A GENETIC ALGORITHM FOR LOCATING ACCEPTABLE STRUCTURE MODELS OF SYSTEMS (RECONSTRUCTABILITY ANALYSIS)

A Master's Thesis

Presented to

Department of Computer and Information Science

State University of New York Polytechnic Institute

Utica, New York

In Partial Fulfillment

of the Requirements for the

Master of Science Degree

Joshua Heath

May 2018

STATE UNIVERSITY OF NEW YORK

POLYTECHNIC INSTITUTE

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Approved and recommended for acceptance as a thesis in
partial fulfillment of the requirements for the degree of
Master of Science in Computer and Information Science

_____

Date

_____

Dr. Roger Cavallo (Advisor)

_____

Dr. Michael Reale

_____

Dr. Saumendra Sengupta

A GENETIC ALGORITHM FOR LOCATING ACCEPTABLE STRUCTURE MODELS OF SYSTEMS (RECONSTRUCTABILITY ANALYSIS)

Except where reference is made to the work of others, the work described here is my own or was done in collaboration with my advisor and/or members of the advisory committee. Further, the content of this work is truthful in regards to my own work and the portrayal of other's work. This work, I claim, does not include proprietary or classified information to the best of my knowledge.

_____

Joshua Heath

Acknowledgements

The following is a list of nouns that I'd like to thank:

- My advisor, Roger Cavallo, for his support and advice during the writing of this thesis.
- Professor Reale and Professor Sengupta, for making themselves available to serve on my committee with relatively short notice and for providing valuable feedback.
- The SUNY Polytechnic staff, for contributing to my education in a positive way.
- The Cincinnatus Central School staff, for contributing to my upbringing in a positive way.
- The Academy.
- Mick Cringle, for inspiring both a love of science and a search for wisdom and joy.
- Nicole Rice, for giving me the opportunity to fall in love with technology, and for always believing in me.
- Obama. *Thanks, Obama*! (It's an old meme)
- My laptop, for sticking it out with me through some good years.
- All the people that I am lucky enough to call my friends.
- My roommates, for constantly trying to pull me away from working on my thesis.
- Lydia, for all that she is, and for pushing me to become a better person.
- My family, for all of their unconditional love and support.

Abstract

The emergence of the field of General Systems Theory (GST) can be best attributed to the belief that all systems, irrespective of context, share simple, organizational principles capable of being mathematically modeled with any of many forms of abstraction. Structure modeling is a well-developed aspect of GST specializing in analyzing the structure of a system - that is, the interactions between the attributes of a system. These interactions, while intuitive in smaller systems, become increasingly difficult to comprehend as the number of measurable attributes of a system increases. To combat this, one may approach an overall system by analyzing its various subsystems and, potentially, reconstruct properties of that system using knowledge gained from considering a collection of these subsystems (a structure model). In situations where the overall system cannot be fully reconstructed based on a given structure model, the benefits and detriments associated with using such a model should both be considered. For example, while a model may be simpler to understand, or require less storage space in memory than the system as a whole, all information regarding that system may not be inferable from that model. As systems grow in size, determining the acceptability of every meaningful structure model of a system in order to find the most acceptable becomes exceedingly resource-intensive. In this thesis, a measure of the memory requirements associated with storing a system or a set of subsystems (a structure model) is defined and is used in defining an objective measure of the acceptability of a structure as a representation of an overall system. A Genetic Algorithm for Locating Acceptable Structures (GALAS) is then outlined, with this acceptability criterion serving as an optimizable fitness function. The goal of this heuristic is to search the set of all meaningful structure models, without the need for exhaustively generating each, and produce those that are the most acceptable, based on predefined acceptability criteria.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

At the beginning of the second half of the 20th century, in response to the ever-present specialization that had long divided and isolated the sciences, a movement for integration began. The presence and subsequent success of interdisciplinary fields, such as Cybernetics and Management Science, inspired many from across academia to work to develop a general framework that could be applied within any field of scientific inquiry, independent of context. From this movement, the study of General Systems Theory, though first conceived decades prior, would gain widespread popularity with those in support of an open scientific environment.

As the study of systems has progressed, many concepts have been found useful for identifying different properties of a general system. One such concept is the description of a system's structure - that is, the interactions between attributes of that system. Knowledge regarding the system that is being modeled/represented can be inferred by observing these inter-attribute relationships. In relevant literature, the study of these relationships and the various perspectives from which one may view a system is referred to as structure modeling. An ideal model allows for the gathering of as much "information" regarding the system as is obtainable from external observations; in other words, compared to all other models, one could infer the most knowledge from the ideal model regarding every possible interaction between the system's attributes (the parts), as well as the properties of the entire system (the whole).

However, in the majority of applications, systems are too elaborate to be modeled in such an idealistic way. The construction and analysis of structure models is typically performed using computers, and, despite their advancements, the modern-day computer is limited by its computational ability and storage capacity. Therefore, we are forced use structure models less demanding of resources. This restriction of the ideal model can be considered a compression

process, since we are effectively condensing the knowledge we would otherwise have about a system. Unfortunately, compression, in this context, is almost always lossy, meaning that the process cannot be reversed without losing information. In this work, a structure model that balances information loss with resource demand - specifically, storage requirement - is desired.

Structure modeling is a well-developed branch of General Systems Theory and has found applications in, among other topics, database theory [7]. Through the work of Cavallo and Klir [4,5,6], a systems framework, referred to as the General Systems Problem Solver (GSPS) has been outlined and is accompanied by a set of computer-implementable procedures for studying systems and their structure. These procedures are categorized into one of three classes, described below:

1. Generate possible structure models for a given system
2. Evaluate the acceptability of a given structure model(s)
3. Search for a model(s) from the set of structure models for a given system that meet predefined acceptability criteria

As systems grow in size, the goal of class 1 procedures becomes increasingly demanding in terms of both memory and computing power. The goal of this thesis is to define a measure used for evaluating the acceptability of a structure model (class 2), and to implement this measure within a search heuristic in an attempt to find the most acceptable model(s) (class 3) of a given system, without requiring the exhaustive generation and evaluation of the entire set of possible structure models. The search heuristic to be implemented is one of a class of "Genetic Algorithms".

Through his ground-breaking book *Adaptation in Natural and Artificial Systems* [14], John Holland introduced genetic algorithms. Soliciting the aid of natural phenomena for practical purposes would appear to be a relatively modern concept. Yet, over 40 years ago, one of Holland's most notable contributions to science was inspired by nature - specifically, evolutionary biology. Genetic algorithms are search/optimization algorithms, and their nonlinear, metaheuristic nature allows for them to outperform modern optimization tools in certain settings, for certain types of problems.

In this thesis, I define a measure of the memory requirements associated with storing a system or a structure model in memory and use this in defining an objective measure of the acceptability of a structure model as a representation of a given system. I then outline and implement, in the R programming language, a Genetic Algorithm for Locating Acceptable Structures (GALAS) using this criterion as an optimizable fitness function, with the goal of finding the "most acceptable" structure model of an overall system.

The structure of the remainder of this thesis is as follows: Chapter 2 introduces preliminary systems terminology, defines storage requirement for systems and structure models, and discusses measures of information in both relational and probabilistic systems; Chapter 3 discusses genetic algorithms, including a brief history, component analysis, and comparison to conventional, non-heuristic techniques; Chapter 4 outlines and analyzes GALAS and presents results of testing GALAS; Chapter 5 consists of closing remarks, potential applications of GALAS, and future work; Appendix A contains any inconveniently large tables; Appendix B contains a user manual for using GALAS; Appendix C contains all source code for GALAS.

Chapter 2

General Systems and Information Theory

This chapter gives a brief history of the development of structure modeling as a branch of general systems theory. Preliminary definitions regarding system modeling are presented as they have been defined by Cavallo in [3]. Structure reconstruction techniques are discussed alongside evaluative methods for considering the acceptability of these structures as representations of an overall system. These methods include information-theoretic concepts and an objective measure of the memory requirements associated with storing a system or structure model. The latter is introduced in this thesis as a means of acknowledging the inevitable need for complex systems, and their models, to be stored and manipulated using computers.

2.1 - Introduction to General Systems Theory

General systems theory is the study of systems. The word system comes from the Latin systēma, meaning “whole compounded of several parts”. Merriam-Webster defines a system as “a regularly interacting or interdependent group of items forming a unified whole” [23]. In order to relate to the conventional idea of a “system”, one will typically relate to some example of a system, be it biological, mechanical, social, etc. However, this association is irrelevant from the systemic perspective taken by systems researchers; the term system implies a lack of context. When considering systems, systems theorists focus their attention on properties of that system, such as its structure, that are independent of function or purpose.

While understanding the relationships between attributes of a simpler system is fairly intuitive from a purely observational perspective, these relationships become far more difficult to grasp and process as the number of measurable attributes of a system increases. However, one should not fear such subjectively complex systems, assuming they are willing to attack them using a systems theoretic approach. In his paper, “Constraint Analysis of many-

dimensional relations" [1], W. Ross Ashby, a trailblazer for systems science, states that "relations between large numbers of variables are often not as complex as they seem, for they are often constructed from simpler sub-relations, and retain something of their simplicity." The emergence of the field of general systems theory can be best attributed to the belief that all systems, irrespective of context, share simple, organizational principles capable of being mathematically modeled with any of many forms of abstraction.

A pioneer of systems theory, George Klir, epistemologically classifies the perspectives we may take on systems into a hierarchy. As the levels increase, so too does our knowledge regarding the system. The lowest level (0) consists of three, so-called dataless systems: those defined only by their contextual attributes and the measurable values of those attributes (object systems), those general object systems that omit any context (image systems), and those systems containing both an object system and an image system as well as some function for mapping between the two (source system). This level does not include information regarding which states of the system can and cannot be observed. Level 1 introduces data systems - those level 0 image systems for which previously unknown state occurrences are given. Level 2 systems are referred to as generative systems because, at this level, we become aware of individual relations among groups of the system attributes that allow for the generation of the state information seen in level 1. Level 3 consists of structure systems, with which this thesis is especially concerned with. Structure systems are defined by a set of generative systems and the relationship(s) between them. Level 4 systems (metasystems) and systems at higher levels of the hierarchy are defined recursively as the set of systems from the level immediately preceding the level of the system being defined; these systems are not explicitly addressed in this work. [17]

2.2 - Defining a system

A *system* S is defined as a four-tuple (V, Δ, dom, f) [3], where

- V is a non-empty set of n distinct symbols called *attributes*;
- Δ is a non-empty set of sets of values called *domains*;
- dom: V → Δ is a function that associates a domain with each attribute;
- f is a function defined over the domain $T = \times_{(v_i \in V)} \mathrm{dom}(v_i)$

- If the codomain of f is {0,1}, then S is called a *relational* system
- If the codomain of f is the interval [0,1], then S is a *fuzzy* system
- If the codomain of f is the interval [0,1] with the restriction that $\sum_{t\in T} f(t) = 1$, then S is a *probabilistic* system

For practical purposes, we will assume that V is a finite set and, for each $v_i \in V$, dom(v_i) is also a finite set of size 2 or greater. In other words, a system has at least one attribute, and each attribute of that system can have one of two or more possible values. In general, the attributes of a system, each represented by a $v_i \in V$, are measurable and discrete; there is a clear divide between each element of dom(v_i). The attributes of a system are also referred to as the *variables* of that system. From a geometric perspective, for all $v_i \in V$, the set containing all dom(v_i) can be viewed as the set of all dimensions of the system; T is the multi-dimensional space denoting the set of all possible states (tuples) of the system. The cardinality of set T, denoted |T|, is equal to $\prod_{\mathbf{v_i}\in\mathbf{V}} |\mathbf{dom}(\mathbf{v_i})|$. In this thesis, relational and probabilistic systems are addressed, while fuzzy systems are not discussed.

If all attributes of the system under investigation that one wishes to represent are elements of V, the system is referred to as the *overall* system. This is obviously a case-by-case definition and to define a system as overall would imply having contextual knowledge. We are not concerned with this context, and any overall systems referenced in this thesis contain none; it is the job of the investigator to associate a specific, real-world system with a general, mathematical definition of that system.

In a relational system, if state $t \in T$ is observed when measuring the attributes of that system, then f(t) = 1, and if it is not observed, f(t) = 0. In a probabilistic system, f(t) is equal to the probability that, when measured, the system is observed in state t. For a system definition to appropriately indicate the actual system being measured, a statistical rule of thumb that has been used is that the number of measurements of that system should be at least 5 times more than |V|. While this is the recommended minimum number, a system should be defined using as many measurements as are available, to ensure the mathematical definition properly reflects the actual system. In both the relational and probabilistic cases, function f, typically referred to as the “characteristic” or “behavior” function, is essential; without this, a system is a dataless

(level 0 source system) and is of little use to investigators. The above system definition is specifically for data systems (level 1) because of the inclusion of the characteristic function.

Example 2.2.1 Tables 2-1(a) and 2-1(b) contain a relational system and a probabilistic system, S^R and S^P, respectively. Both systems are defined by (V, Δ, dom, f), with only the codomain of function f differing between the two - f^R vs. f^P. Both systems have 3 attributes - $V = \{v_1, v_2, v_3\}$; Each attribute is binary, i.e. it can take the value of either 0 or 1 - $\Delta = \{\{0,1\}\}$; $dom(v_1) = dom(v_2) = dom(v_3) = \{0,1\}$. The states of the systems are denoted by the rows of the tables. A state will be referred to by the concatenation of the values of all attributes in ascending order of index number, e.g. the state of the system where $v_1 = 0$, $v_2 = 0$, $v_3 = 0$ - the first row in both tables - is referred to as 000. An attribute v_i may also be referred to by solely its numerical index i in discussions regarding the attributes of an overall system, e.g. systems S^R and S^P can be referred to as 123. The rightmost column of the tables defines the range of the functions f^R and f^P. In Table 2-1(a), $f^R(000) = 1$, signifying that the system state 000 is a possible state that the system S^R may be observed in; in Table 2-1(b), $f^P(101) = 0.15$, signifying that state 101 will be seen when observing S^P with a probability of 0.15. Probabilistic systems are more general than relational systems, but can easily be converted to such by applying the ceiling function to the elements of the range of the characteristic function. By applying the ceiling function to $f^P(t)$ in Table 2-1(b) the result is $f^R(t)$ in Table 2-1(a), suggesting that data collected for both S^R and S^P are from the same source, but S^R is the relational definition of this system and S^p is the probabilistic definition.

v_1	v_2	v_3	$f^R(t)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

Table 2-1(a) Relational System S^R

v_1	v_2	v_3	$f^P(t)$
0	0	0	0.25
0	0	1	0.1
0	1	0	0.15
0	1	1	0.35
1	0	0	0
1	0	1	0.15
1	1	0	0
1	1	1	0

Table 2-1(b) Probabilistic System S^P

2.3 - Subsystems of a system

A system $S_i = (V_i, \Delta, dom|V_i, f_i)$ is a *subsystem* of another system S if $V_i \subseteq V$, $dom|V_i$ is the restriction of dom to V_i, and f_i is the projection of f onto V_i. The set V_i is referred to as a subsystem scheme [3] and T_i is the domain of the function f_i; that is, $\mathbf{T_i = \times_{(v_j \in v_i)} dom(v_j)}$. Assume that subsystems referenced in this thesis are subsystems of some overall system, referred to by S. In this case, a subsystem S_i can be viewed as the system S with one or more attributes being disregarded. In a relational setting, for all $t_i \in T_i$, $f(t_i) = 1$ if there exists a state $t \in T$ of the system S where $f(t) = 1$ and the values of the attributes of V_i in state t_i are identical to the values of the attributes of V_i in state t; otherwise, $f(t_i) = 0$. In a probabilistic setting, $f(t_i) = \sum_{t \in T} f(t)$ where the measurements of the attributes within V_i in state t_i are identical to the measurements of the attributes of V_i in state t. In a probabilistic setting, f(T) is a probability distribution over T and $f(T_i)$ is a marginal distribution over T_i relative to T. In the hierarchy of systems, subsystems are considered generative systems and make up level 2. At this level we are aware of relationships between attributes of S, and the subsystem S_i, is essentially a description of the relationship between the attributes of V_i.

Example 2.3.1 Tables 2-2, 2-3, and 2-4 contain the tabular representations of three subsystems of S^R and S^P from Example 2.2.1, S_{12}, S_{23}, and S_{13}, respectively. $S_i = (V_i, \Delta, dom|V_i, f_i)$, where $V_{12} = \{1,2\}$, $V_{23} = \{2,3\}$, $V_{13} = \{1,3\}$; $\Delta = \{\{0,1\}\}$; $dom(1) = dom(2) = dom(3) = \{0,1\}$; the range of the functions f_i for S^R and S^P are the two rightmost columns of the Tables below. Subsystem S_{12} contains attributes {1,2}, thus the domain of dom is restricted to exclude attribute 3. The first state in Table 2-2, 00, does occur in a relational setting - $f_{12}^R(00) = 1$ - and has a 0.35 probability of occurring in a probabilistic setting - $f_{12}^P(00) = 0.35$. The former is true because at least one state $t \in T$, where $v_1 = 0$ and $v_2 = 0$, occurs in S^R. The latter value (0.35) is calculated by adding the probabilities of all states from the overall system in which attributes 1 and 2 both have the value of 0, i.e. in the overall system S^P: f(000) + f(001) = 0.25 + 0.1 = 0.35. Similarly, in subsystem S_{13}^P (Table 2-4), the second state listed, 01, has a 0.45 probability of occurring because $f^P(001) + f^P(011) = 0.1 + 0.35 = 0.45 = f_{13}^P(01)$. This process can be extended to all states of all subsystems of an overall system.

v_1	v_2	$f_{12}^R(t)$	$f_{12}^P(t)$
0	0	1	0.35
0	1	1	0.5
1	0	1	0.15
1	1	0	0

Table 2-2: Subsystem S_{12}

v_2	v_3	$f_{23}^R(t)$	$f_{23}^P(t)$
0	0	1	0.25
0	1	1	0.25
1	0	1	0.15
1	1	1	0.35

Table 2-3: Subsystem S_{23}

v_1	v_3	$f_{13}^R(t)$	$f_{13}^P(t)$
0	0	1	0.4
0	1	1	0.45
1	0	0	0
1	1	1	0.15

Table 2-4: Subsystem S_{13}

2.4 - Structure systems

Given overall system S with attributes V, a *structure system* is a set of subsystems SS = {S_1, … , S_k}, and the set {V_1, … , V_k} is called a *structure*. The terms structure system and structure will be used synonymously, since, in regards to the same overall system, a unique structure corresponds to a unique structure system. A structure is defined as any family of subsets of V. The set of all possible structures S' is defined as {SS_i | $SS_i \subseteq \mathcal{P}(V)$}, where $\mathcal{P}(.)$ denotes the power set [4].

Example 2.4.1 Examples of valid structures defined on the system 123 (Table 2-1) include: {{1,2}} (Table 2-2); {{1,2}{2,3}} (Tables 2-2 & 2-3}; {{1,2},{1}}; {{1,2},{2,3},{1,3}} (Tables 2-2, 2-3, & 2-4); {{1,3},∅}; {{1},{2,3}}. For notational convenience, the example structures - and all structures in this work - will be written in the following form, respectively: 12, 12/23, 12/1, 12/23/13, 13/∅, 1/23. In this notation, each subsystem within a structure is separated by a forward slash (/); the subsystem with no attributes is denoted using the empty set ∅. As was previously mentioned, structure systems make up level 3 of the epistemological hierarchy of systems and are a primary concern of this work.

In [4], Cavallo & Klir classify structures into the following families of subsets of S' that are worth noting in this work: S_α , S_H , S_G, S_C; additional classes were defined, but are not defined here as they are not immediately relevant. $\mathbf{S_\alpha = \{\alpha_i \mid \alpha_i \subset \mathcal{P}(V), \bigcup_{V_j \in \alpha_i} V_j = V\}}$, i.e. a structure system is an "*α-structure*" if its structure covers the set of all attributes V. $\mathbf{S_H = \{H_i \mid H_i \subset (\mathcal{P}(V) - \{\emptyset\}), \bigcup_{V_j \in H_i} V_j = V\}}$, i.e. "*H-structures*" are α-structures that exclude the empty subsystem. $\mathbf{S_G = \{G_i \mid G_i \subset (\mathcal{P}(V) - \{\emptyset\}), \bigcup_{V_j \in G_i} V_j = V, \text{(for each pair } V_a, V_b \in G_i) } V_a \not\subset V_b)\}$, i.e. a "*G-structure*" is an H-structure with no subsystem within the structure being a subsystem of another subsystem within that same structure. A G-structure is also known as a reduced cover of the set V.

In order to define the elements of S_C, we must first define some preliminary terms. An *undirected graph* G is denoted by the pair (V, E), where $V = \{v_1, v_2, \dots, v_m\}$ is a set of *vertices* (a.k.a. nodes) and E is a set of edges. An *edge* $e \in E$ is a size 2 subset of V signifying the presence of an edge connecting the vertices within that edge. The vertices within an edge are said to be "connected". A *clique* C in an undirected graph G is a subset of V where, for every unique pair of elements $v_i, v_j \in C$, v_i and v_j are connected. A *maximal clique* C_M in an undirected graph G is a clique C in G for which there does not exist a $v_i \in V$, where $v_i \notin C$, for which every element of C is connected to v_i. The *primal graph* of a structure SS, denoted P(SS), is defined by the pair (V, E), where for each $S_i \in SS$, if two unique vertices v_i, v_j are elements of the same subsystem ($v_i, v_j \in V_i$), then v_i and v_j are connected in P(SS). A "*C-structure*" is a G-structure for which each V_i associated with $S_i \in SS$ is a maximal clique of the primal graph of that C-structure and each maximal clique of this primal graph is equal to a V_i. S_C is the set of all C-structures - also called reduced conformal hypergraphs. It is also true that, for all systems, $S_C \subseteq S_G \subseteq S_H \subseteq S_\alpha$.

All H-structures can be viewed as hypergraphs on the set V. The definition of a hypergraph is essentially an extension of the definition of a graph, with the addition of allowing edges of size other than 2. A *hypergraph* H is denoted by the pair (V, E), where E is a set of hyperedges. A *hyperedge* $e \in E$ is a non-empty subset of V signifying the presence of a hyperedge connecting those $v_i \in e$. Note the similarities between a hyperedge and a non-empty subsystem of an overall system. Any structure system that excludes the empty subsystem can be mapped to a unique hypergraph, implying that any H-structure can be represented by a Hypergraph. With that, any G-structure is also a hypergraph because a G-structure is also an H-structure. Additionally, a G-structure is a reduced hypergraph; a *reduced hypergraph* is a hypergraph with no hyperedge being a subset of another hyperedge in that hypergraph. In addition to the textual notation previously specified, a hypergraph may be graphically represented using K-diagrams ("K", short for Klir) [4], as seen in Figure 2-1. In these diagrams, a square represents a hyperedge, a line represents an attribute v_i (labeled as i) and a line connected to a square signifies that v_i is an element of the hyperedge represented by that square. The K-diagram notation can also be used to represent H-structures because of the parallel that can be made between H-structure and hypergraphs.

