Abstract

Constraint logic programming is a popular interpretation of logic programming, focused on using constraint satisfaction to provide a declarative approach to otherwise intractable problems. This paper will focus on providing an overview of constraint logic programming and, in particular, efforts to apply it to graphs, as well as an overview of some alternative logic programming approaches to graph problems.

1 Introduction

The purpose of this paper is to provide a literature search of recent applications of constraint logic programming and related techniques to graph analysis, as well as the basic overview of core concepts for those techniques, with an ultimate eye towards supporting further research into general CLP graph-analysis.

The paper is divided up into a broad overview of CLP, followed by an analysis of applications to graphs, divided up by the broad problems and types of graphs they were intended to deal with. Other approaches are touched on only lightly, followed by a brief summary of the applications CLP graph analysis techniques have been devoted to.

2 Constraint Logic Programming

Constraint Logic Programming is a programming paradigm that applies constraint solving techniques to logic programming [23]. It has been applied to numerous applications, including circuit analysis [36], scheduling, biochemical networks [12], DNA sequencing [6], image analysis [14], and many other combinatorial search problems.

In it, problems are defined as a set of variables, which must be assigned values from an appropriate domain, and as a set of constraints on the combinations of values that these variables can take. A valid solution is one that assigns values that satisfy all the problem’s given constraints.

More formally, a constraint satisfaction problem $\langle X, D, C \rangle$ consists of a set of variables, $\{X_1...X_n\}$; a set of corresponding domains $\{D_1...D_n\}$, with each $D_i$ serving as a set of possible values for variable $X_i$; and a set of constraints $C$, each of which applies to a subset of the variables and imposes a relation between their domains. A solution to a CSP is therefore an assignment to all variables which satisfies all of their corresponding constraints.

An undirected graph called a constraint network, pictured in figure 1, is used to represent the relation between constraints and variables. Edges in this graph represent constraints, while nodes represent the variables. In the given example, the variables are $A$, $B$, and $C$, constrained with $A != B$, $B > C$, and $A != C$.

The range of possible assignments can be divided into a search space, which can be represented as a second, directed graph with nodes for each possible set of assignments and edges that represent making a particular assignment to a variable; finding a solution is therefore a matter of traversing this graph from the
start node to the solution. Efficiency is determined by the order in which edges are followed and the ability to prune out choices that cannot lead to a satisfying answer, so significant improvements can be made by relying on strategies such as lookahead to guide the search. [21]

An example search space for the constraints given above is shown below, with each variable given a domain of \{1, 2, 3\}. Notice that, for example, starting from the initial assignment of \(C = 3\) in the upper-right will never lead to a valid assignment.

2.1 Variables and Domains

Finite domains assign variables to values from a given list; often, this list is from the set of integers, although non-integer finite domains are also possible. Variables are given an initial finite domain (such as an integer range), which is limited by constraint propagation - before we traverse the search space, we take the constraints and reductions on one domain and use them to reduce domains connected to it by constraints in order to remove 'unsupported' assignments, assignments that cannot be satisfied. Only once these constraints have reduced the domain down to a reasonably small size is the backtracking search invoked,
exploring the graph of the search space and backtracking when a particular path leads nowhere.

Set variables allow constraints based on set relations, such as supersets and subsets. The lack of ordering in sets is inherently valuable for many CLP applications, since it breaks symmetries that would otherwise appear when representing them in FD using ordered lists [15]. Set variables are generally assigned values representing a combination of members of a given superset; their initial domain can therefore be defined using the superset of which possible values of the variable will be a subset.

Real variables allow constraints with real number values; they were an early type of CLP constraint implemented by Jaffar to demonstrate the expressive power of constraint logic programming. Constraints on these variables can be expressed with mathematical operations (ranging from simple less-than, greater-than, statements and so forth, all the way to far more complicated algebraic expressions), allowing arithmetic comparisons in an intuitive fashion [24].

Little research to date has been done for graph domains, and all of those have built on top of existing domain variables. Dooms et al [12], for instance, built their CLP(g) on top of finite domains and finite sets; the details of their implementation will be covered in more depth further down.

