Anoma Research Topics | TECHNICAL REPORT

Constraint Satisfaction Problems: A Survey for Anoma

Anthony Hart[a]

[a]Heliax AG

* *E-Mail: anthony@heliax.dev*

Abstract

This paper serves as a broad introduction to Constraint Satisfaction Problems (CSPs) tailored for professionals working on Anoma, a decentralized financial framework. Our primary aim is to build a robust intuition for CSPs, preparing the Anoma community for its application in optimizing intents—defined as desires for future states of the financial system. We explore a wide array of CSP examples, including but not limited to Boolean Satisfiability and Integer Linear Programming, to equip readers with a general understanding of the field. We also include an overview of CSP tractability, aiming to give the reader an understanding of the limitations of solvability. A unique emphasis is placed on compositional methods, facilitating distributed problem-solving across multiple agents. Towards the end, we formalize the notion of 'intents' within the framework of Constraint Optimization.

Keywords: Constraint Satisfaction; Intents; Formal Logic; Automated Reasoning; Surveys;

(Received: October 10, 2023; Published: October 24, 2023; Version: October 24, 2023)

Contents

DOI: 10.5281/zenodo.10019113

1. Introduction to CSPs

Constraint Satisfaction Problems (CSPs) represent a versatile and powerful area of study, offering a general framework that can be applied to a myriad of domains—from logistics and scheduling to artificial intelligence and financial optimization. The true strength of CSPs lies in their generality; new problems from developing fields can often be translated into CSPs, allowing one to leverage a wealth of mature, efficient solver technologies. CSPs are an indispensable tool for modeling complex systems with interacting variables with restrictions. For a good, thorough reference on the subject, see (Rossi et al., 2006) and (Tsang, 1993).

1.1. What CSPs Are. A CSP (Constraint Satisfaction Problem) is fundamentally built out of a set of generating relations, over a domain d. A CSP starts with a finite set of "atomic relations", each with a fixed arity. A relation of arity n can be defined as any set of n-tuples of elements of d. These atomic relations serve as the building blocks for defining more complex relations within the CSP. Such construction is achieved through "relational composition", a process involving both conjunction and existential quantification over shared variables. For instance, given relations $R_1(x, y)$ and $R_2(y, z)$, a new relation can be defined as

$$R_3(x, z) := \exists y.R_1(x, y) \wedge R_2(y, z) \qquad [1]$$

It should be noted that we allow variables to be freely deleted, duplicated, and equated as part of relational definitions. An algebra of relations closed under relational composition is called a "relational clone". A familiar example is linear programming, where atomic relations like $x + y \leq z$ and $x = y$ can be composed to form more intricate linear systems. In that example, the variable domain might be the set of rationals.

Given a CSP, an instance of this CSP consists of

- X: a set of variables
- C: a set of constraints (called "clauses") consisting of expressions of the form $R(x_1, x_2, ..., x_n)$, where R is an n-ary, atomic relation from our CSP, and $x_i \in X$, for all i

An assignment of an instance consists of a map from X onto d, and the truth value (or valuation) of an instance relative to an assignment is the evaluation of the conjunction of all constraints with respect to the assignments. If an assignment produces a true valuation, then the instance is said to be "satisfiable", and the assignment is called a "satisfying assignment" for that instance. One may identify CSP instances with the 0-ary relations definable within the CSP.

Remark 1. Not all authors make a clear distinction between CSPs and their instances. Some will refer to CSP instances as just "CSPs". Research on CSP complexity is consistent on making this distinction, while research on distributed CSPs is not. We've decided to be clear, as confusion may arise if a distinction is not made. One does not need a fixed set of variables to define the CSPs describing linear programming or boolean satisfiability, for example, while one does need a fixed set of variables to define their instances.

Remark 2. There is a close relationship between relational database theory and the theory of CSPs. This connection will not be emphasized here, but some terminology from this link should be noted. Rarely, a CSP will, itself, be called a "database", as, in the finite domain case, it is the same thing as a relational database. What's more common is the usage of the word "query" to describe CSP instances, which one will most often see in the field of constraint programming.

1.2. Example: Boolean Satisfiability. By far the most well-studied CSPs are from the domain of boolean satisfiability. Each problem has the same domain; the booleans, 0 and 1. There are several versions of the problem. The simplest is called 2-SAT and consists of three binary relations, $x \vee y$, $\neg x \vee y$, and $\neg x \vee \neg y$. By commutativity, we can also define $x \vee \neg y$ as $\neg y \vee x$. An instance of a boolean satisfiability problem will generally be a formula in conjunctive normal form. As an example, we have the following instance;

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2) \qquad [2]$$

We can note that this instance has three variables, x_1, x_2, and x_3. 2-SAT is a tractable CSP, meaning that we can always efficiently calculate if a solution exists, and what it is if it does. The standard reasoning method for 2-SAT is called binary resolution, and states that;

$$\frac{C_1 \vee p \quad C_2 \vee \neg p}{C_1 \vee C_2} \tag{3}$$

this allows us to derive new clauses from the old ones by connecting them with shared variables of opposite polarity. We can, in particular, apply it to the clauses $x_1 \vee \neg x_2$ and $\neg x_1 \vee \neg x_2$, allowing us to derive $\neg x_2 \vee \neg x_2 = \neg x_2$. This tells us that x_2 must be 0. Substituting this into our original problem and simplifying leads to

$$(\neg x_1 \vee x_3) \wedge \neg x_3 \tag{4}$$

We can clearly see that x_3 must be 0 since it appears negated and alone within a clause. Making this observation and substituting the implied value is called "unit propagation". By doing this, we end up with

$$\neg x_1 \tag{5}$$

forcing x_1 to be 0. Ultimately, we conclude that this problem has a satisfying instance, that being the map $\langle x_1 \to 0; x_2 \to 0; x_3 \to 0 \rangle$.

While it's convenient to work with 2-SAT, it isn't very expressive. Most boolean SAT solving is done using at least 3-SAT. This version of the problem has the relations, $x \vee y \vee z$, $\neg x \vee y \vee z$, $\neg x \vee \neg y \vee z$, and $\neg x \vee \neg y \vee \neg z$.

Solvers often don't enforce a specific argument number. If we need to express a clause in 3-SAT that has more than three clauses, we can split it up by introducing dummy variables. For example,

$$\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4 \vee x_5 \tag{6}$$

can be defined as

$$\exists d_1, d_2.(\neg x_1 \vee \neg x_2 \vee d_1) \wedge (\neg d_1 \vee \neg x_3 \vee d_2) \wedge (\neg d_2 \vee x_4 \vee x_5) \tag{7}$$

by applying binary resolution, we can recover the original formula from its 3-SAT rendition. A similar procedure cannot be used to turn a 3-SAT formula into a 2-SAT formula, indicating a true jump in complexity. One does not gain meaningful expressiveness for N-SAT beyond N=3.

There does not exist a known method to efficiently solve 3-SAT problems, and it's strongly believed that such a method cannot exist. 3-SAT is, perhaps, the Ur-Example of a complete, hard problem. It is NP-complete, meaning that any problem whose solution is efficiently verifiable can be efficiently expressed as a 3-SAT problem. This also means that it is at least as hard as the hardest of such problems, since, if 3-SAT can be solved efficiently, then all problems efficiently expressible in 3-SAT can as well. It is also known that all finite-domain CSPs are, at worst, NP-hard, meaning they can all be efficiently translated into 3-SAT. 3-SAT is only one of infinite many NP-complete finite domain CSPs; see section 5.1 for more details on this.

Despite its theoretical difficulty, much practical progress has been made. There are a wide variety of methods that can solve special cases quite efficiently. Research into 3-SAT is one of the most active areas in computer science, and for good reason. Its flexibility means that industrial benefits from solver improvements can have wide-reaching consequences.

See the handbook (Biere et al., 2020) for reference.

1.3. Example: Diophantine Equations. Diophantine Equations are an interesting example of a CSP with, in some sense, maximal expressiveness. It is generated from all polynomial equations over the domain of the integers. The most famous example of such a polynomial is the equation at the center of Fermat's Last Theorem, now known to have no satisfying positive assignments.

$$x^3 + y^3 = z^3 \tag{8}$$

By Matiiasevich's theorem, we know that Diophantine Equations are Turing complete. This means that all computable relations are expressible as some system of Diophantine Equations. For example, despite exponentiation between two variables not being a polynomial, such a relation is definable as a system of polynomials, though the construction is quite involved (Matiiasevich, 1993). This expressiveness implies that solving Diophantine Equations, in general, is as hard as solving the hardest possible problem, since that problem can be expressed as a system of Diophantine Equations.

Remark 3. Matiiasevich's theorem was actually proven over the domain of natural numbers, that is the non-negative integers, including 0. The version over the integers is a simple corollary since an integer, n, can be restricted to be a natural number using the expression $\exists i_1, i_2, i_3, i_4.n = i_1^2 + i_2^2 + i_3^2 + i_4^2$, by Lagrange's four-square theorem. Such expressions can be added to any system of polynomials over the naturals to create a system over the integers defining the same relation.

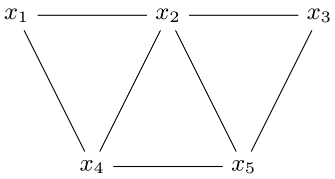
Diophantine Equations are an example of an infinite-domain CSP. They are also an example where instances are generally undecidable. That is, one cannot, in general, even know if a solution exists, even with infinite effort. They are, however, semi-decidable. That is, we can define an algorithm that will find a solution if one exists but may run forever if a solution does not exist. This can be done by counting through tuples of assignments (using some pairing function to define a canonical ordering (Rosenberg, 2003)), and evaluating the system of polynomials at each assignment. This will always tell you if the assignment is satisfying in finite time, meaning that, if a solution exists, it will eventually be found by this procedure.

See (Matiiasevich, 1993) for reference.

1.4. Example: Three Coloring. Graph three coloring is a common example used to introduce CSPs to new audiences. The goal is to color the nodes of a graph with three colors such that no two same-colored nodes touch. The CSP itself consists of a single binary relation asserting the inequality between three elements, typically named after primary colors.

$$\neq = \{(Blue, Red), (Blue, Green), (Red, Blue), (Red, Green), (Green, Blue), (Green, Red)\} \tag{9}$$

Graphs become instances by translating edges into binary relations. Take this graph as an example;



We can simply list the edges using the inequality relation to produce the corresponding CSP instance;

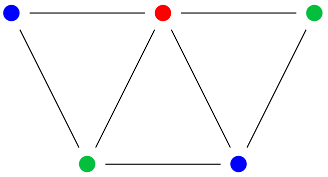
$$x_1 \neq x_2 \wedge x_1 \neq x_4 \wedge x_2 \neq x_3 \wedge x_2 \neq x_4 \wedge x_2 \neq x_5 \wedge x_3 \neq x_5 \wedge x_4 \neq x_5 \tag{10}$$

A solution to the graph coloring problem is then an assignment of colors that satisfies the instance.

If we think about the structure of this graph, we notice that it has three triangles, the left and right side triangles (x_1, x_2, x_4) and (x_2, x_3, x_5), and the central triangle (x_2, x_4, x_5). Each must be a triple of distinctly colored points. The edge between x_4 and x_5 forces those two to have distinct colors, while the two side triangles share a color at x_2. These two observations force us to conclude that the left triangle has the colors of the right triangle flipped along the bisector going through x_2. If we arbitrarily select x_1 to be $Blue$ and x_2 to be Red, we are then forced into the coloring;

$$\langle x_1 \rightarrow Blue; x_2 \rightarrow Red; x_3 \rightarrow Green; x_4 \rightarrow Green; x_5 \rightarrow Blue\rangle, \tag{11}$$

which will satisfy this instance. We can place these colors back on the graph to verify the solution visually.



Like 3-SAT, graph coloring is an NP-complete problem. However, translating problems into graph coloring is often more clunky and non-intuitive than 3-SAT, and solving methods are far less developed. But this problem has similar broad applicability for expressing a wide variety of problems. For example, the original proof that zero-knowledge proofs exist was based on graph coloring (Numberphile, 2009).

Remark 4. The original ZKP setup is usually described in terms of map coloring rather than graph coloring; though, this is merely a different way to draw the same problem. Formally, a map, in this sense, is defined to be a loopless, planar graph.

1.5. Multi-domain CSPs. In some applications, CSPs may have multiple domains. An example of this is mixed-integer linear programming, which has variables ranging over integers, rationals, or booleans, and relations sensitive to each of these domains. A multi-domain CSP will, instead of having a single domain, have a set of domain sets, D. Additionally, for each n-ary relation, R, we will have a tuple $T^R \in D^n$ indicating the domain of each argument.

Instances for multi-domain CSPs can be defined as

- X: a set of variables
- d: a function mapping X to D
- C: a set of constraints (called "clauses") consisting of expressions of the form $R(x_1, x_2, ..., x_n)$, where R is an n-ary, atomic relation from our CSP, and $x_i \in X$ and $d(x_i) = T_i^R$, for all i

Assignments can be extended to this setup in a natural way, as a dependent map from $x \in X$ to $d(x)$.

It's worth noting that multi-domain CSPs can be seen as special cases of single-domain CSPs. The single-domain CSP will have $\bigcup D$ as its domain. We can forget T^R and treat each relation from the multi-domain CSP as a relation over $\bigcup D$. If need be, we can add predicates to the CSP indicating which set of D the argument initially came from, allowing one to restrict the domain of variables; though this is usually unnecessary as domains will already be restricted by the relations those variables appear in. This construction illustrates that we don't gain expressiveness when moving from single to multi-domain CSPs; although keeping domains separated may still be pragmatic for the purposes of implementation.

1.6. CSPs and Graph Homomorphisms. Graph homomorphisms are the notion of structure-preserving map between graphs.

Definition 5. *A graph, for our purposes, is a set of vertices, V, and a set of ordered pairs of Vs, E, representing edges.*

Definition 6. *A graph homomorphism from a graph $G = (V_G, E_G)$ to a graph $H = (V_H, E_H)$ is a function $f : V_G \to V_H$ that satisfies the following condition:*

$$\forall (u, v) \in E_G, \quad (f(u), f(v)) \in E_H. \qquad [12]$$

This means that the function f preserves the edge structure of the graph. In other words, edge-connected vertices in G are always mapped to edge-connected vertices in H.

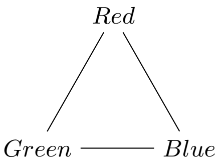
This is enough to model the case of binary CSPs with only a single relation, R. The "model graph" associated with the CSP will have $V = d$, the domain, and $E = R$. As you can see, no interpretation is needed. The model graph of a CSP is literally just the CSP with different labels; no actual interpretation is required.

