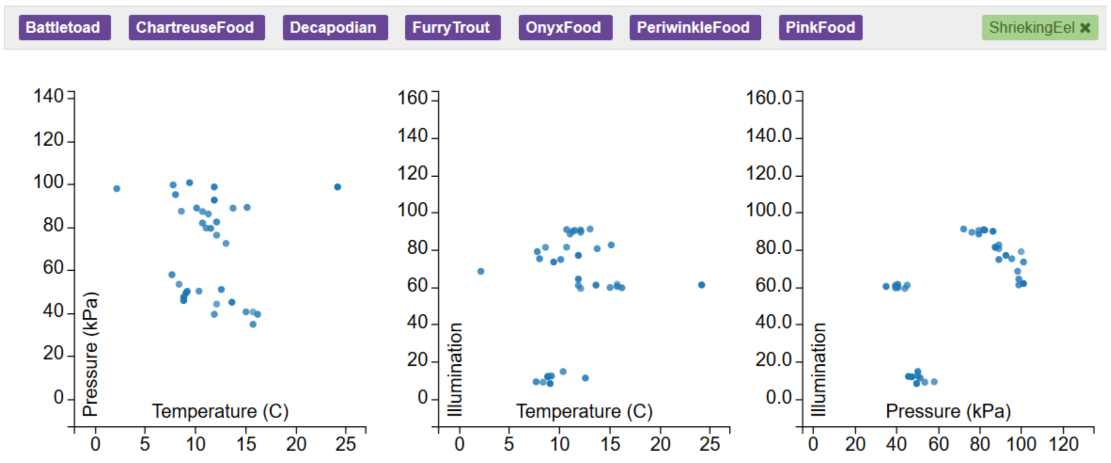


Figure 6.4: *ShriekingEel*

6.1.3 POTENTIAL DISCOVERY

Shrieking Eels appear to have a wide tolerance for Temperatures and Illumination values as expected from subsection 3.1.3. Here while browsing the level 1 set *ShriekingEel* in Figure 6.4 the human partner immediately notices that *Battletoad* is present as a visible node in the dialectic structure. That should not be the case, as the two species are thought to never coexist due to their competitive exclusion in the presence of food. The human partner clicks the *ShriekingEel* set and traverses down to node *Battletoad* ∩ *ShriekingEel* as shown in Figure 6.5.

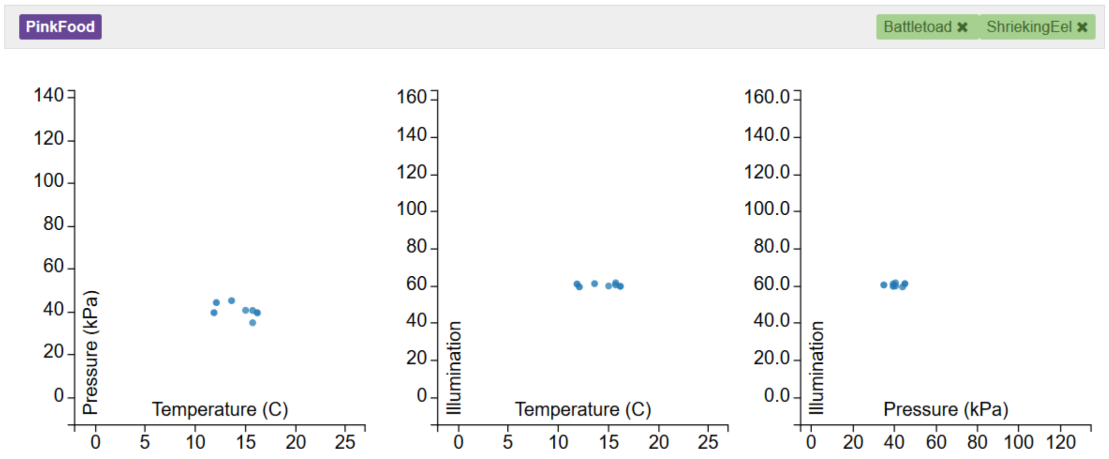


Figure 6.5: *Battletoad* ∩ *ShriekingEel*

6.1.4 Exploration

By observation of the groupings of quantitative data points *Temperature*, *Pressure*, and *Illuminance* it is clear that this is taking place in one specific chamber in the scenario. What is unclear is what property of that room is enabling this previously unknown relationship. Seeing that *PinkFood* is a visible node in the flow, The human partner selects it in order to see if the presence of *PinkFood* stimulates the coexistence of the two species.

Now at layer 3 of the structure we see two more sets that are subsets of *Battletoad*∩*ShriekingEel*∩ *PinkFood*, namely *Decapodian* and *PeriwinkleFood*. Per the scenario rules, Decapodians are known to frequently coexist with Shrieking Eels and Figure 6.7 supports this hypothesis. These species exist together but the low opacity of the data points in the plots show low membership values in the

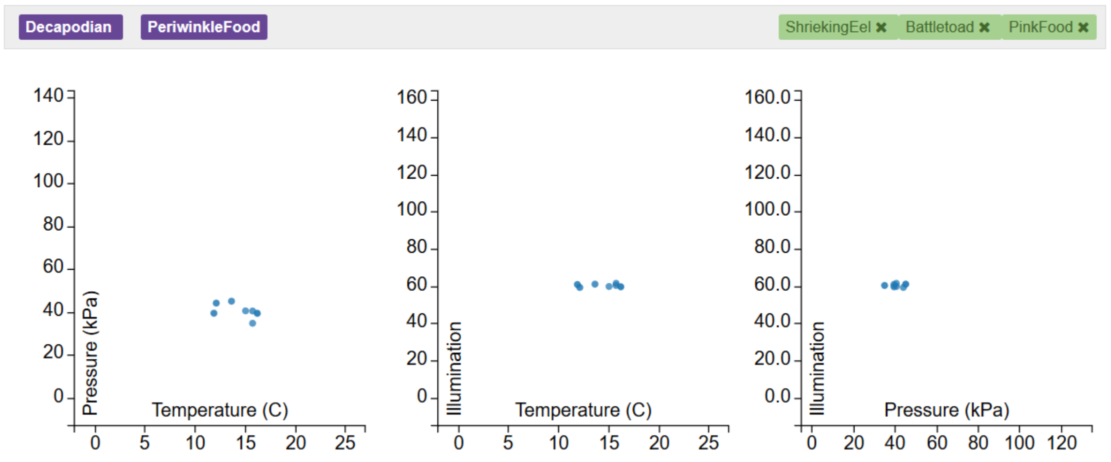


Figure 6.6: *Battletoad* ∩ *PinkFood* ∩ *ShriekingEel*

current set. In fact the opacity of the majority of points is in the 50 − 60% range which is barely above the threshold set for display by the dialectic algorithm. Decapodians are then clearly not an influencing factor in this new discovery.