2.2 Constraints

CLP constraints are, functionally, like sub-problems whose results are then propagated between each other via their impact on the domains of the variables they apply to [34]. This allows for completely distinct approaches to different aspects of the problem to be easily combined. The application of a constraint to one variable reduces its domain in a filtering step. This reduction can then be used to reduce the domains of other variables which are linked to it by constraints in the constraint network, propagating constraints across the solution space.

Finally, we search the available space by attempting to assign values to variables, applying filtering and propagation after these assignments and then making new assignments until we either reach a solution or hit a point where no valid values remain, forcing us to backtrack [34].

Constraints divided into local constraints, applied independently to individual variables, and global constraints, applied to a group of variables at once. Global constraints allow for the use of algorithms that can work more efficiently by accessing the domains of multiple variables directly, rather than relying on the propagation of domain reductions achievable from a purely local view.

Beyond this, constraints cover a wide range, including numerical constraints, which constrain a specific value with a numerical function; functional constraints, which constrain a pair of variables with a function establishing some relation between them; value occurrence constraints, which constrain how many times certain values can appear across a set of variables; value distribution constraints, which constrain the distribution of a value across a set in other ways; sequencing constraints, which constrain the sequence in which values can appear for an ordering on a set of variables; scheduling constraints, intended for
resource scheduling; packing constraints, for knapsack or bin-packing problems; and objective function constraints, intended for maximizing or minimizing some value [32].

2.3 Constraint Propagation

The simplest way to assign values to variables is to repeatedly try different combinations, backtracking on hitting a point where it’s no longer possible to make an assignment. However, doing so can lead to repeatedly attempting similar combinations of assignments if, for instance a choice was made early on which could not lead to a correct answer. Summarizing these problems, Mackworth [25] proposed a concept of local consistency, which could be enforced by propagating constraints through the domains of connected variables. A propagator for a constraint would therefore enforce a constraint by reducing the domains of connected variables as necessary whenever a variable was assigned or a domain reduced.

Mackworth’s original consistencies were node consistency, which ensures every variable is consistent with the unary constraints that directly apply to it; arc consistency, which ensures all pairs of variables connected by constraints (‘arcs’) are consistent with the binary constraints that apply to them; and path consistency, which ensures that, for a pair of nodes, there exists a pair of assignments for them which satisfies their constraints while also satisfying their binary constraints with any third variable. More formally, Mackworth defines these forms of consistency for nodes $i$ and $j$, unary predicates $P_x$, and binary predicates $P_{x,y}$ as follows:

- A node $i$ is node consistent iff for any value $x \in D_i$, $P_i(x)$ holds.
- An arc $(i, j)$ is arc consistent iff for any value $x \in D_i$ such that $P_i(x)$ holds, there is a value $y \in D_j$ such that $P_j(y)$ and $P_{ij}(x,y)$ holds.
- A length-$m$ path through the nodes $(i_0, i_1, ..., i_m)$ is path consistent iff for any values $x \in D_{i_0}$ and $y \in D_{i_m}$ such that $P_{i_0}(x)$ and $P_{i_m}(y)$ and $P_{i_{0:m}}(x,y)$ all hold, there is a sequence of values $z_1 \in D_{i_1}, ..., z_{m-1} \in D_{i_{m-1}}$ such that, first, $P_{i_1}(z_1)$ and ... and $P_{i_{m-1}}(z_{m-1})$ holds; and second, $P_{i_{0:1}}(x,z_1)$ and $P_{i_{1:2}}(z_1,z_2)$ and ... and $P_{i_{m-1:m}}(z_{m-1}, y)$ holds.

Of these, node consistency is the simplest to enforce, involving only a single pass to remove all values that are inconsistent with the constraints on each variable.