A CSP instance will also become a graph. In that case, $V = X$, the set of variables, and

$$E = \{(x, y) \mid R(x, y) \text{ is a clause of the instance}\} \qquad [13]$$

At this point, we'd like you to consider what a homomorphism from the instance graph to the model graph would represent. It would assign every variable an element of the domain such that each clause is mapped to an edge validating the relationship between domain values. A homomorphism is the same thing as a satisfying assignment, and we may understand the goal of a CSP as finding a homomorphism to its model graph.

To give a simple example, consider the three coloring case, that matches our setup. In that case, the relation is symmetric, so we can draw the model and instance graphs as undirected; though that won't be possible in general. The model graph would be;



and the instance graph is literally just the problem graph with no modification. As we can see, the CSP problem can be formulated in a completely non-syntactic manner; without variables or clauses and such. It is a purely graph-theoretic problem, though it's not always helpful to think of it as such. Graph coloring, because it's already about graphs, is quite naturally expressed in these terms. Since the CSP and the model graph are the same, we can identify each graph with a CSP over a single binary relation. In particular, the complete graph of size k corresponds to the model graph for k-coloring.

To generalize to binary CSPs with multiple relations, we must switch to an edge-colored graph.

Definition 7. *Fix a set of colors, C. An edge-C-colored graph is a set of vertices, V, and a set of ordered pairs indexed by C, $\{E_i\}_{i \in C}$, representing an edge with an associated color.*

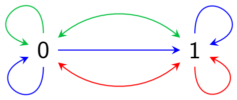
Definition 8. *An edge-C-colored graph homomorphism from a graph $G = (V_G, \{E_{G,i}\}_{i \in C})$ to a graph $H = (V_H, \{E_{H,i}\}_{i \in C})$ is a function $f_V : V_G \to V_H$ that satisfies the following condition:*

$$\forall c \in C, (u, v) \in E_{G,c}, \quad (f_V(u), f_V(v)) \in E_{H,c}. \qquad [14]$$

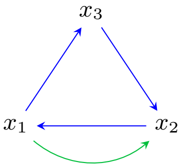
This means that the function f preserves the edge and color structure of the graph. In other words, edge-connected vertices in G are always mapped to edge-connected vertices in H of the same color.

This definition does allow for parallel edges, but such edges must be of distinct colors.

To translate a problem into this framework, we can follow a similar procedure to before, merely noting that each distinct relation gets mapped to a distinct color. 2-SAT provides a good example of a problem fitting in this framework. We can characterize the three relations as three edge colors. The model graph will have two nodes, for 0 and 1, and every true instance for each relation will become an edge. If we assign the color red to $x \vee y$, blue to $\neg x \vee y$, and green to $\neg x \vee \neg y$, we get the following model graph



Note that parallel edges of the same color going in opposite directions have been consolidated into two-headed arrows for conciseness. The 2-SAT problem can be recast as the goal of finding a homomorphism into this graph from an instance graph. Going back to our example from 1.2, we can turn it into the instance graph.



The model graph representation allows us to more efficiently reason about the problem. The cycle of blue arrows can only be mapped onto a self-loop by any homomorphism, meaning that all variables must be mapped to the same point. Further, the green arrow must then be mapped onto a self-loop, the only green one being at 0, allowing us to rapidly conclude the same mapping we found by resolution earlier.

We may also give a graphical interpretation of resolution, at least in this restricted case. The existence of a parallel green and blue arrow forces us to assign 0 to the variable at the source of the blue arrow. This comes from the fact that all parallel green and blue arrows of the model graph have 0 in the source of the blue arrow. Similarly, a parallel red and blue arrow forces us to assign 1 to the variable at the target of the blue arrow. Also, a parallel red and green arrow forces us to conclude that the variables are not equal, though it does not force a specific assignment on its own.

In order to capture all CSPs, we must generalize to edge-colored hypergraphs.

Definition 9. *Fix a set, C, of colors, and an arity-function, a, mapping C to $\mathbb{N}^*$, the non-zero natural numbers. An edge-(C, a)-colored hypergraph is a set of vertices, V, and a set of ordered $a(c)$-tuples indexed by $c \in C$, $\{E_i\}_{i\in C}$, representing a hyper-edge with an associated color.*

Definition 10. *An edge-(C, a)-colored hypergraph homomorphism from a hypergraph $G = (V_G, \{E_{G,i}\}_{i\in C})$ to a hypergraph $H = (V_H, \{E_{H,i}\}_{i\in C})$ is a function $f_V : V_G \to V_H$ that satisfies the following condition:*

$$\forall c \in C, t \in E_{G,c}, \quad \prod_{i<a(c)} f_V(t_i) \in E_{H,c}. \tag{15}$$

Where $\prod_{i<n}$ is used to construct an n-tuple.

The translation should be obvious at this point. The domain becomes the set of vertices, relations become colors, and the arity of the colors is the same as the arity of their relation.

At this point, drawing the model graph explicitly becomes unwieldy. 3-SAT has three colors, each with 7 hyper-edges, and several with self-loops of some kind. Many common ways of drawing hypergraphs don't support self-loops and the ones that do create a quite unreadable representation. I will not try to draw an example, as a consequence, but this completes our presentation of CSPs in terms of graph homomorphism. What I've called edge-colored hypergraphs are usually called "relational structures" in the CSP literature (Feder and Vardi, 1998). That is the terminology I'll use for the rest of this survey.

Remark 11. It was noted earlier that 3-coloring is an NP-complete problem. It was also noted that all finite-domain CSPs are, at most, NP-hard. This means we can translate all CSPs into graph homomorphism problems efficiently, eliminating the need for colors or hyper-edges in the process. However, such representations are far less direct than the colored hyper-graphs mentioned here. There is a trade-off in conceptual simplicity in exchange for sticking to a more familiar setting.

Representing the model graph directly is generally only feasible for small, finite-domain CSPs. CSPs over large domains, such as those over large finite fields, will have model graphs proportional to the size of their domain, which can be extremely large. As such, implicit representations would have to be used.

It should also be noted that our definitions do not restrict ourselves to the finite domain. Infinite graphs, such as the Rado graph, can also be interpreted as CSPs without conceptual difficulty.

2. CSP Variants

While constraint satisfaction is a powerful paradigm for reasoning about problems in broad generality, there are several modifications of it that seek to increase its power. Generally, this involves either attempting to characterize some approximation to CSP solving, or else increasing its representation power.

2.1. Promise CSP. The Promise Constraint Satisfaction Problem is an attempt to characterize an approximation of CSPs by assuming we already have a solution to a CSP and want to use it to solve another CSP faster. Formally, a PCSP consists of a pair of relational structures, A and B, and a homomorphism between them. An instance of a PCSP consists of a relational structure representing a query, I, and asks for a homomorphism from I to B, assuming we know that some homomorphism from I onto A already exists. We don't assume a particular homomorphism, just that some homomorphism exists.

A good example is the "1-in-3-SAT versus NAE-3-SAT" problem. The domain is the booleans. We first have the CSP, T, defined by the 1-in-3 relation.

$$\{(1,0,0),(0,1,0),(0,0,1)\} \qquad [16]$$

Secondly, we have the not-all-equal CSP, H_2, defined as having the sole relation.

$$\{(x,y,z) \mid \neg(x=y=z)\} \qquad [17]$$

Obviously, there's a canonical homomorphism from T onto B mapping the satisfying triples of T into H_2. What's interesting about this problem is that T and B are both known to be NP-complete. Despite this, the promise going from T to B can be solved in polynomial time. To illustrate how this can be done, take the following CSP instance;

$$R(x_1,x_2,x_3) \wedge R(x_2,x_4,x_5) \wedge R(x_1,x_3,x_6) \wedge R(x_1,x_5,x_6) \qquad [18]$$

The promise CSP problem tells us that we know there exists some homomorphism from this into T, which is to say, interpreting R as the 1-in-3 relation here will have some satisfying assignment. Indeed, such an assignment *does* exist in this case; however, we don't actually know what this assignment is, just that it exists. In general, finding such an assignment is very hard; that we know some assignment exists acts as a structural restriction on the problem that we may exploit to find a homomorphism in H_2, which is, itself, usually hard.

How can we use this fact to derive a solution in H_2? First, we can note that T can be relaxed to this problem over the integers;

$$x_1+x_2+x_3=1 \wedge x_2+x_4+x_5=1 \wedge x_1+x_3+x_6=1 \wedge x_1+x_5+x_6=1 \qquad [19]$$

We've just replaced R with the relation $x+y+z=1$. If a solution in T exists, then this system of linear equations has a solution. Furthermore, this is a relaxation since a solution to this system does not necessarily give us a solution to T. We can use an algorithm such as Gaussian Elimination to get a solution to any linear system over the integers efficiently if it exists. For example, we have the following assignment satisfying the linear equations;

$$\langle x_1 \to 5, x_2 \to -47, x_3 \to 43, x_4 \to 5, x_5 \to 43, x_6 \to -47\rangle \qquad [20]$$

With this in hand, we can transform this assignment for the linear equations into a satisfying assignment for the NAE problem by replacing positive values with 1 and negative values with 0. The reason this works is because the presence of any negative integer in a solution must be canceled out by a positive integer. So long as the solution has no 0s, we will necessarily have both positive and negative numbers as part of the solution, which holds in this case. Doing this swap yields

$$\langle x_1 \to 1, x_2 \to 0, x_3 \to 1, x_4 \to 1, x_5 \to 1, x_6 \to 0\rangle \qquad [21]$$

Substituting this into H_2, we'd have

$$\neg(1=0=1) \wedge \neg(0=1=1) \wedge \neg(1=1=0) \wedge \neg(1=1=0) \qquad [22]$$

which is clearly satisfied. We can also clearly read off a solution for T, but this will not usually be the case.

This method is called "making a sandwich", where T and H_2 are the metaphorical bread and linear equations over $\mathbb{Z}$ is the metaphorical filling. This method is currently the most common for creating polytime PCSPs. Interestingly, it's been proven that there are no polytime sandwiches between T and H_2 with a finite domain; one must transfer to an infinite domain.

PCSPs are still a very new and understudied subject. Few general results are known, and many of the most effective techniques for other CSP variants haven't worked in this domain. Still, it is a very active area of research. For a modern overview of the subject, see (Krokhin and Opršal, 2022).

2.2. Quantified CSP. Quantified CSPs extend the CSP paradigm from the propositional to the first-order case. It allows for the usage of quantifiers with unrestricted swapping between universal and existential quantification prior to stating the clauses. The most widely studied is the boolean case. Take the example boolean QCSP;

$$\forall x_1, \exists x_2, \forall x_3, \exists x_4, (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_2 \vee x_3 \vee x_4) \qquad [23]$$

To solve this problem, we are effectively building functions for each existentially quantified variable. The arguments to these functions are the universally quantified variables appearing prior to the existentially quantified one. This means we want functions $f_2 : \mathbb{B} \to \mathbb{B}$ and $f_4 : \mathbb{B}^2 \to \mathbb{B}$, which assign values to x_2 and x_4, making

$$\forall x_1, \forall x_3, (\neg x_1 \vee \neg f_2(x_1) \vee x_3) \wedge (\neg f_2(x_1) \vee x_3 \vee f_4(x_1, x_3)) \quad [24]$$

true. This process of eliminating existential quantifiers via function synthesis is called "Skolemization", and such functions are the analog of a satisfying assignment for QCSPs.

In the finite domain case, these functions can always be described as decision trees that split on each variable; and that's how they're usually presented. In our case, if we define

$$f_2(x) := \neg x \quad [25]$$

$$f_4(x, y) := \neg y \quad [26]$$

then we'd have

$$\forall x_1, \forall x_3, (\neg x_1 \vee \neg\neg x_1 \vee x_3) \wedge (\neg\neg x_1 \vee x_3 \vee \neg x_3) \quad [27]$$

which is true by excluded middle.

QSAT, the quantified version of 3-SAT, is PSpace-Complete, giving it more expressive power and allowing any problem in PSpace to be efficiently translated into it. QCSP more generally is an active area of research that is not as well developed as CSP, though it is generally better understood than PCSP.

There is a further extension of QSAT called "dependency quantified SAT". In this setting, quantifiers are allowed to specify exactly which variables they depend on, instead of assuming they depend on all universally quantified variables appearing beforehand. Despite seeming to be similar in nature to QSAT, DQSAT is NEXPTIME-Complete, giving it significantly more expressive power. Unlike with boolean SAT, dependency-quantified versions of CSPs in general don't seem to appear in the literature.

For a modern overview of QSAT, see chapters 29-31 of (Biere et al., 2020). We cannot find an overview of QCSPs in general which is worth recommending, though the introduction of (Zhuk et al., 2023) gives pointers to significant work across this area in addition to being an important recent paper on the topic.

2.3. MaxCSP. MaxCSP is the problem of, not solving a CSP, but maximizing the number of clauses satisfied within a CSP. Like with other areas, the most well-studied version of this problem is MaxSAT, the boolean version of MaxCSP.

In the most basic version of the problem, we assign each clause, c_i, a boolean, b_i. For each assignment, we can determine if c_i is satisfied by that assignment. If it is, then $b_i = 1$ and is 0 otherwise. The ultimate goal of MaxCSP is to maximize the value

$$\sum_i b_i, \quad [28]$$

which will reach its maximum value if all clauses can be satisfied. We will often want to assign more importance to some clauses over others. To do this, we associate, with each clause, a positive, real weight, w_i. Such problems will often be called "Weighted CSPs", and seek to maximize

$$\sum_i w_i b_i. \quad [29]$$

More often than not there will be more complex weighting schemes. The most common is to allow the weight of a clause to be $-\infty$ in the case that it's false, essentially forcing any solution to satisfy that clause. Such "must have" clauses are common in many applications, and are often called "hard". Appropriately, non-hard clauses are often called "soft" in the same context. Such problems are called "Partial MaxCSPs".

Remark 12. There are further weighting schemes for more sophisticated applications, such as those mapping clauses onto complex numbers or some other metric, though such are outside the scope we consider here. See, for example, (Cai and Chen, 2017).

MaxCSPs generally arise when we have over-constrained problems. We have $100 and want to buy $200 worth of stuff. Each thing we want to buy would become a constraint stating that we bought the thing, and there would be a further "must have" constraint limiting the total spent money to $100. This forces some of the "did buy" constraints to fail, and the actual choice of what to buy must be done by weighing our purchasing options using some utility function.