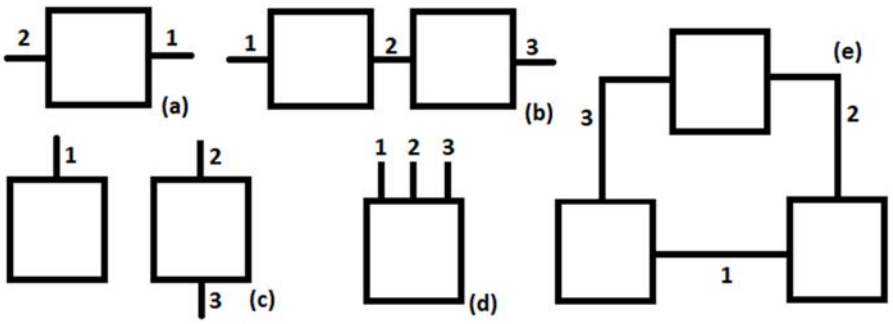

Figure 2-1: K-diagrams for (a) 12 (b) 12/23 (c) 1/23 (d) 123 (e) 12/23/13

In this thesis, only G-structures are considered as potential models of a system, since S', S_α, and S_H each include structures that are not as well-suited for structure modeling purposes for the following reasons. S' contains structures that do not cover the set of all attributes; an attribute of an overall system may not be defined in any subsystem within such a structure. S_α contains structures that contain the empty subsystem, which poses no added benefit, nor detriment, to the amount of information that can be interpreted from a structure. S_H contains structures that contain redundant subsystems. A redundant subsystem S_i in a structure system is a subsystem that is a subsystem of another subsystem S_j in that same structure system, and is referred to as such because the information that can be inferred from S_i can also be inferred from S_j; thus, S_i is unnecessary. S_G contains only structures that cover the set of all attributes, do not contain the empty subsystem, and do not contain redundant subsystems. Since $S_C \subseteq S_G$, C-structures also meet these criteria. However, given any structure, it is more difficult to test if that structure is a C-structure than it is to test if that structure is a G-structure, because finding the maximal cliques of a graph is a recognized NP-complete problem, in the general case. Since the purpose of the algorithm described in Chapter 4 is to serve as an alternative to less efficient techniques for searching a set of structures, the process of evaluating each structure should be as efficient as possible, within reason.

Two unique G-structures G_x ,G_y may be related by the comparative concepts of refinement and aggregation. G_x is a *refinement* of G_y - or G_y is an *aggregate* of G_x -, denoted $G_x \preccurlyeq G_y$, if and only if (*iff*) each $S_i \in G_x$ is a subsystem of at least one $S_j \in G_y$. G_x is an *immediate refinement* of G_y - or G_y is an *immediate aggregate* of G_x - *iff* $G_x \preccurlyeq G_y$ and there exists no G_z such that $G_x \neq G_z \neq G_y$ and $G_x \preccurlyeq G_z \preccurlyeq G_y$. When applied to structures, the act of refining a system may be conceptualized as the act of "looking deeper into the attributes of that system". In doing so,

you are analyzing the attributes of a system from a more refined perspective by viewing the holistic relationship between all attributes as a set of lesser-order relationships. Conversely, the aggregation procedure generalizes a structure by taking into account the relationships between larger groups of attributes, rather than viewing such a relationship as the “sum” of the lesser-order relations that comprise it [3].

2.5 - Emergence

The change in perspective provided by structure models may provide insight into the nature of the underlying relationships between attributes of an otherwise complex system. An example of a system that can be viewed from different resolutions is water. At the level that we as humans come into contact with water on a daily basis, it is simply a liquid. As you begin to refine (“zoom in on”) the definition of water, the properties that are inherent to water begin to disappear. One such property is the notion of “wetness”. If we were to view water as a collection of water molecules, an individual molecule is not usually considered to be wet. At what level of refinement does the system that is water lose its wetness property? Conversely, if we are to view water as a collection of H_2O molecules, then at what level of generalization does wetness become a property of the system? A property, like wetness, that emerges as systems become more abstract or disappears as a system becomes more refined is known as an *emergent property*. In this example, water behaves the way it does through both the action of the individual water molecules *and* through the interactions between the molecules, subject to the natural laws at the given resolution level. The latter allows for these unexpected, emergent properties to appear. The idea of emergence is complexity rising from simplicity, and it encapsulates the famous idea (typically attributed to Aristotle) that “the whole is greater than the sum of its parts”. The posed questions regarding where to draw the “wetness” line is thought-provoking and it is reasonable to believe that taking such a refined or generalized perspective on any system can be beneficial to an investigator looking to understand it.

Emergence is, quite literally, everywhere. Properties of an individual water molecule emerge from the interactions between three atoms - two hydrogen and one oxygen. Swarm intelligence, most notably in insects, is the collective behavior that manifests when a group of individuals behaves as an organized unit. Even the system that is the human body is the result of the interactions between its subsystems. One of these subsystems, the nervous system,

contains perhaps the most complex, yet tangible systems we have yet to fully understand - the brain. Some secular people theorize that consciousness itself is even an emergent property - it is believed to be a manifestation of billions of neuronal cells interacting. Unfortunately, it will be years before we may comprehend what is actually occurring in the biological processors that define us. At the moment, it is generally unknown as to why our mathematical models for neuronal activity (artificial neural networks) are as powerful as they are. But eventually, we'll reach a time in which we will fully understand our minds, and it is only then that we can truly define life in its most literal sense.

2.6 - Lattices on S_G and S_C

On a set of G-structures, the binary relation "$\preccurlyeq$", as it has been defined, will partially order the set. A binary relation R on a set X - denoted by the pair (X,R) - is a partial ordering of X iff R is a reflexive, antisymmetric, and transitive relation on X. The partially ordered set (X,R) is referred to as a poset. The partially ordered quality that results from defining "$\preccurlyeq$" on any set of G-structures is important because it implies structural organization. For example, the poset can be visualized using a Hasse diagram.

Assume a set $X = \{x_1 , x_2 , \dots , x_n\}$ is partially ordered by relation R. Also, assume that the following definitions are relevant to some subset of X, denoted by Y. The set of *upper bounds* of a set $Y \subseteq X$, is the set $U \subseteq X$, where $x_i \in U$ *iff* $x_j \, \mathrm{R} \, x_i$ for all $x_j \in Y$. The *least upper bound* (LUB) of a set $Y \subseteq X$, is $x_i \in U$, such that $x_i \, \mathrm{R} \, x_j$ for all $x_j \in U$. The set of *lower bounds* of a set $Y \subseteq X$, is the set $L \subseteq X$, where $x_i \in L$ *iff* $x_i \, \mathrm{R} \, x_j$ for all $x_j \in Y$. The *greatest lower bound* (GLB) of a set $Y \subseteq X$, is $x_i \in L$, such that $x_j \, \mathrm{R} \, x_i$ for all $x_j \in L$. The *universal upper bound* of X is x_i, such that $x_j \, \mathrm{R} \, x_i$ for all $x_j \in X$. The *universal lower bound* of X is x_i, such that $x_i \, \mathrm{R} \, x_j$ for all $x_j \in X$. In words, the upper bounds U of a set Y is the set of elements that all elements in Y are related, by relation R, to, while L is the set of elements that are related to all elements in Y. The LUB is the element of U that is related to all elements of U, and the GLB is the element of L that all elements of U are related to. The universal upper bound is the element that every element is related to, and the universal lower bound is the element that is related to every element. If every pair of elements of X has both an LUB and a GLB, then the poset (X,R) is a *lattice*. The lattice quality implies that any two members of the set, that may otherwise be incomparable by relation "R", are in some way "related" through their LUB and GLB.

Any set of G-structures, including S_G, ordered by relation $\preccurlyeq$ is a lattice. Therefore, any two G-structures G_x, G_y can be "related" by their least upper bound (denoted $G_x + G_y$) and greatest lower bound (denoted $G_x * G_y$). Cavallo refers to the minimal number of immediate refinements necessary to find G_x, beginning with the universal upper bound of the lattice (the least refined G-structure - the overall system), as the length(G_x). This number is given by n in the following sequence: $G_x = G_0 \preccurlyeq G_1 \preccurlyeq \ldots \preccurlyeq G_n = S$, where each G_i is an immediate refinement of G_{i+1}. In this work, the phrase "moving down" a lattice will refer to the action of simultaneously considering multiple sequences of immediate refinements, beginning with the universal upper bound of the lattice. The lattice structure establishes a hierarchy by which we can objectively consider, using this length measurement, the resolution levels that we may view a system from [3].

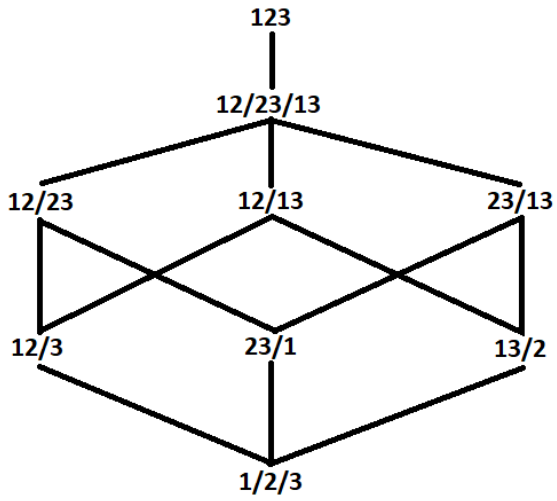

Figure 2-2: Lattice of G-structures of S (|V| = 3)

Example 2.6.1 The Hasse diagram of the lattice of G-structures of the overall system S defined in Example 2.2.1 is seen in Figure 2-2. 123 is the universal upper bound; 1/2/3 is the universal lower bound. The LUB of 23/13 and 12/3 is 12/23/13. The length(12/3) is 3, since a minimum of three immediate refinements are necessary from 123 to 12/3. An example of these immediate refinements (in order) is the following sequence: $12/3 \preccurlyeq 12/23 \preccurlyeq 12/23/13 \preccurlyeq 123$.

One purpose of representing a system by one of its structure systems is to view the system from a more refined perspective. The least refined structure of any overall system S is the system itself. At this level, the holistic relationship between all attributes is being considered, and the value of any attribute of S is assumed to be dependent on the values of all other attributes of S. The overall system is always the "universal upper bound" of the lattice defined on S_G because all structures in the lattice are refinements of it. The most-refined G-structure, denoted G_k, of S is the structure consisting of only and all possible subsystems of S defined on a single attribute - $|V_i| = 1$. At this level, each attribute is viewed independently of the other attributes and no interaction is considered. This structure is the "universal lower

bound" of the lattice in Figure 2-2 because it is a refinement of all structures in the lattice. Modeling an overall system using a G-structure is, essentially, assuming some knowledge regarding the interactions between the attributes of that system For example, when modeling a 3-attribute system using the G-structure 12/23, there is an emphasis placed on the importance of the relationships between attributes 1 and 2 and between attributes 2 and 3.

Any set of C-structures, including S_C, ordered by relation $\preccurlyeq$ is also a lattice; this makes sense, since any set of C-structures is also a set of G-structures. Every unique undirected graph defined on the set V is the primal graph of a unique C-structure. Given two C-structures $C_i, C_j \in S_C$, if $C_i \preccurlyeq C_j$, then every edge that is in $P(C_i)$ is also an edge in $P(C_j)$. If there exists no other $C_k \in S_C$, where $C_i \preccurlyeq C_k \preccurlyeq C_j$, then C_i is said to be an immediate C-structure refinement of C_j, in which case, the number of edges in $P(C_i)$ is exactly one less than the number of edges in $P(C_j)$. That is, as you "move down" the lattice of C-structures, each C-structure's primal graph is the previous C-structure's primal graph with a single edge removed [3].

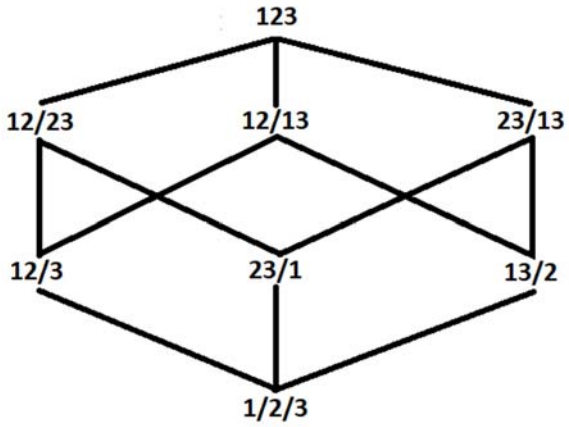

Figure 2-3: Lattice of C-structures of S (|V| = 3)

Example 2.6.2 The Hasse diagram of the lattice of C-structures of the overall system S defined in Example 2.2.1 is shown in Figure 2-3. When |V| = 3, the only G-structure that is not a C-structure is 12/23/13. This is because P(12/23/13) is the complete graph on 3 vertices, and there is only one maximal clique of that graph: 123. The C-structure associated with that primal graph is 123.

2.7 - Storage Requirement of systems and structure systems

As was mentioned in Section 2.2, a system can be defined as the four-tuple (V, Δ, dom, f). If we assume that being able to store a direct representation of a real-world system on a computer is a positive thing, whether this be for the sole purpose of having information or for future consideration and study, then it may be worth considering the resource demands associated with doing so. In terms of storage requirement, |V| is simply the number of system attributes, Δ grows linearly alongside V, and the number of domain mappings from elements in

V to elements in Δ grows linearly with V also. However, the number of mappings from elements of T (the set of all states) to the respective codomain ({0,1} for relational, [0,1] for probabilistic) that are required to define a system increases dramatically. Using V, Δ, and dom, one can infer all elements of T, yet the mappings of each of these states to elements in the codomain of f need to be specified. In a relational setting, this is declaring, for each state t ∈ T, whether that state can be observed (f(t) = 1) or cannot be observed (f(t) = 0). In a probabilistic setting, this is declaring, for each state, the probability that that state will be observed.

In smaller systems - say, less than 10 attributes (|V| < 10) -, the size of the domain of function f - that is, |T| - is not significant, at least by today's memory standards. Although it wasn't always the case, memory is currently very cheap. However, storing larger systems becomes significantly more demanding as |V| increases. This is shown by the following facts:

- If |V| increases by 1, |T| becomes |T| * dom(v_i), where v_i is the new attribute in V
- If |dom(v_i)| increases by 1, |T| becomes |T| + $\prod_{v_j \in V, v_j \neq v_i} |\mathrm{dom}(v_j)|$

For example, when |V| = 10, and each attribute is as simple as possible (binary), the total number of possible states that must be addressed is 2^{10} = 1024. With each additional binary attribute, this value doubles from the previous. To illustrate this dramatic growth, I will point out that there are more states in a system with 70 binary attributes, than there are believed to be stars in the observable universe. When defining and storing systems, the growth of |T| must be a concern. Therefore, the *storage requirement*, denoted ℂ(S), associated with storing a system S in memory is the total number of states of that system (|T|). Symbolically:

$$\mathbb{C}(S) = \prod_{v_i \in V} |\mathrm{dom}(v_i)|$$

If larger systems are to be stored in memory, one must consider modeling these systems in a way that requires addressing fewer states. This is possible through the use of structure systems. The *storage requirement* for any structure SS is defined as the total number of states that require addressing across all subsystems in SS. Symbolically:

$$\mathbb{C}(SS) = \sum_{i=1}^{m} |T_i|$$

Changes in the discretization of a real-world system would result in a change in the definition of that system. It may also result in changes in $\mathbb{C}(SS)$ for any structure $SS = \{S_1, \dots, S_m\}$ affected by the change. If the number of attributes were to increase by 1 ($V = V \cup \{v_{new}\}$), then the storage requirement for the altered structure $SS^{new} = \{S_1^{new}, \dots, S_m^{new}\}$ would differ from $\mathbb{C}(SS)$ in the following way:

$$\mathbb{C}(SS^{new}) = \mathbb{C}(SS) + \sum_{v_{new} \in S_i^{new}} |T_i| * (|dom(v_{new})| - 1)$$

If the number of possible values that an attribute v_y can take were to change by x, where x is any integer ($|dom(v_y)| = |dom(v_y)| + x$), then:

$$\mathbb{C}(SS^{new}) = \mathbb{C}(SS) + \sum_{v_y \in S_i^{new}} |T_j| * x$$

where T_j is the set of all states of the subsystem S_j and $V_j = V_i - \{v_y\}$.

As we move down the lattice of G-structures, the storage requirement of each G-structure does not monotonically decrease or increase in terms of "$\preccurlyeq$" - that is, for G-structures G_i, G_j, $G_i \preccurlyeq G_j$ does not imply that $\mathbb{C}(G_i) \leq \mathbb{C}(G_j)$ or $\mathbb{C}(G_i) \geq \mathbb{C}(G_j)$. Therefore, there are some G-structures that have a storage requirement greater than or equal to that of the least refined G-structure (the system itself), as shown by the following counterexample: in Example 2.6.1, $12/23/13 \preccurlyeq 123$, yet $\mathbb{C}(12/23/13) = 12 \geq 8 = \mathbb{C}(123)$. From the viewpoint of storage requirement, these G-structures should not be considered as prospective models of the overall system, because storing them is at least as demanding as storing the system itself, and doing so would have no benefit in terms of information, since these models cannot contain more information about the overall system than that which is in the overall system. However, if storage requirement is disregarded, one may still choose one of these models for other reasons, such as simplicity.

As we move down the lattice of C-structures, the storage requirement of each C-structure **does** monotonically decrease in terms of "$\preccurlyeq$" - that is, for two C-structures C_i, C_j, $C_i \preccurlyeq C_j$ implies that $\mathbb{C}(C_i) \leq \mathbb{C}(C_j)$. This is proven in the following section.

2.7.1 - Monotonicity of storage requirement on S_C

Assume two C-structures C_j,C_k of an overall system S, where C_j is an immediate C-structure refinement of C_k. That is, $P(C_k)$, with a single edge (v_a,v_b), where $v_a,v_b \in V$, removed, is $P(C_j)$. $\mathbb{C}(C_k)$ is equal to the sum of $|T_i|$ for each $S_i \in C_k$, while $\mathbb{C}(C_j)$ is equal to the sum of $|T_i|$ for each $S_i \in C_j$. In both cases, $|T_i|$ is $\prod_{\mathbf{v_j} \in \mathbf{V_i}} |\mathbf{dom}(\mathbf{v_j})|$**.** The removal of edge (v_a,v_b) from the primal graph of C_k will "break up" any subsystem $S_i \in C_k$, where $v_a,v_b \in V_i$, into two subsystems S_x, S_y, since the cliques in $P(C_k)$ representing these subsystems are now no longer cliques. $|T_i|$ remains the same for any $S_i \in C_k$, where $v_a \notin V_i$ **or** $v_b \notin V_i$, and thus the change from $\mathbb{C}(C_k)$ to $\mathbb{C}(C_j)$ is due only to the breaking up of those subsystems that contain both v_a **and** v_b.

The goal is now to show that, after breaking up any subsystem $S_i \in C_k$ containing both v_a and v_b, the sum of the total number of states in the subsystems S_x, $S_y \in C_j$, where $V_x = V_i - \{v_a\}$ and $V_y = V_i - \{v_b\}$, that formed as a result of breaking up S_i, is less than or equal to $|T_i|$. If this can be proven when a single subsystem in C_k is broken up by the removal of (v_a,v_b) from $P(C_k)$, then it must hold that $\mathbb{C}(C_k) \geq \mathbb{C}(C_j)$ when multiple subsystems in C_k are broken up by this removal. If S_x is not a subsystem of any other subsystem in C_j, then $S_x \in C_j$; likewise for S_y. In the case where S_x and S_y are both not in C_j, then it is obvious that $\mathbb{C}(C_j) \leq \mathbb{C}(C_k)$, since S_i is effectively removed from C_k. In the case where either S_x or S_y is not in C_j, it is also obvious that $\mathbb{C}(C_j) \leq \mathbb{C}(C_k)$, since S_i is effectively replaced by either S_x or S_y and $|V_x| = |V_y| = |V_i| - 1$. Removing an attribute from a system will always result in a decrease in the storage requirement of that system, since the domain of any attribute of a system is a set with a size of at least 2.

In the case where both $S_x \in C_j$ and $S_y \in C_j$, the inequality to be proven is: $|T_i| \geq |T_x| + |T_y|$

which equates to $\prod_{\mathbf{v_j} \in \mathbf{V_i}} |\mathbf{dom}(\mathbf{v_j})| \geq \prod_{\mathbf{v_j} \in \mathbf{V_x}} |\mathbf{dom}(\mathbf{v_j})| + \prod_{\mathbf{v_j} \in \mathbf{V_y}} |\mathbf{dom}(\mathbf{v_j})|$.

Note that $\prod_{\mathbf{v_j} \in \mathbf{V_x}} |\mathbf{dom}(\mathbf{v_j})|$ and $\prod_{\mathbf{v_j} \in \mathbf{V_y}} |\mathbf{dom}(\mathbf{v_j})|$

are equivalent to $\frac{\prod_{\mathbf{v_j} \in \mathbf{V_i}} |\mathbf{dom}(\mathbf{v_j})|}{|\mathbf{dom}(\mathbf{v_a})|}$ and $\frac{\prod_{\mathbf{v_j} \in \mathbf{V_i}} |\mathbf{dom}(\mathbf{v_j})|}{|\mathbf{dom}(\mathbf{v_b})|}$, respectively,

because, in terms of the storage requirement of S_i, removing an attribute from V_i, effectively divides out the size of the domain of the attribute that is removed from $\mathbb{C}(S_i)$.

The inequality then becomes:

$$\prod_{v_j \in V_i} |\mathrm{dom}(v_j)| \geq \frac{\prod_{v_j \in V_i} |\mathrm{dom}(v_j)|}{|\mathrm{dom}(v_a)|} + \frac{\prod_{v_j \in V_i} |\mathrm{dom}(v_j)|}{|\mathrm{dom}(v_b)|}$$

The following algebraic manipulation shows that this inequality is equivalent to the inequality $|\mathbf{dom}(\mathbf{v_a})| * |\mathbf{dom}(\mathbf{v_b})| \geq |\mathbf{dom}(\mathbf{v_a})| + |\mathbf{dom}(\mathbf{v_b})|$, which is a tautology when both $|\mathrm{dom}(v_a)|$ and $|\mathrm{dom}(v_b)|$ are greater than or equal to 2, which is always the case when working with system attributes as they have been defined.

$$\prod_{v_j \in V_i} |\mathrm{dom}(v_j)| \geq \prod_{v_j \in V_i} |\mathrm{dom}(v_j)| \left(\frac{1}{|\mathrm{dom}(v_a)|} + \frac{1}{|\mathrm{dom}(v_b)|}\right)$$

$$1 \geq \frac{1}{|\mathrm{dom}(v_a)|} + \frac{1}{|\mathrm{dom}(v_b)|}$$

$$1 \geq \frac{1}{|\mathrm{dom}(v_a)|}\left(\frac{|\mathrm{dom}(v_b)|}{|\mathrm{dom}(v_b)|}\right) + \frac{1}{|\mathrm{dom}(v_b)|}\left(\frac{|\mathrm{dom}(v_a)|}{|\mathrm{dom}(v_a)|}\right)$$

$$1 \geq \left(\frac{|\mathrm{dom}(v_b)|}{|\mathrm{dom}(v_a)| * |\mathrm{dom}(v_b)|}\right) + \left(\frac{|\mathrm{dom}(v_a)|}{|\mathrm{dom}(v_b)| * |\mathrm{dom}(v_a)|}\right)$$

$$1 \geq \frac{|\mathrm{dom}(v_a)| + |\mathrm{dom}(v_b)|}{|\mathrm{dom}(v_a)| * |\mathrm{dom}(v_b)|}$$

$$|\mathrm{dom}(v_a)| * |\mathrm{dom}(v_b)| \geq |\mathrm{dom}(v_a)| + |\mathrm{dom}(v_b)|$$

2.7.2 - Applications of the storage requirement measure

The primary consideration for this measure is storage-efficiency, a consideration similar, at least for probabilistic systems, to that first presented in P.M. Lewis' "Approximating Probability Distributions to Reduce Storage Requirements" [18]. Lewis was one of the first researchers to suggest workarounds to the fact that storing high order discrete probability distributions in memory was costly. This was especially true in 1959, when computer memory was nowhere near as inexpensive as it is today. However, it is no less relevant when studying larger systems. Rather than storing entire probability distributions, Lewis suggested storing

smaller marginal distributions and using them to approximate the actual probability distribution - possibly sacrificing accuracy for memory.

Up to this point, storage requirement has been defined as dimensionless. This unit of measurement is dependent on the setting (relational/probabilistic) and the means of actually storing f(t) for each state. For relational systems, the range of the characteristic function need only be a bit string of length $\mathbb{C}(S)$, where each bit corresponds to the value f(t) for all $t \in T$, while, for probabilistic systems, this range is a list of probabilities - the probability distribution over T. Storing a relational structure, requires storing a collection of bit strings, with $\mathbb{C}(SS)$ bits in total, while storing a probabilistic structure requires storing a collection of marginal distributions of the overall probability distribution, with $\mathbb{C}(SS)$ probabilities in total. Naturally, for relational systems and structures, the storage requirement could be measured in bits of memory. For relational systems, each probability is represented by a decimal number. Representing these decimal numbers using single-precision floating point numbers would require 32 bits (4 bytes) per value, while double-precision requires 64 bits (8 bytes). The choice of representation for probabilities is dependent on the precision of the most precise probability. For the sake of simplicity in this work, storage requirement will be regarded as unitless.