2.3.1 Arc consistency

Arc consistency is more complex. The first solution Mackworth proposed, which he called AC-1, was to simply iterate over all binary constraints and remove any values which lacked a possible value on the other side of the constraint that would make them consistent; every time such a domain reduction is made, AC-1 will iterate over all constraints again from the beginning.
Iterating over all constraints again after every reduction, however, results in a large amount of wasted effort, as AC-1 would repeatedly re-check binary constraints that were not directly affected by the removal. AC-2 remedies this by only re-examining binary constraints that include the variable which had its domain reduced, rather than all of them. Rather than restarting from the beginning, therefore, AC-2 uses a queue to track the nodes in the constraint network one step away from a node \( i \) under consideration, and adds them to the queue if the domain of \( i \) is revised. While an improvement, this can still result in redundancy, since a variable can end up in the queue multiple times simultaneously. AC-3 solves this using a queue of all arcs instead, which are successively examined and have their variables’ domains revised based on their binary constraints; if a variable from an arc that has already been examined has its domain reduced, that arc is re-inserted into the queue.

With a constraint network consisting of \( n \) nodes, \( a \) constraints (or arcs), and \( e \) edges, AC-3 has a worst-case time of \( O(ea^3) \); its queues give it a space complexity of \( O(ea^3) \). To improve on this time complexity, Mohr and Henderson [27] proposed AC-4, which achieved a running time of \( O(ea^2) \) by working on the basis of measuring support for each value; specifically, since potential values are removed from a domain when they have no way of satisfying a given constraint, every possible value for another variable which allows it to satisfy that particular constraint counts as a support for that constraint. The number of supports a value has are tracked, and every time a value is removed as an option, all of the other values it supported have their supports decremented by one; when a value has no support on a particular constraint, it is removed in turn. The space complexity of this approach is \( O(ea^2) \), the same as AC-3.

An example of how this works is shown in the figures below; the variables \( A, B, \) and \( C \) each have an initial domain of \( 1, 2, 3 \), with the constraints \( A < B \) and \( B < C \). Values that lack a support are highlighted in red. Initially, as seen in figure 3, \( A = 3 \) and \( B = 1 \) lack support on the constraint \( A < B \), while \( B = 3 \) and \( C = 1 \) lack support on the constraint \( B < C \). Removing those unsupported values produces figure 4, where \( A = 2 \) and \( C = 2 \) are now deprived of constraints; removing those in turn produces figure 5, our answer.

In practice, AC-4 has a better worst case than AC-3, but AC-3 has a better average case [38] [3].

AC-5, devised by Hentenryck et al [22], refined AC-3 by having each entry in the reconsideration-queue consist not just of an arc, but an arc and the value that was removed. Each removal causes a new entry to be placed in the queue for each value that was removed from that arc. For functional and monotonic constraints, it achieves a time complexity of \( O(ed) \), for a number of arcs \( e \) with a largest domain \( d \).

Perlin [31] produced an algorithm based on AC-4 which they also called AC-5. This AC-5 is similar in the sense that it attempts to exploit graph structures to gain an advantage with specific types of graphs.

Bessiere [2] refined AC-3 and AC-4 into AC-6, which achieves an average time complexity superior to that of AC-3, while matching the superior \( O(ed_4) \)
worst time complexity of AC-4 with a reduced space complexity of $o(ed)$. It does this by reducing the number of supports it tracks. AC-4 finds every support for each value in its initial setup (a significant overhead that explains its high average time complexity) and then saves all of them so it can track the exact number of support each value has (which gives it its high space complexity.) But we only actually need to know that a value has a single support for it to remain in our domain; therefore, AC-6 finds and tracks only a single support arbitrary for value. If that support is lost, it searches for another, and removes the value if it cannot find one.

Bessiere et al [3] later refined AC-6 itself into AC-7, which takes advantage of the bidirectionality of supports to reduce unnecessary support checks; that is to say that if a value A supports a value B on a constraint, we can infer that B
supports A on that constraint (and, conversely, if A does not support B, then B does not support A).

2.3.2 Path consistency

With arc consistency, it is possible to have two variables with values in their domains that satisfy any binary constraints between them, but which are together inconsistent with constraints including a third variable (that is, a "path" of length two formed by those constraints is inconsistent with those results). Path consistency detects and eliminates such inconsistencies.