MaxCSPs also allow us to deal with conflicting requirements. A common one is "best price" and "best performance". One can almost never have both, so some compromise must be made, weighing price constraints and performance constraints in order to judge proposed solutions. This specific example may be modeled by having a gradient of constraints. For price, we might have a set of constraints stating "price was less than x" for a wide swath of xs. The lower the price is, the more of these constraints are satisfied. Continuous optimization is dealt with more elegantly using constraint optimization rather than maximization. This will be discussed in more detail in section 2.4.

As an example, consider the following modification of our previous 2-SAT example;

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \qquad [30]$$

This formula is not satisfiable by any possible assignment. We can choose to ignore some of the clauses in the hopes of finding an approximate solution. We can ignore the last clause to get our previous problem which is satisfiable. We can also ignore the first clause, allowing for the solution;

$$\langle x_1 \to 0; x_2 \to 1; x_3 \to 0 \rangle \qquad [31]$$

which has the same value as our previous solution. As one may notice, the goal is no longer to find a solution, but to find a problem close to our original that we can solve.

MaxCSP is, at this point, quite well understood. For an overview of MaxSAT, see chapters 23-24 of (Biere et al., 2020). We could not find a decent modern survey of MaxCSP in general.

2.4. Constraint Optimization Problems. Constraint optimization is similar to MaxCSP, except the optimization function is defined over the assignment of variables rather than the satisfaction of clauses. Its setup is the same as the ordinary CSP problem, but there is an additional utility function, U, which assigns a score to any given assignment.

In contrast to MaxCSP where problems are generally overconstrained, constraint optimization problems should be underconstrained. That is. there should be many possible solutions, and we are trying to optimize over them.

The most well-known example of a constraint optimization problem is Integer Linear Programming (ILP). The relations of ILP are all linear equalities over the integers; that is, expressions of the form

$$Ax + By + ... + Cz \leq D \qquad [32]$$

along with, for each variable, x, in the problem, the additional constraint

$$x \geq 0 \qquad [33]$$

Incidentally, this constraint implies that the domain is really the natural numbers, but the allowance of negative coefficients suggests using these constraints instead for uniformity, but the semantics doesn't change in either case.

Utility functions are restricted to linear functions of the variables.

Consider the following example (taken from Wikipedia).

$$-x + y \leq 1 \wedge 3x + 2y \leq 12 \wedge 2x + 3y \leq 12 \qquad [34]$$

I have elided the constraints forcing x and y to be natural numbers, but know they are there. The utility function is

$$U(x, y) = y \qquad [35]$$

Our goal, then, is to find the combination of x and y which maximizes y.

By looking at the second and third clauses, we can restrict the ranges of x and y. Obviously, they must both be less than or equal to 12, but we can be more precise. If we set $y = 0$, then x's maximum value is 4, according to that second clause. Similarly, setting $x = 0$ tells us that y's maximum value is 4 by the third clause. This tells us that both integers are between 0 and 4. Since we are trying to maximize y, let's set y to 4 and see if we can make it work. The first clause will then restrict x to be either 3 or 4 while the second will restrict x to be 0 or 1; so y cannot be 4. If we set y to be 3, then the first equation will restrict x to $[2, 4]$ while the third restricts x to $[0, 1]$; thus, y cannot be 3 either. If we now try $y = 2$, the first equation will restrict x to $[1, 4]$, the second will restrict x to $[0, 2]$ and the third will restrict x to $[0, 3]$. The intersection of these is $\{1, 2\}$, telling us that y will be at a maximum at $(x, y) = (1, 2)$ and $(x, y) = (2, 2)$.

Integer programming is a widely studied field with a long history of texts giving good coverage. Two examples are the homonymous (Wolsey, 2020) and (Michelangelo et al., 2014). Also, see section 3.6 for some more insights into solving techniques in ILP. Also note that ILP often incorporates boolean and continuous variables, creating "Mixed" ILP (MILP).

Remark 13. Integer programming is a special case of linear programming, which is itself a widely studied subject, especially in the context of convex optimization. An optimization problem over a system of linear inequalities over a continuous domain is a COP. They are generally much easier to solve but are less expressive in exchange.

Before ending this section, we should note that MaxCSP can be framed as a special case of constraint optimization. Given a clause, c_i, in the original problem, we associate with it a boolean variable, b_i. We then replace each clause in the instance with

$$b_i = c_i \qquad [36]$$

This will force the boolean variable to take on the value of the clause. Since there are no other constraints on b_i, this makes the new problem trivial to satisfy for any assignment of the other variables. We then define the utility function to be

$$U(...) = \sum_i b_i, \tag*{[37]}$$

This generates the same problem as MaxCSP. A similar construction captures weighted CSPs.

Note that we can go the other way in the special case of non-mixed integer programming. So long as all variables are discrete, we can get a rough max value and use this to generate a weighted bit decomposition. Say, for example, we have the utility function $U = 3x + 4y$ where $x, y \leq 4$. We can decompose them into bits and rewrite the utility function as

$$U = 3(x_0 + 2x_1) + 4(y_0 + 2y_1) = 3x_0 + 6x_1 + 4y_0 + 8y_1 \tag*{[38]}$$

These bits can then be used as soft clauses with their coefficients as their weights in a weighted MaxSAT problem. The ILP constraints are then turned into hard MaxSAT constraints, completing the translation. A similar procedure doesn't exist for continuous optimization, making a perfect translation from MILP impossible, though discretization methods, like those described briefly in 2.3, can be used for approximation.

We could not find a general survey of constraint optimization; though some do exist with a specific emphasis on distributed solving. See, for example, (Fioretto et al., 2018). It should be noted that the previously mentioned representation of weighted CSPs as COPs has led some authors to equate the two and use the terms COP and WCSP interchangeably. At the very least, insights from one subject can be used in the other.

3. Preprocessing Techniques

Understanding how to manipulate CSPs is necessary to gain an intuition for the limits of solving. There isn't a universal theory for such things, but broad categories of techniques are useful to know about. This section will go through some of the most significant.

3.1. Simplification. Simplification is, intuitively, the first thing someone might want to do. Simplification will transform a CSP instance into another, smaller instance with the "same" solution.

These techniques are generally going to be highly specialized to the specific domain. Often, they will be limited to specific forms. For example, in boolean SAT, $x \vee x \vee y$ can be simplified into $x \vee y$. The most well-known simplification methods come from equational theories. Some methods are not immediately applicable due to CSPs splitting all their relations into clauses. For example, the simple factoring property, $(x - 1)(x + 1) = x^2 - 1$ does not immediately apply since the left and right sides will not generally represent atomic constraints. We may, instead, have the following available atomic constraints;

$$a = C \text{ for any constant } C \tag*{[39]}$$

$$a = b + c \tag*{[40]}$$

$$a = b \times c \tag*{[41]}$$

We may set up the left-hand-side of the factoring example as the following instance;

$$o = 1 \wedge n = -1 \wedge a_1 = x + o \wedge a_2 = x + n \wedge m = a_1 \times a_2 \tag*{[42]}$$

The factoring property would transform this into

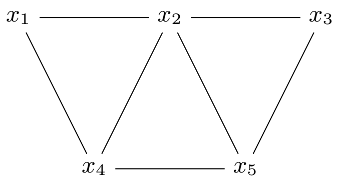
$$n = -1 \wedge s = x \times x \wedge m = s + n \tag*{[43]}$$

This isn't a simplification according to a naive interpretation of the "same solution". Notice that the variables are different, meaning a solution to this new problem would not actually be a solution to the old problem.

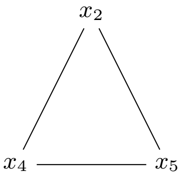
There is a certain sense that this problem is equivalent to the old one. The new problem is satisfiable if and only if the old one is. Further, the values of unshared variables can be uniquely determined by variables that are shared. As such, there is a 1-to-1 correspondence between solutions to the new instance and the old. More than that, the shared variables between the instances will share values within their satisfying assignments. This would allow one to easily calculate solutions to the old instance using solutions to the new instance. This defines a sense in which such transformed instances are "the same".

Some previously mentioned techniques simplify problems. For example, fusing the boolean SAT clauses $x \vee y \vee \neg z$ and $z \vee w \vee v$ into $x \vee y \vee w \vee v$, using resolution as described in section 1.2, so long as z is not mentioned in any other clause.

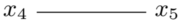
We can further use this idea to reason about our three-coloring example;



Given the naive interpretation of "the same" semantics, it seems like no simplification can be done. If we want to preserve semantics on the nose, then we must preserve nodes and can seemingly only change the edges. But, if we instead preserve semantics in the broader sense just described, we can make an immediate observation. The x_1 and x_3 nodes will have their color uniquely determined by the only two other nodes attached to them. This means we can eliminate them without changing the number of colorings. This gives us;

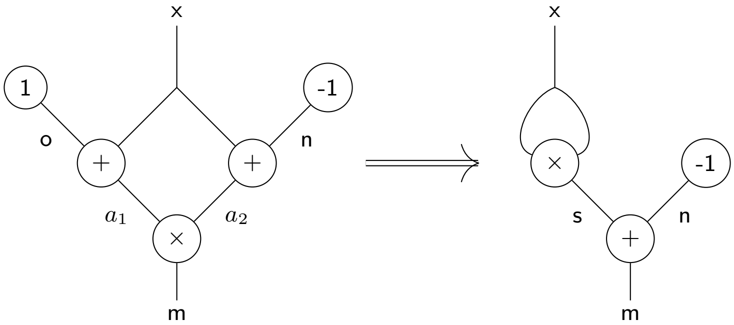


We can further note that x_2 also has its color uniquely determined by the two nodes it's connected to. This, too, allows its elimination, leaving;



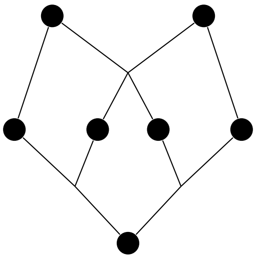
This is now as simple as it can be. We can pick any pair of distinct colors for this pair, and it will determine a unique coloring for our original graph. This implies, in particular, that the number of colorings in our original graph is the same as the number of distinct color pairs; 6.

This idea can be nicely captured by string-diagram representations of CSP instances. In it, each relation becomes a node on a graph and each variable becomes a string connecting nodes. Variables mentioned by three or more relations get combined into "spider" nodes, represented by a lack of a node marker. This string diagram representation is a kind of duel of the other graph representation we've been using; swapping edges and nodes. The utility of this representation can be seen in the rewrite rule for our factoring example;

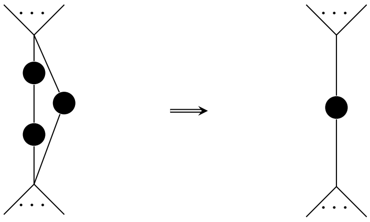


Drawing this rule using the ordinary graph representation would be difficult due to the many nodes an edge can have, but string diagrams handle such cases elegantly.

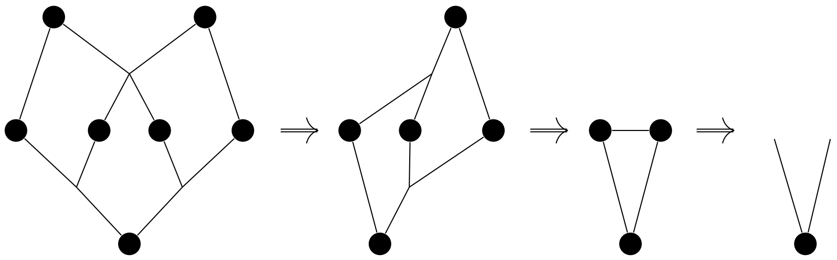
The string diagram for our recurring graph example would look like this;



We can, further, draw the transformation rule eliminating nodes for graph coloring through the following rule;



Repeatedly applying this to our string diagram, we'd get the sequence;



Ending with a diagram equivalent to a single edge with two nodes in our previous representation. If we want to reconstruct a solution to the original CSP instance, we must also keep track of what rewrites we did, as those will correspond to functions calculating implied values for the original instance's variables.

This representation turns the simplification of a CSP into the actual simplification of a concrete data structure where the elimination of a variable becomes the elimination of an edge and the introduction of a variable corresponds to the introduction of an edge. Essentially all typical examples of algebraic simplification methods can be recast as local string-diagram rewrite rules.

Tactics along these lines are studied widely within the subject of string diagrams. See (Bonchi et al., 2017) for a broad account of relational theories presented as string diagrams. This also covers methods beyond simplification, such as strengthening instances to new ones which, if true, would imply a satisfying instance of the whole, but which are not equivalent. Such methods are important for implementing search, but will not be covered here. Some specific CSPs have fully worked out string diagram calculi that are algebraically complete; see (Gu et al., 2023) on boolean SAT, for example.

3.2. Redundancy Elimination. The concepts from the previous section can be generalized to a notion of redundancy elimination. When solving a CSP, we care less about preserving the number of solutions than we do finding some solution. A transformation that preserves satisfiability, but not solution number, is fine for most purposes.

In general, we can ask if, given an instance $F \wedge C$, if removing C preserves satisfiability. If so, then C is redundant in the instance. This is all we need if we only care about the existence of a satisfying assignment. However, we generally actually want to find a solution. The most common method to reconstruct solutions through redundancy elimination are special cases of what's called "propagation redundancy".

We can assess whether some clause is propagation redundant with respect to CSP instances by asking about assignments blocked by that clause. For example, if we have the clause $x = 1$, then the assignment $\langle x \to 0\rangle$ is blocked by that clause. As another example, the clause $\neg x \vee y \vee z$ blocks the assignment $\langle x \to 1, y \to 0, z \to 0\rangle$.

Given a partial assignment, α, and a CSP instance, F, we denote the substitution of α info F by $F_{|\alpha}$, creating the new, smaller CSP instance.

Given a clause, C, and a partial assignment, α, blocked by that clause, we say that C is propagation redundant with respect to the instance F if there exists another partial assignment ω satisfying C, such that every assignment satisfying $F_{|\alpha}$ also satisfies $F_{|\omega}$, for every blocked assignment α. We can frame this as the existence of a function, f, over partial assignments such that, for any blocked assignment, α, any satisfying assignment for $F_{|\alpha}$ also satisfies $F_{|f(\alpha)}$.

The idea of this definition is that we have a CSP instance $F \wedge C$. Hypothetically, there is an assignment which we are looking for. It has a sub-assignment on the variables of C, α, which may be blocked by C. That C is redundant implies that we can replace if needed, the sub-assignment with another, ω, which satisfies C without causing F to be unsatisfied. This means that given an assignment with sub-assignment α which satisfies F but not C, an ω that we can swap with α will always exist so that $F \wedge C$ is satisfied in its entirety.