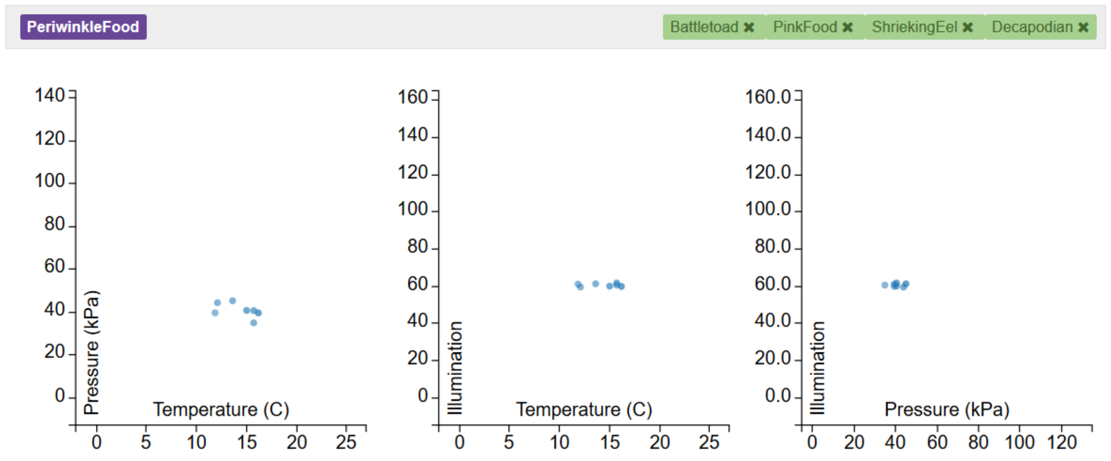


Figure 6.7: *Battletoad* ∩ *Decapodian* ∩ *PinkFood* ∩ *ShriekingEel*

The intersection of all five sets: *Battletoad*, *PinkFood*, *ShriekingEel*, *Decapodian*, and *PeriwinkleFood* shows an even lower correlation. In fact there is only one very low opacity data point. One data point does not a correlation make. So the human partner again browses up (subsection 2.3.1) in the structure to remove categories. [*]

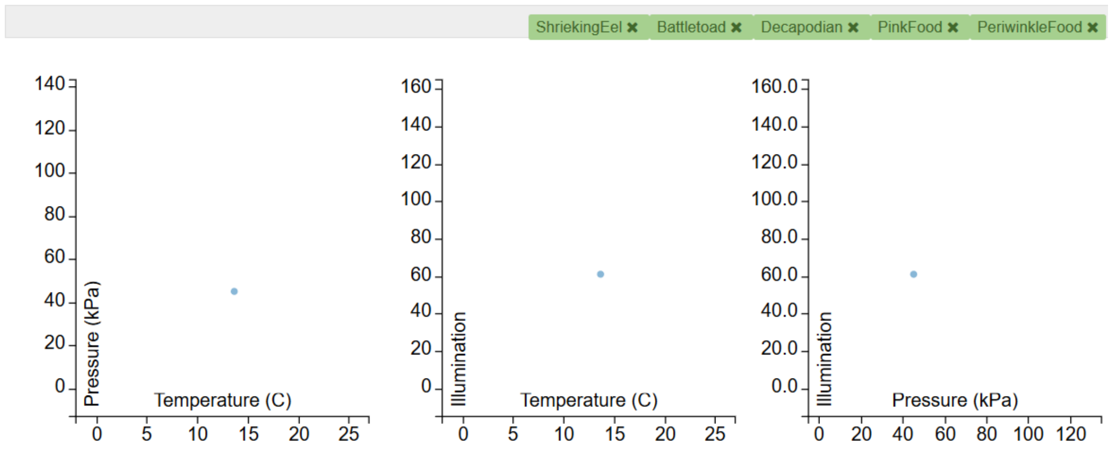


Figure 6.8: *Battletoad* ∩ *Decapodian* ∩ *PeriwinkleFood* ∩ *PinkFood* ∩ *ShriekingEel*

Navigating "up" in the structure to *Battletoad* ∩ *PinkFood* ∩ *PeriwinkleFood* ∩ *ShriekingEel* reveals a very strong correlation in the data. These points have noticeably higher opacity.

[*]This browsing up is an attempt by the human partner to locate an *antithesis flow* to the original *thesis* flow (subsection 2.3.1) that the two species are always competitively exclusive.

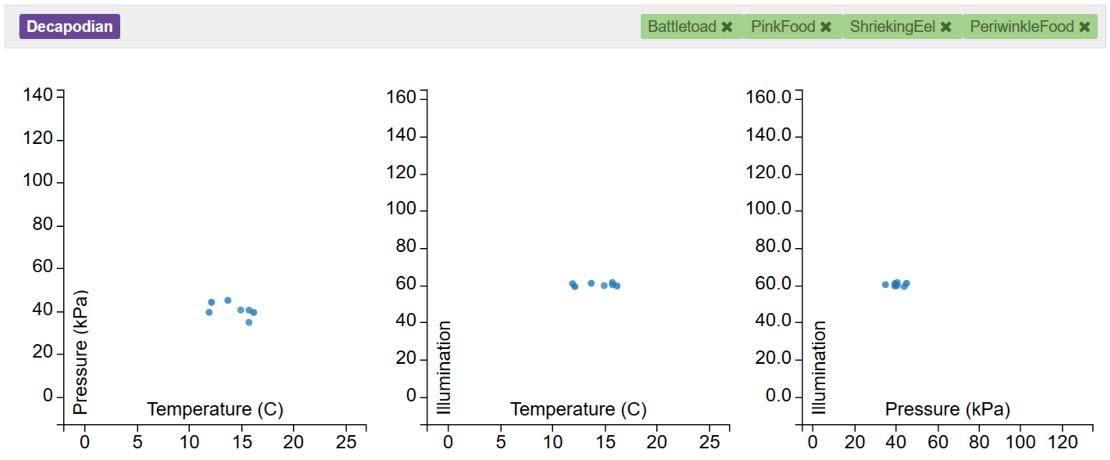


Figure 6.9: *Battletoad* ∩ *PinkFood* ∩ *PeriwinkleFood* ∩ *ShriekingEel*

This can be confirmed by browsing up a level again to see if the correlation remains strong or is reduced (Figure 6.10).

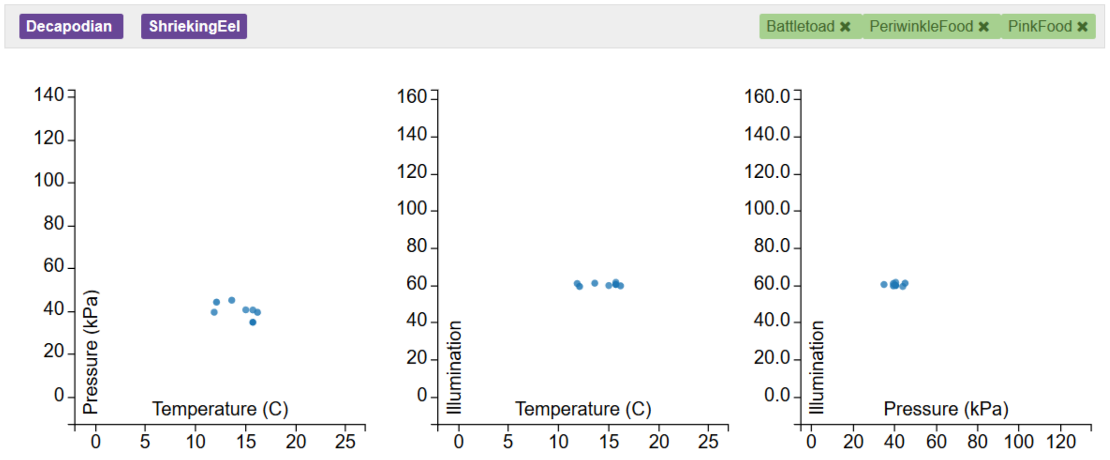


Figure 6.10: *Battletoad* ∩ *PeriwinkleFood* ∩ *PinkFood*

Figure 6.11 shows that *PinkFood* occurs throughout the map, and cannot be the sole factor that influences the coexistence of our two presumably incompatible species.