Montanari [29] described the basic problem of path consistency and proved that establishing consistency for paths of length 2 was sufficient to establish path consistency. Montanari’s solution was comparable to AC-1; iterate over the entire set of length-2 paths, repeatedly removing invalid combinations until it was path-consistent. Mackworth classified this algorithm as PC-1 and proposed a refinement, PC-2, along the lines of AC-3 – maintaining a queue of length-2 paths that must be examined, and re-enqueuing any paths that were potentially invalidated by a removal if they are not already in the queue [25].

One important factor to consider when searching for a solution is the existence of symmetry in the problem; this is when multiple solutions fall into the same equivalence class, representing the same basic concept with, for example, a different ordering. We normally want only one solution from each equivalence class, since it is a waste of computation time to repeatedly find trivial permutations of the same answer; so it is important to recognize and handle equivalences.
3 CLP on Graphs

Graph analysis is key to many intractable problems, and therefore a natural place to apply constraint logic programming. In particular, graph matching and subgraph isomorphism, pathfinding and reachability, and graph labeling contain many relevant problems and have therefore seen work from various angles.

3.1 Graph Matching and Subgraph Isomorphism

Graph matching and subgraph isomorphism deals with determining the similarity or difference between two graphs and, sometimes, quantifying the changes necessary to turn one into the other. It is particularly relevant in genetics, biochemistry [12], and pattern recognition. [7]

3.1.1 Dooms et al

Dooms et. al [11] [12] wrote CP(Graph), a general-purpose computation domain for graphs focused on finding subgraphs using constraints; while it has broad applications, it was primarily intended for biochemical network analysis. They implemented graph domain variables using finite sets, with graphs represented by sets of nodes, arcs, and arcnodes (the latter representing the connections between nodes and arcs); weights are represented by constraints rather than variables. While they designed CP(Graph) for directed graphs, and used that in most of their examples, it can also support undirected graphs. Many constraints work for both, so only a few additional constraints are required.

Building CP(Graph) on top of finite domains and finite sets allows them to include and exploit existing finite domain and finite set constraints, such as set inclusion, intersection, difference, cardinality, membership, inequality, and weight.

Graphs are defined as a set of nodes, \( SN \), and a set of edges, \( SA \), subject to the constraint \( SA \subseteq SN \times SN \). Their graphs have a partial ordering based on graph inclusion; given \( g_1 = (sn_1, sa_1) \) and \( g_2 = (sn_2, sa_2) \), \( g_1 \subseteq g_2 \) iff \( sn_1 \subseteq sn_2 \) and \( sa_1 \subseteq sa_2 \).

Dooms illustrates the graph ordering as follows:

![Figure 6: Source: Dooms et al](image)

Graphs therefore have a domain bounded by the greatest lower bound and the least upper bound within this ordering; the least upper bound defines all nodes and edges that could possibly be part of a graph, while the greatest lower bound contains all nodes and edges that must be part of that graph.
The most basic constraints on these variables are 'kernel' constraints, which define the structure of a graph and which can be composed with finite domain and finite set constraints to make additional constraints. The kernel constraints are \(\text{Arcs}(G, SA)\), defining \(SA\) as the set of edges for \(G\); \(\text{Nodes}(G, SN)\), defining \(SN\) as the set of nodes for \(G\); and \(\text{ArcNode}(A, N_1, N_2)\), defining \(A\) as an edge from node \(N_1\) to node \(N_2\).

Formally, the basic domains for a set of edges, \(SA\), are defined by the least upper bound on that set of edges, \(\text{sa}_l\) and corresponding greatest upper bound, \(\text{sa}_u\); therefore, \(\text{dom}(SA) = [\text{sa}_l, \text{sa}_u]\). The domain for nodes is defined similarly as \(\text{dom}(SN) = [\text{sn}_l, \text{sn}_u]\), and a graph domain is defined similarly as \(\text{dom}(G) = [\text{gl}, \text{gu}]\) with \(\text{gl} = (\text{gsn}_l, \text{gsa}_l)\) and \(\text{gu} = (\text{gsn}_u, \text{gsa}_u)\).