As a basic example, consider this 3-SAT problem;

$$(x \vee y) \wedge (x \vee z) \wedge (\neg x \vee y \vee z) \wedge x \qquad [44]$$

we may observe that $C = x$ blocks the assignment $\alpha = \langle x \to 0\rangle$. If we remove this clause, we'd have.

$$F = (x \vee y) \wedge (x \vee z) \wedge (\neg x \vee y \vee z) \qquad [45]$$

We may also note that

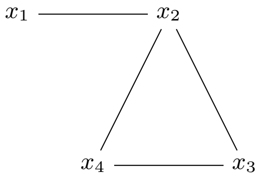
$$F_{|\alpha} = y \wedge z \qquad [46]$$

We may observe that the only other partial assignment is $\omega = \langle x \to 1\rangle$, and

$$F_{|\omega} = y \vee z \qquad [47]$$

It's not hard to see that every satisfying assignment of $F_{|\alpha}$ also satisfies $F_{|\omega}$. If we have a satisfying assignment for F, there are two cases. Either it already also satisfies $F \wedge C$, in which case we're done, or it only satisfies F. As an example of the latter case, take the assignment $\langle x \to 0, y \to 1, z \to 1\rangle$. In the latter case, it must be a satisfying assignment of $F_{|\alpha}$. Since the sub-assignment $\langle y \to 1, z \to 1\rangle$ also satisfies $F_{|\omega}$, we can replace the α sub-assignment with ω, getting $\langle x \to 1, y \to 1, z \to 1\rangle$ which satisfies $F \wedge C$.

As a second example, consider three coloring on the following graph;



This corresponds to the CSP instance;

$$x_1 \neq x_2 \wedge x_2 \neq x_3 \wedge x_2 \neq x_4 \wedge x_3 \neq x_4 \tag{48}$$

It seems intuitively obvious that

$$C = x_1 \neq x_2 \tag{49}$$

ought to be redundant as we should always be able to find an assignment satisfying it given an assignment for the rest;

$$F = x_2 \neq x_3 \wedge x_2 \neq x_4 \wedge x_3 \neq x_4 \tag{50}$$

We may note that there are three assignments blocked by C. Without loss of generality, fix

$$\alpha = \langle x_1 \to B, x_2 \to B\rangle \tag{51}$$

We may note that;

$$F_{|\alpha} = B \neq x_3 \neq x_4 \tag{52}$$

We may identify the following;

$$\omega = \langle x_1 \to R, x_2 \to B\rangle \tag{53}$$

and further, we may observe that $F_{|\alpha} = F_{|\omega}$, meaning that any assignment satisfying one will also satisfy the other. That ω will satisfy $F \wedge C$ and that analogous assignments will exist for all blocked clauses implies that C is propagation redundant.

In theory, we only need to find one ω as an alternative to the blocked α and swap to it if needed. However, finding such can be nontrivial. In general, judging if a clause is propagation redundant is, itself, an NP-complete problem. As such, much work has been done to characterize clauses whose propagation redundancy is efficiently detectable. The most popular of these are the so-called "blocked clauses" and their variants. Alternatively, some recent methods, such as the so-called "satisfaction-driven clause learning" methodology, recursively call an SAT solver to detect redundancy.

There is a further generalization of redundancy. Instead of asking for every assignment of $F_{|\alpha}$ to satisfy $F_{|f(\alpha)}$, we may instead ask for a function, f, over partial assignments such that any assignment ρ of $F_{|\alpha}$ is also an assignment of $F_{|f(\rho,\alpha)}$, and, of course, $f(\rho, \alpha)$ should satisfy the redundant clause C. This allows ω to vary based on what assignment for F we found, and f is exactly the method to reconstruct assignments for larger problems from smaller sub-problems. Using this, we can establish $x_3 \neq x_4$ as redundant in our previous example with

$$f(\rho, \alpha) = \langle x_3 \to \text{if } \rho(x_2) \in \{R, B\} \text{ then } G \text{ else } R, x_4 \to \text{if } \rho(x_2) \in \{R, G\} \text{ then } B \text{ else } R\rangle. \tag{54}$$

Note that $x_3 \neq x_4$ is not propagation redundant since, for example, the blocked assignment $\alpha = \langle x_3 \to B, x_4 \to B\rangle$ will produce

$$F_{|\alpha} = x_1 \neq x_2 \wedge x_2 \neq B, \tag{55}$$

and we won't be able to find an ω such that every assignment of F_α is also an assignment of F_ω. If we try, for example, $\omega = \langle x_3 \to R, x_4 \to B\rangle$, then

$$F_{|\omega} = x_1 \neq x_2 \wedge x_2 \neq R \neq B \tag{56}$$

which is clearly more, not less, specific than $F_{|\alpha}$. Further, if we try $\omega = \langle x_3 \to R, x_4 \to G\rangle$, we'd get

$$F_{|\omega} = x_1 \neq x_2 \wedge x_2 \neq R \neq G \tag{57}$$

which is not satisfied by $\langle x_1 \to R, x_2 \to G\rangle$, despite that being a satisfying assignment of $F_{|\alpha}$.

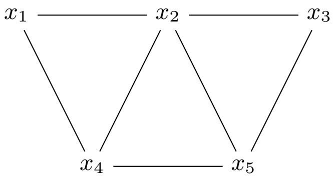
We may also turn redundancy notions into rewrite rules along the lines of those described in 3.1.

For a modern treatment, see (Heule et al., 2017) and (Barnett and Biere, 2021).

3.3. Vertex Splitting. Vertex splitting is a generic method to split up CSP problems when instances are almost disconnected. Consider a CSP problem of the form $F \land G$ where the two sub-instances, F and G, share a single variable, x, in common. Rather than solving the problem as a whole, we can, instead, eliminate the shared variable. We can do this by instantiating the variable with each possible value. In the boolean case, this creates four sub-problems, $F_{|\langle x \to 0\rangle}$, $F_{|\langle x \to 1\rangle}$, $G_{|\langle x \to 0\rangle}$, $G_{|\langle x \to 1\rangle}$. Assuming F and G are roughly equal in size, the size of the problem doubles. However, due to the exponential worst-case complexity of CSP solving, we go from $O(2^m)$ to $O(4 \times 2^{\frac{m}{2}}) = O(2^{\frac{m}{2}+2})$, where m is the number of variables, which will generally be much, much faster.

We can generalize this by identifying vertex separators instead of individual vertices. One can identify a vertex separator of the instance graph which splits the problem into two problems only connected by the fact that they share the variables in the separator. We then fix the values of all the variables in the separator. Assuming there were n variables in the separator, we now have 2^{n+1} sub-problems, each, roughly, half the size of the original problem. This means the new (total) work will be $O(2^{n+1}2^{\frac{m}{2}}) = O(2^{n+\frac{m}{2}+1})$. Compared to the original work of $O(2^m)$, if n is small, this could be a considerable asymptotic efficiency boost. However, random SAT problems can't generally be efficiently decomposed like this since their minimal vertex separator will have roughly the same number of variables as $m/2$, meaning we'd be back to $O(2^m)$ work and wouldn't gain any advantage. Also, finding minimum vertex separators is, itself, an NP-hard problem, but decent approximation algorithms do exist. See (Amir and McIlraith, 2001).

Vertex elimination can also be used to make a problem more tree-like. Take our recurring example graph as an example;



splitting x_2 would not split the problem, but it would make it entirely linear, allowing the sub-problems to be efficiently solved using the methods described in section 4.1.

3.4. Bound Variable Elimination. We can utilize the resolution method mentioned in section 1.2 to eliminate variables in general. Restricted to the boolean SAT setting, if we want to eliminate a variable, x, we resolve all clauses where x appears positively with every clause where it appears negatively. This creates the "clause distribution" of the variable. We then delete all the clauses mentioning x and replace them with the clause distribution. As an example, consider the following;

$$(x \lor y) \land (\neg y \lor z) \land (w \lor y) \land (\neg y \lor w) \tag*{[58]}$$

Let's eliminate y. Since y appears in all these clauses, we'll be replacing the whole instance. We can create the following table showing the clause distribution:

	$\neg y \lor z$	$\neg y \lor w$
$x \lor y$	$x \lor z$	$x \lor w$
$w \lor y$	$w \lor z$	w

After eliminating y, the new instance becomes:

$$(x \lor z) \land (x \lor w) \land (w \lor z) \land (w) \tag*{[59]}$$

Finding an assignment, at this point, becomes a trivial exercise. Given an assignment for the new instance, we can derive an assignment for y by applying that assignment to the positive clauses for y, along with assigning y to be false. If any of them are false, then y must be true, otherwise, it may be false in a full assignment. This choice is somewhat arbitrary; we could have applied the assignment to the negative clauses for y, along with assigning y to be true, getting a similar analysis.

Two special cases are important to note. Firstly, if a variable only appears as positive or negative, then the clause distribution will be empty. This reflects the fact that we may eliminate such a variable by assigning it a value that satisfies all the clauses it appears in. Secondly, if the variable appears in a singleton/unit clause, then the distribution will be logically equivalent to either the positive or negative clauses with the variable removed. This reflects the fact that unit clauses force a value for their variable.

If we take a step back, we can understand what resolution is doing. We may restate the resolution as stating;

$$\frac{p = 0 \to C_1 \quad p = 1 \to C_2}{C_1 \lor C_2} \tag*{[60]}$$

In the boolean case, we have full coverage with the two cases. In the case of three-valued CSPs, we may have more complex resolution variants;

$$\frac{p=0\to C_1 \quad p=1\to C_2 \quad p=2\to C_2}{C_1\vee C_2\vee C_3} \quad \frac{(p=0\vee p=1)\to C_1 \quad p=2\to C_2}{C_1\vee C_2}$$
$$\frac{(p=0\vee p=2)\to C_1 \quad p=1\to C_2}{C_1\vee C_2} \quad \frac{(p=1\vee p=2)\to C_1 \quad p=0\to C_2}{C_1\vee C_2} \qquad [61]$$

We may even extend this into the infinite domain. If our domain is the integers, then the following is a valid resolution variant;

$$\frac{p<0\to C_1 \quad p\geq 0\to C_2}{C_1\vee C_2} \qquad [62]$$

Though it becomes more complicated as the domain grows, there's a natural extension of clause distribution. Applicability will depend on the extent we may interpret a clause as an implication of the appropriate form. If we take three-coloring as an example, we only have one kind of clause, of the form $x\neq y$. We can rephrase this as;

$$\begin{aligned}(x=R)&\to(y=G\vee y=B)\wedge\\(x=G)&\to(y=R\vee y=B)\wedge\\(x=B)&\to(y=R\vee y=G)\end{aligned} \qquad [63]$$

This will expand the formula, but put every clause in a form that's resolution compatible. Similarly, there is a form that places y in the antecedent. Using this form, we may eliminate a variable, although we will be switching to a more flexible CSP since our clause distribution will not exclusively contain clauses of the form $x\neq y$.

3.5. Cost Preservation. We've talked mostly about satisfaction, but, for the purposes of optimization, additional steps must be taken to ensure costs are preserved by transformations. Consider the following problem;

$$(x\vee y)\wedge(\neg y\vee z)\wedge(w\vee y)\wedge(\neg y\vee w)\wedge(\neg x\vee\neg w)\wedge\neg w \qquad [64]$$

Assume our goal is to maximize the number of satisfied clauses. A maximum can be reached by assigning $\langle x\to 0, y\to 1, z\to 1, w\to 0\rangle$, which satisfies 5 clauses, but this is not the only assignment that does this.

It's entirely possible to eliminate all the variables using bound variable elimination. This will merely tell us that the statement is unsatisfiable.

If we used unit propagation to eliminate w, we'd get

$$(x\vee y)\wedge(\neg y\vee z)\wedge y\wedge\neg y \qquad [65]$$

This is also not satisfiable. The assignment $\langle x\to 1, y\to 0, z\to 1\rangle$ satisfies 3 clauses, which is the max, though this is not the only way to do this. Unfortunately, this doesn't tell us anything about the maximum satisfiability of the original problem. If we reconstruct an assignment for the full problem from this by assigning $\langle w\to 0\rangle$ (since the original unit clause negated w), we'd only satisfy 4 of the original clauses, which is suboptimal.

How do we fix this?

The short answer is that we must introduce new variables whose value is directly tied to the value we seek to maximize, and we must ensure that these are not targeted for elimination. We start by adding a unique labeling variable to each clause;

$$(x\vee y\vee l_1)\wedge(\neg y\vee z\vee l_2)\wedge(w\vee y\vee l_3)\wedge(\neg y\vee w\vee l_4)\wedge(\neg x\vee\neg w\vee l_5)\wedge(\neg w\vee l_6) \qquad [66]$$

We can now reframe our utility function as the sum of the negation of these labels. This new instance can be trivially satisfied by assigning them all true; we want to minimize the number we have to set to true.

We can start eliminating variables, ensuring that our labels remain untouched. First, eliminate w, getting;

$$(x\vee y\vee l_1)\wedge(\neg y\vee z\vee l_2)\wedge(\neg x\vee y\vee l_3\vee l_5)\wedge(y\vee l_3\vee l_6)\wedge(\neg x\vee\neg y\vee l_4\vee l_5)\wedge(\neg y\vee l_4\vee l_6) \qquad [67]$$

We can then eliminate x, getting

$$(\neg y\vee z\vee l_2)\wedge(y\vee l_1\vee y\vee l_3\vee l_5)\wedge(y\vee l_3\vee l_6)\wedge(y\vee l_1\vee\neg y\vee l_4\vee l_5)\wedge(\neg y\vee l_4\vee l_6) \qquad [68]$$

We can remove the fourth of these clauses, since it's trivial, and simplify the second to get;

$$(\neg y\vee z\vee l_2)\wedge(y\vee l_1\vee l_3\vee l_5)\wedge(y\vee l_3\vee l_6)\wedge(\neg y\vee l_4\vee l_6) \qquad [69]$$

We can eliminate z, getting;

$$(y\vee l_1\vee l_3\vee l_5)\wedge(y\vee l_3\vee l_6)\wedge(\neg y\vee l_4\vee l_6) \qquad [70]$$

Incidentally, this eliminates the label l_2, which we can assign false since its covered by z. This tells us that allowing the second clause to be false cannot possibly help us find a maximizing assignment. Lastly, we can eliminate y, getting

$$(l_4 \vee l_6 \vee l_1 \vee l_3 \vee l_5) \wedge (l_4 \vee l_6 \vee l_3 \vee l_6) \quad [71]$$

and this can be cleaned up into;

$$(l_1 \vee l_3 \vee l_4 \vee l_5 \vee l_6) \wedge (l_3 \vee l_4 \vee l_6) \quad [72]$$

We may also notice that the second clause is more specific than the first, meaning we can drop the first clause without changing semantics (this is called "clause subsumption"), also eliminating l_1 and l_5 in the process. This gets;

$$l_3 \vee l_4 \vee l_6 \quad [73]$$

This gives us a full characterization of what we need to maximize satisfiability. We must have one of the clauses labeled by one of these to be false. If we, for example, choose l_4 to be the failing clause, then we are forced to assign $\langle y \to 1, w \to 0 \rangle$, since that's the only assignment blocked by our fourth clause, $\neg y \vee w$. Removing this clause and substituting the blocked assignment, we'd get a formula that must be satisfiable. In this case, we'd get;

$$z \quad [74]$$

Which forces z to be true, and allows x to be anything. Any of these assignments will maximize satisfiability, and we can characterize all the other maximizing assignments by looking at the l_3 and l_6 branches.