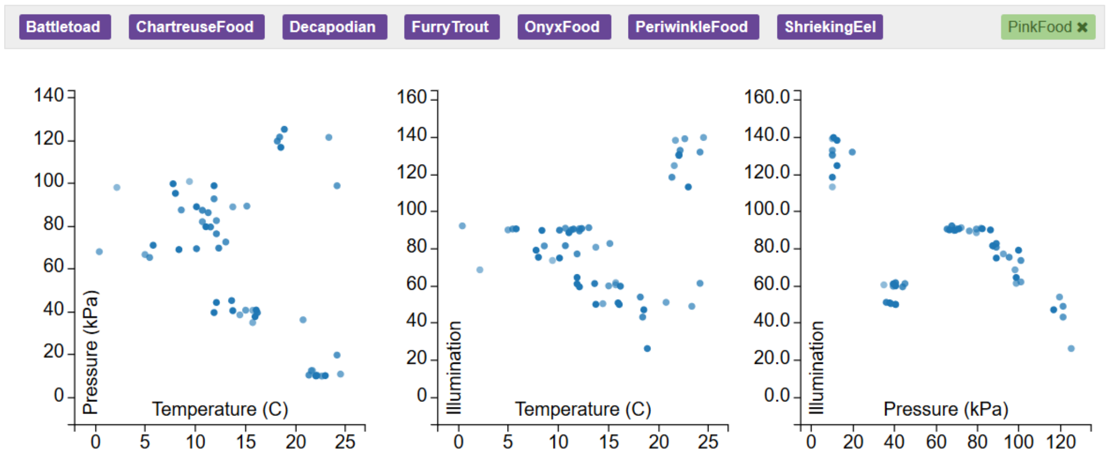


Figure 6.11: *PinkFood*

Likewise, *PeriwinkleFood* by itself cannot be the lone factor (Figure 6.12).

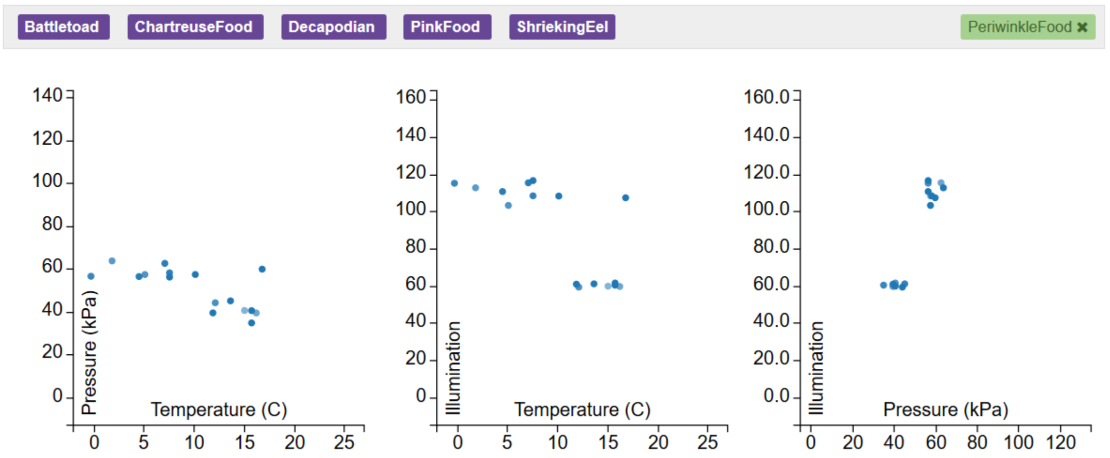


Figure 6.12: *PeriwinkleFood*

Notably, once these two categories are selected, the addition of another category: either *ShriekingEel* of *Battletoad* causes almost no change in the quantitative plots and the data point opacity.

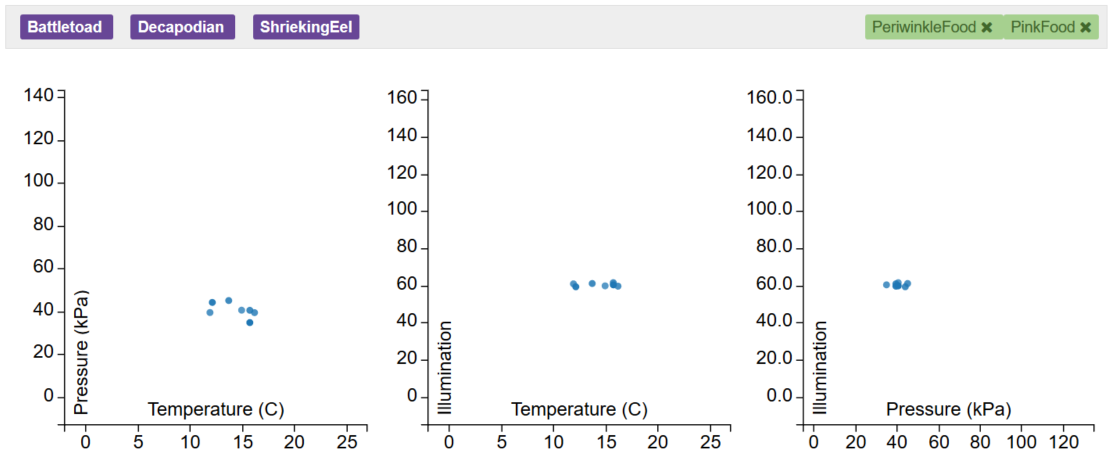


Figure 6.13: *PeriwinkleFood* ∩ *PinkFood*

This suggests a causal relationship between the presence of the two foods and the coexistence of the two species. Figure 6.10 Figure 6.14.

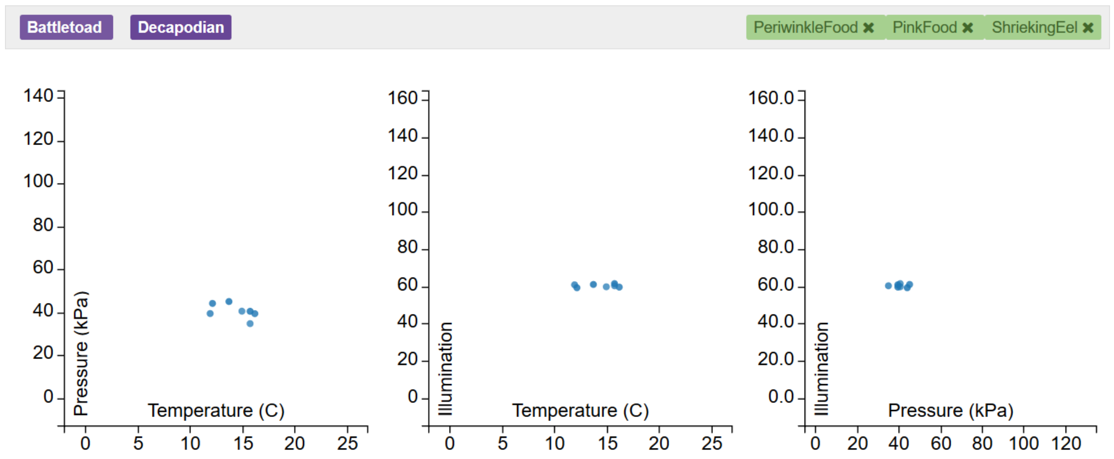


Figure 6.14: *PeriwinkleFood* ∩ *PinkFood* ∩ *ShriekingEel*

It is only in the presence of *both* foods that we see the intersection of sets *Battletoad*∩*ShriekingEel*.