Example 2.7.2 The structure 1/23 contains subsystems S_1^P (Table 2-5) and S_{23}^P (Table 2-3), and serves as a model of the probabilistic system S^P (Table 2-1(b)). In this tabular representation, $\mathbb{C}(S^P)$ is equal to the total number of rows containing possible tuples in Table 2-1(b), i.e. $\mathbb{C}(S^P) = \mathbb{C}(123) = 8$. The storage requirement of 1/23, $\mathbb{C}(1/23) = \mathbb{C}(1) + \mathbb{C}(23) = 2 + 4 = 6$, is equal to the sum of the total number of rows containing tuples in Tables 2-3 and 2-5.

v_2	v_3	$f_{23}^R(t)$	$f_{23}^P(t)$
0	0	1	0.25
0	1	1	0.25
1	0	1	0.15
1	1	1	0.35

Table 2-3: Subsystem S_{23}

v_1	$f_1^R(t)$	$f_1^P(t)$
0	1	0.85
1	1	0.15

Table 2-5 Subsystem S_1

2.8 - Reconstructing an overall system from a structure system

In order to measure the amount of information about the overall system that is inferable by a structure model, the structure must first be converted into a comparable form, as is required by the information measures described and used in this thesis. This conversion process is known as "reconstructing" an overall system. The reconstruction procedure used differs between relational and probabilistic settings. The *reconstruction* of a structure SS will be denoted by r(SS). This reconstruction is an estimate of the actual overall system that is based on the information obtainable from the structure. It is very common for there to be more than one plausible reconstruction of a structure. In other words, there may be multiple systems whose corresponding subsystems are identical to those in SS. This set of possible overall systems is referred to as the *reconstruction family* of SS and is denoted by $\mathcal{R}$(SS). A system S is considered to be identifiable by SS if, from the constraints placed by the subsystems of SS, one can reconstruct S - that is, S = r(SS); when $|\mathcal{R}(SS)| = 1$, this is always the case. In both the relational and probabilistic settings, the identifiability of S from the reconstruction of SS implies certain facts about the relationships between the attributes of V (See Example 2.8.1).
"Reconstructability Analysis is viewed as the process of investigating the possibilities of reconstructing desirable properties of overall systems from the knowledge of corresponding properties of their various subsystems." [4]

Example 2.8.1 Assume that S^R (Example 2.2.1) and r(1/23) (Example 2.7.2) are equal. We can say the following: if we have data collected regarding both attribute 1 and pairwise data collected regarding the interactions of attributes 2 and 3, then we know everything there is to know about the relationship between all 3 attributes together. From this information, we can then state that attributes 1 and 2 occur independently of each other - likewise, for 1 and 3. Pairwise data between 12 and 13 is not needed to fully reconstruct S^R, implying that the relationships between these attributes are nonexistent, and thus, the attributes are independent. When SS = 12/23, and r(12/23) = S^R we can say that attributes 1 and 3 are conditionally independent, given 2, because their being together is not needed to reconstruct S^R, so long as pairwise information regarding 12 and 23 is available.

2.8.1 - Reconstructing relational structures using the relational join

Given a relational structure SS, an overall system r(SS) ∈ $\mathcal{R}$(SS), where SS is made up of valid subsystems of r(SS), can be found through repeated use of the relational join procedure. This is accomplished by performing a relational join between two subsystems in the structure, then joining that result with another subsystem in the structure, and so on for all other subsystems. If S is not identifiable by SS, then one or more states that are known to not be observable (f(t) = 0) in S, are said to be observable (f(t) = 1) in r(SS), thus r(SS) ≠ S. However, it should be noted that if S is not identifiable by SS, then it is impossible for a state that is known to be observable in S to be unobservable in r(SS), since SS is a collection of subsystems of S.

Given two subsystems S_i, S_j the result of performing the *relational join* between the two will be a system S_{ij} defined over the set of attributes $V_{ij} = V_i \cup V_j$. If $V_i \cap V_j \neq \{\}$, then if a state $t_i \in T_i$ is observed ($f(t_i) = 1$) in S_i and $t_j \in T_j$ is observed in S_j, then a state $t_{ij} \in T_{ij}$ will be observed in S_{ij} if (1) the measurement of the attributes of V_i in states t_i and t_{ij} are identical, (2) the measurement of the attributes of V_j in states t_j and t_{ij} are identical, and (3) the measurements of the attributes of $V_i \cap V_j$ in states t_i and t_j are identical. If $V_i \cap V_j = \{\}$, then (1) if a state $t_i \in T_i$ is observed in S_i, then a state $t_{ij} \in T_{ij}$ will be observed in S_{ij} if the measurements of the attributes of V_i in states t_i and t_{ij} are identical, and (2) if a state $t_j \in T_j$ is observed in S_j, then a state $t_{ij} \in T_{ij}$ will occur in S_{ij} if the measurements of the attributes of V_j in states t_j and t_{ij} are identical [5].

Example 2.8.2 The structure 12/23 contains subsystems S_{12}^R (Table 2-2) and S_{23}^R (Table 2-3). Table 2-6 is the tabular representation of the result of performing the relational reconstruction procedure on SS, where SS = 12/23. Notice that r(SS) is not equivalent to S^R (Table 2-1(a)). This means that S^R is not identifiable from SS and $|\mathcal{R}(SS)| > 1$.

v_1	v_2	v_3	$f^R(t)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Table 2-6: Relational reconstruction r({S_{12},S_{23}})

v_1	v_2	v_3	$f^R(t)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

Table 2-7: Relational reconstruction r({S_{12},S_{13}})

Example 2.8.3 The structure 12/13 contains subsystems S_{12}^R (Table 2-2) and S_{13}^R (Table 2-4). Table 2-7 is the tabular representation of the result of performing the relational reconstruction procedure on SS, where SS = 12/13. Notice that r(SS) = S^R, i.e. S^R is identifiable from SS.

2.8.2 - Reconstructing probabilistic structures using the probabilistic join and the Iterative Proportional Fitting Procedure (IPFP)

In a probabilistic setting, if $|\mathcal{R}(SS)| \neq 1$, then $|\mathcal{R}(SS)|$ is infinite because of the nature of probability distributions. Although there is controversy surrounding its use, the member of (SS) with maximum entropy (See Section 2.9.2) will be considered the best estimate of the overall probabilistic system. E.T. Jaynes, one of the most notable advocates for the use of the *maximum entropy member*, defines the maximum entropy principle as the belief that "when we make inferences based on incomplete information, we should draw them from that probability distribution that has the maximum entropy permitted by the information that we do have." [15]

This is supported by the condition that if all members of (SS) are possible overall systems, then we should choose the reconstruction that contains the most entropy, which equates to choosing the reconstruction that contains the least amount of information. This is because, in all other scenarios where the actual distribution is not this maximum entropy distribution, we are assuming additional information by using any other plausible distribution. If we were to select, say, the minimum entropy member of $\mathcal{R}(SS)$, and the actual distribution was anything but, then we would be biased in assuming more information from the reconstruction about the overall system than that which is actually available. In their papers on reconstructability analysis, Cavallo & Klir also advocate for the maximum entropy principle; they use the phrase "unbiased reconstruction" to allude to the maximum entropy element of $\mathcal{R}(SS)$ [5].

The goal of the probabilistic reconstruction procedure is to find the member of $\mathcal{R}(SS)$ with maximum entropy. Finding this member is essentially an optimization problem, where the search space is $\mathcal{R}(SS)$ and the objective function to be maximized is the formula for Shannon's entropy. The problem of finding this element of $\mathcal{R}(SS)$ is analogous to maximum likelihood estimation in statistics - a method of estimating parameters of a statistical model using a set of

observations. Numerous optimization techniques have been employed for solving this problem. In this work, the *Iterative Proportional Fitting Procedure* (IPFP) is used - specifically, the IPFP as it was adapted for probabilistic systems by Cavallo and Klir [5]. The procedure can be applied to a set of marginal distributions to produce the maximum entropy member of the set of overall distributions with those marginal distributions. This can be extended to structure reconstruction, since a probabilistic structure is, effectively, a set of marginal distributions of an overall distribution (the range of the behavior function of an overall system). This procedure makes use of the probabilistic join, the probabilistic extension of the relational join described in the previous Section 2.8.1.

The following is a modified version of the *probabilistic join* procedure that receives as input two distributions. In this work, these distributions are the ranges of the behavior functions of two systems S_i and S_j, where $V_i \cap V_j \neq \{\ \}$. The output is a system S_{ij}, where $V_{ij} = V_i \cup V_j$ and $f(t_{ij})$, where $t_{ij} \in T_{ij}$, is equal to $\mathbf{f(t_i) * (f(t_j) / \sum_{t_j' \in T_j'} f(t_j'))}$, where (1) $t_i \in T_i$, where the measurements of the attributes of V_i in states t_i and t_{ij} are identical, (2) $t_j \in T_j$, where the measurements of the attributes of V_j in states t_j and t_{ij} are identical, and (3) $T_j' \subseteq T_j$ contains all states t_j' in which the measurements of the attributes of $V_i \cap V_j$ in states t_i and t_j are identical.

Example 2.8.4 Table 2-8 is the result of taking the probabilistic join of S_{12} (Table 2-2) and S_{23} (Table 2-3). f(000) in Table 2-8 is equivalent to the probability of the state of system S_{12} where $v_1 = 0$ and $v_2 = 0$, i.e. $f_{12}(12 = 00)$, multiplied by the conditional probability that $v_3 = 0$, given that $v_2 = 0$, in S_{23}, i.e. $f_{23}(3 = 0 \mid 2 = 0)$. That is, $f(000) = f_{12}(12 = 00) * f_{23}(3 = 0 \mid 2 = 0) = 0.35 * (0.25 / (0.25 + 0.25)) = 0.35 * (0.25 / 0.5) = 0.175$. Another example: $f(011) = f_{12}(12 = 01) * f_{23}(3 = 1 \mid 2 = 1) = 0.5 * (0.35 / (0.15 + 0.35)) = 0.5 * (0.35 / 0.5) = 0.35$.

v_1	v_2	v_3	$f^P(t)$
0	0	0	0.175
0	0	1	0.175
0	1	0	0.15
0	1	1	0.35
1	0	0	0.075
1	0	1	0.075
1	1	0	0
1	1	1	0

Table 2-8: Probabilistic Join of S_{12} and S_{23}

The IPFP receives a probabilistic structure SS as input and outputs the maximum entropy element of $\mathcal{R}$(SS). The algorithm, similar to Algorithm 4 in [5], is as follows:

1. Create an overall system S with the range of the behavior function initialized to the uniform probability distribution - f(t) = 1/|T| for all t ∈ T
2. Input S and the first subsystem $S_1 \in$ SS to the defined probabilistic join procedure
3. Overwrite S with the results of step 2
4. Repeat steps 2 - 3 for each subsystem in the structure
5. Repeat steps 2 - 4 until the range of the behavior function of S converges, given some predefined precision. In this work, IPFP is run until the sum of the absolute values of the differences in probabilities for each f(t) is below 0.0001 when comparing the previous iteration's probability distribution to the current iteration's probability distribution.

Technically, the distribution of the initial S (step 1) need not be the uniform distribution; any distribution will work. Essentially, in IPFP, small changes are made to this initial distribution, so that the resulting overall distribution is consistent with each subsystem - i.e. each subsystem in the structure is a subsystem of the overall system with this overall distribution. However, this process needs to repeat until the distribution converges, because inconsistencies are introduced with structures that are considered cyclic (See [4]).

2.9 - Information in a system

In addition to the storage requirement of a structure SS, the amount of information regarding the overall system S that can be inferred by the constraints placed by the subsystems in SS need also be considered when evaluating its acceptability. The following section will discuss information and how it is measured in a relational system vs. a probabilistic system, for the purpose of comparing the information in r(SS) to the information in S.

2.9.1 - Information in a relational system

Information in a relational system S is defined as a measure of the amount of constraint that need be placed on a system with no information to result in S. The relational system with no information is the system in which all potential states of that system are observable. Information is acquired when it becomes known that a state of that system will never occur.

Therefore, I am defining the amount of information in S as the number of states that are known to never occur. An overall system will always have an amount of information that is greater than or equal to that of the reconstruction of a structure of that overall system. This is because: for any state t, if f(t) = 0 in r(SS), then f(t) = 0 in S, but if f(t) = 0 in S, then f(t) is not necessarily 0 in r(SS). Therefore, I am defining the amount of information loss that occurs as a result of using r(SS) as a model of S, as the number of states in r(SS) that are inaccurately said to be observable, when, in fact, they are not actually observable. I refer to this measure as the "*number of Incorrectly Predicted States*" (IPS) of that structure. Symbolically,

$$\text{IPS}(r(SS)) = \sum_{t\in T, f_r(t)=1} f_r(t) - \sum_{t\in T, f(t)=1} f(t)$$

where f_r is the behavior function of r(SS). IPS is always nonnegative.

Given two structures SS_i and SS_j, if $IPS(r(SS_i)) > IPS(r(SS_j))$, then the amount of information that can be inferred about the overall system from SS_j is more than that which can be inferred from SS_i. This measure allows for the objective comparison of relational structure reconstructions from an information-loss standpoint. IPS is effectively the difference between the amounts of constraint found in S and r(SS). Constraint, in a relational system, is related to the number of states that are known to not occur, and IPS is defined as a measure of how much of the constraint that is present in S is not captured by the reconstruction of SS. If r(SS) captures all of the constraint in S, then r(SS) = S and IPS(r(SS)) = 0.

Example 2.9.1 Given SS = 12/23 (Example 2.8.2), IPS(r(SS)) = 1. When observing Tables 2-6 (r(12/23)) and 2-1(a) (S^R), it is clear that $f_r(100) \neq f(100)$, hence IPS(r(12/23)) = 1. Given SS = 12/13 (Example 2.8.3), IPS(r(SS)) = 0, as seen in Table 2-7, i.e. r(12/13) = S^R and IPS(r(12/23)) = 0.

2.9.2 - Shannon's entropy & information in a probabilistic system

In [22], Claude Shannon introduced an objective measure of uncertainty within a probability distribution. Given a discrete probability distribution P(X) over a discrete variable X with possible values $\{x_1, x_2, \dots, x_n\}$, *Shannon's entropy*, denoted H(X), is given by the formula **H(X) =** $-\sum_{i=1}^{n} \mathbf{p}(\mathbf{x_i}) \log_b \mathbf{p}(\mathbf{x_i})$**.** In this work, b = 2, in which case entropy is measured in bits. Shannon's entropy is intended to be a measurement of the amount of uncertainty there is in a given distribution, i.e. the amount of uncertainty surrounding which value $x_i \in X$ will occur. A

probability distribution in which one state will occur with probability of 1, while all other states occur with probability 0, contains the least amount of entropy (H(X) = 0); this is synonymous with saying that this distribution contains the most information regarding the value of variable X. This is intuitive, since knowing with complete certainty which value X will take is more informative than all cases with lesser certainty, since we are guaranteed to correctly predict the value of X. The uniform distribution contains the maximum entropy ($\log_2(1/n)$ bits of entropy) of all possible probability distributions over n possible values, i.e. the uniform distribution contains the least amount of information. The objective amount of information present in a probability distribution is calculated relative to this maximum entropy measurement. Specifically, it is the difference in uncertainty between the maximum entropy distribution, $H(X_{max})$, and the distribution, H(X), in question, i.e. $H(X_{max}) - H(X)$.

This can naturally be applied to probabilistic behavior functions. Shannon's entropy is used to measure the amount of uncertainty there is in the state of a system. Similar to relational systems, the information in r(SS) will never be greater than the information in S. Therefore, the entropy of r(SS), denoted H(r(SS)), minus the amount of information that S contains, denoted H(S), gives the amount of information that is lost if r(SS) were used as a model of S. The entropy of a system is also referred to as unresolved information.

Example 2.9.2 Given SS = 12/23 (Example 2.8.2), the amount of uncertainty in r(12/23) is 2.381 bits. The amount of uncertainty in S^R (Table 2-1(b)) is 2.183 bits. The amount of uncertainty in the uniform distribution over 8 states is 3. Therefore, the amount of information in r(12/23) is 0.619, while the amount of information in S^R is 0.817, thus the amount of information loss that would occur if we were to approximate S^R using r(12/23) is 0.198, which is equivalent to both 2.381 - 2.183 and 0.817 - 0.619.

Information monotonically decreases as you move down a lattice defined on any set of G-structures. That is, given two G-structures G_x, G_y where $G_x \preccurlyeq G_y$, it is always the case that $IPS(r(G_x)) \geq IPS(r(G_y))$, for relational systems, and the entropy of the maximum entropy member of $\boldsymbol{\mathcal{R}}(G_x)$ will always be at least the entropy of the maximum entropy member of $\boldsymbol{\mathcal{R}}(G_y)$, for probabilistic systems ($H(r(G_x)) \geq (H(r(G_y))$. This is because the act of taking the refinement of a G-structure, effectively, removes constraints on the reconstructed system, which in turn reduces the amount of information in that reconstructed system.

2.10 - Acceptability of structure systems

The number of possible G-structures, $|S_G|$, of an overall system increases aggressively as the number of system attributes increases, as seen in Table 2-9 [16].

$\|V\|$	1	2	3	4	5	6	7
$\|S_G\|$	1	2	9	114	6,894	7,785,062	2,414,627,396,434

Table 2-9: Number of G-structures when |V| is [1-7]

An acceptable structure of an overall system will meet some set of predefined acceptability criteria. Acceptability criteria used to evaluate G-structures in this thesis are discussed in Section 4.3. Brute-force techniques for finding the most acceptable model involve creating the entire lattice of G-structures and evaluating each G-structure. This becomes infeasible as |V| grows, especially if the process of evaluating a G-structure's acceptability is inefficient. To combat the poor scaling that comes with this problem, a search heuristic, known as a Genetic Algorithm, has been designed. The goal of this heuristic is to search the set of all meaningful structure models, without the need for exhaustively generating and evaluating each, and produce the optimal (most acceptable) model(s). The following chapter discusses this class of algorithms.

Chapter 3

The Genetic Algorithm

The purpose of this chapter is to introduce, identify, and describe the theoretical components of a genetic algorithm, provide example implementations of these components that have been proven to be successful in practice, and compare genetic algorithms to more conventional search and optimization techniques.

3.1 - Introduction to and brief history of genetic algorithms

Numerous classes of algorithms, once considered to be poor alternatives to conventional techniques, have found themselves amid a great resurgence, largely because of the exponential growth of the computational capacities of modern processors over the 30 years prior to the new millennium. Evolutionary algorithms (EAs) - optimization algorithms inspired by biological evolution - are among this set of classes. This type of algorithm sparked interest in the 1980s and 1990s, only to experience a decade-long hiatus while more deductive methods were pursued. However, within the last 10 years, EAs have begun to receive significant attention once more. The application of EAs in other popular fields of computer science and mathematics, such as machine learning, has resulted in the expansion of the number of EAs (e.g. Neural Networks + Evolutionary Algorithms = Neuroevolution). Genetic algorithms (GAs) are arguably the most influential and well-known type of EA.

The “Genetic Algorithm” was first proposed by John Holland in his ground-breaking book, *Adaptation in Natural and Artificial Systems* [14]. Holland, an avid systems researcher, specialized in adaptive systems. An adaptive system is similar to a static system (defined in Chapter 2) with the extension that the definition of an adaptive system can change over time. Holland drew inspiration for the genetic algorithm from his observations of the evolution of biological populations by means of natural selection, and sought to mathematically model the genetic adaptations by which the evolutionary process occurs in a population. By the mid-1970s, artificial evolution was already a recognized optimization technique and, through

Holland's efforts, GAs soon become the most notable type of EA. However, due to the computational limitations of this time, GA research was theoretical until around the mid-1980s, when implementation of efficient GAs became much more feasible.

A GA is what is known as a metaheuristic. Just as an algorithm is a problem-independent solution to a general problem, a metaheuristic is a problem-independent framework used to develop a problem-specific heuristic. A heuristic, and by extension a metaheuristic, is any method of finding a solution to a problem that is not guaranteed to provide the optimal solution, but will typically find a solution that is considered acceptable. Heuristics are often used when finding an optimal solution to a problem using classic algorithms is inefficient, in which case accuracy is sacrificed for an increase in speed.

The following is a list of the various components and operations that define a genetic algorithm:

- Representation of solutions
- Population size & Initialization procedure
- Fitness (Evaluation) function
- Selection protocol
- Breeding strategy
- Replacement protocol
- Repetition & Termination conditions

Sections 3.2 - 3.8 of this chapter discuss these elements. Section 3.9 summarizes John Holland's schema theorem; this is Holland's explanation for why GAs are successful. In Section 3.10, techniques for mapping solutions to a genetics-friendly representation are given. Problems for which GAs outperform conventional algorithms are discussed in Section 3.11.

3.2 - Representation of solutions

Given a problem and the set of all possible solutions to that problem, denoted $\mathbb{S}$, the goal of a genetic algorithm is to search $\mathbb{S}$ and find the optimal solution to that problem, where this optimality is determined by a predefined measure of solution acceptability (See Section 3.4). The first step in accomplishing this goal is to represent all solutions in a general and manageable form. Each solution is slightly different from another solution and these

distinctions are what may make one solution a better or worse solution than another. The representation of a solution should capture these distinctions in a way that is consistent for all solutions. In other words, the definition of a solution must be standardized. This is accomplished by defining a function to serve as the mapping between $\mathbb{S}$ and the set containing representations for members of $\mathbb{S}$, denoted $\mathbb{G}$. This function $M: \mathbb{S} \rightarrow \mathbb{G}$ is known as the encoding function and is, in many cases, a bijection - that is, a unique solution has a unique representation and vice versa. Therefore, if M is a bijection, then $M^{-1}(M(\mathtt{I})) = \mathtt{I}$, for every $\mathtt{I} \in \mathbb{S}$. Regardless of the bijectivity of the mapping, the encoding and decoding functions should be deterministic to ensure that a solution will always map to the same representation and vice versa.

In biological terms, every organism has a unique set of traits. Given a trait (e.g. hair color, eye color, etc.), the expression of that trait is known as the trait's *phenotype* (blue, brown, etc.), while the genetic material responsible for this expression is the trait's *genotype* (DNA, RNA). Because of the biological connection, $\mathbb{S}$ is sometimes referred to as the *phenotype space* and the elements (solutions) of this space are referred to as *phenomes* (or *individuals*), while $\mathbb{G}$ is sometimes referred to as the *genotype space* and elements of this space are known as *genomes*. A phenome is defined by a combination of variants (*alleles*) of its measurable attributes (*genes*). Using a genomic representation of phenomes allows for the use of simple, problem-independent techniques for altering and combining gene-variants of phenomes (See Section 3.6).

The most common method of representing a phenome is by a bit string, although other representations do exist. When using bit string representations, $\mathbb{G}$ is the set of all possible bit strings of length n, where n is the number of genes (attributes) defined for any phenome. Each gene is represented by a bit in a bit string, and the value of that bit signifies which of the two variants of that gene is held by a given solution. A common implementation of bit string representation is to define a number of attributes that any solution could have and have each bit's value signify whether an individual solution has (1) or does not have (0) one of these attributes.

Example 3.1.1 Assume a problem where a solution can have between 0 and 3 attributes - a_1, a_2, and a_3. $\mathbb{S}$ consists of all possible collections of these attributes; each solution can either *have* or *not have* each attribute. $\mathbb{G}$ consists of all possible bit strings of length 3, where the value of the ith bit of a bit string signifies that the solution that maps to this bit string either has attribute a_i (1) or does not have attribute a_i (0). The genomic representation of: 1) the solution that has attributes a_1 and a_3 is 101, 2) the solution that has only attribute a_2 is 010, 3) the solution that has none of these attributes is 000.