The propagation rule for the arc constraint sets new bounds for \(\text{sa}\) and \(\text{gsa}\), \(\text{sa'}_l\) and \(\text{gsa'}_l\). The lower bound consists of the union of required arcs from both \(\text{sa}\) and \(\text{gsa}\), while the upper bound is the intersection of arcs present in both \(\text{sa}\) and \(\text{gsa}\):

\[
\text{sa'}_l = \text{gsa'}_l = \text{sa}_l \cup \text{gsa}_l \\
\text{sa'}_u = \text{gsa'}_u = \text{sa}_u \cap \text{gsa}_u
\]

The \(\text{Nodes}(G, SN)\) constraint is propagated similarly:

\[
\text{sa'}_l = \text{gsa'}_l = \text{sa}_l \cup \text{gsa}_l \\
\text{sa'}_u = \text{gsa'}_u = \text{sa}_u \cap \text{gsa}_u
\]

The arcs constraint is complexity \(O(1)\), since it just performs a single set of set unifications and intersections; the nodes constraint is complexity \(O(d)\) on the max degree \(d\) in \(\text{gu}\), since removing a node can trigger up to \(d\) arc removals.

Finally, they describe the propagation for the \(\text{ArcNode}(A, N_1, N_2)\) constraint, constraining it to the nodes in the node domains and the contents of the arc domains:

\[
\text{dom}'(A) = \text{dom}(A) \cap (\text{dom}(N1) \times \text{dom}(N2))
\]

\[
\text{dom}'(N_1) = \{n_1 \in \text{dom}(N_1) | \exists n_2 \in \text{dom}(N_2), \langle n_1, n_2 \rangle \in \text{dom}(A)\}
\]

\[
\text{dom}'(N_2) = \{n_2 \in \text{dom}(N_2) | \exists n_1 \in \text{dom}(N_1), \langle n_1, n_2 \rangle \in \text{dom}(A)\}
\]

Other constraints can be built from these and from set constraints; for instance, they define the subgraph constraint \(\text{SubGraph}(G_1, G_2)\) as:

\[
(G_1, G_2) \equiv \text{Nodes}(G_1) \subseteq \text{Nodes}(G_2), \text{Arcs}(G_1) \subseteq \text{Arcs}(G_2)
\]

For linear optimization problems, they define a weight constraint \(\text{Weight}(G, w, I)\) which constrains the total weight of graph variable \(G\) to the value of the variable \(I\) according to weight function \(w\) (with its restriction to the edge or node domains being \(w_a\) or \(w_n\)):

\[
\text{Weight}(G, w, I) \equiv I = \text{Weight}(\text{Nodes}(G), w_n) + \text{Weight}(\text{Arcs}(G)), w_a
\]
They define the set of neighbors, \( SN \), with edges leading to \( N \) with 
\[
\text{InNeighbors}(G, N, SN) = SN \subseteq \text{Nodes}(G) \land (\#SN > 0 \Leftrightarrow N \in \text{Nodes}(G)) \land
\forall n \in \text{Nodes}(gU) \cap SN : n \in SN \Leftrightarrow (n, N) \in \text{Arcs}(G)
\]

Their reachable constraint, \( \text{Reachable}(G, N, SN) \), implements the reachability propagator defined by Quesada et al [33] (described in more detail in the pathfinding and reachability section below.) It relies on a \( \text{QuasiPath}(G, SN, N, n_2) \) constraint, which defines a subset of \( G, SN \), as a path from source \( N \) to sink \( n_2 \) which may have some extra mutually disjoint cycles branching off of it; it is formed by making every node in \( SN \), except the sink and source, have an in-degree and out-degree of 1. They define it as:

\[
\text{QuasiPath}(G, SN, N_1, N_2) \equiv N_1 \in SN \land N_2 \in SN \land
\forall n \in SN : O = \text{OutNeighbors}(G, n) \cap SN \land \#O \leq 1 \land (n \neq N_2) \Rightarrow \#O = 1 \land
\forall n \in SN : I = \text{InNeighbors}(G, n) \cap SN \land \#I \leq 1 \land (n \neq N_1) \Rightarrow \#I = 1
\]