6.2 Discovered Structure

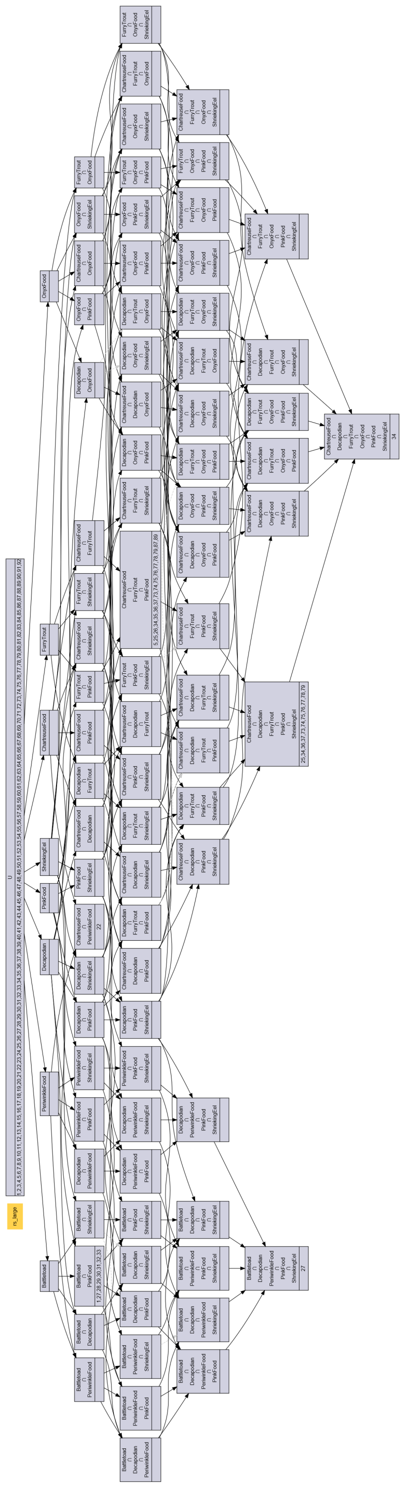


Figure 6.15: Full dialectic structure resulting from simulation

7
Conclusion

To review, the system is broken into modules of simulation, categorization, storage, retrieval, and analysis. Sensor data can be processed into fuzzy sets by a fuzzy logic engine, written to a database, read by the fuzzy dialectical algorithm, and displayed by a user interface, as shown in Figure 3. This instantiation utilized Python for the sensor simulation and fuzzy categorzation engine, PostgreSQL for the database, and Ruby for the presentation engine. The modular organization of the information system separates categorization from analysis and presentation enhancing the resiliency and flexibility of the system. In some instances, once continuous streams of data are categorized and stored, they are no longer time-sensitive, and can be batch-processed by the fuzzy dialectical algorithm. Each module can be run on a different physical system at a different geographical location, depending on the requirements of the individual implementation.

The human-robot interface framework that emerges fuses information structure, category, sequence, and dialectic. A simple interface represents complex information that contains elements both quantitative and qualitative in nature. The integration of quantitative and qualitative information in this straight-forward, intuitive interface is designed to enhance information integration and amplify human intelligence. This information system improves on traditional architectures in both speed and ease of use.

Further work with this information system could be into the enhancement of larger human-robot teams such as swarms. By leveraging the modular nature of the architecture swarms of robots in teams could feed into a user interface managed by a single human team-member. This technology also has applications outside the field of robotics which may warrant investigation. The discovery of structure innate to the data could make it possible to graph networks of systems in novel ways. For example, Content-addressable memory tables on network switches from differing vendors containing MAC(Media Access Control) Addresses could be imported and parsed with the dialectic algorithm. The result would be a vendor agnostic, dynamically generated diagram containing the location of every active device on the computer network. Another thusfar unexplored application is in monitoring and reporting of distributed systems. This interface may prove valuable in log analysis in a variety of use cases such as security event management (SEM) or the oversight of smart manufacturing facilities.

These results show that it is possible to leverage the fuzzy dialectic architecture to enable discovery of unknown relations. This ability can be enhanced by expanding the capabilities of the information system. Information types, such as video, text-based information in paragraph form, images, etc. could be incorporated into the fuzzy dialectic architecture. The information system could also be improved through usability enhancements to the presentation engine. A human partner could be given the ability to type in a category and immediately jump to that node in the structure, for example. A module in the user interface for the importation and integration of text based data is another area for improvement.