See (Belov et al., 2013) and follow-up work for more information on this method.

3.6. Decomposition in Integer Linear Programming. The most popular method for solving integer programming problems involves decomposing the problem into a smaller problem. There are many ways to do this; this section will focus on the "branch and bound" method since it's the most popular.

Consider the following problem;

Maximize $5x_1 + 8x_2$ subject to the constraints $2x_1 + 3x_2 \leq 5$ and $4x_1 + 9x_2 \leq 12$. Our ultimate goal is to split up the problem into two easier sub-problems while removing regions guaranteed to not contain the optimum.

We may start by solving the relaxed problem. This means we solve the problem assuming x_1 and x_2 are rationals instead of integers. This will not generally give us an integer solution, but we can use it as an approximation that can be leveraged later on, and it can be found much more efficiently using non-integer linear programming methods. In this case, a maximum of $\frac{77}{6}$ is reached when $x_1 = \frac{3}{2}$ and $x_2 = \frac{2}{3}$.

From here, we create two new sub-problems by cutting on integer approximations of this solution. In this case, we will round x_1 up to 2 in one problem, and down to 1 in the other. This creates two sub-problems;

$$x_1 \leq 1 \wedge 2x_1 + 3x_2 \leq 5 \wedge 4x_1 + 9x_2 \leq 12 \quad [75]$$

and

$$x_1 \geq 2 \wedge 2x_1 + 3x_2 \leq 5 \wedge 4x_1 + 9x_2 \leq 12 \quad [76]$$

One of these smaller regions is guaranteed to have the true, integer maximum. If we solve the relaxed version of these problems, we get the maximum at $x_1 = 1$ and $x_2 = \frac{8}{9}$ for the first problem, and at $x_1 = 2$ and $x_2 = \frac{1}{3}$ for the second. x_2 isn't an integer for either of these, so we can cut on it, creating four sub-problems. In this case, the cut on both sub-problems will approximate x_2 down to 0 and up to 1, but, in general, cutting the same variable on different sub-problems will produce different approximations. In this case, our sub-problems are;

$$x_1 \leq 1 \wedge x_2 = 0 \wedge 2x_1 + 3x_2 \leq 5 \wedge 4x_1 + 9x_2 \leq 12 \quad [77]$$

$$x_1 \leq 1 \wedge x_2 \geq 1 \wedge 2x_1 + 3x_2 \leq 5 \wedge 4x_1 + 9x_2 \leq 12 \quad [78]$$

$$x_1 \geq 2 \wedge x_2 = 0 \wedge 2x_1 + 3x_2 \leq 5 \wedge 4x_1 + 9x_2 \leq 12 \quad [79]$$

$$x_1 \geq 2 \wedge x_2 \geq 1 \wedge 2x_1 + 3x_2 \leq 5 \wedge 4x_1 + 9x_2 \leq 12 \quad [80]$$

As it turns out, the last of these problems is unsatisfiable, so we throw it away. The first of these problems reaches its relaxed max at $x_1 = 1$ and $x_2 = 0$, with a max value of 5. Since the relaxed max is an integer solution, this is also the max when restricted to integers. This isn't necessarily the global max, but we'll mark it down as the best possible solution found so far.

The other two problems reach a max at $x_1 = \frac{3}{4}$ and $x_2 = 0$ and $x_1 = \frac{5}{2}$ and $x_2 = 1$, respectively. These have a max value of $\frac{47}{4}$ and $\frac{25}{2}$, respectively. If either of these were less than 5, we could eliminate them since they could not possibly have better integer solutions than what we've found so far, however, both are larger so we must continue. Integer approximating values of x_1 produces the four sub-problems;

$$x_1 = 0 \land x_2 \geq 1 \land 2x_1 + 3x_2 \leq 5 \land 4x_1 + 9x_2 \leq 12 \tag*{[81]}$$

$$x_1 = 1 \land x_2 \geq 1 \land 2x_1 + 3x_2 \leq 5 \land 4x_1 + 9x_2 \leq 12 \tag*{[82]}$$

$$x_1 \geq 3 \land x_2 = 0 \land 2x_1 + 3x_2 \leq 5 \land 4x_1 + 9x_2 \leq 12 \tag*{[83]}$$

$$x_1 = 2 \land x_2 = 0 \land 2x_1 + 3x_2 \leq 5 \land 4x_1 + 9x_2 \leq 12 \tag*{[84]}$$

The middle two problems are unsatisfiable, so we throw them away. The last problem has a relaxed integer solution at $x_1 = 2$ and $x_2 = 0$, with a value of 10; so that's our new best found so far. That first problem reaches a relaxed max at $x_1 = 0$ and $x_2 = \frac{4}{3}$. Its max value is $\frac{32}{3}$, which is more than 10, so we continue. We cut on that first problem to get two sub-problems;

$$x_1 = 0 \land x_2 = 1 \land 2x_1 + 3x_2 \leq 5 \land 4x_1 + 9x_2 \leq 12 \tag*{[85]}$$

$$x_1 = 0 \land x_2 \geq 2 \land 2x_1 + 3x_2 \leq 5 \land 4x_1 + 9x_2 \leq 12 \tag*{[86]}$$

The second problem is unsatisfiable, while the first has a relaxed integer solution at $x_1 = 0$ and $x_2 = 1$, with a max value of 8. This is the final sub-problem, so we know that our previous integer solution with a value of 10 is, indeed, the maximum value possible.

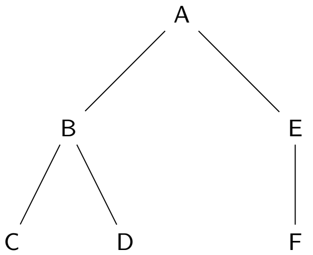
This is just one decomposition method. There are myriad more ways to perform cuts and to decompose integer linear programming problems. See (Galati, 2010) for a modern overview.

3.7. Other Methods. There are numerous other preprocessing methods, far too many to cover here. A notable missing method is that of implication graphs, used for equality detection and clause learning. For a thorough treatment of techniques in the context of boolean SAT, see chapter 9 of (Biere et al., 2020).

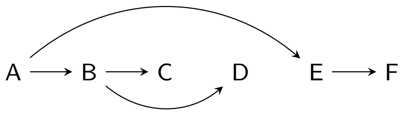
4. Tractable Instances

CSP instances are, in general, hard to solve. To compensate for this, we may ask for characterizations of queries that are guaranteed to be efficiently solvable.

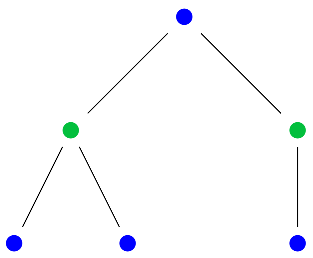
4.1. Acyclic Queries. Most work on tractable instances focuses on the concept of acyclicity. If an instance is shaped like a tree, then we may efficiently determine a solution in a single pass. Consider the three-coloring problem over this tree;



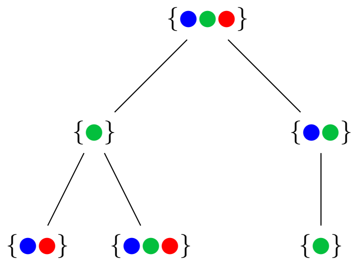
We can start by linearly ordering the tree based on depth-first order. This specific order doesn't matter, as long as the parents come before the children.



We can simply go in this order, making a choice of color, only needing to ensure it's consistent with the nodes that came before. Let's say we always try choosing blue, then green, then red. We assign A to blue. We move onto B, which cannot be blue, so we assign it green. We move onto C, which can be blue, so that's what we assign it. Same with D. We move onto E, which cannot be blue, so we assign it green. We may assign F to blue, so we do. We now have a complete, valid three-coloring.

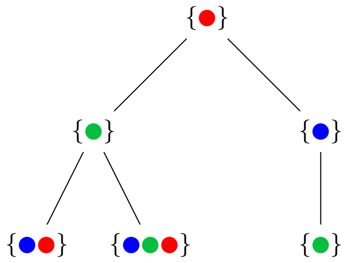


Of course, this doesn't generalize to more complex CSPs. Often, an initial choice may prevent any valid later assignments, forcing us to backtrack, which could ultimately take exponential time to find a solution. For the sake of illustration, let's assume the variable domains are restricted to the following sets.

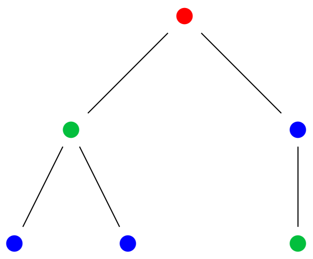


If we started by assigning A to blue, we'd eventually get a contradiction, as E would have to be assigned to green, leaving no consistent assignment for F. We modify our procedure to go in reverse first, filtering out values for the parent that are incompatible with all possibilities. This is called "arc consistency".

We start with F which must be green, so E cannot contain green. E would then be restricted to {🔵}, meaning this must be filtered from the options of A. The next is D, which implies no restrictions on B. We then go to C which also implies no restrictions on B since both options are compatible with all possibilities in B. We then move onto B, which must be green, and so that possibility is filtered from A. After this backward sweep, we have



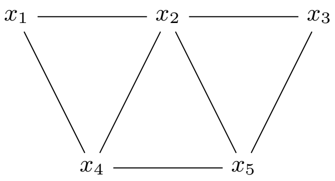
And applying the forward sweep, we'd have the following three-coloring.



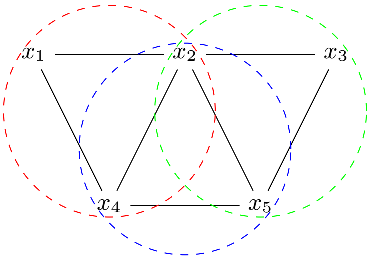
This does generalize to other CSPs; so long as our instances are tree-shaped, we can evaluate them efficiently. Cyclicity itself does not prevent us from solving a problem efficiently, though there is a sense in which the more tree-like a problem can be framed, the more efficiently it can be solved. See section 4.2 for details.

4.2. CSP Decomposition. Trees are not the only efficiently solvable instances. There is a large body of research focusing on characterizing efficient instances. Work on giving a full characterization will be described in section 4.3. Immediately, we end up with two separate issues; identifying if a query can be cast as an efficiently solvable problem, and actually solving it. In general, it may be as hard to frame the problem properly as it would be to solve using more brute-force methods. The field of CSP decomposition is dedicated to studying efficient methods for both transforming and solving queries. Each generally defines some notion of cyclicity (the exact definition will vary based on the method), and will guarantee a solution time that is polynomial in the size of the problem while being exponential in the cyclicity measure. See (Gottlob et al., 2000) for details and a comparison of different methods.

To demonstrate a simple method, consider this graph for the three coloring problem from section 1.4;



This is clearly not a tree. But, consider the following clusterings of variables;



Now, we'll use these clusters to define three new variables; $x_{124} = (x_1, x_2, x_4)$, $x_{245} = (x_2, x_4, x_5)$, and $x_{235} = (x_2, x_3, x_5)$, whose domain is the permutations of three colors,

$$ColorPerms = \{\text{🔵🟢🔴}, \text{🟢🔵🔴}, \text{🔴🟢🔵}, \text{🔵🔴🟢}, \text{🔴🔵🟢}, \text{🟢🔴🔵}\} \qquad [87]$$

We also have constraints relating the values of shared variables between different clusters. For our problem, the important constraints are

$$E^{23}_{12} = \{((x, y, z), (y, z, w)) \mid (x, y, z), (y, z, w) \in ColorPerms\} \qquad [88]$$

$$E^{13}_{13} = \{((x, y, z), (x, w, z)) \mid (x, y, z), (x, w, z) \in ColorPerms\} \qquad [89]$$

Using these, we can reformulate our CSP into

$$E^{23}_{12}(x_{124}, x_{245}) \wedge E^{13}_{13}(x_{245}, x_{235}) \qquad [90]$$

This is clearly tree-like. In fact, it doesn't even branch; it's a line. As such, we can now solve it efficiently. Applying the forward pass from the previous section, we can start by assigning;

$$x_{124} = \text{🔵🟢🔴} \qquad [91]$$

We can then use E^{23}_{12} to see that the remaining options for x_{245} are

$$\{\text{🟢🔴🔵}\} \qquad [92]$$

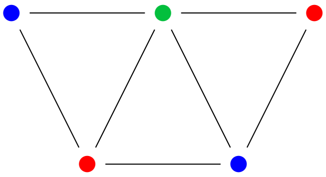
forcing a single option. Using this along with E^{13}_{13} we can restrict x_{235} to the same set. This means we have;

$$x_{124} = (x_1, x_2, x_4) = \text{🔵🟢🔴} \qquad [93]$$

$$x_{245} = (x_2, x_4, x_5) = \text{🟢🔴🔵} \qquad [94]$$

$$x_{235} = (x_2, x_3, x_5) = \text{🟢🔴🔵} \qquad [95]$$

We can then use this solution to fill in the graph colors, obtaining;



It's worth thinking about the complexity of this approach. Assuming we can cluster variables so that they form a tree, the time complexity will be linear with respect to the size of the clusters. However, the domains of the cluster variables will be, in general, exponentially larger than the domains we started with. This means the complexity scales exponentially with respect to cluster size. So long as the clusters can be kept manageable, the instance can be solved in a reasonable amount of time.

The most general known method for cluster creation in such a way that the resulting query is efficiently solvable is called "fractional hypertree decomposition"; see (Grohe and Marx, 2014) for technical details, and see (Marx, 2013) for a broader overview.