3.3 - Population & Initialization

The first step in the execution of a genetic algorithm is to initialize the *population* of solutions. The population $\mathbb{P}$ is a subset of the search space $\mathbb{S}$, with the additional stipulation that multiple copies of a solution are allowed within $\mathbb{P}$. This population is the medium by which evolution occurs. A solution itself does not change while it is in $\mathbb{P}$. Instead, the population as a whole is a dynamic entity - consisting of static elements - that changes over time. Minimally, the parameters of a population need only be its size ($|\mathbb{P}|$) - this is the number of (not necessarily unique) solutions that, at any given time, are in $\mathbb{P}$. During execution of a GA, this size is typically kept constant.

The population is first initialized by creating $|\mathbb{P}|$ solutions. This initial population is known as the first *generation* of individuals. If no information is known about what characteristics/traits the optimal solution may have, then the population is randomly generated; that is, $|\mathbb{P}|$ random genomes are created and decoded. If some information is known about what region of the search space the optimal solution may be in, then the initial population can be generated with a bias towards that region. In this case, the algorithm has an advantage because it is essentially given “hints” regarding where to look, potentially reducing the overall computational burden of the search.

The *diversity* of the population is a term used to describe the amount of variance that the population has. I do not explicitly mention an objective diversity metric in this work because, while there are a number of possibilities to choose from, each involves terminology (e.g. variety, spread) whose definition(s) are subjective and, therefore, may vary across different settings. Regardless, population diversity must still be considered. In the

aforementioned cases (no prior information vs. some prior information), a randomly initialized population would be more diverse than a biased initial population. Solutions in the former population are randomly spread across the search space, while solutions in the latter population are randomly spread across a subset of the search space and would thus be more similar. If a population consists solely of the same solution, then that population is said to have no diversity [10,20].

3.4 - Evaluation

The ultimate goal of a GA is to find an optimal solution to a problem. An optimal solution is an $\mathtt{I} \in \mathbb{S}$, for which there exists no other $\mathtt{J} \in \mathbb{S}$ that is a "better" solution than $\mathtt{I}$ is. This implies that an objective measure of the "goodness" of a solution is necessary, so that all solutions are objectively comparable. This measure, referred to as a solution's *fitness*, is obtained by inputting a solution to a *fitness function* $\mathrm{F}: \mathbb{S} \rightarrow \mathbb{V}$, where $\mathbb{V}$ is a set of possible fitness scores. In general, if $\mathtt{I}$ is considered a better solution than $\mathtt{J}$, based on a predefined measure of goodness, then $\mathrm{F}(\mathtt{I}) > \mathrm{F}(\mathtt{J})$. The fitness function of a GA is also the objective function of the underlying optimization problem. Following population initialization, all solutions in the population are evaluated, i.e. the fitness of each is calculated [10,20].

3.5 - Selection

Following the evaluation of the members of $\mathbb{P}$, a set $\mathbb{M} \subseteq \mathbb{P}$ is chosen as the mating pool of the current generation. Pairs of solutions (*parents*) in this pool have a chance to reproduce and create new solutions (*offspring)*, passing on their gene-variants to solutions in the next generation. The method by which the members of $\mathbb{M}$ are chosen is known as the selection protocol. In general, the solutions with the highest fitness scores (best-performing) should have the highest chance of being selected, while the solutions with the lowest fitness scores (worst-performing) should have the lowest chance. The primary assumption behind this is that the reason a highly-rated solution is rated highly is because it contains a combination of gene-variants that makes it a better solution to the problem, relative to other solutions. By passing these variants to future solutions, the hope is that the performance of the population, as a whole, will improve and the search will move towards finding the optimal solution. The number of times that a solution in the population is selected to occupy a spot in the mating pool is determined by the selection protocol used; this number can be greater than 1.

There are multiple selection protocols that have been developed and tested in practice. Assuming that the desired size of the mating pool is $|\mathbb{M}|$, Table 3-1 contains a description of four well-known selection protocols used to select $|\mathbb{M}|$ solutions [10,20].

Truncation	The $|\mathbb{M}|$ best-performing individuals are selected.
Tournament	An additional predefined integer $\mathbb{T}$ is specified, where $\mathbb{T} \in [2,|\mathbb{P}|]$. $\mathbb{T}$ Individuals are randomly sampled from the population; this set of individuals is known as a "tournament". The best-performing individual in the tournament is placed in the mating pool. This is repeated $|\mathbb{M}|$ times.
Roulette Wheel (a.k.a Fitness proportional)	The probability that an individual $\mathrm{I} \in \mathbb{P}$ is selected is $\mathrm{F}(\mathrm{I}) / \sum_{\mathrm{X} \in \mathbb{P}} \mathrm{F}(\mathrm{X})$, i.e. the probability is proportional to its fitness relative to the fitnesses of all other solutions in the population. This is a stochastic distribution from which $|\mathbb{M}|$ individuals are selected (with replacement).
Stochastic Universal Sampling	Assign each individual an amount of "territory" on a number line (0 to 1) that is proportional to the individual's fitness relative to the sum of the fitnesses of all members of $\mathbb{P}$. Divide this line into $|\mathbb{M}|+1$ equally spaced sections. If the border separating two of these sections falls within an individual's territory, then that individual is selected.

Table 3-1: Common selection protocols

Although it may seem counterintuitive, it is not always beneficial to select only the best-performing solutions. Truncation selection does not always work well because of the higher chance of premature convergence. Ideally, the goal of a GA is for the population to eventually *converge* on the optimal solution - that is, the population contains only this solution. *Premature convergence* occurs when a population's diversity drops to inescapable levels before the most optimal solution is found. From an optimization standpoint, the goal of a genetic algorithm is to discover all "peaks" of the objective (fitness) function defined over $\mathbb{S}$, and "climb" these peaks in parallel. If the algorithm converges around a sub-optimal peak, then this convergence is considered premature. It is often necessary to allow "bad" solutions to propagate their gene-variants, as these solutions may contain some "good" variants that are overshadowed by

especially “bad” variants. Doing this may preserve population diversity until $\mathbb{S}$ can be properly skimmed and all peaks can be found.

3.6 - Breeding

In a GA, breeding is the two-step process by which new solutions are formed. The first step, crossover, results in a set of new solutions (known as offspring), while the second step, mutation, involves making minor changes to the offspring solutions in this set. The total number of offspring that will be created during crossover can be defined either statically or dynamically. In the former case, the number of offspring is defined prior to algorithm execution, while, in the latter case, a pair of individuals in the mating pool are bred together based on a predefined probability - the “rate of crossover”. In most GAs, crossover occurs between two parents and results in the formation of two offspring, though it is not uncommon to see one-child reproduction. Crossover and mutation are operations performed on the genomes of the selected solutions.

3.6.1 - Genetic Crossover (Recombination)

Crossover (a.k.a. recombination) is a binary operation that receives two parent solutions as input and outputs two offspring solutions. The purpose of crossover is to create offspring that contain a combination of their parent’s features, with the hope of producing at least one child that holds a desired combination of features. This concept is similar to that of selective breeding, which humans have successfully practiced for millennia through the domestication of plant and animal species. With the exception of simpler organisms, two-parent sexual reproduction is the dominant driving force of natural evolution. In this work, crossover refers to this two-parent type.

There is a variety of crossover techniques that have been proposed, and a description of those most prominent in the literature is given in Table 3-2 [10,20]. In the following, assume crossover is being performed on two parent genomes G_i and G_j, where $\mathtt{I},\mathtt{J} \in \mathbb{M}$, $M(\mathtt{I}) = G_i$, and $M(\mathtt{J}) = G_j$, and results in two offspring solutions, whose genomes are denoted by O_1 and O_2.

Single-point	A random point, separating two genes on G_i and G_j, is chosen. O_1 receives the alleles for the genes on one side of this point from G_i and the alleles for the genes on the other side from G_j - vice versa for O_2.
Multi-point	A set of multiple random points on G_i and G_j are chosen. From left to right, O_1 (O_2) receives the alleles of the genes of G_i (G_j) until the first point. From this point to the next, the alleles of the genes of G_j (G_i) are then assigned to the corresponding genes in O_1 (O_2). This process is repeated for all pairs of points, alternating between alleles from G_i and G_j.
Shuffle	Single-point or multi-point crossover with two additional steps: randomly shuffle the order of the genes before crossover is performed, and undo this shuffling afterwards.
Reduced Surrogate	Single-point or multi-point crossover, where a point is possible only if it separates adjacent genes that each have different alleles in G_i and G_j.
Uniform	For each gene in G_i, O_1 has a 50% chance of inheriting that gene's allele from G_i and a 50% chance of inheriting it from G_j. If a gene in O_1 receives its allele from G_i, then the same gene in O_2 receives its allele from G_j - vice versa.

Table 3-2: Common crossover techniques

The superior method of crossover is very much problem-specific. However, there are biases associated with each that should be noted. Positional bias is the tendency for the alleles of genes that are near each other on a genome to remain together in future offspring. Single-point crossover is positionally biased because adjacent genes are rarely "split up" when compared to genes on opposite ends of a genome. Positional bias in single-point or multi-point crossover can be reduced by using shuffle crossover, while positional bias can be completely eliminated by using uniform crossover. Having positional bias is not necessarily a bad thing; some genes may be closely related, and thus, should not be split up. One must also account for the distributional bias associated with a crossover method. Distributional bias is the tendency for offspring to contain unequal amounts of genes from each of its parents. For example, single-point crossover has a higher distributional bias than uniform crossover. Like positional bias, distributional bias is not necessarily good or bad to have, and can be used to a GA developer's advantage [10,20].

3.6.2 - Mutation

Mutation is a unary operation that receives the allele of a single gene as input and outputs another allele of that gene. Mutation is a random, unbiased operation that is used to alter the offspring solutions created during crossover - usually, very slightly. The most common type of mutation, bit flipping, is implemented alongside a bit string genomic representation scheme. A bit-flip mutation is performed on a gene by flipping the bit that corresponds to that gene. Mutation has a chance of occurring on all genes of all offspring. The "rate of mutation" is the probability that, given a gene, the allele of that gene will be changed to another of that gene's alleles. Mutation, like crossover, is inspired by evolutionary biology and plays a significant role in the success of a genetic algorithm. It introduces variety into the mating pool, which aids in preventing premature convergence and, when paired with crossover, ensures that the search space is connected and all solutions are discoverable, given sufficient time [10,20].

3.7 - Replacement

After breeding, the newly created offspring are inserted into the population. The chosen replacement protocol is dependent on the amount of offspring produced during crossover. A *generational* replacement strategy results in an entirely new population consisting of the offspring produced by the previous population. A *steady-state* replacement strategy involves replacing a number - less than $|\mathbb{P}|$ - of the worst-performing solutions in $\mathbb{P}$ with the offspring solutions. Steady-state replacement is elitist in nature, since the best-performing solutions in $\mathbb{P}$ will also be a part of the next generation. Like selection protocols, replacement protocols may also be stochastic, in which case each member of $\mathbb{P}$ is assigned a probability of being replaced that is inversely proportional to its relative fitness. However, most GAs implement deterministic replacement because, unlike with selection protocols, the use of a stochastic replacement protocol, in most cases, does not result in a significant increase in algorithm success or efficiency [10,20].

3.8 - Repetition & Termination

Following replacement, the evolutionary process (Evaluation → Selection → Crossover → Mutation → Replacement) repeats with the new population. Every succeeding population is a new generation and each evolutionary cycle is referred to as one iteration of the GA. Over

time, a well-developed GA will move closer to finding the solution with the optimal fitness. However, the GA provides no guarantee that the best solution in $\mathbb{P}$, at any time during algorithm execution, is the optimal solution in $\mathbb{S}$. The strongest statement that can be made is that the best solution found by the GA is the best solution that has been found *so far*. Additional terminating conditions may be required, since the probabilistic nature of the GA does not guarantee that the population will converge on the optimal solution in a reasonable amount of time [10,20]. It is common practice to implement one of the following stopping conditions:

- A maximum number of iterations has been reached
- A period of real-time has passed
- A number of iterations have occurred with average overall fitness improvement below a threshold value
- A given period of real-time has passed with average overall fitness improvement below a threshold value
- A chosen measure of population diversity falls below an acceptable threshold

3.9 - Holland's Schema Theorem

To this day, there is no widely accepted proof of a GA's ability to search for an optimal solution; their stochastic nature makes them particularly difficult to analyze. *Holland's Schema Theorem* [14] - originally referred to as the "fundamental theorem of genetic algorithms" - was proposed by John Holland as an attempt to explain the success of GAs. Since its release, the theorem has been disputed, but it is worth considering, nonetheless.

The theorem states that a GA is able to search multiple regions of a search space simultaneously because the genome of a single individual belongs to numerous regions of that search space. Holland explains this with the help of the "wildcard" symbol, denoted by *. Using a bit string representation scheme, * may be used alongside 0 and 1 when defining genomes; its presence signifies that the allele of the gene corresponding to its position in a genome could be any one of that gene's alleles, but which one it actually is is unknown. For example, given a bit string representation scheme, the string 1**01 is the genome of any solution that has the attributes represented by the first and fifth bits and does not have the fourth attribute; the presence of the second and third attributes is unknown. 1**01 is an example of a *schema* - a

representation of a region of $\mathbb{S}$. Specifically, 1**01 is the region containing any individual whose genomic representation is either 10001, 10101, 11001, or 11101. Any one individual occupies many different regions of $\mathbb{S}$ in parallel - that is, an individual's genome is a part of multiple schemata. Therefore, a set of solutions (such as $\mathbb{P}$) can cover wide areas of $\mathbb{S}$.

Searching $\mathbb{S}$ requires a balance between *exploration* and *exploitation*. The entire space should be explored, moving from region to region, and occasionally a region should be exploited, or further analyzed, to find those well-performing solutions within it. Exploration is conducted through crossover and mutation. When crossover is performed on two solutions, the resulting offspring are a part of a collection of regions in $\mathbb{S}$ that each of their parents were in. When a gene of a solution is mutated, that solution is removed from some regions of $\mathbb{S}$ and becomes a part of other regions. Both operators output a solution(s) that occupies different regions of $\mathbb{S}$ than the input solution(s) does. Exploitation is conducted through selection. Given a set of solutions, each occupying various regions of $\mathbb{S}$, those that are considered better solutions are given a higher chance of reproducing. If a solution is selected to be bred, then the regions that this solution occupies are being exploited. This is because the offspring that this solution is a parent of will also be in some of the regions that the parent is in. Selection ensures that "good" gene-variants remain in the population, and that the regions containing solutions with these "good" variants are exploited.

3.10 - Constraint Handling

GAs are useful for a vast assortment of search and optimization problems. However, for some problems, the validity of a solution may be contingent on that solution satisfying a set of predefined constraints. An invalid solution is a solution in $\mathbb{S}$ that does not satisfy these constraints, yet maps to a (valid) genome in $\mathbb{G}$. There are numerous techniques used to incorporate predefined constraints into the evolutionary process, and these techniques are typically filed under the title '*Constraint Handling*'.

Eiben and Smith [10] classify search/optimization problems into three classes: Free Optimization Problems (FOPs), Constraint Satisfaction Problems (CSPs), and Constrained Optimization Problems (COPs). FOPs are problems for which the optimal solution is the "best" solution in $\mathbb{S}$ based on an objective measure of "goodness"; problems in the preceding sections

of this chapter were assumed to be of this class. CSPs are problems for which an optimal solution is a solution that satisfies all predefined constraints. COPs are the most common type of search/optimization problem. This class consists of problems for which the optimal solution is the solution that both satisfies all constraints and is objectively optimal. Most real-world optimization problems are COPs to some degree, e.g. a combinatorial optimization problem is known as a discrete COP. A discrete COP is a problem with solutions consisting of discrete traits - that is, each trait has a finite number of possible expressions. The remainder of this section will describe both indirect and direct constraint handling techniques for discrete COPs.

3.10.1 - Indirect vs. Direct constraint handling

Indirect constraint handling occurs *prior to* execution of the GA and involves transforming the constraints into optimization objectives, i.e. the underlying COP is changed to an FOP. A commonly used indirect technique is to institute a penalty function that modifies an individual's fitness if that individual is invalid. Examples of penalty functions include extinctive - an invalid solution's fitness is lowered to the point that it would almost never be selected - and distance-based - an invalid solution's fitness is lowered by a value proportional to some measure of its invalidity.

Direct constraint handling occurs *during* the evolutionary process. This involves enforcing the constraints in all members of the population. One direct constraint handling technique is to initialize a population of valid solutions and define crossover and mutation operations that output valid solution(s), thus ensuring an entirely valid population across all generations. Another direct approach is to "repair" an invalid solution by making changes to it to make it valid; this approach is known as a repair mechanism. A third direct approach is to implement a specialized decoder function that is guaranteed to map each genome to a valid solution. When implemented, this method usually results in multiple genomes mapping to the same solution [10].

3.10.2 - Example application: 0-1 knapsack problem

A common problem that is used to illustrate the effectiveness of GAs and constraint handling is the 0-1 knapsack problem, defined as follows:

> Given n items {$item_1$,$item_2$, … , $item_n$}, each with its own associated value v_i and cost c_i, find the combination of items that maximizes the sum of all values within a combination while keeping the sum of the costs of these items below a capacity (c).

There is no known algorithm for finding the optimal solution to this problem in polynomial time. However, using a GA, a near-optimal solution can be reliably found in linear time.

The following is a description of the encoding and decoding functions used in the GA, described in [10], developed to solve this problem. A solution to the 0-1 knapsack problem is naturally represented by a bit string, where each bit corresponds to an item and the value of that bit denotes whether that item is (1) or is not (0) a part of that solution. However, there is the possibility that a genome maps to an invalid solution - one that violates the constraint that the sum of the items' costs must be below c. The authors of [10] implement a specialized decoder function that ensures that only valid solutions are evaluated. In their implementation, when a genome is decoded, an empty set is first created. The genome is evaluated from left to right, and items with corresponding bits equal to 1 are added to this set until the sum of the costs of the items currently in the set is above c. The last item - the one that put the set over capacity - is removed and the resulting combination of items is the solution that that genome maps to.

3.11 - Comparison to other algorithms

A GA is seen as an alternative to classic problem-solving methods. For optimization over a small, linear, unimodal space, the GA is overshadowed, by other techniques. For example, gradient-based hill climbing or linear programming are more efficient than a GA when "climbing" a single hill. However, linear techniques, like the Simplex method, assume that the underlying objective function is linear, and nonlinear techniques, like gradient ascent/descent, perform best when the objective function is unimodal. If these conditions are not met, these

techniques do not perform well and are susceptible to becoming stuck in local optima. In contrast, a GA is capable of finding near-optimal solutions regardless of the satisfaction of these conditions. A GA is a significantly better alternative than the previously mentioned techniques when the underlying problem has an extremely large search space, a multimodal search space, and/or no efficient method of finding the best solution [12].

Chapter 4

Genetic Algorithm for Locating Acceptable Structures (GALAS)

In this chapter, a Genetic Algorithm for Locating Acceptable Structures (GALAS) is outlined. The goal of this algorithm is to find the optimal (most acceptable) G-structure model(s) of an overall system, where a G-structure's acceptability is a function of both the information loss associated with using the reconstruction of that structure as a model of the overall system, *and* the change in storage requirement that occurs as a result of having to store the structure in memory rather than the overall system. This measure of acceptability is described in Section 4.3. The remainder of this chapter includes a description of the various components of GALAS, the results of testing GALAS, and a comparison of these results to the results of testing two alternative approaches for finding the optimal G-structure(s): a random search, and a brute-force search.

GALAS is implemented in the R programming language; the source code for this implementation is provided in Appendix C. The algorithm allows the user the option to customize the values of a variety of algorithmic parameters before execution. If the user abstains from supplying a value for a parameter, that parameter is set to its default value. Default values were chosen because they resulted in subjectively "good" GALAS performance, based on both the algorithm's ability to find the optimal solution(s), and the speed at which it does so. It is common practice, when implementing a GA, to test a few different combinations of parameter values until one set of values is found that "appears to work well". This is because there are usually multiple parameters, and finding the combination of values that yields the best performance is an optimization problem in its own right. The set of default values for the parameters of GALAS is not necessarily the optimal set when searching S_G for all overall systems, but is a set that provided reasonable results during testing. GALAS' customizable parameters, and their default values, are also defined in this chapter.

4.1 - Representation of G-structures

Given the problem of finding the "best" G-structure of an overall system, the set of all G-structures of that system is a reasonable search space. A bit string representation is a natural way of representing any G-structure; each bit denotes the presence or absence of a subsystem of the overall system in that G-structure. However, there is no direct mapping from $\mathbb{G}$ to S_G when using bit string representations, i.e. there are some combinations of subsystems that map to valid bit strings, but are not G-structures. Therefore, a one-to-one mapping from $\mathbb{G}$ to a search space larger than S_G is first defined. This search space is then constrained during the evolutionary process to ensure that only G-structures within this space are evaluated. This larger search space, denoted S'', is the set containing all structures of the overall system that do not contain the empty subsystem or the overall system, i.e. $\mathbf{S'' = \{SS_i \mid SS_i \subseteq (\mathcal{P}(V) - \{\emptyset, V\})\}}$. S'' is nearly identical to S', but excludes all structures that contain the empty subsystem or the overall system because incorporating these structures would be counterintuitive. This is because 1) there is no G-structure that contains the empty subsystem, and 2) there is only one G-structure ({S}) that contains the overall system, but the goal of GALAS is to find an alternative to representing S using {S}, since the two are, effectively, the same.

Given an overall system S with a set of attributes V, the number of unique subsystems of S is $2^{|V|}$ ($|\mathcal{P}(V)|$). When excluding the empty subsystem and the subsystem that is equal to S, this number is $2^{|V|} - 2$. The set of subsystems of S, excluding $\emptyset$ and S, is referred to as "*the set of meaningful subsystems*" because an element of S'' may contain any combination of these subsystems and, therefore, each subsystem's presence/absence is meaningful when defining an element of S''. The set of meaningful subsystems of S, denoted $P_{|V|}$, is given in Table 4-1 for systems defined on 1, 2, 3, and 4 attributes.

$\|V\|$	$P_{\|V\|}$
1	{}
2	{1, 2}
3	{1, 2, 12, 3, 13, 23}
4	{1, 2, 12, 3, 13, 23, 123, 4, 14, 24, 124, 34, 134, 234}

Table 4-1: $P_{|V|}$ when $|V| = [1\text{-}4]$

Given an SS ∈ S'', SS will contain between zero and all ($2^{|V|}$ - 2) meaningful subsystems of S. Using a bit string genomic representation, SS maps to a bit string of length $2^{|V|}$ - 2, where each bit represents the presence (1) or absence (0) of a meaningful subsystem of S within SS. A meaningful subsystem is represented by the bit that corresponds to that subsystem's order in the enumeration of all meaningful subsystems; this order is shown in Table 4-1. With this genomic representation, the mapping between S'' and $\mathbb{G}$ is one-to-one,

Example 4.1.1 Assume an overall system S, where |V| = 3. Given SS = 1/12/3/23, SS maps to the genome 101101. When decoded, the bit string 111111 becomes the structure 1/2/12/3/13/23, and the bit string 000000 becomes the trivial structure { }.

In order to enforce that only G-structures are allowed to be members of the population, two additional constraints must be placed on the members of S'' (See Section 3.10 - Constraint Handling). These constraints take the form of repair mechanisms and are enforced immediately prior to the evaluation of the population. If a structure SS ∈ S'' is not a G-structure, it becomes one through a repairing process. The first constraint is that SS must be reduced. This is enforced by "reducing" SS prior to evaluation: any system in SS that is a subsystem of another subsystem in SS, is removed. The second constraint is that, for SS = {S_1, S_2, … , S_k}, the following must be true: $\mathbf{V} = \cup_{\mathbf{V_i} \in \mathbf{SS}} \mathbf{V_i}$**.** This is enforced by adding singleton subsystems - subsystems defined on one attribute - $|V_i|$ = 1 - to S until this condition is met, i.e. for each $\mathbf{v_j \in (V - \cup_{V_i \in SS} V_i)}$, a subsystem S_a, defined solely on v_j, is added to SS. When one considers the set of all H-structures of a system defined on the attributes in (V - $\cup_{V_i \in SS} V_i$), the H-structure containing only singleton subsystems is guaranteed to have the smallest storage requirement, in the general case. Therefore, the addition of these singleton subsystems to SS will always result in the smallest increase to $\mathbb{C}$(SS) - the importance of keeping the storage requirement of SS minimal will be seen in Section 4.3. Also, these additions never result in the removal, due to enforcing the first constraint, of any system in SS, because it is not possible for a system already in SS to be a subsystem of a singleton subsystem.