A

Python Simulation Code Listing

```
#!/usr/bin/env python
#  -*- coding: utf-8 -*-
"""Input a 16 color bitmap file and read into an array
Robot size is 1 pixel
simulate starting at pixel 0,0
move the robot step by step, taking a reading at each step

Example:
    To view which numbers get assigned to which colors

    $ python sim.py ColorPalette.bmp
```

```
If run with no arguments default to filename map.bmp
"""
# TODO CREATE ANIMATION OF ROBOT MOVING AROUND MAP
# TODO SEED RANDOM NUMBER GENERATOR
import sys
import numpy as np
import PIL.Image as Image
import psycopg2
import skfuzzy as fuzz
import matplotlib.pyplot as plt
import pandas as pd

# constants
π = np.pi
σ_α = 0.1
σ_φ = 0.15
robotSteps = 92
seedFile = 'SeedValues.csv'
np.random.seed(0)

if __name__ == '__main__':
    try:
        filename = sys.argv[1]
    except IndexError:
        filename = 'map.bmp'
"""Import Bitmap file.
Colors: 0=black (wall), 15= white(robot) 1-14=room colors
"""
colors = pd.read_csv(seedFile, sep='\s*,', index_col='code', encoding='UTF-8',
 ↪   engine='python')
print(colors)
# define fuzzy universe and membership functions
# TODO: fuzzy stuff hardset for now, improve by setting ranges programmatically
u_T = np.arange(0, 40, 1)
u_P = np.arange(0, 150, 1)
u_L = np.arange(0, 150, 1)
# Generate fuzzy membership functions
mf_T_lo = fuzz.trimf(u_T, [0, 8, 15])
mf_T_md = fuzz.trimf(u_T, [0, 15, 30])
```

```
mf_T_hi = fuzz.trapmf(u_T, [20, 35, 40, 40])
mf_P_lo = fuzz.trimf(u_L, [0, 0, 50])
mf_P_md = fuzz.trapmf(u_L, [10, 60, 90, 140])
mf_P_hi = fuzz.trimf(u_L, [100, 150, 150])
mf_L_lo = fuzz.trapmf(u_L, [0, 0, 20, 75])
mf_L_md = fuzz.trimf(u_L, [25, 75, 125])
mf_L_hi = fuzz.trapmf(u_L, [75, 120, 150, 150])

# noinspection PyUnboundLocalVariable
im = Image.open(filename)
the_map = np.array(im)
(rows, columns) = np.shape(the_map)

# Use this section to make sure the map is as expected when drawn
# mapcheck = np.empty([rows, columns], dtype=np.dtype('U10'))  # Only for visual
 ↪  verification that map imported correctly
# for row in np.arange(rows):
#     for column in np.arange(columns):
#         color = the_map[row, column]
#         mapcheck[row, column] = colors['color'][color]
# print(mapcheck)
# open database
try:
    # conn = psycopg2.connect("dbname='RoboSim' user='RoboSimFull'
     ↪  host='localhost' password='l7S4Q5mm'")
    conn = psycopg2.connect("dbname='RoboSim' user='RoboSimFull'
     ↪  host='172.16.0.130' password='l7S4Q5mm'")
except:
    print("I am unable to connect to the database")
    exit()

cur = conn.cursor()

# clear old data
cur.execute("truncate rs_large;")
# define format for database entries.  Actual data within while loop
SQL = """INSERT INTO "rs_large" (step, x, y, color_code, "Temperature",
 ↪  "Pressure", "Illumination", "ShriekingEel",
        "Decapodian", "JaguarShark", "FurryTrout", "Battletoad", "OnyxFood",
 ↪  "PinkFood", "ChartreuseFood", "IndigoFood",
        "PeriwinkleFood", "T_lo", "T_md", "T_hi","P_lo", "P_md", "P_hi", "L_lo",
 ↪  "L_md", "L_hi")
        VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s,
 ↪  %s, %s, %s, %s, %s, %s, %s, %s, %s, %s);
        """
position = np.array([0, 0], int)  # Robot starts at point 0, 0
orientation = np.array([1, 0], int)  # Pointed down first column of bitmap
i = 0
# while i < np.count_nonzero(the_map):
while i < robotSteps:
    i = i + 1
    # if(i > np.count_nonzero(the_map)):
    #     break
    # gather sensor info at current position
    color_code = int(the_map[tuple(position)])
    T = colors['σ_T'][color_code] * np.random.randn() +
     ↪  colors['μ_T'][color_code]
    P = colors['σ_P'][color_code] * np.random.randn() +
     ↪  colors['μ_P'][color_code]
    L = colors['σ_L'][color_code] * np.random.randn() +
     ↪  colors['μ_L'][color_code]
    # α1 = int(np.random.random_sample() > colors[color_code]['th_α1'])
    α1 = min(max(0, σ_α * np.random.randn() + colors['μ_α1'][color_code]), 1)  #
     ↪  Using min, max to ensure randomness
    α2 = min(max(0, σ_α * np.random.randn() + colors['μ_α2'][color_code]), 1)  #
     ↪  doesn't generate a number outside 0-1
    α3 = min(max(0, σ_α * np.random.randn() + colors['μ_α3'][color_code]), 1)
    α4 = min(max(0, σ_α * np.random.randn() + colors['μ_α4'][color_code]), 1)
    α5 = min(max(0, σ_α * np.random.randn() + colors['μ_α5'][color_code]), 1)
    φ1 = min(max(0, σ_φ * np.random.randn() + colors['μ_φ1'][color_code]), 1)
    φ2 = min(max(0, σ_φ * np.random.randn() + colors['μ_φ2'][color_code]), 1)
    φ3 = min(max(0, σ_φ * np.random.randn() + colors['μ_φ3'][color_code]), 1)
    φ4 = min(max(0, σ_φ * np.random.randn() + colors['μ_φ4'][color_code]), 1)
    φ5 = min(max(0, σ_φ * np.random.randn() + colors['μ_φ5'][color_code]), 1)
    # Find fuzzy membership values for numerical sensor data
    T_lo = fuzz.interp_membership(u_T, mf_T_lo, T)
    T_md = fuzz.interp_membership(u_T, mf_T_md, T)
    T_hi = fuzz.interp_membership(u_T, mf_T_hi, T)
    P_lo = fuzz.interp_membership(u_P, mf_P_lo, P)
    P_md = fuzz.interp_membership(u_P, mf_P_md, P)
```

```
P_hi = fuzz.interp_membership(u_P, mf_P_hi, P)
L_lo = fuzz.interp_membership(u_L, mf_L_lo, L)
L_md = fuzz.interp_membership(u_L, mf_L_md, L)
L_hi = fuzz.interp_membership(u_L, mf_L_hi, L)
# TODO: get rid of retyping with int() and use cur.mogrify() from psycopg2
line = (i, int(position[0]), int(position[1]), color_code, T, P, L, α1, α2,
 ↪  α3, α4, α5, φ1, φ2, φ3, φ4, φ5, T_lo, T_md, T_hi, P_lo, P_md, P_hi,
 ↪  L_lo, L_md, L_hi)
# Write line to database
cur.execute(SQL, line)
# calculate map position of possible moves
θ = np.arctan2(orientation[1], orientation[0])
turn_left = np.array([np.cos(θ + π / 2), np.sin(θ + π / 2)], dtype=int)
turn_around = np.array([np.cos(θ + π), np.sin(θ + π)], dtype=int)
turn_right = np.array([np.cos(θ + π * 1.5), np.sin(θ + π * 1.5)], dtype=int)
left = turn_left + position
forward = orientation + position
back = turn_around + position
right = turn_right + position
if (0 <= left[0] < rows) & (0 <= left[1] < columns):
    if the_map[left[0], left[1]]:
        # print('Left is Valid')
        # turn left and move 1
        orientation = turn_left
        position += orientation
        continue
    # else:
        # print('Left is a Wall')
# else:
    # print('Left is Out of Bounds')
if (0 <= forward[0] < rows) & (0 <= forward[1] < columns):
    if the_map[forward[0], forward[1]]:
        # print('Forward is Valid')
        # Maintain Orientation and move one space
        position += orientation
        continue
    # else:
        # print('Forward is a Wall')
# else:
    # print('Forward is Out of Bounds')
    if (0 <= right[0] < rows) & (0 <= right[1] < columns):
        if the_map[right[0], right[1]]:
            # turn right and move 1
            # print('Right is Valid')
            orientation = turn_right
            position += orientation
            continue
        # else:
            # print('Right is a Wall')
    # else:
        # print('Right is Out of Bounds')
    if (0 <= back[0] < rows) & (0 <= back[1] < columns):
        if the_map[back[0], back[1]]:
            orientation = turn_around
            position = back

# Make the changes to the database persistent
conn.commit()
# Close communication with the database
cur.close()
conn.close()
```

B

Robot Navigation Animation Code

```
#!/usr/bin/env python
#  -*- coding: utf-8 -*-
"""Read Robot Navigation data generated by sim.py
    from a database.  draw the map used to generate the date
    Animate robot position at each step
    map file, robot image file, database and table are set in variables

Example:
    After editing variables

    $ python animate.py
```

```
"""
import numpy as np
import pygame
import PIL.Image as Image
import psycopg2
from psycopg2.extras import DictCursor
import sys
import subprocess

# Define some colors
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
GREEN = (0, 255, 0)
RED = (255, 0, 0)
mapFile = 'map.bmp'
robotFile = 'bbr.bmp'
scale = 55

im = Image.open(mapFile)
the_map = np.array(im)
(map_rows, map_columns) = np.shape(the_map)
# open database
try:
    # conn = psycopg2.connect("dbname='RoboSim' user='RoboSimFull'
     ↪  host='localhost' password='l7S4Q5mm'")
    conn = psycopg2.connect("dbname='RoboSim' user='RoboSimFull'
     ↪  host='172.16.0.130' password='l7S4Q5mm'")
except:
    print("I am unable to connect to the database")

cur = conn.cursor(cursor_factory=DictCursor)
# cur.execute("""SELECT x, y from rs_small""")
cur.execute("""SELECT * from rs_large""")
rows = cur.fetchall()
# Close communication with the database
cur.close()
conn.close()
filenamelist = [0]*(len(rows))
pygame.init()

# Set the width and height of the screen [width, height]
```

```
size = (map_columns*scale, map_rows*scale)
screen = pygame.display.set_mode(size)
print(size[0])
print(size[1])

pygame.display.set_caption("RoboSim")

# Loop until the user clicks the close button.
done = False

# Used to manage how fast the screen updates
clock = pygame.time.Clock()

# Load graphics
background_image = pygame.image.load(mapFile).convert()
robot_image = pygame.image.load(robotFile).convert()

# -------- Main Program Loop -----------
while not done:
    # --- Main event loop
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True
        elif event.type == pygame.MOUSEBUTTONUP:
            None
        elif event.type == pygame.MOUSEBUTTONDOWN:
            None
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_ESCAPE:
                done = True

    # --- Game logic should go here
    for row in rows:
        print(row)
        screen.blit(pygame.transform.scale(background_image, size), [0,0])
        screen.blit(pygame.transform.scale(robot_image, (scale, scale)),
         ↪  [row['y'] * scale, row['x'] * scale])
        pygame.display.flip()
        filenamelist[row['step']-1] = "animate" + str(row['step']) + ".png"
        pygame.image.save(screen, filenamelist[row['step']-1])
        clock.tick(60)
        pygame.event.pump()
    done = True
print(filenamelist)
# Close the window and quit.
convertexepath = "C:/Users/jlentz/Downloads/ImageMagick-7.0.7-28-portable-Q16-
 ↪   x64/convert.exe"  #
 ↪   Hardcoded
convertcommand = [convertexepath, "-delay", "20", "-size", str(size[0]) + "x" +
 ↪   str(size[1])] + filenamelist + [" anim.gif"]
subprocess.call(convertcommand)

pygame.quit()
sys.exit()
```

C
Further browsing of the dialectic Structure

The following are more images of the dialectic user interface as generated by the presentation engine.