4.3. Complete Tractable Query Languages. Rather than starting with a difficult in general CSP which we then filter to tractable instances, it may be prudent to try designing a query language capable of expressing exactly and only the tractable instances for any CSP. This problem is "Gurevich's Problem" (Gurevich, 1985), and is, in its most general form, still open.

The most famous positive result is the Immerman-Vardi theorem (Immerman, 1982). This states that a query language generated by relational calculus plus a least fixed point operator defines a language for queries that are exactly the queries computable in polynomial time. A variant of this stating that "the restriction of second-order logic to formulae whose first order part is a universal Horn formula" captures all P-time queries over ordered structures was proven in (Grädel, 1991).

Remark 14. Note that least-fixed-point logic is essentially just bare Datalog.

The limitation of this is the requirement for total ordering. What this means is that some total ordering must be definable from the relations given in the CSP. An example of a poly-time query not expressible without a total ordering is the parity of an unordered set (determining if the number of elements in a set is even or odd). An unordered set corresponds to a trivial CSP where one has a domain but no actual relations. Counting the size of the domain, then the parity of that size, is clearly efficiently doable. If one had a total ordering, they could associate numbers with each element of the domain and the query would just calculate the highest element, according to this relation. Without such a relation, the query isn't expressible.

An intrinsic ordering isn't present on arbitrary graphs. If the graph is canonicalized, then this canonicalization implies an intrinsic ordering. The existence of a polynomial time graph canonization algorithm is still open; it is known to be doable in quasi-polynomial time (Wiebking, 2021).

The goal is to create a query language whose expressive power is independent of how the model graph is laid out in memory. The only major candidates are a logic called "Choiceless Polynomial Time", and extensions of it. Specifically, the variant with the ability to count is the minimal, still possible candidate (Blass et al., 1999).

CPT with counting is essentially the weakest logic such that all queries are in PTime and there are no known PTime queries that are not expressible in it.

The basic idea of the logic is to extend the language of relations with the following abilities, intended to manipulate sets of solutions to define new relations;

- Take the unions of sets.
- Take unordered pairs of sets.
- Determine if a solution is unique (e.g. map $\{b\}$ onto b and everything else onto $\{\}$).
- Count the number of elements in a set.
- Taking the iterated compositions of definable relations.

The semantics of the language is further mediated by providing an explicit polynomial during evaluation for the sake of limiting compute time.

There is much ongoing research trying to separate this logic from P. There are also attempts to minimally extend CPT so that its canonization invariance is more explicit. The most promising is Choiceless Polynomial Time with Witnessed Symmetric Choice (Lichter and Schweitzer, 2022). Also see that paper for a more detailed, streamlined presentation of regular CPT.

5. Tractable CSPs

There is a wide swath of research attempting to characterize CSPs that are intrinsically tractable. The goal is to classify all CSPs within a large class. All finite-domain CSPs have been classified, and that's where we will spend most of the attention in this section.

5.1. Polymorphisms and Tractability. In 2017, the full dichotomy theorem for finite-domain CPS was proven in (Bulatov, 2017) and (Zhuk, 2020). This section will not try to describe the proof but will describe the basic result. Review section 1.1 for a description of CSP as relational clones.

Each CSP language has a corresponding algebra of "polymorphisms". A polymorphism of a CSP is a function that preserves all relations. That is, given a relation, $R(X, Y, ..., Z)$, in the CSP, an n-ary function, f, conserves R if $R(a_1, b_1, ..., c_1) \wedge R(a_2, b_2, ..., c_2) \wedge ... \wedge R(a_n, b_n, ..., c_n)$ implies $R(f(a_1, a_2, ..., a_n), f(b_1, b_2, ..., b_n), ..., f(c_1, c_2, ..., c_n))$. If f satisfies the analog of this for all relations within the CSP, it's a polymorphism.

Remark 15. This has a connection to programming language theory through Reynold's theory of relational parametricity. There, polymorphism (in the programming sense), is defined through a function preserving all relations. This is the same concept, but in a higher-order, infinite domain, while we're dealing with a first-order, finite domain. At first-order, the only parametrically polymorphic functions are the projection functions (including identity as a special case). As such, projections are always polymorphisms for any CSP. Additionally, with some thought, we can conclude that, if f and g preserve a relation, then so does their composition. Beyond this, we don't think anyone's studied this connection with any seriousness.

This gets us an algebra of functions that includes all the projection functions and is closed under composition. This is called a "functional clone".

Remark 16. Incidentally, functional clones are usually just called "clones", and such function algebras are the domain of clone theory (see, e.g., (Lau, 2006)). Relational clones don't seem widely studied.

Modern approaches to CSP tractability focus on the structure of this polymorphism algebra. The low-resolution version of the dichotomy theorem states that

> Any finite-domain CSP is either NP-complete or all queries are P-time solvable

A higher-resolution version of this theorem would state

> Any finite-domain CSP whose polymorphism algebra is essentially a clone of projections (in other words, whose polymorphism algebra is trivial) is NP-complete, otherwise, all queries are P-time solvable.

Remark 17. Note that I'm hiding some technical details by using the word "essentially"; something can be "essentially just a bunch of projections" without literally being that. A polymorphism is "essentially a projection" if all height-1 identities (i.e. equations with exactly one occurrence of the function on each side) are also satisfied by projections.

This has some significant consequences in that it shows there are no finite-domain CSPs that are between P and NP-hard; there are no quasi-polynomial time CSPs, etc. Also, tractability can be framed by studying the algebra of polymorphisms. All previous dichotomy results in the literature, e.g. those presented in (Grohe, 2006), are special cases of this.

For a survey of the general approach, see (Brady, 2022) and (Barto et al., 2017). These, unfortunately, do not go into the proof of the dichotomy theorem. The most cogent presentation is (Barto et al., 2021). The original proofs used different, hard-to-grok methods; the goal of that paper is to unify them under a single framework.

5.2. Example: 2-SAT. I will give an example of a powerful polymorphism within 2-SAT, and explain how it relates to efficient problem solving.

We will start by identifying a non-trivial polymorphism. The smallest, non-trivial polymorphism is the following function;

$$f(x, y, z) := (x \wedge y) \vee (x \wedge z) \vee (y \wedge z) \quad [96]$$

This function will return true so long as at least two of its inputs are true, and false otherwise. This is often called the "majority function".

The three relations in 2-SAT are

$$R_1(x, y) := x \vee y \quad [97]$$

$$R_2(x, y) := \neg x \vee y \quad [98]$$

$$R_3(x, y) := \neg x \vee \neg y \quad [99]$$

That f is a polymorphism boils down to three observations.

1 The polymorphism property of R_1 merely states that, if we know three disjunctions are true, then either at least two of the left disjuncts must be true or at least two of the right disjuncts must be true.

2 The polymorphism property of R_2 merely states that, if we know three implications are true, then if at least two of the antecedents are true then at least two of the consequents must also be true.

3 The polymorphism property of R_3 merely states that, if we know three disjunctions between negated literals are true, then either at most one of the left disjuncts can be true or at most one of the right disjuncts can be true.

All of these can be concluded intuitively with some thought, and so we may conclude that f is a polymorphism for 2-SAT.

The polymorphism doesn't automatically tell us how to derive a solution; rather, it verifies that certain methods will always work by restricting the possible structure of the CSP. In the case of majority functions, if such a polymorphism exists, this guarantees that a greedy search on values (i.e. assigning arbitrary values to variables, and simply ensuring no future contradictions) will end efficiently so long as the width of the instance graph is small enough. Further, the existence of a majority polymorphism also guarantees that any instance graph can be transformed into another instance graph with the required treewidth, with the same solutions in polynomial time (Feder and Vardi, 1998).

5.3. Tractability for CSP Variants. There have been many directions of work trying to apply the so-called "Algebraic Approach" to CSPs to other domains.

The most active is promise CSPs. Most polymorphism techniques have not been so successful in this domain. Many definitions that universal algebra relies on do not make sense here. See (Krokhin and Opršal, 2022).

Quantified CSPs have had much more success in adopting this method. There was a trichotomy conjecture asserting that all finite domain QCSP problems are either PSpace-hard, NP-hard, or P-time tractable, but this has since been disproven. See (Zhuk and Martin, 2022). Full classification of QCSPs is still an open problem. Some large classes are classified, see for example (Zhuk, 2021) and (Börner et al., 2009).

There is also some work on infinite domain CSPs. This is a bit vague, so additional restrictions are added. Common ones are being numerical domain relations, being omega-categorical, or being equipped with an oligomorphic permutation group. See (Bodirsky, 2012).

Another relevant area is parameterized complexity of CSPs. This seems understudied, but the idea is to loosen the notion of polymorphism to that of a partial polymorphism that preserves some but not all relations of the CSP. The details of this can allow one to classify queries by difficulty, allowing a worst-case untractable CSP to have a large collection of tractable queries. See (Couceiro et al., 2019).

Robust satisfiability is an attempt to characterize problems where, if a problem has an almost satisfying assignment, then such an assignment can always be found. Note that this doesn't necessarily cover MaxCSP in its entirety; a MaxCSP problem could still be tractable in practice even if it isn't robustly satisfiable. This would mean that some instances are easily approximable, but, in general, we couldn't do better than guessing. This doesn't mean that we can't efficiently maximize satisfied clauses either on average or efficiently with respect to some parameter other than problem size. The most important paper is (Barto and Kozik, 2016). This proves that any bounded width CSP is robustly satisfiable. Bounded width, in this case, means that the CSP cannot encode linear equations over abelian groups. Further, any efficiently robustly satisfiable CSP must have bounded width. This means that being of bounded width is the only condition that must be met to efficiently and robustly maximize assignments, and one cannot do better.

6. Intents as CSPs

In this section, we'd like to start giving concrete formalizations of intents using CSPs. We will not aim to thoroughly cover all intent examples but will stick to some basic, non-trivial examples that have been formalized as part of Anoma's "Kudos" sub-project.

6.1. Intents as ILP programs. An intent, for the purposes of our formalization, will be conceptualized as a stated desire to obtain some collection of resources given a stated ability to trade some collection of other resources. Issues like verifiable ownership will not be dealt with here; rather, we concern ourselves only with the collection of stated desires that may be used to formulate trades.

Let's start by considering a specific kind of intent. It consists of partial transactions containing the following data;

- Wh_i: A float weighing intent i.
- $H_{i,r}$: A natural number indicating how much of resource r intent i is willing to trade.
- $W_{i,r}$: A natural number indicating how much of resource r intent i needs to initiate a trade.
- $V_{i,r}$: A float indicating the value intent i gives to resource r.

We are interpreting a partial transaction as stating that we intend to trade, at most, some collection of resources, specified by the Hs, for exactly some other collection of resources, specified by Ws.

The goal can be stated as best satisfying the intent. We may formalize this notion of intents within mixed integer programming. Additionally, for each resource in each intent, there will be an integer indicating how much of each resource was traded away. For each intent, i, and each resource, r, there will be the following data;

- SAT$_i$: boolean indicating if intent i is satisfied.

- $\mathsf{GOT}_{i,r}$: a natural number indicating how many rs were received by intent i.
- $\mathsf{GAVE}_{i,r}$: a natural number indicating how many rs were spent by intent i.

We then turn these into a full problem by asserting appropriate constraints. Firstly, if an intent is satisfied, then the amount given and gotten must reflect this;

$$\mathsf{GOT}_{i,r} \leq \mathsf{SAT}_i \times W_{i,r} \tag{100}$$

$$\mathsf{GAVE}_{i,r} = \mathsf{SAT}_i \times H_{i,r} \tag{101}$$

Further, the total resources transferred must be balanced;

$$\sum_i \mathsf{GOT}_{i,r} = \sum_i \mathsf{GAVE}_{i,r} \tag{102}$$

Lastly, we must cast this as a constraint optimization problem by asserting a utility function. We can simply sum the differences in traded resources, weighted by the intent;

$$U := \sum_i Wh_i \left(\left(\sum_r V_{i,r} \mathsf{GOT}_{i,r} \right) - \left(\sum_r V_{i,r} \mathsf{GAVE}_{i,r} \right) \right) \tag{103}$$

This is, of course, just one way of interpreting "best satisfying". In essence, this attempts to get the best aggregate deal, as subjectively interpreted by each intent, weighted by some objective weight indicating the importance of the intent.

This is, of course, just one kind of intent. To demonstrate an alternative, we may consider a different kind of exclusive-weighted partial transaction. These contain the same data as before, but we can now only trade one collection of resources for another. We are reinterpreting the intents as stating we intend to trade one of a collection of resources, specified by the Hs, for exactly one from some other collection of resources, specified by Ws. To formalize this, we now have the following data for each intent;

- $\mathsf{TR}_{i,r}$: boolean indicating if intent i traded resource r.
- $\mathsf{RC}_{i,r}$: boolean indicating if intent i received resource r.
- $\mathsf{GOT}_{i,r}$: a natural number indicating how many rs were received by intent i.
- $\mathsf{GAVE}_{i,r}$: a natural number indicating how many rs were spend by intent i.

To enforce exclusivity, we must ensure that, at most one of the $\mathsf{TR}_{i,r}$ and $\mathsf{RC}_{i,r}$ booleans is true;

$$\sum_r \mathsf{TR}_{i,r} \leq 1 \tag{104}$$

$$\sum_r \mathsf{RC}_{i,r} \leq 1 \tag{105}$$

and, furthermore, we need to enforce that an intent won't trade something if it doesn't receive something.

$$\sum_r \mathsf{TR}_{i,r} = \sum_r \mathsf{RC}_{i,r} \tag{106}$$

The constraints for each partial trade now become;

$$\mathsf{GOT}_{i,r} \leq \mathsf{RC}_{i,r} \times W_{i,r} \tag{107}$$

$$\mathsf{GAVE}_{i,r} = \mathsf{TR}_{i,r} \times H_{i,r} \tag{108}$$

Everything else can stay the same, but we are now enforcing exclusivity. We can further vary this, and we will fuse these two notions of partial transactions in the demo given in section 6.2.

6.2. Example: Optimizing Intents with SCIP. With a clear idea to formalize, we can create an actual intent optimizer using an off-the-shelf solver. For this demo, we will use the SCIP solver through the Google OR Tools library within Python. We may begin by importing the library and declaring our solver;

```python
from ortools.linear_solver import pywraplp

# Create the solver
solver = pywraplp.Solver.CreateSolver('SCIP')
```

The notion of intent we will formalize will be a combination of the two notions described in section 6.1. Each partial transaction will have an exclusive set of proposed haves and wants. Each of those will have a collection of resources that can be traded.