\( \text{Reachables}(G, N, SN) \) can then constrain \( SN \) to be the set of nodes in graph \( G \) reachable from node \( N \) using this, as follows:

\[
\forall n \in SN : \exists SN' \subseteq \text{Nodes}(G) : \text{QuasiPath}(G, SN', N, n)
\]

They use Reachability in turn to define their directed acyclic graph constraint \( \text{DAG}(G) \), stating that graph \( G \) contains no cycles. It just has to constrain the in-neighbors and reachable set for each node to be disjoint:

\[
\text{DAG}(G) \equiv \forall n \in \text{Nodes}(G) : \text{InNeighbors}(G, n) \cap \text{Reachable}(G, n) = \emptyset
\]

Taking QuasiPath and restricting its number of edges to one more than the number of nodes removes all of its disjoint cycles and lets them define the path constraint \( \text{Path}(G, N_1, N_2) \), which declares that \( G \) is a path from node \( N_1 \) to node \( N_2 \):

\[
\text{Path}(G) \equiv \text{QuasiPath}(G, \text{Nodes}(G), N_1, N_2) \land \#\text{Nodes}(G) = \#\text{Aracs}(G) + 1
\]

They can then combine these constraints in turn to solve problems; for instance, as an example, they describe the traveling salesman problem in a graph \( g \) with weights \( w \) as a problem to minimize \( \text{Weight}(G, w) \) (using an additional \( \text{Cycle}(G) \) constraint that defines \( G \) as a closed directed path) such that:

\[
\text{SubGraph}(G, g) \land \text{Cycle}(G) \land \text{Nodes}(G) = \text{nodes}(g)
\]

They implemented a prototype version of CP(Graph) using Oz/Mozart, a constraint programming framework, and applied it to biochemical network analysis on graphs ranging from 50 to 300 nodes; time ranged from a fraction of a second to a little under three hours.
3.1.2 Solnon et al

Solnon et al [7] created a constraint-based modeling language for graph matching in which each node in one graph would be represented by a variable with a domain of either integers or sets of integers, depending on whether a problem was set up to find one-to-one matches or to match one node to many. They built their system on top of Comet, an existing constraint-based modeling language.

They define two graphs to match, $G_1 = (N_1, E_1, L_1)$ and $G_2 = (N_2, E_2, L_2)$, with $E \subseteq N \times N$ as a set of directed edges, and a function $L : N \cup E \rightarrow \mathbb{N}$ for labeling nodes and edges. They are searching for a matching between these two graphs, $f : N_1 \rightarrow N_2$, matching every node of $G_1$ to either a node in $G_2$ or to a set of nodes in $G_2$; or, in other words, they are looking for a matching $M \subseteq N_1 \times N_2$, which for a node $u$ assigns it as $\forall u \in N_1, M(u) = \{ v \in N_2 | (u, v) \in M \}$ and $\forall v \in N_2, M(v) = \{ v \in N_1 | (u, v) \in M \}$.

With nodes $u, v \in N_1 \cup N_2$ (and $U$ as a set of such nodes), bounds $ub, lb \in \mathbb{N}$, labels $L = L_1 \cup L_2$, and $D \subseteq (N_1 \cup N_2) \times (N_1 \cup N_2)$, they define their constraints as follows.

First, they define a set of constraints on the maximum and minimum number of nodes a given node can be matched to in $M$:

- $MinMatch(M, u, lb) \equiv lb \leq \#M(u)$
- $MinMatch(M, U, lb) \equiv \forall u \in U : MinMatch(M, u, lb)$
- $MaxMatch(M, u, ub) \equiv \#M(u) \leq ub$
- $MaxMatch(M, U, lb) \equiv \forall u \in U : MaxMatch(M, u, lb)$
- $CardMatch(M, u, lb, ub) \equiv MinMatch(M, u, lb) \land MaxMatch(M, u, ub)$
- $CardMatch(M, U, lb, ub) \equiv \forall u \in U : CardMatch(M, u, lb, ub)$