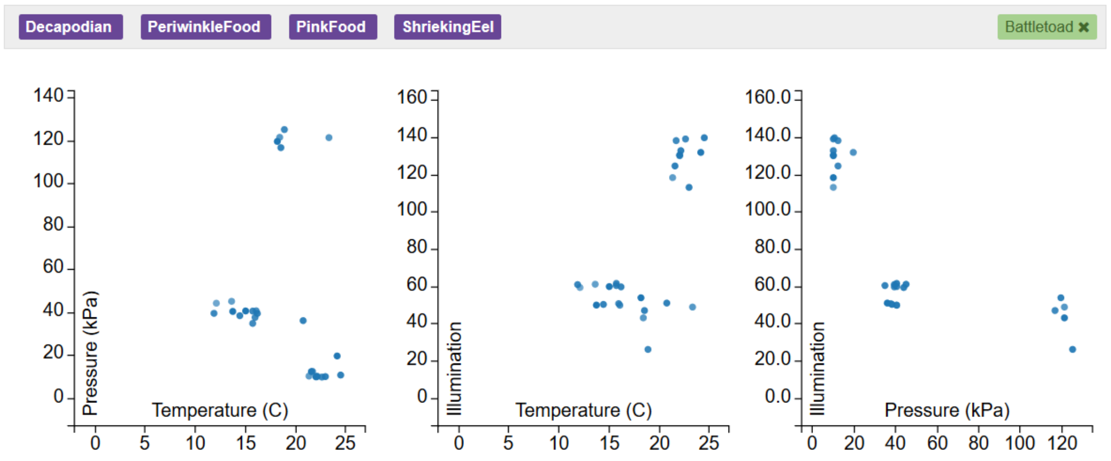


Figure C.1: *Battletoad*

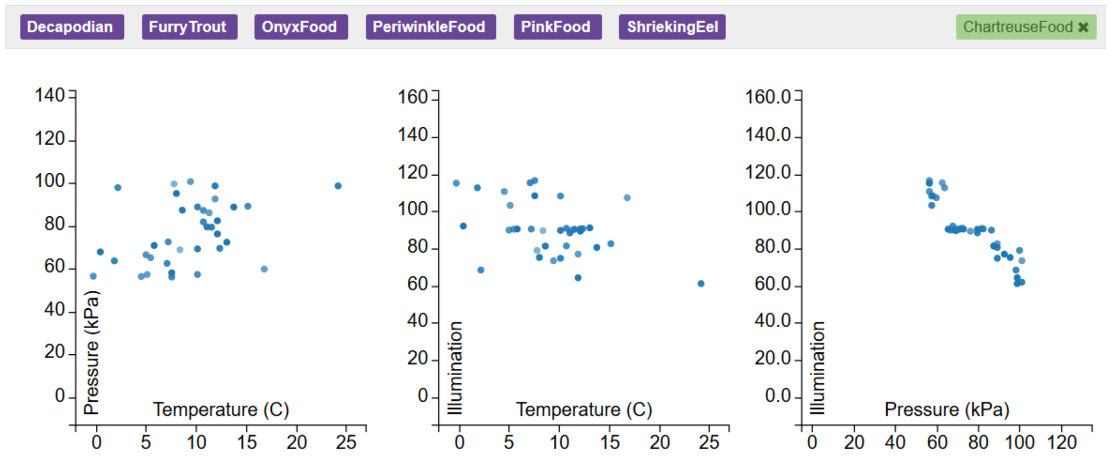


Figure C.2: *ChartreuseFood*

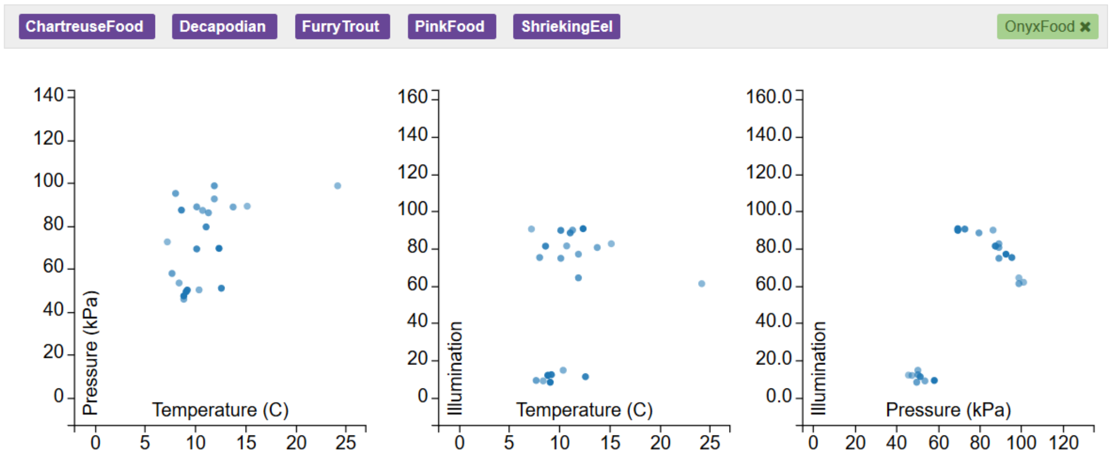


Figure C.3: *OnyxFood*

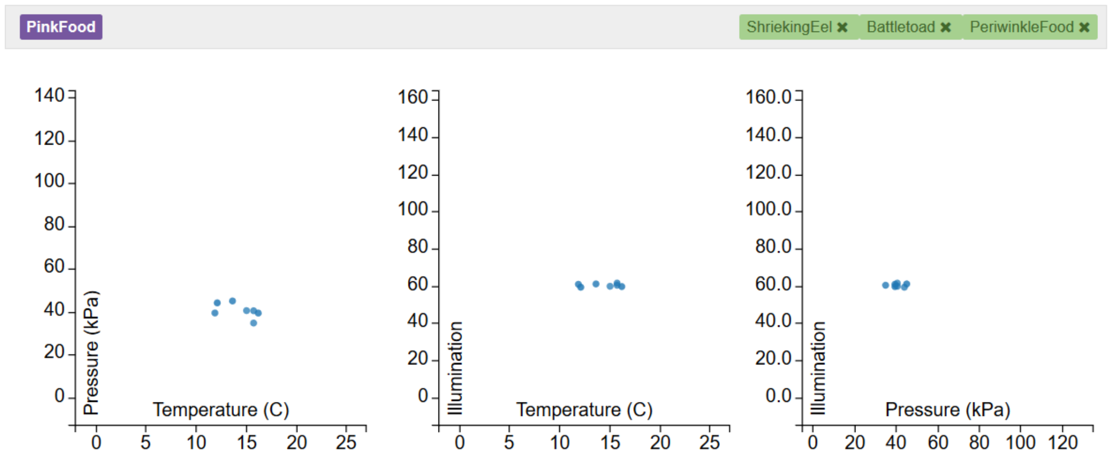


Figure C.4: *Battletoad* ∩ *PeriwinkleFood* ∩ *ShriekingEel*

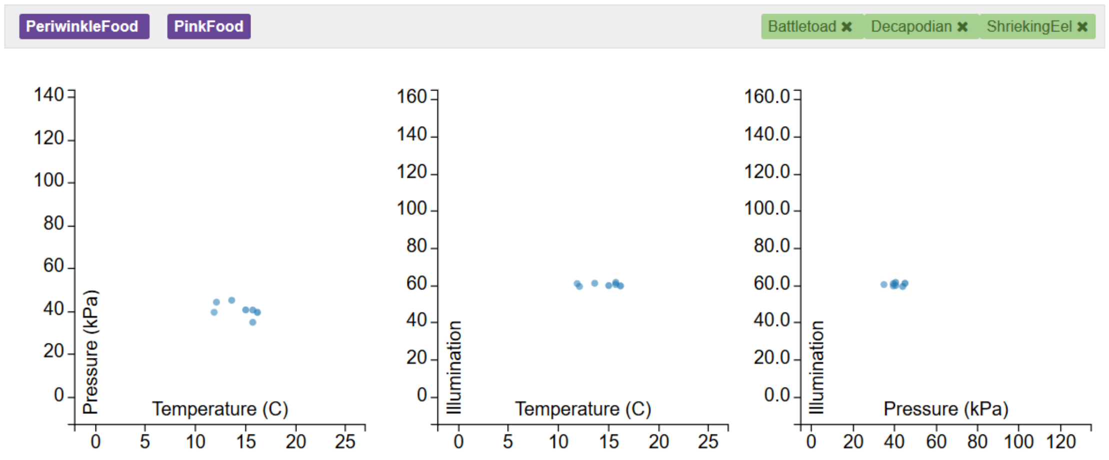


Figure C.5: *Battletoad ∩ Decapodian ∩ ShriekingEel*

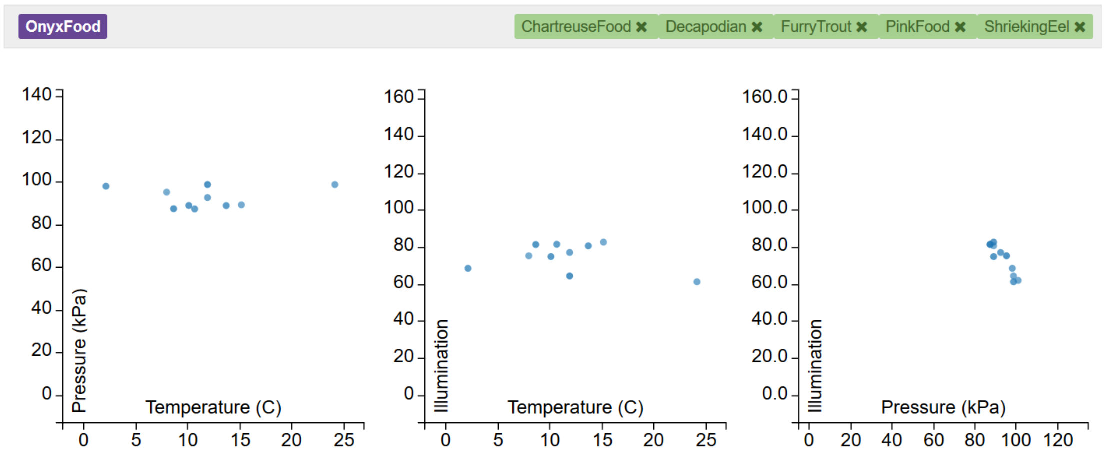


Figure C.6: *ChartreuseFood ∩ Decapodian ∩ FurryTrout ∩ PinkFood ∩ ShriekingEel*

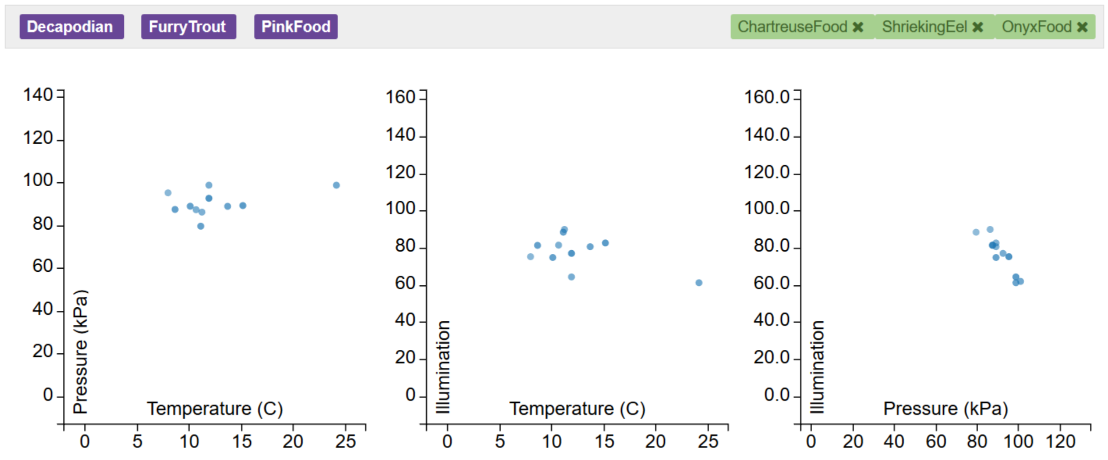


Figure C.7: *ChartreuseFood* ∩ *OnyxFood* ∩ *ShriekingEel*

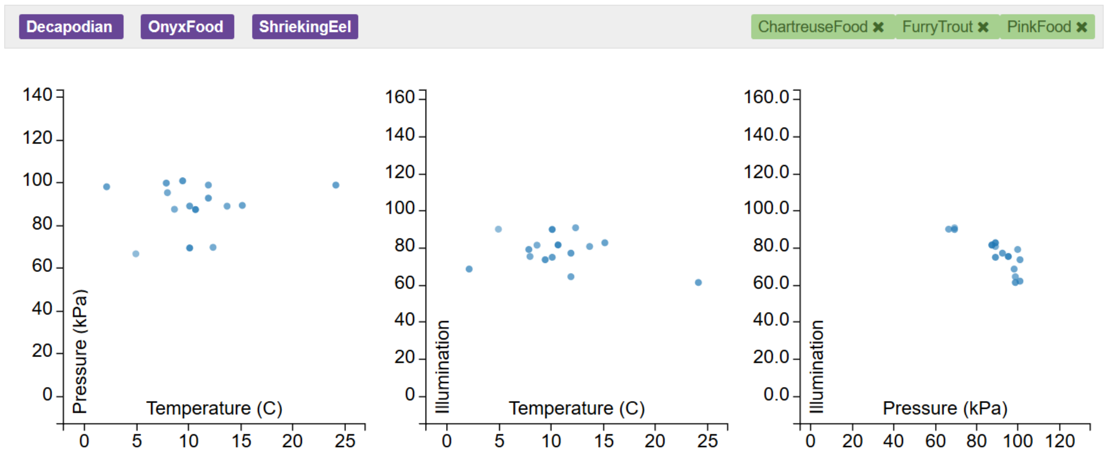


Figure C.8: *ChartreuseFood* ∩ *FurryTrout* ∩ *PinkFood*

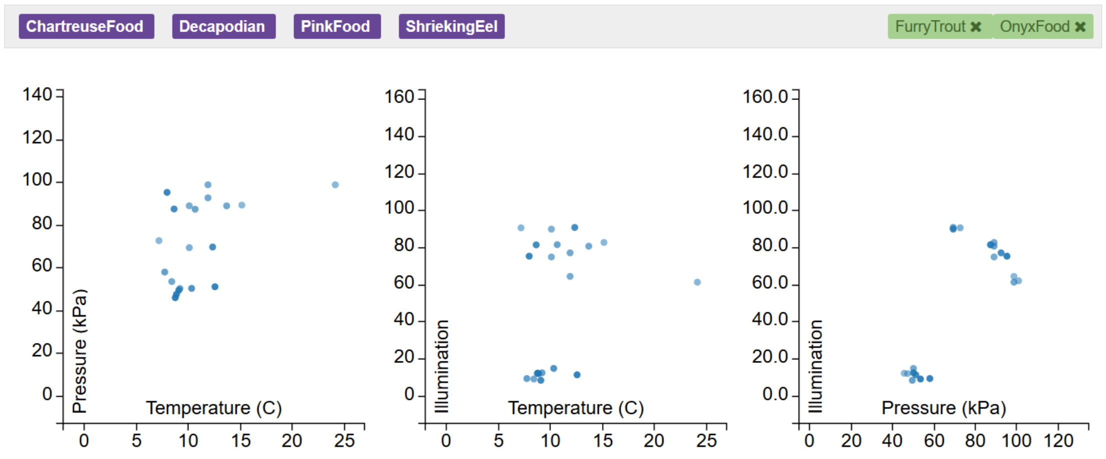


Figure C.9: *FurryTrout* ∩ *OnyxFood*

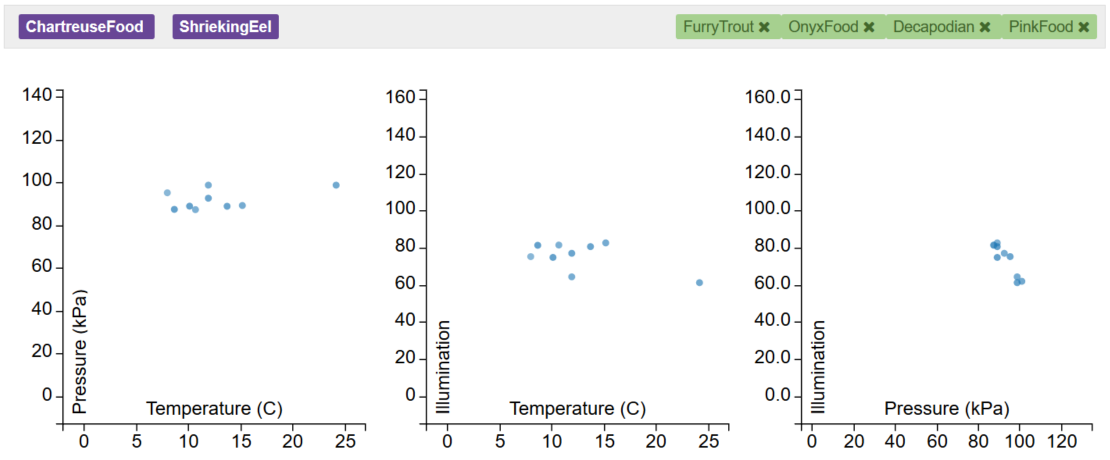


Figure C.10: *Decapodian* ∩ *FurryTrout* ∩ *OnyxFood* ∩ *PinkFood*

References

[1] Bondy, A. & Murty, U. (2011). *Graph Theory*. Graduate Texts in Mathematics. Springer London.

[2] Clark, Alex (2018). Pillow.

[3] Daniele Varrazzo (2017). psycopg: Postgresql adapter for the python programming language.

[4] Garrett, J. (2010). *Elements of User Experience,The: User-Centered Design for the Web and Beyond*. Voices That Matter. Pearson Education.

[5] ISO/IEC 9075-1:2016 (2016). *Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework)*. Standard, International Organization for Standardization, Geneva, CH.

[6] Jeremy Evans (2017). Sequel: The database toolkit for ruby.