4.2 - Population Size & Initialization

The initial population is created by first generating a number of random genomes of the appropriate length ($2^{|V|}$ - 2). These genomes are then decoded and the G-structure constraint is

enforced on each resulting structure. The resulting solutions represent the first generation of solutions. The size of the population used in GALAS is a customizable parameter. The optimal population size is dependent on |V|. A larger population occupies more regions of S_G in parallel, which may result in an overall shorter time required to find the optimal G-structure. However, a larger population requires the evaluation of more solutions during each iteration of the algorithm and requires more time to converge, both of which result in a decrease in efficiency. When |V| < 4, generating, evaluating, and comparing all G-structures of an overall system to find the optimal G-structure is almost always faster than using GALAS to do so. Therefore, GALAS is recommended for use with systems defined on 4 or more attributes. When |V| = 4, the default population size is set to 10; when |V| > 4, the default size is 30.

4.3 - Fitness Function

The ultimate goal of GALAS is to find the most acceptable G-structure model of an overall system S, given an objective measure of acceptability. While any fitness function can be defined for use in GALAS, the implemented measure of G-structure acceptability that is defined in this section considers both the potential loss of information that occurs when representing S as the reconstruction of a G-structure, and the potential change in storage requirement that occurs as a result of having to store that G-structure in memory, instead of having to store S. The acceptability of a G-structure G_i is defined as a measure of how efficiently G_i uses the space required to store it to essentially contain information about S. When $\mathbb{C}(G_i) < \mathbb{C}(S)$, this measure, denoted $R(G_i)$, is the ratio of information loss to decrease in storage requirement and is given by the following formula, in the relational case,

$$R(G_i) = \frac{IPS(r(G_i))}{\mathbb{C}(S) - \mathbb{C}(G_i)}$$

and by the following formula, in the probabilistic case

$$R(G_i) = \frac{H(r(G_i)) - H(S)}{\mathbb{C}(S) - \mathbb{C}(G_i)}$$

Theorem 4.3.1: Given two G-structures G_x,G_y, where $\mathbb{C}(G_x) < \mathbb{C}(S) > \mathbb{C}(G_y)$, if $R(G_x) < R(G_y)$, then G_x is a more acceptable model of S than G_y is because G_x more efficiently uses its storage requirement to contain information about S.

While this ratio captures the idea of storage efficiency for G-structures with storage requirement below that of the overall system, it does not do the same for all other G-structures. If $\mathbb{C}(G_i) = \mathbb{C}(S)$, the ratio is invalid (divide-by-zero), and if $\mathbb{C}(G_i) > \mathbb{C}(S)$, the denominator is negative, thus $R(G_i)$ is nonpositive, since the numerator is always nonnegative. $R(G_i)$ is the amount of information that is lost per unit *decrease* in storage requirement, and so the absolute value of a negative ratio indicates the amount of information that is lost per unit *increase* in storage requirement. G-structures with negative ratios are inconsistent with Theorem 4.3.1. There are also numerous additional cases in which using this ratio to compare G-structures gives undesired results. For example, if, for two G-structures G_i,G_j, $\mathbb{C}(G_i) < \mathbb{C}(G_j)$ and the overall system S is equivalent to both $r(G_i)$ and $r(G_j)$, meaning that no information is lost, then $R(G_i) = R(G_j) = 0$. In this scenario, the same amount of information is in $r(G_i)$ and $r(G_j)$, yet G_i has less storage requirement, so, in terms of storage efficiency, G_i is a superior model to G_j. However, this is not reflected when using this measure. For these reasons, this ratio alone is not a suitable measure of the acceptability of any G-structure.

An ideal measure of G-structure acceptability is one that is valid for all G-structures and captures the idea of storage efficiency. The method that is used in this thesis to ensure that these statements are true is to define G-structure acceptability as a distance-based metric. S_G is partitioned into three subsets: S_N, containing “neutral” G-structures; S_A, containing “acceptable G-structures whose acceptabilities are measured as a distance from the set S_N; and S_U, containing “unacceptable” G-structures whose acceptabilities are also measured as a distance from S_N, but are negative.

A G-structure G_i is considered neutral if 1) $R(G_i)$ is valid, 2) $\mathbb{C}(G_i) \leq \mathbb{C}(S)$, and 3) $R(G_i) = R(G_k)$, where G_k is the most-refined G-structure in S_G. G_k serves as the G-structure to which all other G-structures are compared. This is because: 1) every lattice defined on S_G has a single most-refined G-structure, 2) unlike all other G-structures (except the least-refined G-structure), G_k is trivial to generate, and 3) this G-structure is guaranteed to have the smallest storage requirement of all G-strictures. In terms of storage efficiency, neutral G-structures are identical,

and thus no neutral G-structure is considered a better representation of S then another. The least-refined G-structure, denoted G_S, is also an honorary member of S_N, since it is basically the overall system and we are neutral about selecting G_S as a model of S, i.e. there are better models of S, but there are also worse models.

A G-structure G_i is considered unacceptable if 1) $R(G_i) > R(G_k)$, or 2) $R(G_i)$ is invalid. If the storage requirement of G_i is greater than that of S then $R(G_i) > R(G_k)$, and G_i would never be chosen as a model of S. Instead the neutral model G_S would be chosen because 1) every lattice defined on S_G has a least-refined G-structure, 2) G_S is trivial to generate, 3) the information in $r(G_i)$ is never greater than the information in $r(G_S)$ - which is equal to S. In terms of storage efficiency, there is never any benefit in selecting G_i over G_S when $\mathbb{C}(G_i) > \mathbb{C}(S)$. If $R(G_i) > R(G_k)$ and $\mathbb{C}(G_i) < \mathbb{C}(S)$, then G_k is a better model of G_i (See Theorem 4.3.1). In both cases, it is not beneficial to select an unacceptable G-structure as a model of S.

A G-structure G_i is considered acceptable if it is not unacceptable or neutral - that is, if 1) $R(G_i) < R(G_k)$ and $\mathbb{C}(G_i) < \mathbb{C}(S)$. Compared to all neutral and unacceptable G-structures, any acceptable G-structure is a more storage-efficient model than each.

Example 4.3.1 Assume a probabilistic system S defined on 4 binary attributes. G_k = 1/2/3/4, $H(r(G_k) = 3.8$, $\mathbb{C}(G_k) = 8$; S = 1234, H(S) = 3.4, $\mathbb{C}(S) = 16$. Figure 4-1 is a graph of Storage Requirement (X) vs. Entropy (Y). If S were a relational system, the y-axis of this graph would be IPS (See Section 2.9.1). The two points on the graph correspond to where G_k and S plot to. The line passing through these points signifies the space that a neutral G-structure will plot to. This line serves as a “border” separating the acceptable and unacceptable G-structures. If a G-structure plots to a point below this line, then it is an acceptable G-structure; if it falls above this line, it is unacceptable. Note that all G-structures will plot to points within the red lines seen in Figure 4-2. This constraint indicates that, given a G-structure G_i, $I(r(G_k)) \leq I(r(G_i)) \leq I(S)$ and $\mathbb{C}(G_i) \geq \mathbb{C}(G_k)$.

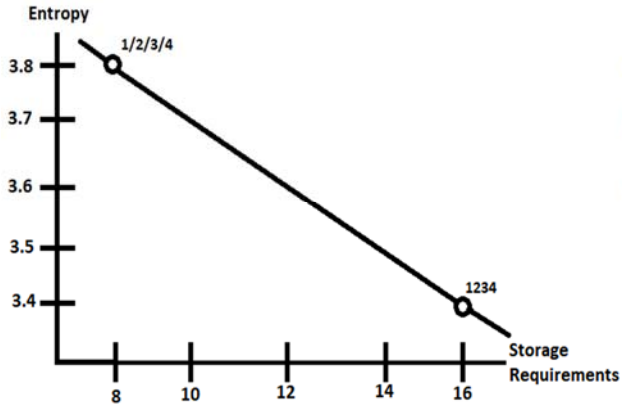

Figure 4-1: The border between acceptable and unacceptable G-structures

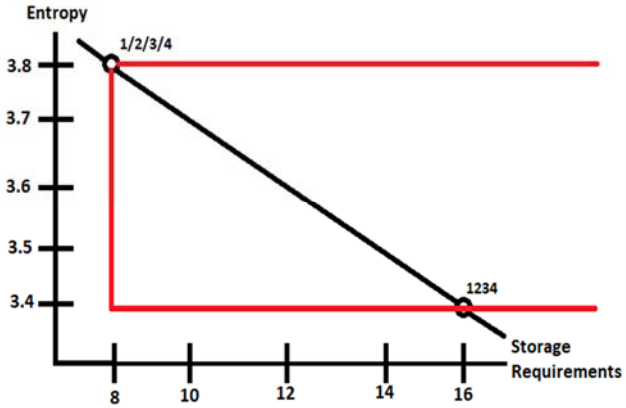

Figure 4-2: The constraint on the position of a possible G-structure

Using this border, any G-structure model can be classified as either acceptable, unacceptable, or neutral. However, this classification is not a sufficient evaluation method for use in a GA. The codomain of a GA's fitness function F must be larger, otherwise, the peaks of F over $\mathbb{S}$ will be too steep for an optimizer, like a GA, to "climb". If a fitness function with a ternary codomain was implemented in a GA, the "best" G-structure, to that algorithm, would be any one that is deemed acceptable. This approach does not aid in finding the *most* acceptable G-structure. Therefore, a larger, preferably continuous, codomain is necessary.

The fitness of any G-structure G_i is defined as the difference between the amount of information in the reconstruction of G_i, and the amount of information in the reconstruction of a hypothetical neutral G-structure G_x with identical storage requirements to G_i. Using this measure, every G_i is directly comparable to a neutral G-structure, G_x (which may not exist). Because the storage requirements of G_x and G_i are equal, a direct comparison can be made, since the only difference between the two G-structures, in terms of storage efficiency, is the amount of information in each. The equation for calculating the fitness of a G-structure G_i is $F(G_i) = I(r(G_i)) - I(r(G_x))$, which is equivalent to

$$F(G_i) = (R(G_k) * (\mathbb{C}(S) - \mathbb{C}(G_i)) - IPS(r(G_i))$$

in a relational setting, and

$$F(G_i) = H(S) + (R(G_k) * (\mathbb{C}(S) - \mathbb{C}(G_i)) - H(r(G_i))$$

in a probabilistic setting. If we expand $R(G_k)$, the fitness function for evaluating a relational G-structure is

$$F(G_i) = \left(\left(\frac{IPS(r(G_k))}{\mathbb{C}(S) - \mathbb{C}(G_k)} \right) * \left(\mathbb{C}(S) - \mathbb{C}(G_i) \right) \right) - IPS(r(G_i))$$

and the fitness function for a probabilistic G-structure is

$$F(G_i) = H(S) + \left(\left(\frac{H(r(G_k)) - H(S)}{\mathbb{C}(S) - \mathbb{C}(G_k)} \right) * \left(\mathbb{C}(S) - \mathbb{C}(G_i) \right) \right) - H(r(G_i))$$

If G_x existed, it would plot to the point on the border (Figure 4-1) that shares the same x-coordinate (storage requirement) as G_i. When G_i is plotted, $F(G_i)$ is the vertical distance that G_i is below the point on the graph representing G_x. If G_i falls above G_x (above the neutral line), then $F(G_i) < 0$; if G_i falls on the neutral line, $F(G_i) = 0$; if G_i falls below G_x (below the neutral line), then $F(G_i) > 0$. In other words, an acceptable G-structure has a positive fitness, a neutral G-structure has a fitness of 0, and an unacceptable G-structure has a negative fitness.

Example 4.3.2 Assume G-structures G_y, G_z are models of the probabilistic system S specified in Example 4.3.1, where $H(r(G_y)) = 3.5$, $H(r(G_z)) = 3.7$, and $\mathbb{C}(G_y) = \mathbb{C}(G_z) = 12$ (Figure 4-3). G-structure G_x is the neutral G-structure with identical storage requirement to both G_y and G_z. The following is true:

1. $R(G_k) = R(G_x)$, i.e. (3.8 - 3.4)/(16 - 8) = 0.05 = (3.6 - 3.4)/(16 - 12)
2. $F(G_y) = H(r(G_x)) - H(r(G_y)) = 3.6 - 3.5 = 0.1$
3. $F(G_z) = H(r(G_x) - H(r(G_z)) = 3.6 - 3.7 = -0.1$
4. In terms of storage-efficiency, G_y is a better model of S than G_z is.

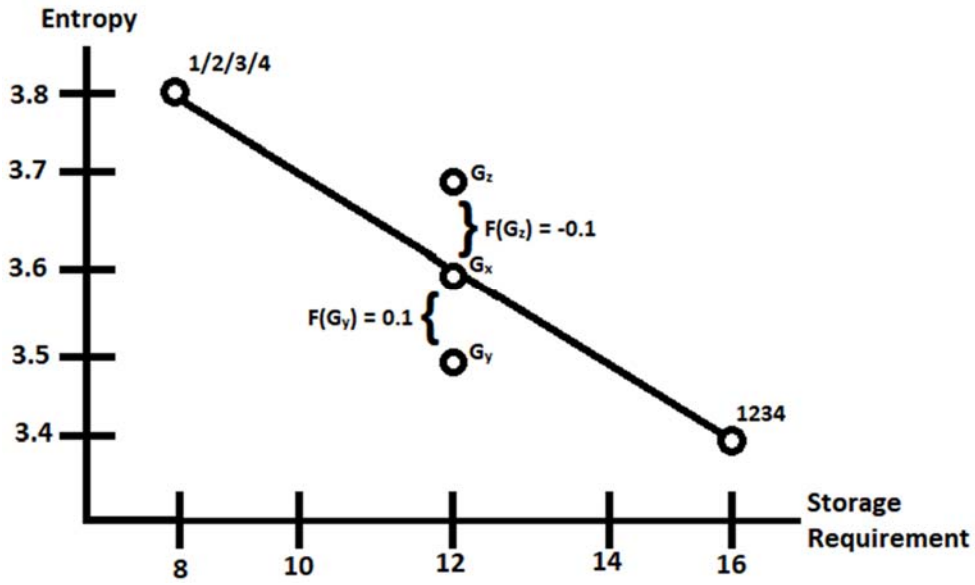

Figure 4-3: Example G-structures G_x, G_y, and G_z

Example 4.3.3 Table 4-2 contains the storage requirement, fitness, and IPS of the reconstruction of each of the 9 G-structure models of relational system S^R (Table 2-1(a)). Table 4-3 contains the storage requirement, fitness, and entropy of the maximum entropy member of the reconstruction family of each of the 9 G-structure models of probabilistic system S^P (Table 2-1(b)). These G-structures are plotted in Figures 4-4 (relational) and 4-5 (probabilistic).

$G_i \in S_G$	Storage Requirement	IPS($r(G_i)$)	$F(G_i)$
123	8	0	0
12/23/13	12	0	-6
12/23	8	1	-1
12/13	8	0	0
13/23	8	1	-1
1/23	6	3	0
2/13	6	1	2
3/12	6	1	2
1/2/3	6	3	0

Table 4-2: The elements of S_G modeling S^R

$G_i \in S_G$	Storage Requirement	$H(r(G_i))$	$F(G_i)$
123	8	2.183	0
12/23/13	12	2.183	-0.795
12/23	8	2.381	-0.198
12/13	8	2.289	-0.105
13/23	8	2.427	-0.244
1/23	6	2.550	0.030
2/13	6	2.458	0.123
3/12	6	2.412	0.169
1/2/3	6	2.581	0

Table 4-3: The elements of S_G modeling S^P

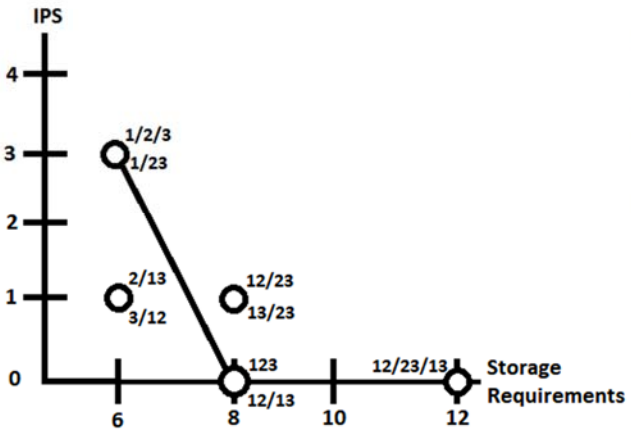

Figure 4-4: Plot of G-structures modeling S^R

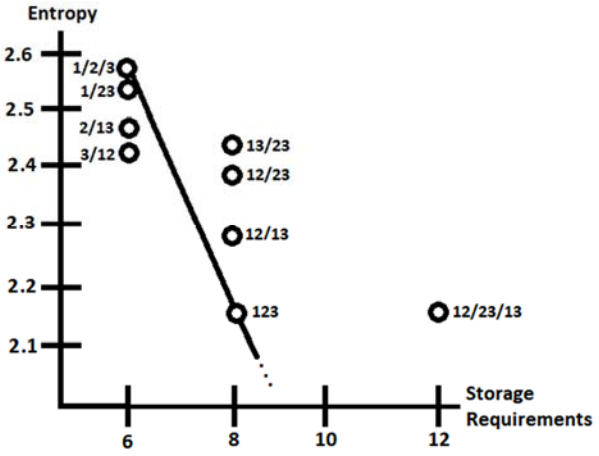

Figure 4-5: Plot of G-structures modeling S^P

This fitness measure is well-suited for evaluating G-structures based on both information loss and decrease in storage requirement, as it captures the intent of the initial ratio, but does so for all G-structures, including those for which R() is invalid or 0. The following is a compilation of scenarios presented to further justify the use of this fitness measure.

- If, for G-structures G_i, G_j of an overall system S, $\mathbb{C}(G_i) < \mathbb{C}(G_j)$ and $S = r(G_i) = r(G_j)$, then $F(G_i) > F(G_j)$. In other words, G_i is a more acceptable model of S than G_j is. In terms of storage efficiency, this is true because $r(G_i)$ contains the same amount of information as $r(G_j)$, but requires less storage space. Figure 4-6 shows an example of this scenario with G-structure models of the probabilistic overall system defined in Example 4.3.1.

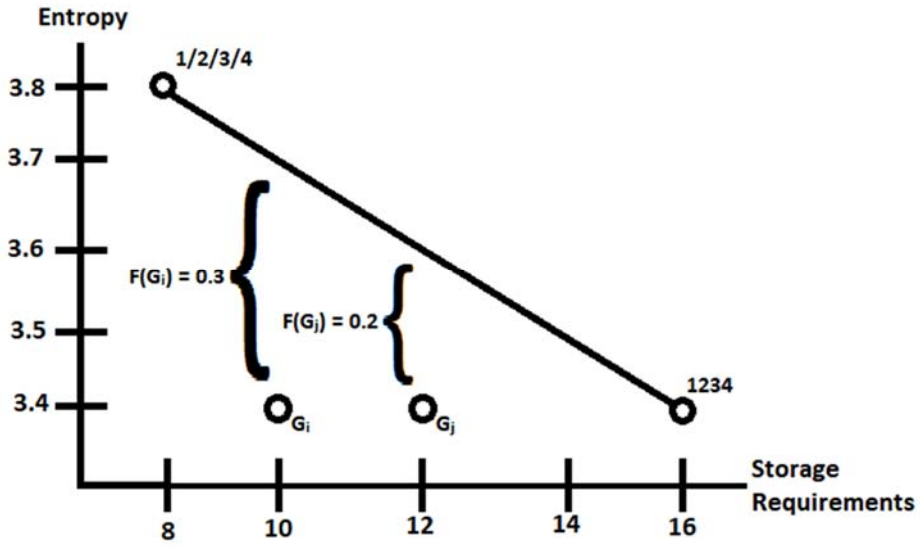

Figure 4-6: $\mathbb{C}(G_i) < \mathbb{C}(G_j)$

- If, for G-structures G_i, G_j, $\mathbb{C}(G_i) = \mathbb{C}(G_j)$ and $I(r(G_i)) < I(r(G_j))$, then $F(G_i) < F(G_j)$. In terms of storage efficiency, G_j is a more acceptable G-structure than G_i because both G-structures require the same amount of memory, but the reconstruction of G_j contains more information than the reconstruction of G_i. An example of this is given in Figure 4-7.

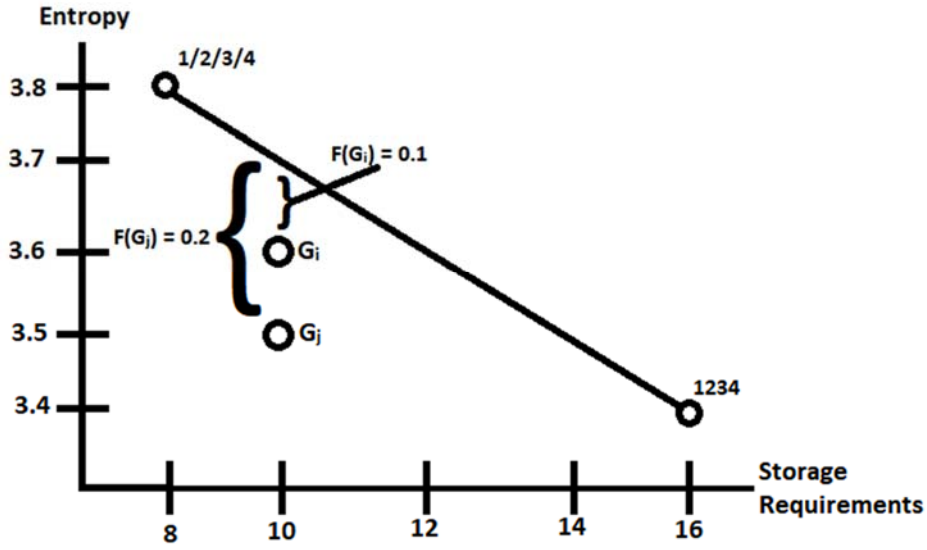

Figure 4-7: $\mathbb{C}(G_i) = \mathbb{C}(G_j)$ and $I(r(G_i)) < I(r(G_j))$

- If, for G-structures G_i, G_j, $R(G_i) = R(G_j)$ and $\mathbb{C}(G_i) < \mathbb{C}(G_j)$, then $F(G_i) > F(G_j)$. Since both G-structures have the same R() value, the better model, in terms of storage efficiency, is the one with less storage requirement (Figure 4-8).

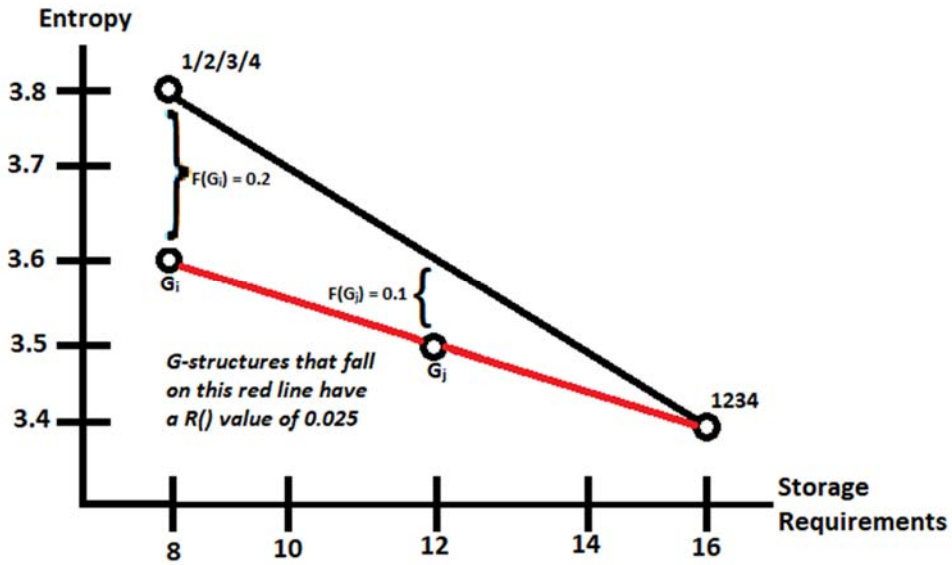

Figure 4-8: $R(G_i) = R(G_j)$ and $\mathbb{C}(G_i) < \mathbb{C}(G_j)$

- If, for G-structure G_i, $\mathbb{C}(G_i) \geq \mathbb{C}(S)$ and $r(G_i) \neq S$, then $F(G_i) < 0$ (Figure 4-9). If, for G-structure G_j, $\mathbb{C}(G_j) = \mathbb{C}(S)$ and $r(G_j) = S$, then $F(G_j) = 0$.