We can start by defining a data formalization. Each intent will consist of a partial transaction that possesses

- Wh_i: A float weighing intent i.
- $H_{i,o,r}$: A natural number indicating how much of resource r intent i is willing to trade at option o.
- $W_{i,o,r}$: A natural number indicating how much of resource r intent i needs to initiate a trade with option o.
- $V_{i,r}$: A float indicating the value intent i gives to resource r.

We can use this as a guide to implement example partial transactions. We will use the following in this demo;

```python
# Define available resources
resources = ['a', 'b', 'c', 'd', 'e', 'f', 'g']

# Define test intents
# Note: 'have_exprs' and 'want_exprs' are lists representing the 'xor' expressions
# For example, 'have_exprs': [{'a': 3, 'b': 5}, {'a': 2, 'd': 1}]
# means "choose (3a or 5b) xor (2a or 1d)"
intents = [
    {'weight': 1.0,
     'have': [{'a': 3, 'b': 4, 'c': 5}, {'a': 2}, {'e': 3, 'g': 1}],
     'want': [{'d': 3}, {'f': 1}],
     'r_weights': { 'a': 1.0, 'b': 0.5, 'c': 0.6, 'd': 0.9
                  , 'e': 0.5, 'f': 10.0, 'g': 0.85}
     },
    {'weight': 2.0,
     'have': [{'c': 2, 'd': 1}, {'e': 4}, {'f': 2}],
     'want': [{'a': 2, 'b':1}, {'b': 1, 'f': 2}],
     'r_weights': { 'a': 0.7, 'b': 0.2, 'c': 0.9, 'd': 0.9, 'e': 0.85, 'f': 1.0}
    },
]
```

We can further declare the variables used to actually formalize the constraint problem. We will have the following;

- $\text{TR}_{i,o}$: boolean indicating if option $o \leq Oh$ was selected in intent i.
- $\text{RC}_{i,o}$: boolean indicating if option $o \leq Ow$ was selected in intent i.
- $\text{GOT}_{i,r}$: a natural number indicating how many rs were received by intent i.
- $\text{GAVE}_{i,r}$: a natural number indicating how many rs were spend by intent i.

These can be implemented by declaring each list of variables, along with their domains, like so;

```python
have_choice_vars = []
want_choice_vars = []
have_qty_vars = []
want_qty_vars = []

# Declare variables
for i, intent in enumerate(intents):
    # Boolean variables for each "have" option
    have_bools = [solver.BoolVar(f"have_choice_{i}_{j}") for j in range(len(intent['have']))]
    have_choice_vars.append(have_bools)
```

DOI: 10.5281/zenodo.10019113

```
    # Boolean variables for each "want" option
    want_bools = [solver.BoolVar(f"want_choice_{i}_{k}") for k in range(len(intent['want']))]
    want_choice_vars.append(want_bools)

    # Integer variables for each resource traded away
    have_qty = {r: solver.IntVar(0, max_qty, f"have_qty_{i}_{r}")\
                for option in intent['have'] for r, max_qty in option.items()}
    have_qty_vars.append(have_qty)

    # Integer variables for each resource received
    want_qty = {r: solver.IntVar(0, qty, f"want_qty_{i}_{r}")\
                for option in intent['want'] for r, qty in option.items()}
    want_qty_vars.append(want_qty)
```

We may then declare the constraints which guarantee that only one option is chosen and that a want option is chosen if and only if a have option is chosen.

$$\sum_{o} \mathsf{TR}_{i,o} \leq 1 \tag{109}$$

$$\sum_{o} \mathsf{RC}_{i,o} \leq 1 \tag{110}$$

$$\sum_{o} \mathsf{TR}_{i,o} = \sum_{o} \mathsf{RC}_{i,o} \tag{111}$$

This can be implemented as

```
# Declare constraints
# For each intent, we'll ensure that at most one "have" and one "want"
# boolean can be true, and they must coincide.
for i in range(len(intents)):
    solver.Add(solver.Sum(have_choice_vars[i]) <= 1)
    solver.Add(solver.Sum(want_choice_vars[i]) <= 1)
    solver.Add(solver.Sum(have_choice_vars[i]) == solver.Sum(want_choice_vars[i]))
```

Next, we must ensure the traded quantities are appropriate. The quantities given away must max out at the quantity stated within the have option chosen;

$$\mathsf{GAVE}_{i,r} \leq \sum_{o} \mathsf{TR}_{i,o} \times H_{i,o,r} \tag{112}$$

and the quantities got have to be exactly those specified by the chosen want option;

$$\mathsf{GOT}_{i,r} = \sum_{o} \mathsf{RC}_{i,o} \times W_{i,o,r} \tag{113}$$

We can implement these constraints as;

```
# Link "have" and "want" quantities to their choice variables
for i, intent in enumerate(intents):
    for j, have_option in enumerate(intent['have']):
        for r in resources_set(intent, 'have'):
            solver.Add(have_qty_vars[i][r] <=\
                sum(option.get(r, 0) * have_choice_vars[i][j]
                    for j, option in enumerate(intent['have'])))

    for k, want_option in enumerate(intent['want']):
        for r in resources_set(intent, 'want'):
            solver.Add(want_qty_vars[i][r] ==\
                sum(option.get(r, 0) * want_choice_vars[i][j]
                    for j, option in enumerate(intent['want'])))
```

where `resources_set` is a function that calculates the set of resources within an intent;

```
def resources_set(intent, key):
    return list(set(resource for option in intent[key] for resource in option.keys()))
```

The last constraint asserts that resources are conserved;

$$\sum_i \mathsf{GOT}_{i,r} = \sum_i \mathsf{GAVE}_{i,r} \qquad [114]$$

this can be implemented as;

```
for resource in resources:
    inflow = solver.Sum(have_qty_vars[i].get(resource, 0) for i in range(len(intents)))
    outflow = solver.Sum(want_qty_vars[i].get(resource, 0) for i in range(len(intents)))
    solver.Add(inflow == outflow)
```

We now must implement the utility function for maximization. This simply subtracts the weighted resources traded away from those received;

$$U := \sum_i Wh_i \left(\left(\sum_r V_{i,r} \mathsf{GOT}_{i,r} \right) - \left(\sum_r V_{i,r} \mathsf{GAVE}_{i,r} \right) \right) \qquad [115]$$

This can be implemented as;

```
# Declare objective
objective = solver.Objective()

for i, intent in enumerate(intents):
    for r, qty_var in have_qty_vars[i].items():
        coef = - intent['weight'] * intent['resource_weights'].get(r, 1)
        objective.SetCoefficient(qty_var, coef)

    for r, qty_var in want_qty_vars[i].items():
        coef = intent['weight'] * intent['resource_weights'].get(r, 1)
        objective.SetCoefficient(qty_var, coef)

objective.SetMaximization()
```

Now we can actually solve the system using;

```
status = solver.Solve()
```

The following will print the output in a nice format;

```
if status == pywraplp.Solver.OPTIMAL:
    print(f"Objective value = {objective.Value()}")
    for i, intent in enumerate(intents):
        if sum(var.solution_value() for var in have_choice_vars[i]) == 1 and \
            sum(var.solution_value() for var in want_choice_vars[i]) == 1:
            traded_haves = []
            traded_wants = []
            for resource, var in have_qty_vars[i].items():
                if var.solution_value() > 0:
                    traded_haves.append(f"{var.solution_value()} {resource}")
            for resource, var in want_qty_vars[i].items():
                if var.solution_value() > 0:
                    traded_wants.append(f"{var.solution_value()} {resource}")
            print(f"Intent {i + 1} trades {' and '.join(traded_haves)}\
                    for {' and '.join(traded_wants)}")
else:
    print("The problem does not have an optimal solution.")
```

Running that will produce the final solution;

```
Objective value = 8.700000000000001
Intent 1 trades 2.0 a and 1.0 b for 1.0 f
Intent 2 trades 1.0 f for 2.0 a and 1.0 b
```

7. Distributed CSPs

Rather than designing a central system to solve a CSP, it may be prudent to distribute the workload across several solvers. The idea is that we have numerous agents working in parallel, each responsible for a finite segment of the CSP problem (generally, a small set of variables or a single variable). They repeatedly use knowledge of their local environment to search for solutions and continuously send messages to locally relevant agents in order to approach a cooperative maximum.

The following quote from (Rossi et al., 2006) should be noted as the main guiding principle to determine if distributed CSP methods are appropriate;

> In general, distributed techniques work well only when the problem is sparse and loose, i.e. each variable has constraints with only a few other variables, and for each constraint, there are many value combinations that satisfy it. When problems are dense and tight, usually the distributed solution requires so much information exchange among the agents that it would be much more efficient to communicate the problem to a central solver.

For a broad overview of the subject, see (Fioretto et al., 2018), especially sections 4.3 and 4.4, which summarize algorithms and trade-offs in more detail than here.

7.1. Complete Algorithms. The most basic algorithms simply distribute search over all the agents. Each agent is assigned a variable, and they synchronously "count" through variable combinations. This requires the agents to be linearly ordered so that higher priority agents only change their assignment after the lower priority ones exhaust their options. This will clearly require an exponential number of messages to be passed between the agents.

There are various ways this setup can be improved. Rather than broadcasting assignments, each agent could, instead, broadcast optimistic scores for certain assignments, and receive messages from neighboring agents dictating if such assignments are possible. If they aren't, then backtracking must be done. This is still asymptotically exponential, but is guided in a smarter way; being best-first instead of depth-first.

These search-based methods can benefit heavily from preprocessing, such as the decomposition of isolated parts; see sections 3.3 and 4.2. This would allow parts of the search to be performed in parallel in relative isolation; though this possibility is heavily problem-dependent. There are many useful preprocessing techniques, and this would require a centralized preprocessor.

There are also algorithms based, not on search, but on inference. These will calculate a representation of all good options; essentially a tree representing each value combination, pruned for redundancy. This entire database is propagated to other agents, so they may further prune and add to the tree based on these new agents' options. Ultimately, this propagates up to the head agent, which makes a final pruning which is the optimal choice. In this setup, we no longer have an exponential number of messages but, instead, a linear number of messages exponential in size. Each agent is also expected to do an exponential amount of work.

These exponential sizes are intrinsic if we want to guarantee some kind of optimum. In many cases, we cannot afford such an effort, and so must accept a best-effort suboptimal solution.

7.2. Incomplete Algorithms. Incomplete algorithms are more varied and often simpler than their complete counterparts. One fundamental approach involves agents performing localized hill-climbing, where they periodically update their variables to an optimal value. This local optimization can be further enriched through inter-agent communication, allowing the agents to consider assignment costs that affect multiple participants. A major challenge to CSP decomposition is the inability to combine optimal solutions to sub-problems to get an optimal solution for the entire problem. Despite this, for practical applications, combining these sub-solutions often yields results that are good enough. What local context is being optimized can vary. Rather than optimizing between actual choices made by neighboring agents, they may optimize against some expectation of values determined by some heuristic, or through a distribution propagated by other agents.

Agents can be given additional powers, allowing them to more smartly decide on good solutions. A solution might look locally good but would cause other agents to react in such a way that it becomes a bad choice. As such, each choice is a gamble, of sorts, where the immediate change in value is merely a hint. Agents can, instead, run bandit algorithms (Lattimore and Szepesvári, 2020) to make better decisions over time.

Perhaps the most relevant incomplete method for our application are stochastic search methods. The most classic method is WalkSAT. This algorithm involves looking up an unsatisfied clause, and randomly changing a variable to make that clause satisfied. This is then repeated until either satisfaction or some timeout. While the original version of the algorithm was sequential, there are many simple ways to make the algorithm more parallel, as well as adaptations of more sophisticated techniques from other approaches to solving (McDonald et al., 2009).

More promising for general combinatorial optimization are the so-called annealing methods. These come in many flavors, but the idea is to replace discrete variables with continuous variables ranging between real values encoding possible assignments. The problem is then turned into a description for a potential well where solutions are global minima and better solutions are lower minima. The values are then evolved according to a stochastic differential equation of some kind in the hopes of landing at a decent minimum.

As a basic example, we may model the inequality relation in three coloring by the following 2-dimensional potential;

$$V(x,y) := \prod_{i,j\in\{-1,0,1\},i\neq j} (x-i)^2 + (y-j)^2 \qquad [116]$$

Here, the three colors are encoded as -1, 0, and 1, and this potential reaches a global minimum of 0 if and only if the input pair encodes an unequal pair of colors. Given a CSP, we can convert each constraint into a potential, and then generate a potential for the whole problem by summing all the constraint potentials. The most basic way to turn this into a dynamical system is to interpret the potential as a gradient flow. We can simulate a particle following the equation $x_i'(t) = -\partial_{x_i(t)} V(x(t))$, for each dimension/variable i. This is just gradient descent. Generally, this method is prone to get stuck in local minima, but there's a vast body of research, mainly coming from machine learning, for improving this method (Ruder, 2016).

Much can be done to improve things like numerical stability and modeling through careful consideration of the potential. The potential we've given as an example is easy to understand, but suboptimal for most purposes. One can replace the squares with absolute values and the product with taking a minimum, and one will get a linearized "triangle" potential with minima at the same points. Furthermore, by restricting the domain, say to $x, y \in [-1, 1]$, the potential will be bounded, and we may then normalize it to $[0, 1]$; appropriate as these model propositions. This assists in further generalizations. With normalized potentials, we can consistently model weighted MaxCSP by simply multiplying each potential by their weights when summing the potentials for each clause. We may then handle hard constraints by setting their weight to be greater than the sum of the weights of the soft constraints. This ensures that every hard clause can overpower all the soft clauses if need be.

There are two classic annealing methods used to minimize these potentials; thermal annealing and chaotic annealing. In thermal annealing, the trajectory of assignments receives a force directly from the potential well assigned to the problem, plus an additional kick from random, usually white, noise (Kirkpatrick et al., 1983). More technically, this would involve simulating something like $x_i'(t) = -\partial_{x_i(t)} V(x(t)) + W(t)$, where $W(t)$ is a random process (usually a Gaussian) sampled at each time-step of the simulation. So long as the temperature/size of kicks decreases over time slowly enough (often exponentially slow), one is essentially guaranteed to settle in a global minimum. Chaotic annealing is another extreme where the dynamic system is entirely deterministic, but chaotic dynamics are introduced to facilitate exploration. Generally, a chaotic function is treated as noise added to the system. So the dynamics might look something like $x_i'(t) = -\partial_{x_i(t)} V(x(t)) + g(t)$, where $g(t)$ is a chaotic function replacing nondeterministic noise. Temperature becomes a coefficient on this function when added to the potential, so the chaos has less of an influence the lower the temperature. The key property of chaotic dynamics is that the same point is never explored twice, accelerating a search for minima (Zhou and Chen, 1997).