[7] Jones, E., Oliphant, T., Peterson, P., et al. (2001–). SciPy: Open source scientific tools for Python. [Online; accessed <today>].

[8] Kaufmann, W. (1966). *Hegel: A Reinterpretation*. A Doubleday Anchor book. Doubleday.

[9] Kroeger, A. E. (1872). The difference between the dialectic method of hegel and the synthetic method of kant and fichte. *The Journal of Speculative Philosophy*, 6(2), 184–187.

[10] Lundh, Fredrik (2005). *Python Imaging Library*. Technical report.

[11] McKinney, W. (2010). Data structures for statistical computing in python. In S. van der Walt & J. Millman (Eds.), *Proceedings of the 9th Python in Science Conference* (pp. 51 – 56).

[12] Microsoft Corporation (2012). Types of bitmaps.

[13] Picone, R., powell, B., & Lentz, J. (2018). Dialectical information architecture.

[14] Picone, R. A. & Powell, B. (2015). A new information architecture: A synthesis of structure, flow, and dialectic. In S. Yamamoto (Ed.), *Human Interface and the Management of Information. Information and Knowledge Design*, volume 9172 of *Lecture Notes in Computer Science* (pp. 320–331). Springer International Publishing.

[15] Picone, R. A. R., Lentz, J., & Powell, B. (2017). The fuzzification of an information architecture for information integration: Human interface and the management of information: Information, knowledge and interaction design. volume 10273 of *Lecture Notes in Computer Science* (pp. 145–157). Springer International Publishing.

[16] pygame Community (2018). pygame.

[17] Ross, T. (2011). *Fuzzy Logic with Engineering Applications*. Wiley, third edition.