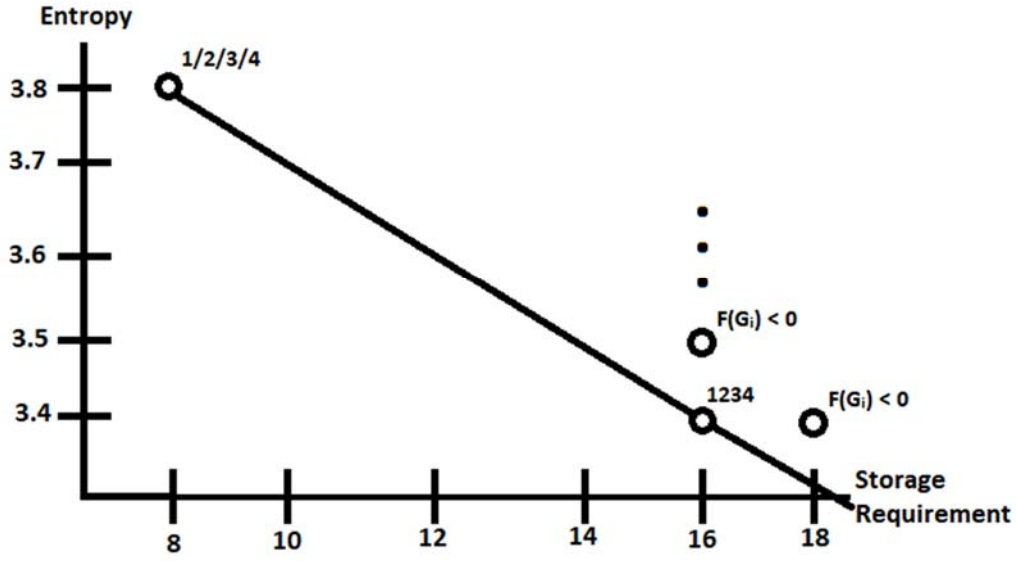

Figure 4-9: $\mathbb{C}(G_i) \geq \mathbb{C}(S)$

The most acceptable G-structure in S_G is the G-structure with the highest fitness score. This is the G-structure whose reconstruction contains the most information, relative to the information in the reconstruction of a hypothetical neutral G-structure with identical storage requirement. The design of this fitness measure was inspired by the fitness function of the genetic algorithm outlined in [9].

In the GALAS implementation, all G-structures evaluated during execution of GALAS that are found to have positive fitnesses are recorded. These G-structures are not printed to the screen during algorithm execution, but are stored in the lists with the names “GoodPhenos” and “GoodPhenosScores” in the R environment.

4.4 - Selection Strategy

Three of the four selection algorithms discussed in Section 3.5 are implemented in GALAS: truncation, tournament (using a preset tournament size of 3), and roulette wheel. The type of selection algorithm is a customizable parameter; the default type is roulette wheel selection. The size of the mating pool is also a customizable parameter, with the default value

being dependent on |V|. When |V| = 4, 6 (not necessarily unique) individuals are selected; when |V| > 4, this number is 20.

4.5 - Crossover of two G-structures

Crossover, in GALAS, occurs between two G-structures, and produces two offspring structures, which are not necessarily G-structures. The G-structures in $\mathbb{M}$ are randomly paired together. Given a pair of G-structures, the probability that crossover occurs between them - the "rate of crossover" - is a customizable parameter, with a default value of 0.8. Because breeding is stochastic, the total number of offspring produced by all pairs in a generation is a dynamic value. The type of crossover that is used is also a customizable parameter; the user has the option of using either single-point, multi-point, or uniform crossover, with multi-point being the default. Each of these types of crossover produces two offspring solutions.

With all three types of crossover, an offspring structure will always inherit a subsystem if that subsystem is in both of its parent G-structures. Using uniform crossover, a subsystem that is in one parent G-structure, but not the other, has a 0.5 probability of being inherited by an offspring structure. The multi-point crossover implemented in GALAS does not use random points, but rather uses preselected points on the G-structure's genome. The positions of these points are selected based on prior knowledge of the genomic representation that is being used in GALAS. When viewing a genome from left to right, each point marks the introduction of an additional attribute that may be defined on by the subsystems in $P_{|V|}$. These points separate the $(|P_{X-1}| + 1)$th and $(|P_{X-1}| + 2)$th bits of the parent bit strings for all integers X in the interval $(1,|V|]$.

Example 4.5.1 Given an overall system S, defined on 4 attributes, and two G-structure models of S, G_x = 2/13/14 and G_y = 12/13/23/14, G_x maps to the genome 01001000100000 and G_y maps to the genome 00101100100000. Assume that G_x and G_y will be bred using multi-point crossover. Rewritten with points denoted by "|", G_x = 0|10|0100|0100000 and G_y = 0|01|0110|0100000. When G_x and G_y are bred, using multi-point crossover, to produce offspring SS_a and SS_b, it is certain that the first bit of the genome of SS_a is 0, since it is 0 in both parent genomes; there is a 50% chance that the next two bits of SS_a are 10, and a 50% chance that they are 01; there is a 50% chance that the next four bits are 0100 and a 50% chance that

they are 0110; it is certain that the next seven bit are 0100000. The set of genomes of possible offspring that can be created by breeding G_x and G_y using this version of multi-point crossover (with "|" left in for clarity) is: {0|10|0100|0100000, 0|01|0110|0100000, 0|10|0110|0100000, 0|01|0100|0100000}.

4.6 - Mutation

Bit-flipping mutation is implemented in GALAS. Every bit of the genomes of all offspring produced during crossover is mutated (flipped) with a predefined probability. This probability is a customizable parameter with a default value of $1/(2^{|V|} - 2)$ - that is, 1 divided by the length of the bit string that the bit is a part of. This probability is commonly used in GAs that implement bit string genomic encoding. Using the default probability, one bit will be flipped per offspring, on average. This equates to either removing a subsystem from the offspring structure ($1 \rightarrow 0$), or adding a meaningful subsystem of S to the offspring structure ($0 \rightarrow 1$) that is not already in it.

Example 4.6.1 Assume an overall system S defined on 3 attributes and a structure model of S, O_x = 12/3. The set of all possible structures, following the mutation of a single bit of O_x's genome, is {1/12/3, 2/12/3, 3, 12, 12/3/13, 12/3/23}. After enforcing the G-structure constraint on each, note that half of these structures, are equivalent to O_x. The possibility of an ineffective mutation is one reason why *every* bit of a genome, rather than just one bit, has a chance of being mutated.

4.7 - Replacement policy

A steady-state replacement strategy is implemented in GALAS. Deterministic replacement of the least acceptable G-structures in the population with the offspring produced during breeding occurs at the end of each iteration. The number of G-structures replaced is equal to the number of offspring produced during crossover; this is implemented to maintain a static population size across all iterations of GALAS. After replacement, the G-structure constraint is enforced on the newly inserted offspring.

4.8 - Repetition & Termination

During each iteration of GALAS, information regarding the status of the search is reported, including: the most acceptable G-structure in the current population, the fitness

score of this G-structure, the average fitness score across the entire population, and the amount of time (in seconds) that GALAS has been running for. The range of the fitnesses is used as an indicator of population convergence; if the solution with the best fitness in the population is equal to the solution with the worst fitness in the population, then all members of the population have the same fitness, and thus the population has converged. GALAS will run indefinitely, until the population has converged or until a maximum number of iterations have occurred. This maximum number of iterations is a customizable parameter, with a default value of 100 iterations.

4.9 - Testing and comparing to alternative procedures

In this section, the results of testing GALAS are presented. These results are then compared to the results of testing potential alternative methods - namely, a brute-force search and a random search - for searching S_G for the most acceptable G-structure. All tests were conducted on a Dell Inspiron 5547 with an Intel Core i7-4510U processor running at 2.6 GHz. For tests on systems defined on 4 attributes, GALAS' customizable parameters were set to their default values. Note that this is not necessarily the most optimal set of parameters for all of the systems tested, but is a set of parameters that was found to produce decent results, in general.

4.9.1 - Results

GALAS was tested on 11 different probabilistic systems defined on 4 binary attributes. The range of the behavior function - that is, the distribution over T - of one of these systems was preselected and served as the primary system that was used to test components of GALAS throughout its development. The tabular representation of this system is given in Table 4-4. The ranges of the behavior functions of the remaining 4-attribute systems were generated randomly. GALAS was run 30 consecutive times for each 4-attribute system, terminating when the population converged or when the maximum number of iterations was reached - whichever occurred first. The results of each of GALAS' runs are presented in Table A-1. For each run, the maximum fitness that was seen during the run and the amount of time (in seconds) that GALAS ran for during the run are both reported. Each run is also

v_1	v_2	v_3	v_4	f(t)
0	0	0	0	0.0625
0	0	0	1	0.0625
0	0	1	0	0
0	0	1	1	0.125
0	1	0	0	0
0	1	0	1	0.1875
0	1	1	0	0
0	1	1	1	0.0625
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0.0625
1	0	1	1	0.0625
1	1	0	0	0.125
1	1	0	1	0.0625
1	1	1	0	0.125
1	1	1	1	0.0625

Table 4-4: Preselected Probabilistic System

assigned an indicator number, which specifies whether during the run GALAS terminated because the population converged (0) or because the maximum number of iterations was reached before the population converged (1).

GALAS was also tested on a single probabilistic system defined on 5 binary attributes. For this test, all of GALAS' customizable parameters were set to default values, except for the maximum number of iterations, which was set to 10. GALAS was run for 10 generations, regardless of the satisfaction of any other terminating conditions. This was done to limit the amount of time the test would require. The results of 30 consecutive runs are presented in Table A-1. For each run, the amount of time (in seconds) required for 10 iterations of the algorithm to occur, and the fitness of the most acceptable model in the population when the algorithm terminated were recorded. The average time GALAS took to terminate and the average of the best fitness scores of the most acceptable models found for all systems tested are seen in the last row of Table A-1.

In this work, I consider two alternative procedures for finding the most optimal G-structure based on the fitness function defined in this thesis - or any fitness function, for that matter. The first method, which I refer to as the "brute force" or exhaustive approach, is to generate the entire lattice of G-structures and calculate the fitness of each. This method was also implemented in R, alongside GALAS, and uses the RG-procedure described in [4]. If this method is used, the most acceptable G-structure is guaranteed to be found. However, this method is computationally expensive when $|V| > 4$. For each of the 12 aforementioned probabilistic systems, the fitness of the most acceptable G-structure of each is provided in Table 4-5. Also included in this table is: the average of the best fitnesses found during 30 runs of GALAS, the average amount of time GALAS took to terminate over 30 consecutive runs, and the amount of time the brute-force search took to evaluate the fitness of all G-structures. For the 5-attribute probabilistic system, the brute-force method required 9.36 hours to find the most acceptable solution - 2.89 hours to generate all 6894 G-structures and 6.47 hours to evaluate them all. The significant increase in time, from 4-attribute to 5-attribute systems, is the reason that only one 5-attribute system was tested.

Distribution	Maximum Fitness	Average Maximum Fitness Found (GALAS)	GALAS (s)	Brute Force (s)
Preselected	0.2657025	0.2657025	33.32	27.99
Random (1)	0.18150652	0.143807496	31.65	25.48
Random (2)	0.10700493	0.104041155	37.33	23.06
Random (3)	0.07616349	0.073917151	38.25	20.20
Random (4)	0.10199852	0.095507153	53.01	20.35
Random (5)	0.10084217	0.099319142	34.13	21.11
Random (6)	0.1442208	0.1442208	36.29	20.82
Random (7)	0.02536194	0.023428826	43.87	18.06
Random (8)	0.03713224	0.027610086	39.89	20.24
Random (9)	0.08999404	0.086741665	31.68	20.05
Random (10)	0.05796067	0.052812423	100.33	20.28
5-attribute	0.060162	0.041485108	178.76	9.36 hrs
Average for 4-attribute Random Distributions	~	~	44.64	20.96

Table 4-5: GALAS vs. Brute-Force

The second alternative method is to randomly create and evaluate G-structures until the most acceptable is found. This method assumes that the best solution is known a priori; this assumption need not be made when using GALAS. However, without prior knowledge of the most acceptable solution, we can never guarantee that the best G-structure found by the random search or by GALAS is the most acceptable G-structure in S_G; we can only say that it is the best that has been found so far. The random search was carried out by randomly initializing G-structures using the same technique used for population initialization in GALAS. Although this technique is random when producing an element of S'', it is not truly random when producing a G-structure. This is shown in Table A-3 for G-structures of a 4-attribute system. In this table, the 114 G-structure models of a system defined on 4 attributes are listed. For each G-structure, the total number of elements of S'' that, after enforcing the G-structure constraint, are equivalent to that G-structure is given, along with the probability that a random bit string of length $2^{|V|} - 2$ will map to that G-structure. The random search involved randomly generating genomes until a genome was found that mapped to the most acceptable G-structure in S_G (found beforehand). 30 probabilistic overall systems were created and a single random search was run on each. The amount of time that the random search took to find the most acceptable G-structure of each system is given in Table A-2; the average of these times is seen in the last row. A random search using a 5-attribute overall system was not attempted.

4.9.2 - Discussion

For the 4-attribute systems, GALAS outperformed the random search but performed worse than the brute-force search. The random search required an average of 4.4 minutes to find the most acceptable G-structure of each of the 30 random 4-attribute systems. However, in a real-world setting, there is no guarantee that any G-structure found by the random search is the most acceptable G-structure of a system in question. Of course, this is also true for GALAS - but, for those same 10 systems, GALAS was able to converge on an optimal, or near-optimal, solution in under 45 seconds. With that, it is clear that GALAS outperformed the random search. The brute-force search required an average of 20.96 seconds to evaluate all G-structures of the 4-attribute systems - far less than GALAS required. The brute-force search is also guaranteed to find the optimal solution, while GALAS was unable to converge on the optimal solution during all runs. For 4-attribute systems, GALAS is outclassed by a brute-force search.

For the single 5-attribute system tested, GALAS, on average, required 2.98 minutes to complete 10 iterations. In this time, GALAS was unable to find the optimal G-structure, but was able to find acceptable G-structures with fitnesses relatively close to that of the optimal G-structure (0.060162). In comparison, the brute-force search required 9.36 hours to find this optimal G-structure. While GALAS is not guaranteed to find this G-structure, finding near-optimal G-structures is a reasonable secondary goal when considering the extremely poor scalability of this problem. GALAS can be used to quickly sift through large amounts of unacceptable G-structures and find near-optimal G-structures in a relatively small amount of time. This becomes more beneficial as the number of attributes that an overall system is defined on increases.

Regardless of the fitness function that is used to evaluate G-structures, these results are a testament to GALAS' ability to search S_G. Systems defined on 5 or more attributes were tested minimally because an initial exhaustive search to determine the most optimal G-structure is incredibly time-consuming. When using larger systems, the default parameters may need to be tweaked, but this is made easy with the implementation of customizable parameters. The potential for enormous improvements in efficiency as $|V|$ increases is present, since $|S_G|$ grows so aggressively. Although the systems that GALAS was tested on were defined solely on binary

attributes, GALAS can be used in the general case on any overall system. The restriction to binary attributes affects the fitness scores of individual G-structures, but has no effect on the overall goal of the algorithm, since our primary concern is with the interactions *between* the attributes.

4.9.3 - Limitations

There are some cases in which using GALAS to find the most acceptable model of an overall system is of little practical use. As was mentioned in Section 4.2, when the overall system is defined on 3 attributes or less, it is typically more efficient to generate all G-structures of the system and calculate each of their fitnesses than it is to use GALAS. When the overall system contains no information, GALAS is also not useful, since the reconstruction of every possible G-structure will contain the same amount of information (none). This occurs when the range of that system's characteristic function contains only 1s, for a relational overall system, or the distribution defined over T is uniform, for a probabilistic overall system. In this situation, the slope of the border line in the graph of information loss vs. storage requirement is 0, and all G-structures are neutral.

GALAS is also susceptible to the same limitation - namely, scalability - that is present for problems related to reconstructability analysis. As the number of attributes of an overall system grows larger, so too does the number of calculations required for most of GALAS' processes. For example, the speed at which a reconstruction procedure is performed on a structure system is highly dependent on the number of subsystems in that structure system - structures containing more subsystems will take longer to reconstruct. This is especially true for probabilistic structures because IPFP, in general, requires more time for the overall distribution to converge for structures containing more subsystems. The maximum number of subsystems a single G-structure can contain increases as |V| grows. This number is $\binom{|V|}{2}$, and signifies the total number of unique subsystems of size 2 that can be made using |V| attributes.

Chapter 5

Conclusion

The following chapter presents a summary of the work conducted in this thesis, a discussion of the potential applications that this work may be relevant to, and a description of possible extensions/improvements that may be made to the components of GALAS.

5.1 - Summary

Systems-related terminology, relevant to the field of reconstructability analysis, are defined and exemplified to ensure clarification. A measure of the memory requirements associated with storing a system or a set of systems in memory is defined. A proof of the monotonicity of this storage requirement measure on the refinement lattice defined on S_C is given. Information metrics for both probabilistic (Shannon's Entropy) and relational (IPS) systems are defined.

The "Genetic Algorithm" is introduced and explanations of both the purpose of its various components and common implementations of these components are given. Holland's Schema Theorem is presented as a pseudo-explanatory proof of the success of genetic algorithms. Constraint handling techniques for ensuring a population of valid solutions are discussed. Genetic algorithms are compared to more conventional problem-solving methods.

A Genetic Algorithm for Locating Acceptable Structures (GALAS) is outlined. The GALAS implementation of those GA components discussed in Chapter 3 is described in Chapter 4. A ternary measure of the acceptability of a G-structure ("Acceptable"/"Unacceptable"/"Neutral") is defined that considers both the information loss associated with representing an overall system as the reconstruction of a G-structure model of that system, and the change in storage requirements that results from having to store that G-structure, rather than having to store its overall system. An objective extension of this measure is defined for any G-structure and serves

as the fitness function for GALAS. GALAS is tested alongside an exhaustive search and a random search, and the results of these tests are compared and discussed. Bottlenecks and other limitations on the speed and efficiency of GALAS are noted.

5.2 - Applications

The storage requirement measure that is defined in this thesis could serve useful in various fields of academia because it, like general systems theory, was developed with the goal of application-less generalization in mind. For example, in relational database theory, a relational database schema and the tables (relations) within it are parallels of a relational structure system and the subsystems within it. The storage requirement of a table is the maximum number of tuples that can be stored in that table, while the storage requirement of the entire database schema is the maximum total number of tuples that can be stored in that database - assuming no duplicates are allowed. If the measure could serve useful for working with relational databases, then the extent of its impact could reach any system that implements this form of data storage.

One can also represent any probability distribution P as the range of the behavior function of an overall probabilistic system S. The storage requirement of S is the number of probabilities in P. A set of marginal distributions of P can be represented as a structure of S, in which case the storage requirement of that structure is the total number of probabilities across all marginal distributions defined by the systems in that structure. GALAS can be used to find storage-efficient sets of marginal distributions of P.

5.3 - Future work

There are a number of directions that this work could be taken in. The following is a list of potential extensions that should be pursued:

- Extend the G-structure constraint on the members of $\mathbb{P}$ to a C-structure constraint, i.e. further constrict the search space to S_C. This may be beneficial for a number of reasons:
    - The reconstruction procedures would scale better. The maximum number of subsystems in a C-structure is $|V|$; the C-structures that contain this number of

subsystems are those whose primal graphs can be drawn as a ring. The maximum number of subsystems in a G-structure is $\binom{|V|}{2}$; this is the G-structure that contains only and all subsystems defined on each pair of attributes. In general, C-structures tend to contain less subsystems than G-structures. Since the speed of a reconstruction procedure is heavily dependent on the number of subsystems in a structure, only working with C-structures may increase overall efficiency.

- The growth of the search space would be much less aggressive. Given an overall system S, $|S_C| = 2^{\binom{|V|}{2}}$. This growth is shown in Table 5-1. For example, when |V| = 7, 2 million C-structures, while still a lot, is much more manageable than 2.4 trillion G-structures (Table 2-9).

$\|V\|$	1	2	3	4	5	6	7
$\|S_C\|$	1	2	8	64	1,024	32,768	2,097,152

Table 5-1: Number of C-structures when |V| is [1-7]

- The fitness measure may also be reworked, since we would only need to accommodate for C-structures with storage requirement less than or equal to that of the overall system.
- Overall, the decrease in efficiency that comes with enforcing a C-structure constraint may be overshadowed by the increase in efficiency resulting from the aforementioned benefits.

- The fitness measure, as it has been defined, should be evaluated further to determine intricacies that were not considered in this work. Classes of structures, mentioned in [4] but excluded from this work, should be analyzed from the perspective of objective structure acceptability. Questions that should be addressed include: “Are some types of structures inherently more acceptable than others?”, “Will the most acceptable structure always fall within a certain class of structures?”, etc.
- G-structure acceptability should be extended to allow a user to define the amount of value that is placed on retaining information vs. that which is placed on minimizing storage requirements. Doing so would involve altering the fitness function to weigh

expressions involving one measurement more than those involving the other. The user should also be given the option to enforce additional constraints on solutions. For example, given the constraint that a valid G-structure is one with a storage requirement below a user-specified threshold, GALAS would search for the valid G-structure whose reconstruction contains the most information of the reconstructions of all other valid G-structures in S_G. This example is especially pertinent when memory supply has a hard limit.

- GALAS should also be tested on larger systems. This would, most likely, involve changing the values of GALAS' parameters, which is easily accomplished because many of these parameters have been made customizable in the implementation outlined in this work.