These two methods have different drawbacks. Thermal annealing is very simple, but we have poor guarantees for the rate of convergence. Chaotic annealing can give better guarantees, but the cost of simulating a chaotic system can grow exponentially with the need for accuracy. Chaotic systems are also often quite sensitive to noise, making hardware implementations of such systems less reliable than their theoretical counterparts. More recent work in the theory of stochastics (see (Ovchinnikov, 2016)) has led to the creation of so-called "N-phase" annealing that compromises between the two. Exactly how these should be implemented is still an ongoing project, but there seem to be two main descriptions. In (Ovchinnikov and Wang, 2017), N-phase annealing is described as continuously shifting between a near-thermal phase and a near-chaotic phase based on how long its been since progress has been made on optimization. Without a practical implementation, it's hard to know exactly what this would look like. Another approach is described in (Di Ventra, 2022), where the system is entirely deterministic, and stochastics are assumed to be natural environmental noise to which the system must be robust. The system is non-chaotic but periodically enters critical states that upset the global configuration of the system until a better minimum is found. This is done by adding, for each clause, two variables, one being a "short-term" memory indicating the degree to which the constraint was satisfied in the near-past, and a "long-term" memory, increasing the potential generated by the associated clause the longer it's remained unsatisfied. Schematically, the set of differential equations will be some variant of the following;

$$x_i'(t) = \sum_{m\in\text{Clauses}} -\partial_{x_i(t)} V_m(x(t))\big(s_m(t)l_m(t) + (1 - s_m(t))(1 + \zeta l_m(t))R_{i,m}\big) \qquad [117]$$

$$s_i'(t) = \beta(s_i(t) + \epsilon)(V_i(x(t)) - \gamma) \qquad l_i'(t) = \alpha(V_i(x(t)) - \delta) \qquad [118]$$

Where $s_i(t) \in [0, 1]$, $l_i(t) \geq 0$, the Greek letters are constant hyperparameters that must be tuned, and $R_{i,m}$ is a function assigning a value in $[0, 1]$ indicating the degree to which variable i is responsible for clause m being satisfied. Conceptually, the gradient of each variable descends the potential but is modulated by the short and long-term memory. If the short-term memory is 1 (i.e. not satisfied) the long-term memory is in full force, but if it's 0 (i.e. satisfied), the long-term memory is depressed, and, further, a variable is only affected by the potential of a satisfied clause if that variable is the reason the clause is satisfied (i.e. if R is 1). In our example of three coloring, R will always be 1, but this is not the case in, for example, 3-SAT where a single variable can cause a clause to be satisfied irrespective of the other variables in that clause. See, for example, (Bearden et al., 2020) and (Zhang and Di Ventra, 2023) for an explicit example in the context of boolean sat and details on setting the constants.

When the potential of a long-time unsatisfied clause gets powerful enough to overcome the potentials of the clauses that share variables with it, this triggers a critical transition. With particular variants, the dynamical systems can be made to be dissipative; that is, all volumes decrease over time. This guarantees that the system will eventually settle into a well around some stable equilibria, as this prevents cycles or chaos. In such a case, one can understand the new system as turning all local minima in the original potential into saddle points. Such methods have been specialized for integer programming in (Traversa and Di Ventra, 2018). Such methods have recently shown empirical quadratic scaling for prime factorization on classical computers (Sharp et al., 2023). Existing designs are hand-crafted using expert knowledge, but see (Caravelli et al., 2021) and (Caravelli et al., 2023) for recent work on more systematic transformations of dynamical systems. These methods echo some previous physics-inspired methods, such as the method of discrete Lagrange multipliers; see (Shang and Wah, 1998) and section 6.3 of (Biere et al., 2020), and survey propagation; see (Mézard et al., 2002) and sections 6.5 and 10.8 of (Biere et al., 2020).

8. Acknowledgements

This study on constraint satisfaction was an effort supported by the Anoma Foundation. The author extends sincere gratitude to Christopher Goes for his invaluable guidance on shaping the report's scope, emphasis, and overarching objectives. Acknowledgment is also due to D. Reusche for insightful discussions on the articulation of intents as CSPs. Appreciation is extended to Jonathan Prieto-Cubides for his assistance with formatting. This report was enriched by their collective expertise and dedication.

References

Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006. (cit. on pp. 2 and 27.)

E Tsang. Foundations of constraint satisfaction. *(No Title)*, 1993. (cit. on p. 2.)

Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*. IOS press, 2020. (cit. on pp. 3, 8, 9, 17, and 29.)

IUriĭ V Matiiasevich. *Hilbert's tenth problem*. MIT press, 1993. (cit. on pp. 3 and 4.)

Arnold L Rosenberg. Efficient pairing functions-and why you should care. *International journal of foundations of computer science*, 14(01):3–17, 2003. (cit. on p. 4.)

Numberphile. Zero Knowledge Proof (with Avi Wigderson). https://www.youtube.com/watch?v=5ovdoxnfFVc, 2009. [Online; accessed 19-Sep-2023]. (cit. on p. 4.)

Tomás Feder and Moshe Y Vardi. The computational structure of monotone monadic snp and constraint satisfaction: A study through datalog and group theory. *SIAM Journal on Computing*, 28(1):57–104, 1998. (cit. on pp. 6 and 22.)

Andrei Krokhin and Jakub Opršal. An invitation to the promise constraint satisfaction problem. *ACM SIGLOG News*, 9(3):30–59, 2022. (cit. on pp. 7 and 22.)

Dmitriy Zhuk, Barnaby Martin, and Michał Wrona. The complete classification for quantified equality constraints. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2746–2760. SIAM, 2023. (cit. on p. 8.)

Jin-Yi Cai and Xi Chen. Complexity of counting csp with complex weights. *Journal of the ACM (JACM)*, 64(3):1–39, 2017. (cit. on p. 8.)

Laurence A Wolsey. *Integer programming*. John Wiley & Sons, 2020. (cit. on p. 9.)

Conforti Michelangelo, P Cornujols Grard, and Zambelli Giacomo. Integer programming (graduate texts in mathematics), vol. 271, 2014. (cit. on p. 9.)

Ferdinando Fioretto, Enrico Pontelli, and William Yeoh. Distributed constraint optimization problems and applications: A survey. *Journal of Artificial Intelligence Research*, 61:623–698, 2018. (cit. on pp. 10 and 27.)

Filippo Bonchi, Dusko Pavlovic, and Pawel Sobocinski. Functorial semantics for relational theories. *arXiv preprint arXiv:1711.08699*, 2017. (cit. on p. 12.)

Tao Gu, Robin Piedeleu, and Fabio Zanasi. A complete diagrammatic calculus for boolean satisfiability. *Electronic Notes in Theoretical Informatics and Computer Science*, 1, 2023. (cit. on p. 12.)

Marijn JH Heule, Benjamin Kiesl, and Armin Biere. Short proofs without new variables. In *Automated Deduction–CADE 26: 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6–11, 2017, Proceedings*, pages 130–147. Springer, 2017. (cit. on p. 13.)

Lee A Barnett and Armin Biere. Non-clausal redundancy properties. In *CADE*, pages 252–272, 2021. (cit. on p. 13.)

Eyal Amir and Sheila McIlraith. Solving satisfiability using decomposition and the most constrained subproblem (preliminary report). *Electronic Notes in Discrete Mathematics*, 9:329–343, 2001. (cit. on p. 14.)

Anton Belov, António Morgado, and Joao Marques-Silva. Sat-based preprocessing for maxsat. In *Logic for Programming, Artificial Intelligence, and Reasoning: 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings 19*, pages 96–111. Springer, 2013. (cit. on p. 16.)

Matthew Galati. *Decomposition methods for integer linear programming*. Lehigh University, 2010. (cit. on p. 17.)

Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural csp decomposition methods. *Artificial Intelligence*, 124(2):243–282, 2000. (cit. on p. 19.)

Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. *ACM Transactions on Algorithms (TALG)*, 11(1):1–20, 2014. (cit. on p. 20.)

Dániel Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *Journal of the ACM (JACM)*, 60(6):1–51, 2013. (cit. on p. 20.)

Yuri Gurevich. Logic and the challenge of computer science. Technical report, 1985. (cit. on p. 20.)

Neil Immerman. Relational queries computable in polynomial time. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 147–152, 1982. (cit. on p. 20.)

Erich Grädel. The expressive power of second order horn logic. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 466–477. Springer, 1991. (cit. on p. 20.)

Daniel Wiebking. *A decomposition-compatible canonization framework for the graph isomorphism problem*. PhD thesis, Dissertation, RWTH Aachen University, 2021, 2021. (cit. on p. 20.)

Andreas Blass, Yuri Gurevich, and Saharon Shelah. Choiceless polynomial time. *Annals of Pure and Applied Logic*, 100(1-3):141–187, 1999. (cit. on p. 20.)

Moritz Lichter and Pascal Schweitzer. Choiceless polynomial time with witnessed symmetric choice. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 1–13, 2022. (cit. on p. 20.)

Andrei A Bulatov. A dichotomy theorem for nonuniform csps. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 319–330. IEEE, 2017. (cit. on p. 21.)

Dmitriy Zhuk. A proof of the csp dichotomy conjecture. *Journal of the ACM (JACM)*, 67(5):1–78, 2020. (cit. on p. 21.)

Dietlinde Lau. *Function algebras on finite sets: Basic course on many-valued logic and clone theory*. Springer Science & Business Media, 2006. (cit. on p. 21.)

Martin Grohe. The structure of tractable constraint satisfaction problems. In *Mathematical Foundations of Computer Science 2006: 31st International Symposium, MFCS 2006, Stará Lesná, Slovakia, August 28-September 1, 2006. Proceedings 31*, pages 58–72. Springer, 2006. (cit. on p. 21.)

Zarathustra Brady. Notes on csps and polymorphisms. *arXiv preprint arXiv:2210.07383*, 2022. (cit. on p. 21.)

Libor Barto, Andrei Krokhin, and Ross Willard. Polymorphisms, and how to use them. In *Dagstuhl Follow-Ups*, volume 7. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017. (cit. on p. 21.)

Libor Barto, Zarathustra Brady, Andrei Bulatov, Marcin Kozik, and Dmitriy Zhuk. Unifying the three algebraic approaches to the csp via minimal taylor algebras. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13. IEEE, 2021. (cit. on p. 21.)

Dmitriy Zhuk and Barnaby Martin. Qcsp monsters and the demise of the chen conjecture. *Journal of the ACM*, 69(5):1–44, 2022. (cit. on p. 22.)

Dmitriy Zhuk. The complexity of the quantified csp having the polynomially generated powers property. *arXiv preprint arXiv:2110.09504*, 2021. (cit. on p. 22.)

Ferdinand Börner, Andrei Bulatov, Hubie Chen, Peter Jeavons, and Andrei Krokhin. The complexity of constraint satisfaction games and qcsp. *Information and Computation*, 207(9):923–944, 2009. (cit. on p. 22.)

Manuel Bodirsky. Complexity classification in infinite-domain constraint satisfaction. *arXiv preprint arXiv:1201.0856*, 2012. (cit. on p. 22.)

Miguel Couceiro, Lucien Haddad, and Victor Lagerkvist. Fine-grained complexity of constraint satisfaction problems through partial polymorphisms: A survey. In *2019 IEEE 49th International Symposium on Multiple-Valued Logic (ISMVL)*, pages 170–175. IEEE, 2019. (cit. on p. 22.)

Libor Barto and Marcin Kozik. Robustly solvable constraint satisfaction problems. *SIAM Journal on Computing*, 45(4):1646–1669, 2016. (cit. on p. 22.)

Tor Lattimore and Csaba Szepesvári. *Bandit algorithms*. Cambridge University Press, 2020. (cit. on p. 27.)

Austin McDonald et al. Parallel walksat with clause learning. *Data analysis project papers*, 2009. (cit. on p. 27.)

Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016. (cit. on p. 28.)

Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983. (cit. on p. 28.)

Chang-song Zhou and Tian-lun Chen. Chaotic annealing for optimization. *Physical Review E*, 55(3):2580, 1997. (cit. on p. 28.)

Igor V Ovchinnikov. Introduction to supersymmetric theory of stochastics. *Entropy*, 18(4):108, 2016. (cit. on p. 28.)

Igor V Ovchinnikov and Kang L Wang. Stochastic dynamics and combinatorial optimization. *Modern Physics Letters B*, 31(31):1750285, 2017. (cit. on p. 28.)

Massimiliano Di Ventra. *MemComputing: fundamentals and applications*. Oxford University Press, 2022. (cit. on p. 28.)

Sean RB Bearden, Yan Ru Pei, and Massimiliano Di Ventra. Efficient solution of boolean satisfiability problems with digital memcomputing. *Scientific reports*, 10(1): 19741, 2020. (cit. on p. 28.)

Yuan-Hang Zhang and Massimiliano Di Ventra. Implementation of digital memcomputing using standard electronic components. *arXiv preprint arXiv:2309.12437*, 2023. (cit. on p. 28.)

Fabio L Traversa and Massimiliano Di Ventra. Memcomputing integer linear programming. *arXiv preprint arXiv:1808.09999*, 2018. (cit. on p. 29.)

Tristan Sharp, Rishabh Khare, Erick Pederson, and Fabio Lorenzo Traversa. Scaling up prime factorization with self-organizing gates: A memcomputing approach. *arXiv preprint arXiv:2309.08198*, 2023. (cit. on p. 29.)

Francesco Caravelli, Forrest C Sheldon, and Fabio L Traversa. Global minimization via classical tunneling assisted by collective force field formation. *Science Advances*, 7(52):eabh1542, 2021. (cit. on p. 29.)

Francesco Caravelli, Fabio L Traversa, Michele Bonnin, and Fabrizio Bonani. Projective embedding of dynamical systems: Uniform mean field equations. *Physica D: Nonlinear Phenomena*, 450:133747, 2023. (cit. on p. 29.)

Yi Shang and Benjamin W Wah. A discrete lagrangian-based global-search method for solving satisfiability problems. *Journal of global optimization*, 12:61–99, 1998. (cit. on p. 29.)

Marc Mézard, Giorgio Parisi, and Riccardo Zecchina. Analytic and algorithmic solution of random satisfiability problems. *Science*, 297(5582):812–815, 2002. (cit. on p. 29.)