Another constraint is used to establish that a matching is injective, matching the nodes in $U$ to distinct nodes:

- $Injective(M, U) \equiv \forall u, v \in U, u \neq v : M(u) \cap M(v) = \emptyset$

Their third set of constraints is for pairs of nodes which must be matched to nodes connected by an edge: MatchedToSomeEdges ensures that there exists a pair of matched nodes that have an edge between them, while MatchedToAllEdges ensures that all pairs of matched nodes have edges between them, as follows:

- $MatchedToSomeEdges(M, u, v) \equiv \exists u' \in M(u), \exists v' \in M(v) : (u', v') \in E_1 \cup E_2$
- $MatchedToSomeEdges(M, D) \equiv \forall (u, v) \in D : MatchedToSomeEdges(M, u, v)$
- $MatchedToAllEdges(M, u, ub) \equiv \forall u' \in M(u), \forall v' \in M(v) : (u', v') \in E_1 \cup E_2$
- $MatchedToAllEdges(M, D) \equiv \forall (u, v) \in D : MatchedToAllEdges(M, u, v)$

Finally, their label constraints work similarly, ensuring that pairs of edges or nodes share the same label on either one of their matches, or all of their matches:
MatchSomeNodeLabels(M, u) ≡ ∃ v ∈ M(u) : L(u) = L(v)
MatchSomeEdgeLabels(M, u, v) ≡ ∃ u' ∈ M(u), ∃ v' ∈ M(v) : (u', v') ∈ E_1 ∪ E_2 ∧ L(u, v) = L(u', v')
MatchAllNodeLabels(M, u) ≡ ∀ v ∈ M(u) : L(u) = L(v)
MatchAllEdgeLabels(M, u, v) ≡ ∀ u' ∈ M(u), ∀ v' ∈ M(v) : (u', v') ∈ E_1 ∪ E_2 ∧ L(u, v) = L(u', v')

The model allows for both hard and soft constraints, with soft constraints specifying a cost for violations.

These constraints can be used for a wide variety of graph matching problems; for example, they described how to use it for graph homomorphism as \( \mathcal{GH}(M, G_1, G_2) \equiv \text{CardMatch}(M, N_1, 1, 1) \land \text{MatchedToSomeEdges}(M, E_1) \).

Their implementation of this approach was applied to graphs with 200, 600, and 1000 nodes; only about fifteen percent were solved within their ten-minute time limit at a thousand nodes.

3.1.3 Gay et al

Gay et al [16] described a system for finding subgraph isomorphisms intended for use in studying biochemical reactions. This is accomplished by turning an 'antecedent' graph into a reduced version via the application of transformation rules; the antecedent graph is graph defined by sets of vertices and edges, while the goal graph is represented by 'morphism' variables with a connection to the antecedent graph established by constraints that define viable transformations.

Starting with the antecedent graph, they reduce it using delete and merge rules they define as follows. For a graph \( G = (V, A) \):

Given \( u, v \in V \), the deletion of \( v \), \( d_v(G) \), is defined as \( (V', A') \), with \( V' = V \setminus \{v\} \) and \( A' = A \cap (V' \times V') \). Similarly, the merge of \( u \) and \( v \) into \( uv \), \( m_{u,v}(G) \), is defined as \( (V', A') \) with \( V' = V \setminus \{u, v\} \cup \{uv\} \), \( A' = \{(s_{u,v}(X), s_{u,v}(y)) | (x, y) \in A\} \), \( s_{u,v} : [u \rightarrow uv, v \rightarrow uv, x \notin \{u, v\}, x \rightarrow x] \). Or, in other words, \( d_v(G) \) is the graph \( G \) with node \( v \) deleted; and \( m_{u,v}(G) \) is the graph \( G \) with nodes \( u \) and \( v \) merged together into a new node \( uv \).