Appendix A: Large Tables

	Preselected Distribution			Random Distribution (1)			Random Distribution (2)			Random Distribution (3)		
	Fitness	Time (s)	Indicator	Fitness	Time (s)	Indicator	Fitness	Time (s)	Indicator	Fitness	Time (s)	Indicator
1	0.2657025	53.88	0	0.14077117	11.66	0	0.10578158	50.04	0	0.07616349	1.10	0
2	0.2657025	49.70	0	0.18150652	1.04	1	0.10578158	35.19	0	0.07616349	38.16	0
3	0.2657025	42.04	0	0.14077117	28.85	0	0.10578158	28.71	0	0.07616349	19.04	0
4	0.2657025	25.47	0	0.14077117	17.44	0	0.10578158	57.14	1	0.07616349	9.91	0
5	0.2657025	29.58	0	0.14077117	38.78	0	0.10578158	27.07	0	0.00877338	12.54	0
6	0.2657025	21.00	0	0.10712355	10.58	0	0.10700493	58.97	1	0.07616349	42.56	0
7	0.2657025	41.85	0	0.14077117	25.31	0	0.10578158	23.73	0	0.07616349	46.67	0
8	0.2657025	52.98	0	0.10712355	19.41	0	0.10578158	30.90	0	0.07616349	72.48	1
9	0.2657025	24.72	0	0.10712355	19.61	0	0.10578158	46.79	0	0.07616349	15.75	0
10	0.2657025	16.19	0	0.14077117	35.51	0	0.10700493	27.45	0	0.07616349	49.69	0
11	0.2657025	20.06	0	0.18150652	57.40	1	0.10578158	29.93	0	0.07616349	27.42	0
12	0.2657025	15.82	0	0.14077117	28.52	0	0.08470728	30.40	0	0.07616349	62.45	0
13	0.2657025	21.28	0	0.14077117	33.73	0	0.10578158	14.18	0	0.07616349	12.63	0
14	0.2657025	33.60	0	0.14077117	16.57	0	0.10578158	25.90	0	0.07616349	44.92	0
15	0.2657025	46.45	0	0.14077117	19.00	0	0.10700493	42.42	0	0.07616349	18.49	0
16	0.2657025	50.35	0	0.14077117	33.53	0	0.10700493	62.44	0	0.07616349	64.79	0
17	0.2657025	24.70	0	0.18150652	60.09	1	0.10578158	28.99	0	0.07616349	30.27	0
18	0.2657025	67.83	0	0.18150652	54.31	0	0.10700493	48.99	0	0.07616349	57.66	0
19	0.2657025	33.62	0	0.14077117	40.36	0	0.10578158	11.60	0	0.07616349	36.84	0
20	0.2657025	26.59	0	0.14077117	23.21	0	0.10700493	60.82	1	0.07616349	30.57	0
21	0.2657025	25.73	0	0.14077117	33.44	0	0.10578158	23.63	0	0.07616349	27.57	0
22	0.2657025	27.16	0	0.14077117	23.07	0	0.08470728	25.45	0	0.07616349	26.31	0
23	0.2657025	30.12	0	0.18150652	50.11	0	0.10578158	28.07	0	0.07616349	71.66	0
24	0.2657025	53.71	0	0.14077117	41.54	0	0.10578158	45.42	0	0.07616349	36.88	0
25	0.2657025	35.94	0	0.18150652	56.33	1	0.10578158	77.28	1	0.07616349	33.48	0
26	0.2657025	32.98	0	0.18150652	52.22	0	0.10700493	65.93	0	0.07616349	77.54	1
27	0.2657025	17.39	0	0.08130398	7.79	0	0.10578158	26.83	0	0.07616349	48.43	0
28	0.2657025	26.66	0	0.10712355	20.88	0	0.10700493	29.02	0	0.07616349	18.44	0
29	0.2657025	23.53	0	0.14077117	59.55	1	0.10700493	33.76	0	0.07616349	35.96	0
30	0.2657025	28.63	0	0.14077117	29.69	0	0.08470728	22.73	0	0.07616349	77.28	0
Avg.	0.2657025	33.32		0.1438075	31.65		0.10404116	37.33		0.07391715	38.25	

Table A-1: Results of running GALAS on test systems

Table A-1 (continued)

	Random Distribution (4)			Random Distribution (5)			Random Distribution (6)			Random Distribution (7)		
	Fitness	Time (s)	Indicator	Fitness	Time (s)	Indicator	Fitness	Time (s)	Indicator	Fitness	Time (s)	Indicator
1	0.1015686	81.42	1	0.10084217	23.89	0	0.1442208	72.33	1	0.02536194	18.72	0
2	0.1019985	58.64	0	0.10084217	26.64	0	0.1442208	29.58	0	0.02504616	48.74	0
3	0.1019985	68.51	1	0.10084217	63.22	1	0.1442208	74.00	1	0.02504616	12.11	0
4	0.1019985	7.65	0	0.10084217	31.21	0	0.1442208	45.73	0	0.02536194	63.96	1
5	0.1019985	4.98	0	0.10084217	11.97	0	0.1442208	49.58	0	0.02536194	59.33	1
6	0.1019985	10.24	0	0.10084217	22.05	0	0.1442208	66.37	1	0.02504616	35.10	0
7	0.1019985	8.16	1	0.10084217	13.45	0	0.1442208	22.09	0	0.02536194	40.12	0
8	0.0362025	39.55	0	0.10084217	27.29	0	0.1442208	55.80	0	0.00697815	32.96	0
9	0.1019985	85.58	1	0.05515133	11.44	0	0.1442208	38.02	0	0.02504616	35.64	0
10	0.1019985	80.15	1	0.10084217	39.26	0	0.1442208	74.81	0	0.02536194	61.59	1
11	0.1019985	80.36	1	0.10084217	38.39	0	0.1442208	37.40	0	0.02536194	67.87	1
12	0.1019985	68.58	1	0.10084217	29.45	0	0.1442208	8.37	0	0.00697815	14.78	0
13	0.1019985	62.50	1	0.10084217	55.63	0	0.1442208	34.03	0	0.00697815	21.34	0
14	0.1019985	22.87	0	0.10084217	33.29	0	0.1442208	22.09	0	0.02536194	29.18	0
15	0.1019985	66.29	1	0.10084217	32.10	0	0.1442208	22.42	0	0.02504616	35.31	0
16	0.1019985	60.56	1	0.10084217	26.10	0	0.1442208	23.66	0	0.02504616	14.59	0
17	0.1019985	53.34	0	0.10084217	9.99	0	0.1442208	18.73	0	0.02536194	68.44	1
18	0.1015686	69.17	1	0.10084217	25.70	0	0.1442208	26.05	0	0.02536194	65.16	1
19	0.1019985	65.01	1	0.10084217	17.54	0	0.1442208	31.06	0	0.02536194	48.42	0
20	0.1019985	68.13	1	0.10084217	44.17	0	0.1442208	11.47	0	0.02504616	35.47	0
21	0.0383861	12.22	0	0.10084217	43.31	0	0.1442208	29.16	0	0.02504616	26.45	0
22	0.1019985	60.14	1	0.10084217	59.33	0	0.1442208	17.58	0	0.02536194	63.40	1
23	0.1019985	58.23	1	0.10084217	53.03	0	0.1442208	64.33	1	0.02536194	52.94	0
24	0.1019985	68.19	1	0.10084217	18.43	0	0.1442208	19.79	0	0.02536194	62.96	1
25	0.0383861	16.64	0	0.10084217	28.24	0	0.1442208	38.22	0	0.02504616	14.99	0
26	0.1015686	64.63	1	0.10084217	9.10	0	0.1442208	12.48	0	0.02536194	66.45	1
27	0.1019985	58.64	1	0.10084217	65.78	1	0.1442208	67.17	1	0.02536194	54.04	0
28	0.1019985	63.70	1	0.10084217	63.67	1	0.1442208	21.19	0	0.02536194	41.41	0
29	0.1019985	65.63	1	0.10084217	58.21	1	0.1442208	25.78	0	0.02536194	62.45	1
30	0.1015686	60.61	1	0.10084217	42.03	0	0.1442208	29.47	0	0.02536194	62.25	1
Avg.	0.09550715	53.01		0.09931914	34.13		0.1442208	36.29		0.02342883	43.87	

Table A-1 (continued)

	Random Distribution (8)			Random Distribution (9)			Random Distribution (10)			5-attribute Distribution		
	Fitness	Time (s)	Indicator	Fitness	Time (s)	Indicator	Fitness	Time (s)	Indicator	Fitness	Time (s)	Indicator
1	0.01981905	26.91	0	0.08999404	24.79	0	0.05796067	63.43	0	0.05401738	175.20	1
2	0.02167425	22.53	0	0.03306803	17.24	0	0.03767321	77.97	0	0.04372603	199.80	1
3	0.02167425	34.81	0	0.08999404	43.14	0	0.05796067	116.82	0	0.04372603	164.40	1
4	0.03713224	68.10	1	0.08999404	10.97	0	0.03492214	79.75	0	0.05401738	169.80	1
5	0.02167425	12.39	0	0.08999404	27.35	0	0.03492214	40.26	0	0.04356638	164.40	1
6	0.02167425	22.38	0	0.08999404	66.92	1	0.03767321	57.46	0	0.0461285	182.40	1
7	0.02167425	10.33	0	0.08999404	29.72	0	0.05796067	159.80	1	0.04372603	178.80	1
8	0.01981905	20.80	0	0.08999404	63.05	1	0.05796067	146.06	1	0.060162	169.80	1
9	0.03713224	67.91	1	0.08999404	34.72	0	0.05796067	137.99	1	0.04356638	222.00	1
10	0.03713224	66.02	1	0.0493488	8.05	0	0.05796067	140.77	1	0.05401738	191.40	1
11	0.03713224	62.62	1	0.08999404	9.01	0	0.05796067	182.36	1	0.02734099	182.40	1
12	0.03713224	64.86	1	0.08999404	60.18	0	0.05796067	217.74	1	0.04372603	146.40	1
13	0.03713224	34.80	0	0.08999404	33.87	0	0.05796067	217.36	1	0.02613645	148.20	1
14	0.01981905	10.01	0	0.08999404	26.66	0	0.05796067	2.70	0	0.0461285	164.40	1
15	0.03713224	44.67	0	0.08999404	22.10	0	0.03767321	65.75	0	0.04372603	202.20	1
16	0.03713224	67.39	0	0.08999404	32.00	0	0.05796067	172.26	1	0.04356638	179.40	1
17	0.03713224	66.70	1	0.08999404	26.99	0	0.05796067	153.12	1	0.04356638	193.20	1
18	0.02167425	11.95	0	0.08999404	66.41	1	0.05796067	140.76	1	0.04372603	186.60	1
19	0.02167425	70.15	1	0.08999404	30.32	0	0.03492214	76.75	0	0.02613645	183.60	1
20	0.02167425	34.30	0	0.08999404	38.71	0	0.05796067	165.49	1	0.04372603	167.40	1
21	0.02167425	34.25	0	0.08999404	18.02	0	0.05796067	124.37	0	0.04372603	178.80	1
22	0.02167425	13.74	0	0.08999404	53.73	0	0.03767321	88.26	0	0.04356638	165.60	1
23	0.02167425	12.68	0	0.08999404	13.50	0	0.05796067	63.09	1	0.02734099	160.80	1
24	0.02167425	20.94	0	0.08999404	59.63	0	0.05796067	31.22	0	0.04372603	175.20	1
25	0.02167425	32.67	0	0.08999404	24.40	0	0.05796067	61.40	1	0.04372603	177.60	1
26	0.03713224	55.32	0	0.08999404	29.07	0	0.05586968	30.50	0	0.04372603	175.80	1
27	0.03713224	59.42	1	0.08999404	14.36	0	0.05796067	66.83	1	0.01680025	193.20	1
28	0.02167425	64.50	1	0.08999404	19.04	0	0.05796067	66.42	1	0.04372603	194.40	1
29	0.01981905	18.65	0	0.08999404	35.54	0	0.05586968	27.21	0	0.02005667	201.00	1
30	0.03713224	64.98	1	0.08999404	10.79	0	0.05796067	35.95	0	0.04372603	168.60	1
Avg.	0.02761009	39.89		0.08674167	31.68		0.05281242	100.33		0.04148511	178.76	

	Time found (min)
1	0:28
2	2:34
3	3:32
4	4:42
5	13:59
6	13:07
7	0:25
8	10:19
9	2:16
10	0:15
11	0:24
12	0:55
13	2:09
14	0:18
15	1:07
16	0:24
17	0:57
18	2:37
19	1:59
20	16:42
21	0:32
22	1:35
23	5:45
24	0:37
25	11:00
26	2:19
27	4:22
28	16:03
29	5:37
30	5:13
Avg.	4:24

Table A-2: Results of running a random search on 4-attribute probabilistic systems

G-structure	Number of S'' structures	Probability of random generation
1234	0	0
234/134/124/123	1024	0.0625
234/134/124	1024	0.0625
234/134/123	1024	0.0625
234/124/123	1024	0.0625
134/124/123	1024	0.0625
234/134/12	512	0.03125
234/124/13	512	0.03125
134/124/23	512	0.03125
234/123/14	512	0.03125
134/123/24	512	0.03125
124/123/34	512	0.03125
234/134	512	0.03125
234/12/14/13	128	0.0078125
134/12/24/23	128	0.0078125
234/124	512	0.03125
124/13/34/23	128	0.0078125
134/124	512	0.03125
234/123	512	0.03125
123/14/34/24	128	0.0078125
134/123	512	0.03125
124/123	512	0.03125
234/14/13	128	0.0078125
134/24/23	128	0.0078125
234/12/14	128	0.0078125
234/12/13	128	0.0078125
12/14/13/34/24/23	16	0.000976563
134/12/24	128	0.0078125
134/12/23	128	0.0078125
124/34/23	128	0.0078125
124/13/34	128	0.0078125
124/13/23	128	0.0078125
123/34/24	128	0.0078125
123/14/34	128	0.0078125
123/14/24	128	0.0078125
234/14	128	0.0078125
234/13	128	0.0078125
14/13/34/24/23	16	0.000976563

	G-structure	Number of S'' structures	Probability of random generation
39	134/24	128	0.0078125
40	134/23	128	0.0078125
41	234/12	128	0.0078125
42	12/14/34/24/23	16	0.000976563
43	12/13/34/24/23	16	0.000976563
44	12/14/13/34/24	16	0.000976563
45	12/14/13/34/23	16	0.000976563
46	12/14/13/24/23	16	0.000976563
47	134/12	128	0.0078125
48	124/34	128	0.0078125
49	124/23	128	0.0078125
50	124/13	128	0.0078125
51	123/34	128	0.0078125
52	123/24	128	0.0078125
53	123/14	128	0.0078125
54	234/1	128	0.0078125
55	14/34/24/23	16	0.000976563
56	13/34/24/23	16	0.000976563
57	14/13/34/24	16	0.000976563
58	14/13/34/23	16	0.000976563
59	14/13/24/23	16	0.000976563
60	134/2	128	0.0078125
61	12/34/24/23	16	0.000976563
62	12/14/34/24	16	0.000976563
63	12/14/34/23	16	0.000976563
64	12/14/24/23	16	0.000976563
65	12/13/34/24	16	0.000976563
66	12/13/34/23	16	0.000976563
67	12/13/24/23	16	0.000976563
68	12/14/13/34	16	0.000976563
69	12/14/13/24	16	0.000976563
70	12/14/13/23	16	0.000976563
71	124/3	128	0.0078125
72	123/4	128	0.0078125
73	1/34/24/23	16	0.000976563
74	14/34/24	16	0.000976563
75	14/34/23	16	0.000976563
76	14/24/23	16	0.000976563

	G-structure	Number of S'' structures	Probability of random generation
77	13/34/24	16	0.000976563
78	13/34/23	16	0.000976563
79	13/24/23	16	0.000976563
80	14/13/34/2	16	0.000976563
81	14/13/24	16	0.000976563
82	14/13/23	16	0.000976563
83	12/34/24	16	0.000976563
84	12/34/23	16	0.000976563
85	12/24/23	16	0.000976563
86	12/14/34	16	0.000976563
87	12/14/24/3	16	0.000976563
88	12/14/23	16	0.000976563
89	12/13/34	16	0.000976563
90	12/13/24	16	0.000976563
91	12/13/23/4	16	0.000976563
92	12/14/13	16	0.000976563
93	1/34/24	16	0.000976563
94	1/34/23	16	0.000976563
95	1/24/23	16	0.000976563
96	14/34/2	16	0.000976563
97	14/24/3	16	0.000976563
98	14/23	16	0.000976563
99	13/34/2	16	0.000976563
100	13/24	16	0.000976563
101	13/23/4	16	0.000976563
102	14/13/2	16	0.000976563
103	12/34	16	0.000976563
104	12/24/3	16	0.000976563
105	12/23/4	16	0.000976563
106	12/14/3	16	0.000976563
107	12/13/4	16	0.000976563
108	1/34/2	16	0.000976563
109	1/24/3	16	0.000976563
110	1/23/4	16	0.000976563
111	14/2/3	16	0.000976563
112	13/2/4	16	0.000976563
113	12/4/3	16	0.000976563
114	1/2/4/3	16	0.000976563

Table A-3: Statistics on G-structure initialization (|V| = 4)

Appendix B: User Manual

In this appendix, instructions on how to properly use my GALAS implementation (Source code: Appendix C) are given. When GALAS is first run, you will be prompted to "Enter the filename containing the definition of the system". GALAS requires a specially formatted text file for defining the overall system S that you would like to find the most acceptable G-structure model(s) of. The first line of this file is $|V|$, where $|V| > 2$. The next $|V|$ lines should each contain a single integer. These integers correspond to the $|dom(v_i)|$ for each $v_i \in V = \{v_1, v_2, \dots, v_{|V|}\}$. The remaining lines of the file each contain a single $f(t_i)$ for each $t_i \in T$. GALAS will automatically detect if the system is relational or probabilistic based on these values. The $f(t_i)$'s must be ordered in a way that is consistent with the enumeration of all states of the system. This enumeration scheme is shown in the table in Figure B-1 for a system defined on 3 attributes, where $|dom(v_1)| = 2$, $|dom(v_2)| = 3$, and $|dom(v_3)| = 2$. The $f(t_i)$'s should be listed in increasing order from t_1 to $t_{\mathbb{C}(S)}$. An example of the text file that would be used to define this example system is shown in Figure B-1.

v_1	v_2	v_3	f(t)
0	0	0	t_1
0	0	1	t_2
0	1	0	t_3
0	1	1	t_4
0	2	0	t_5
0	2	1	t_6
1	0	0	t_7
1	0	1	t_8
1	1	0	t_9
1	1	1	t_{10}
1	2	0	t_{11}
1	2	1	t_{12}

```
3
2
3
2
f(t1)
f(t2)
f(t3)
f(t4)
f(t5)
f(t6)
f(t7)
f(t8)
f(t9)
f(t10)
f(t11)
f(t12)
```

Figure B-1:
Enumeration Scheme (left)
& Example text file (right)

After your system has been properly formatted as a text file, ensure that, before starting GALAS, your working directory contains this file. In R, to find your working directory, use the function getwd(); to change your working directory, use the function setwd("*Path_Of_Directory*"). Once GALAS is started, enter the filename of the text file, excluding the .txt extension. The next prompt asks if you would like to use default values for all customizable parameters of GALAS. If any input is given that is not a lowercase "n", GALAS will run using default values for all customizable parameters. If "n" is given, you will need to supply values for each of the customizable parameters. At any point during execution of GALAS, the program may be manually terminated by hitting the ESC key.

Appendix C: Source Code

```
# A Genetic Algorithm for Locating Acceptable Structures (GALAS)
# Written in the R programming language by Joshua Heath - Spring 2018

#Read in the system definition from the supplied text file
FILE = readline("Enter the filename containing the definition of the system: ")
INFO = scan(paste(getwd(),"/",FILE,".txt",sep = ""))
vals = list()
for(i in 1:(INFO[1])){
    vals[[i]] = rep(0,INFO[i+1])
    for(j in 1:INFO[i+1]){
        vals[[i]][j] = j-1
    }
}
distribution = INFO[(i+2):length(INFO)]

#Decide if the system is a Relational or a Probabilistic system
if(sum(distribution) > 1 && sum(distribution) != 0){
    Relational = TRUE
} else {
    Relational = FALSE
}

#Function - Create the overall system
Overall = function(vals, distribution){

    #Count the total number of states
    totalstates = 1
    for(i in 1:length(vals)){
        totalstates = totalstates * length(vals[[i]])
    }

    #Create the tabular representation of the overall system within a matrix
    overall = matrix(0, totalstates, length(vals)+1)
    for(i in 1:length(vals)){
        if(i == 1){
            change = totalstates / length(vals[[i]])
        } else {
            change = prevchange / length(vals[[i]])
        }
        prevchange = change
        j = 1
        for(k in 1:totalstates){
            overall[k,i] = vals[[i]][j]
            if(k %% change == 0){
                j = (j + 1) %% (length(vals[[i]]) + 1)
```

```
                if(j == 0){
                    j = j + 1
                }
            }
        }
    }

    #Add the distribution to the rightmost column
    if(length(distribution) == totalstates){
        overall[,ncol(overall)] = distribution
    } else {
        print("Error: Number of states given unequal to number of possible states")
        stop()
    }

    #Give the columns names
    names = 1:length(vals)
    names[i+1] = "f(t)"
    colnames(overall) = names

    return(overall)
}

overall = Overall(vals, distribution)

#Function - Return a subsystem of the overall system that is defined on the attributes listed in attr
Project = function(overall, vals, attr, Relational){

    #Create reduced list of attributes
    projvals = list()
    j = 1
    for(i in 1:length(attr)){
        projvals[[j]] = unlist(vals[attr[i]])
        j = j + 1
    }
    projstates = 1
    for(i in 1:length(attr)){
        projstates = projstates * length(projvals[[i]])
    }

    #Create the projected system
    projection = matrix(0,projstates, length(attr) + 1)
    for(i in 1:length(projvals)){
        if(i == 1){
            change = projstates / length(projvals[[i]])
        } else {
            change = prevchange / length(projvals[[i]])
        }
```

```
    prevchange = change
    j = 1
    for(k in 1:projstates){
        projection[k,i] = projvals[[i]][j]
        if(k %% change == 0){
            j = (j + 1) %% (length(projvals[[i]]) + 1)
            if(j == 0){
                j = j + 1
            }
        }
    }
}

#Add the marginal distribution to the rightmost column of the subsystem
totalstates = nrow(overall)
for(i in 1:totalstates){
    for(j in 1:projstates){
        DoSum = TRUE
        k = 1
        while(k <= length(attr) && DoSum == TRUE){
            if(overall[i,attr[k]] != projection[j,k]){
                DoSum = FALSE
            }
            k = k + 1
        }
        if(DoSum == TRUE){
            projection[j,length(attr)+1] = projection[j,length(attr)+1] + overall[i,ncol(overall)]
        }
    }
}

#If the overall system is relational, then raise all nonzero probabilities to 1
if(Relational == TRUE){
    for(i in 1:projstates){
        if(projection[i,ncol(projection)] > 0){
            projection[i,ncol(projection)] = 1
        }
    }
}

#Give the columns names
names = c()
for(i in 1:length(attr)){
    names[i] = attr[i]
}
names[i+1] = "f(t)"
colnames(projection) = names
```

```
    return(projection)
}

#Function - Perform a modified version of the relational or probabilistic join on two subsystems
PRJoin = function(sys1, sys2, overall, Relational){

    proj1 = strtoi(colnames(sys1)[1:length(colnames(sys1))-1])
    proj2 = strtoi(colnames(sys2)[1:length(colnames(sys2))-1])

    #Find shared values
    proj1vector = proj2vector = same = rep(0,ncol(overall)-1)
    for(i in 1:length(proj1)){
        proj1vector[proj1[i]] = 1
    }
    for(i in 1:length(proj2)){
        proj2vector[proj2[i]] = 1
    }
    for(i in 1:length(vals)){
        if(proj1vector[i] == 1 && proj2vector[i] == 1){
            same[i] = 1
        }
    }

    #Begin creating the result of the join
    result = overall
    for(i in 1:nrow(sys1)){
        Share = rep(1,nrow(sys2))
        sum = 0
        for(j in 1:nrow(sys2)){

            #Find rows in the sys2 that coincide with the current state of sys1
            for(k in 1:length(same)){
                if(Share[j] == 1 && same[k] == 1){
                    if(sys1[i,toString(k)] != sys2[j,toString(k)]){
                        Share[j] = 0
                    }
                }
            }

            #Keep a sum of probabilities of states in sys2 that coincide with current state of sys1
            if(Share[j] == 1){
               sum = sum + sys2[j,ncol(sys2)]
            }
        }

        #Find the state in the result of the join that corresponds with the state in both sys1 and sys2
        for(j in 1:nrow(sys2)){
            if(Share[j] == 1){
```

```
                IsSame = FALSE
                Q = 0
                while(IsSame == FALSE){
                    Q = Q + 1
                    IsSame = TRUE
                    for(k in 1:length(proj1)){
                        if(sys1[i,toString(proj1[k])] != result[Q,toString(proj1[k])]){
                            IsSame = FALSE
                        }
                    }
                    for(k in 1:length(proj2)){
                        if(sys2[j,toString(proj2[k])] != result[Q,toString(proj2[k])]){
                            IsSame = FALSE
                        }
                    }
                }

                #Assign the calculated probability to the correct state in the result of the join
                if(sum != 0){
                    result[Q,ncol(result)] = sys1[i,ncol(sys1)] * (sys2[j,ncol(sys2)]) / sum
                } else {
                    result[Q,ncol(result)] = 0
                }
            }
        }
    }

    #Raise all nonzero values to 1 if the subsystems are relational
    if(Relational == TRUE){
        for(i in 1:nrow(result)){
            if(result[i,ncol(result)] > 0){
                result[i,ncol(result)] = 1
            }
        }
    }
    return(result)
}

#Function - Perform the IPFP on a structure system
IPFP = function(struct, overall, vals, Relational){

    #Create a system defined on the uniform distribution
    uniform = overall
    uniform[,(length(vals)+1)] = rep(1/nrow(overall),nrow(overall))

    #The procedure begins
    converged = FALSE
    i = 1
```

```
    while(converged == FALSE){

        #Iteratively take the probabilistic join of all subsystems in the structure system
        for(j in 1:length(struct)){
            if(i == 1 && j == 1){
                reconstruct = PRJoin(Project(overall, vals, struct[[j]], Relational), uniform, overall,
                Relational)
            } else {
                reconstruct = PRJoin(Project(overall, vals, struct[[j]], Relational), reconstruct,
                overall,Relational)
            }
        }

        #If the overall distribution was similer to last distribution, then the distribution has converged
        if(i != 1){
            sum = 0
            for(j in 1:nrow(reconstruct)){
                sum = abs(reconstruct[j,ncol(reconstruct)] - prevreconstruct[j,ncol(prevreconstruct)])
            }
            if(sum < 0.0001){
                converged = TRUE
            }
        }
        prevreconstruct = reconstruct
        i = i + 1
    }

    return(reconstruct)
}

#Function - Return the IPS of a reconstructed relational system
IPS = function(sys, overall){
    dif = sys[,ncol(sys)] - overall[,ncol(overall)]
    return(sum(dif))
}

#Function - Return the fitness score for a given structure
Fitness = function(struct, reconstruct, overall, expectedline, vals, Relational){
    if(Relational == TRUE){
        return( ((expectedline[1]*StructSize(struct,vals)) + expectedline[2]) - IPS(reconstruct,overall) )
    } else {
        return( ((expectedline[1]*StructSize(struct,vals)) + expectedline[2]) - Entropy(reconstruct) )
    }
}

#Function - Return the entropy of a probabilistic system
Entropy = function(sys){
    H = 0
```

```
    for(i in 1:nrow(sys)){
        if(sys[i,ncol(sys)] != 0){
            H = H - sys[i,ncol(sys)] * log2(sys[i,ncol(sys)])
        }
    }
    return(H)
}
```