[18] Rossum, G. (1995). *Python Reference Manual*. Technical report, Amsterdam, The Netherlands, The Netherlands.

[19] Strother, J. B., Ulijn, J. M., & Fazal, Z. (2012). *Information Overload: An International Challenge for Professional Engineers and Technical Communicators*. Number ISBN 9781118360491. Wiley-IEEE Press.

[20] The PostgreSQL Global Development Group (2017). Postgresql.

[21] Travis E. Oliphant (2006). *A guide to NumPy*. USA: Trelgol Publishing.

[22] Trudeau, R. (2013). *Introduction to Graph Theory*. Dover Books on Mathematics. Dover Publications.

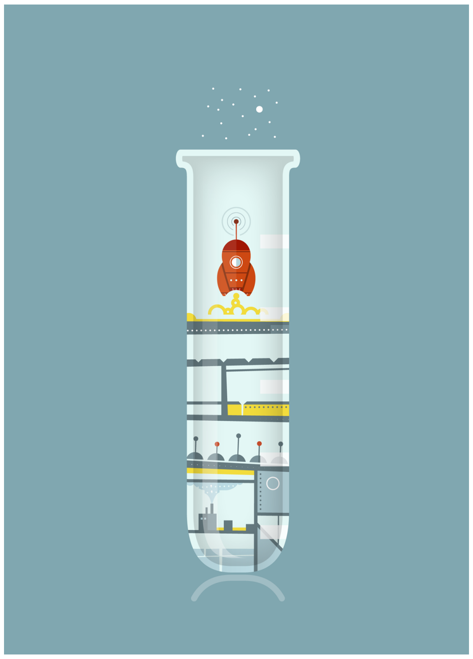
[23] Tufte, E. (2001). *The Visual Display of Quantitative Information*. Graphics Press.

[24] Warner, J. (2016). Scikit-fuzzy: A fuzzy logic toolbox for scipy.

[25] Yukihiro Matsumoto (2016). Ruby.

[26] Zeldes, N., Sward, D., & Louchheim, S. (2007). Infomania: Why we can't afford to ignore it any longer. *First Monday*, 12(8).

[27] Zimmermann, H. (2001). *Fuzzy Set Theory—and Its Applications*. Springer Netherlands.



THIS THESIS WAS TYPESET using LaTeX, originally developed by Leslie Lamport and based on Donald Knuth's TeX. The body text is set in 11 point Egenolff-Berner Garamond, a revival of Claude Garamont's humanist typeface. The above illustration, *Science Experiment 02*, was created by Ben Schlitter and released under CC BY-NC-ND 3.0. A template that can be used to format a PhD dissertation with this look & feel has been released under the permissive AGPL license, and can be found online at github.com/suchow/Dissertate or from its lead author, Jordan Suchow, at suchow@post.harvard.edu.