#Function - Return the storage requirement of a structure system

```
StructSize = function(struct, vals){
    TotalSize = 0
    for(i in 1:length(struct)){
        SizeOfProj = 1
        for(j in 1:length(struct[[i]])){
            SizeOfProj = SizeOfProj * length(vals[[struct[[i]][j]]])
        }
        TotalSize = TotalSize + SizeOfProj
    }
    return(TotalSize)
}
```

#Get the most-refined structure system

```
refinedstructure = list()
for(i in 1:length(vals)){
    refinedstructure[[i]] = c(i)
}
refinedsystem = IPFP(refinedstructure,overall,vals,Relational)
expectedline = Line(overall, refinedsystem, vals, Relational)
```

#Function - Calculate formula for the border line between acceptable and unacceptable G-structures

```
Line = function(overall, refinedsystem, vals, Relational){

    #First entry of "line" is the slope, second is the y-intercept(IPS/Entropy)
    line = rep(0,2)
    if(Relational == TRUE){
        line[1] = IPS(refinedsystem, overall) / (StructSize(refinedstructure,vals) - nrow(overall))
        line[2] = -(line[1] * nrow(overall))
    } else {
        line[1] = (Entropy(refinedsystem) - Entropy(overall)) / (StructSize(refinedstructure,vals) -
        nrow(overall))
        line[2] = Entropy(overall) - (line[1] * nrow(overall))
    }
    return(line)
}
```

#Function - Return the reference table with all meaningful subsystems ($P_{|V|}$) of the overall system

```
Generate = function(vals){
    HashTable = list(rep(0,(2^length(vals)) - 2))
```

```
    entry = 1
    Binary = rep(0,length(vals))

    #Assign each attribute a bit in a bit string and find subsystems for all bit strings of length |V|
    while(!all(Binary == 1)){
        projection = c()
        if(sum(Binary == 1) > 0){
            for(i in 1:length(Binary)){
                if(Binary[i] == 1){
                    projection = append(projection, i)
                }
            }
        }

        #If a valid subsystem is found, add it to the reference table
        if(!is.null(projection)){
            HashTable[[entry]] = projection
            entry = entry + 1
        }

        #Increment the binary represenation
        i = 1
        AllSet = FALSE
        while(i <= length(vals) && AllSet == FALSE){
            if(Binary[i] == 1){
                Binary[i] = 0
                i = i + 1
            } else {
                Binary[i] = 1
                AllSet = TRUE
            }
        }
    }
    return(HashTable)
}

reference = Generate(vals)

#Function - Test if a given structure is an H-structure
Covers = function(struct, vals){
    IsIn = rep(0,length(vals))
    for(i in 1:length(struct)){
        for(j in 1:length(struct[[i]])){
            IsIn[struct[[i]][j]] = 1
        }
    }
    return(IsIn)
}
```

```
#Function - Return the genome that the structure maps to
PhenoToGeno = function(pheno, reference){
    geno = rep(0,length(reference))
    for(i in 1:length(pheno)){
        for(l in 1:length(reference)){
            if(length(reference[[l]]) == length(unlist(pheno[[i]]))){
                if(all(reference[[l]] == unlist(pheno[[i]]))){
                    geno[l] = 1
                }
            }
        }
    }
    return(geno)
}

#Set default values
input = readline(prompt = "Would you like to use default values for customizable parameters? (y/n): ")
if(input == "n"){
    USERINPUT = TRUE
} else {
    USERINPUT = FALSE
}


#Set user's parameter values
if(USERINPUT){
    PopSize = as.integer(readline(prompt = "Enter Population Size: "))
    PoolSize = as.integer(readline(prompt = paste("Enter a mating pool size less than or equal to ",
    PopSize, ": ",sep="")))
    MaxIters = as.integer(readline(prompt = "Enter Maximum number of Iterations: "))
    Selection = toString(readline(prompt = "Type of Selection - Truncation(Tr), Tournament(To), or
    Roullete Wheel(R): "))
    Crossover = toString(readline(prompt = "Type of Crossover - Single-point(S), Multi-point(M), or
    Uniform(U): "))
    ProbOfCrossover = as.double(readline(prompt = "Enter the probability of crossover [0,1]: "))
    ProbOfMutation = as.double(readline(prompt = "Enter the probability of mutation [0,1]: "))
    if(Crossover == "S"){
        SingleCrossPoint = floor(ncol(Genotypes)/2)
    }
    if(PoolSize %% 2 == 1){
        PoolSize = PoolSize + 1
    }
} else {
    if(length(vals) > 4){
        PopSize = 30
        PoolSize = 20
    } else {
```

```
        PopSize = 10
        PoolSize = 6
    }
    MaxIters = 50
    Selection = "R"
    Crossover = "M"
    ProbOfCrossover = 0.8
    ProbOfMutation = 1/length(reference)
}

#Begin Genetic Algorithm
start = Sys.time()

#Create a random population of genomes
Genotypes = matrix(0, PopSize, length(reference))
for(i in 1:PopSize){
    for(j in 1:ncol(Genotypes)){
        Genotypes[i,j] = sample(0:1,1)
    }
}

recalculate = rep(1,PopSize)
Phenotypes = list()
GoodPhenos = list()
GoodGenos = list()
GoodPhenosScores = list()
FitScores = rep(0,PopSize)
NormalizedFitScores = rep(0,PopSize)
converged = FALSE
Generation = 1

#Run until the maximum number of iterations have been reached or the population has converged
while(Generation <= MaxIters && converged == FALSE){

    #Map genomes to actual H-structures
    for(i in 1:PopSize){
        if(recalculate[i] == 1){
            covers = FALSE
            Phenotypes[[i]] = list()
            NumOfProj = 0
            IsIn = rep(0,length(vals))
            j = 1
            while(j <= ncol(Genotypes)){
                if(Genotypes[i,j] == 1){
                    NumOfProj = NumOfProj + 1
                    Phenotypes[[i]][NumOfProj] = reference[j]
                    IsIn = Covers(Phenotypes[[i]], vals)
                }
```

```
            j = j + 1
        }
        if(all(IsIn == 1)){
            covers = TRUE
        }

        #If the structure does not cover the set, then add the singleton subsystems
        if(covers == FALSE) {
            for(k in 1:length(vals)){
                singleton = c()
                if(IsIn[k] == 0){
                    singleton[1] = k
                    NumOfProj = NumOfProj + 1
                    Phenotypes[[i]][NumOfProj] = singleton
                    for(l in 1:length(reference)){
                        if(length(reference[[l]]) == 1){
                            if(k == reference[[l]]){
                                Genotypes[i,l] = 1
                            }
                        }
                    }
                }
            }
        }
    }
}

#Reduce the H-structures
for(i in 1:length(Phenotypes)){
    if(recalculate[i] == 1){
        if(length(Phenotypes[[i]]) > 1){
            hyperedges = matrix(0,length(Phenotypes[[i]]),length(vals))
            for(j in 1:length(Phenotypes[[i]])){
                for(k in 1:length(Phenotypes[[i]][[j]])){
                    hyperedges[j,Phenotypes[[i]][[j]][k]] = 1
                }
            }

            #Remove any subsystems that is a subsystem of another subsystem in that structure
            for(j in 1:(nrow(hyperedges)-1)){
                for(k in (j+1):nrow(hyperedges)){
                    first = 0
                    last = 0
                    for(l in 1:length(vals)){
                        if(hyperedges[j,l] > hyperedges[k,l]){
                            last = 1
                        } else if(hyperedges[j,l] < hyperedges[k,l]){
                            first = 1
```

```
                    }
                }
                if(first == 1 && last == 0){
                    for(l in 1:length(reference)){
                        if(length(reference[[l]]) == length(Phenotypes[[i]][[j]]) &&
                        all(reference[[l]] == unlist(Phenotypes[[i]][j]))){
                            Genotypes[i,l] = 0
                        }
                    }
                    Phenotypes[[i]][j] = 0
                }
                if(first == 0 && last == 1){
                    for(l in 1:length(reference)){
                        if(length(reference[[l]]) == length(Phenotypes[[i]][[k]]) &&
                        all(reference[[l]] == unlist(Phenotypes[[i]][k]))){
                            Genotypes[i,l] = 0
                        }
                    }
                    Phenotypes[[i]][k] = 0
                }
            }
        }
        for(j in length(Phenotypes[[i]]):1){
            if(all(unlist(Phenotypes[[i]][j]) == 0)){
                Phenotypes[[i]][j] = NULL
            }
        }
    }
}

#Calculate the fitness for all members of the population; normalize this data to the range [1 - 100]
for(i in 1:PopSize){
    if(recalculate[i] == 1){
        cat(i," ")
        reconstruct = IPFP(Phenotypes[[i]], overall, vals, Relational)
        FitScores[i] = Fitness(Phenotypes[[i]], reconstruct, overall, expectedline, vals, Relational)

        #If fitness is above 0, record phenotype and fitness score if it has not already been seen
        if(FitScores[i] > 0){
            Geno = PhenoToGeno(Phenotypes[[i]],reference)
            AlreadyFound = FALSE
            j = 1
            while(j <= length(GoodGenos) && AlreadyFound == FALSE){
                if(all(GoodGenos[[j]] == Geno)){
                    AlreadyFound = TRUE
                }
                j = j + 1
```

```
            }
            if(AlreadyFound == FALSE){
                GoodGenos[[length(GoodGenos)+1]] = Geno
                GoodPhenos[[length(GoodPhenos)+1]] = Phenotypes[[i]]
                GoodPhenosScores[[length(GoodPhenosScores)+1]] = FitScores[i]
            }
        }
    }
}

#Check if the population has converged
HighestScore = max(FitScores)
LowestScore = min(FitScores)
if(HighestScore - LowestScore != 0){
    for(i in 1:PopSize){
        if(recalculate[i] == 1){
            NormalizedFitScores[i] = 1 + (((FitScores[i] - (-2))*(100 - 1)) / (2 - (-2)))
        }
    }
} else {
    converged = TRUE
}

#Print the average fitness in the population, highest fitness, and G-structure with that fitness
cat("\nAverage: ", sum(FitScores)/PopSize, "\n")
cat("Max fitness of this generation: ", HighestScore, "\n")
cat("Best G-structure in the population: ")
for(i in 1:length(Phenotypes[[which.max(FitScores)]])){
    cat(unlist(Phenotypes[[which.max(FitScores)]][[i]]))
    if(i < length(Phenotypes[[which.max(FitScores)]])){
        cat(" / ")
    }
}
cat("\n")

#If the population has not converged yet…
if(converged == FALSE){

    #Select the members of the mating pool
    MatingPool = rep(0,PoolSize)
    if(Selection == "Tr"){

        #Truncation Selection
        DecreasingOrderPop = order(NormalizedFitScores)
        for(i in 1:PoolSize){
            MatingPool[i] = DecreasingOrderPop[i]
        }
    } else if(Selection == "To"){
```

```
    #Tournament Selection
    TournSize = 3
    for(i in 1:PoolSize){
        Tourn = rep(0,PopSize)
        for(j in 1:TournSize){
            random = ceiling(runif(1,0,PopSize))
            Tourn[random] = NormalizedFitScores[random]
        }
        DecreasingOrder = order(Tourn, decreasing = TRUE)
        MatingPool[i] = DecreasingOrder[1]
    }
} else if(Selection == "R"){

    #Roulette Wheel Selection (a.k.a. Fitness Proportional Selection)
    sum = sum(NormalizedFitScores)
    ProbOfSelect = rep(0,PopSize)
    CumulativeProb = 0
    for(i in 1:PopSize){
        CumulativeProb = CumulativeProb + (NormalizedFitScores[i]/sum)
        ProbOfSelect[i] = CumulativeProb
    }
    for(i in 1:PoolSize){
        random = runif(1,0,1)
        for(j in 1:PopSize){
            if(j == 1){
                if(random <= ProbOfSelect[j]){
                    MatingPool[i] = j
                }
            } else if(j > 1 && j <= PopSize){
                if(random <= ProbOfSelect[j] && random > ProbOfSelect[j-1]){
                    MatingPool[i] = j
                }
            } else {
                if(random > ProbOfSelect[j-1]){
                    MatingPool[i] = j
                }
            }
        }
    }
}

#Pair mating pool genomes and perform crossover with a probability of ProbOfCrossover
OffspringGeno = list()
NumOfOff = 0
i = 1
while(i <= PoolSize){
    if(runif(1,0,1) < ProbOfCrossover){
```

```
NumOfOff = NumOfOff + 2
if(Crossover == "S"){
    #Single-Point Crossover
    OffspringGeno[[NumOfOff - 1]] = c(Genotypes[MatingPool[i],1:SingleCrossPoint],
    Genotypes[MatingPool[i+1],(SingleCrossPoint+1):ncol(Genotypes)])
    OffspringGeno[[NumOfOff]] = c(Genotypes[MatingPool[i+1],1:SingleCrossPoint],
    Genotypes[MatingPool[i],(SingleCrossPoint+1):ncol(Genotypes)])

} else if(Crossover == "M"){
    #Multi-Point Crossover
    Marks = rep(0,length(vals)-1)
    for(j in 1:length(Marks)){
        Marks[j] = 2^j
    }
    MarksIndex = 1
    for(j in 1:ncol(Genotypes)){
        if(MarksIndex <= length(vals)){
            if(j == Marks[MarksIndex] && MarksIndex == 1){
                if(runif(1,0,1) <= 0.5){
                    OffspringGeno[[NumOfOff - 1]] =
                    c(Genotypes[MatingPool[i],1:(Marks[MarksIndex]-1)])
                    OffspringGeno[[NumOfOff]] =
                    c(Genotypes[MatingPool[i+1],1:(Marks[MarksIndex]-1)])
                } else {
                    OffspringGeno[[NumOfOff - 1]] =
                    c(Genotypes[MatingPool[i+1],1:(Marks[MarksIndex]-1)])
                    OffspringGeno[[NumOfOff]] =
                    c(Genotypes[MatingPool[i],1:(Marks[MarksIndex]-1)])
                }
                MarksIndex = MarksIndex + 1
            }
            if(j == Marks[MarksIndex] && MarksIndex != 1 && MarksIndex <=
            (length(vals)-1)){
                if(runif(1,0,1) <= 0.5){
                    OffspringGeno[[NumOfOff - 1]] = c(OffspringGeno[[NumOfOff - 1]] ,
                    Genotypes[MatingPool[i],(Marks[MarksIndex-
                    1]):(Marks[MarksIndex]-1)])
                    OffspringGeno[[NumOfOff]] = c(OffspringGeno[[NumOfOff]] ,
                    Genotypes[MatingPool[i+1],(Marks[MarksIndex-
                    1]):(Marks[MarksIndex]-1)])
                } else {
                    OffspringGeno[[NumOfOff - 1]] = c(OffspringGeno[[NumOfOff - 1]] ,
                    Genotypes[MatingPool[i+1],(Marks[MarksIndex-
                    1]):(Marks[MarksIndex]-1)])
                    OffspringGeno[[NumOfOff]] = c(OffspringGeno[[NumOfOff]] ,
                    Genotypes[MatingPool[i],(Marks[MarksIndex-
                    1]):(Marks[MarksIndex]-1)])
                }
```

```
                    MarksIndex = MarksIndex + 1
                }
                if(j == Marks[MarksIndex-1] && MarksIndex == (length(vals))){
                    if(runif(1,0,1) <= 0.5){
                        OffspringGeno[[NumOfOff - 1]] = c(unlist(OffspringGeno[[NumOfOff
                        - 1]]) , Genotypes[MatingPool[i],(Marks[MarksIndex-
                        1]):ncol(Genotypes)])
                        OffspringGeno[[NumOfOff]] = c(unlist(OffspringGeno[[NumOfOff]]) ,
                        Genotypes[MatingPool[i+1],(Marks[MarksIndex-
                        1]):ncol(Genotypes)])
                    } else {
                        OffspringGeno[[NumOfOff - 1]] = c(unlist(OffspringGeno[[NumOfOff
                        - 1]]) , Genotypes[MatingPool[i+1],(Marks[MarksIndex-
                        1]):(ncol(Genotypes))])
                        OffspringGeno[[NumOfOff]] = c(unlist(OffspringGeno[[NumOfOff]]) ,
                        Genotypes[MatingPool[i],(Marks[MarksIndex-1]):ncol(Genotypes)])
                    }
                    MarksIndex = MarksIndex + 1
                }
            }
        }
    } else if(Crossover == "U"){
        #Uniform Crossover
        OffspringGeno[[NumOfOff]] = rep(0,ncol(Genotypes))
        for(j in 1:ncol(Genotypes)){
            if(runif(1,0,1) <= 0.5){
                OffspringGeno[[NumOfOff - 1]][j] = Genotypes[MatingPool[i],j]
                OffspringGeno[[NumOfOff]][j] = Genotypes[MatingPool[i+1],j]
            } else {
                OffspringGeno[[NumOfOff - 1]][j] = Genotypes[MatingPool[i+1],j]
                OffspringGeno[[NumOfOff]][j] = Genotypes[MatingPool[i],j]
            }
        }
    }
}
i = i + 2
}

#Replace the worst performing genomes with the offspring
recalculate = rep(0,PopSize)
DecreasingOrderPop = order(NormalizedFitScores)
if(NumOfOff > 0){
    for(i in 1:NumOfOff){
        recalculate[DecreasingOrderPop[i]] = 1
        Genotypes[DecreasingOrderPop[i],] = OffspringGeno[[i]]
    }
}
```

```
        #Mutate the offspring that were just inserted into the population
        for(i in 1:NumOfOff){
            for(j in 1:ncol(Genotypes)){
                if(runif(1,0,1) < ProbOfMutation){
                    Genotypes[DecreasingOrderPop[i],j] = (Genotypes[DecreasingOrderPop[i],j] + 1) %% 2
                }
            }
        }
    }

    end = Sys.time()
    print(end - start)
    Generation = Generation + 1
}
```

References

[1] W. R. Ashby, “Constraint analysis of many-dimensional relation” in *Progress in Biocybernetics*, Vol. 2, pp. 10-18, edited by N. Wiener and J. P. Schade, Elsevier, Amsterdam, 1965.

[2] S. Athmakuri, “A graphical tool for structure modeling”, *M.S. Thesis*, SUNY Institute of Technology, Utica, New York, 1995.

[3] R. E. Cavallo, “Lattice of Structure Models and Database Schemes”, *International Journal of General Systems*, Vol. 20, pp. 247-274, 1992.

[4] R. E. Cavallo and G. J. Klir, “Reconstructability Analysis of Multi-dimensional Relations: A Theoretical Basis for Computer-Aided Determination of Acceptable Systems Models”, *International Journal of General Systems*, Vol. 5, pp. 143-171, 1979.

[5] R. E. Cavallo and G. J. Klir, “Reconstructability Analysis: Evaluation of Reconstruction Hypotheses”, *International Journal of General Systems*, Vol. 7, pp. 7-32, 1981.

[6] R. E. Cavallo and G. J. Klir, “Decision Making in Reconstructability Analysis”, *International Journal of General Systems*, Vol. 8, pp. 243-255, 1982.

[7] R. E. Cavallo and M. Pittarelli, “The Theory of Probabilistic Databases”, *Proceedings of the 13th VLDB Conference*, pp. 71-81. Brighton, 1987.

[8] W. E. Deming and F. F. Stephan, “On a Least Squares Adjustment of a Sampled Frequency Table When the Expected Marginal Totals are Known”, *The Annals of Mathematical Statistics*, Vol. 11, No. 4, pp. 427-444, 1940.

[9] M. K. Dobransky and M. J. Wierman, “Genetic Algorithms: A Search Technique Applied to Behavior Analysis”, *International Journal of General Systems*, Vol. 24, No. 1, pp. 125-135, 1996.

[10] A. E. Eiben and J. E. Smith, “Introduction to Evolutionary Computing”, *Springer*, 2003.

[11] M. A. El-Beltagy and A. J. Keane, “A Comparison of Genetic Algorithms with Various Optimization Methods for Multilevel Problems”, Submitted to *Artificial Intelligence in Engineerin*g, 1999.

[12] P. Godefroid and S. Khurshid. “Exploring Very Large State Spaces Using Genetic Algorithms”, *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of* Systems, pp. 266–280. Springer, 2002.

[13] J. H. Holland, “Genetic Algorithms”, *Scientific American*, pp 66-72, 1992.

[14] J. H. Holland, “Adaptation in Natural and Artificial Systems”, *University of Michigan Press*, 1975.

[15] E.T. Jaynes, “On the Rationale of Maximum-Entropy Methods”, *Proceeding of the IEEE*, Vol. 70, No. 9, pp. 939-952, 1982.

[16] G. J. Klir, “Architecture of Systems Problem Solving”, *Plenum Press*, 1985.

[17] G. J. Klir and H. J. J. Uyttenhove, “On the problem of computer-aided structure identification: some experimental observations and resulting guidelines”, *International Journal of Man-Machine Studies*, Vol. 9, No. 6, pp. 593-628, 1977.

[18] P.M Lewis II, “Approximating probability distributions to reduce storage requirements”, *Information and Control*, Vol. 2, No. 3, pp. 214-225, 1959.

[19] R. F. Madden and W. R. Ashby, “The identification of many-dimensional relations”, *International Journal of General Systems*, Vol. 3, No. 4, pp. 343-356, 1972.

[20] K. F. Man, K. S. Tang, and S. Kwong, “Genetic Algorithms: Concepts and Applications”, *IEEE Transactions on Industrial Electronics*, Vol. 43, No. 5, pp. 519-534, 1996.

[21] D. Naidu, “Tools for Evaluation of Reconstruction Hypotheses - Iterative Proportional Fitting Procedure vs. Optimization by the Lagrange Multiplier Method”, *M.S. Thesis*, SUNY Institute of Technology, Utica, New York, 2001.

[22] C. E. Shannon, “A Mathematical Theory of Communication”, *The Bell Systems Technical Journal*, Vol. 27, pp. 379-423, 623-656, 1948.

[23] "System" Merriam-Webster.com. Accessed May 11, 2018. https://www.merriam-webster.com/dictionary/system.