They illustrate these transformations as shown below:

![Figure 7: Source: Gay et al](image)

In the above diagram, the graph on the left is transformed into the graph on the right. The nodes d and F are deleted, and e is merged with p to produce r. (E becomes the node labeled C.) When a series of delete and merge operations can transform \( G \) to \( G' \) in this manner, they write it as \( G \rightarrow_{md} G' \).

They define the problem using two kinds of variables, morphism variables (which match the elements of \( G \) to the elements of \( G' \)) and antecedent variables
These are then bound by constraints to establish their roles. The morphism
constraints say that if an edge $a$ in $G$ connects to a node $x$, then its corresponding
$a'$ must connect to $x$. With $\pi_1$ and $\pi_2$ as projection functions to refer to the
nodes for the given edges, they define this as:

$$\forall a \in A, \text{element}(A_a, [\pi_1(a_1')...\pi_1(a_k')_1]), X_{\pi_1(a)}$$

$$\forall a \in A, \text{element}(A_a, [\pi_2(a_1')...\pi_2(a_k')_2]), X_{\pi_2(a)}$$

The minimal antecedent constraints establish how the antecedent variables
work:

$$\forall v \in V, \forall v' \in V', X_{v'} = v \Rightarrow X_v = v'$$

$$\forall v \in V, \forall v' \in V', X_v = v' \Rightarrow X_{v'} \leq v$$

$$\forall a \in A, \forall a' \in A', A_a = a' \Rightarrow A_{a'} = a$$

$$\forall a \in A, \forall a' \in A', A_a = a' \Rightarrow A_{a'} \leq a$$

The first and third constraints establish that the antecedent variables reflect
the morphism variables; if an antecedent variable says that a node or edge
corresponds to a specific edge or node in the original graph, then the morphism
variable for that original edge or node must say it is transformed into that
element in the antecedent graph. The second and fourth are similar, but use
less-than rather than equals because deletions and merges to the graph may
result in the corresponding variable being lower than its place in the original.

For comparison, they also coded a SAT solver for the same problem, imple-
menting the CLP solution in GNU Prolog and the SAT version using Glucose.
Both results proved effective, though the SAT solver performed somewhat better,
with more successful solutions and fewer timeouts.

### 3.1.4 Fromherz and Mahoney

Fromherz and Mahoney [14] applied constraint-based graph matching to image
analysis, attempting to identify stick figures in sloppy sketches by converting
them to graphs indicating the connections between elements of an image and
determining its distance from a graph representing an idealized stick figure.
Sloppy stick figures could be drawn in different positions, with limbs overlapping
other parts of the drawing or extending past the joints.

They define an ideal ‘stick figure’ using a model graph, which is divided into
defined limbs. A model is composed of limb statements, which can be tagged as
optional or connected end-to-end with a linked statement. Finally, a minimize
statement can be used to define relative lengths. This model graph is then
matched with a data graph representing a possible sloppy stick figure in order to identify its parts.

They begin with a graph rectification step on the data graph, which attempts to fix errors like lines that overshoot slightly rather than perfectly co-terminating.

In their constraint model, each limb is a pair of nodes representing its endpoint, connected by a bond; if two endpoints from different edges are linked with the linked statement, then they have a link between them. Each node in the model graph has a corresponding variable, with values from the domain of the data graph (plus the option of null, since some may be tagged as optional); finding a correspondence between the two graphs is therefore simply a matter of solving for the model-graph variables.

The constraints they use for this solution establish the rules for correspondence; the primary ones are the link support and unique interpretation constraints.

The link support constraint requires that if two model nodes \( m_1 \) and \( m_2 \) have a link or bond between them, then any two data nodes \( d_1 \) and \( d_2 \) they are matched to must also share that connection. More formally, they define this as follows: If \( m_1 \) and \( m_2 \) are bonded, then either both must be null, or both the associated nodes \( d_1 \) and \( d_2 \) must be bonded. If \( m_1 \) and \( m_2 \) are linked, then either both must be null; or \( m_1 \) is null and \( d_2 \) is unlinked; or \( m_2 \) and \( d_1 \) have no link; or \( d_1 \) and \( d_2 \) are linked.

The unique interpretation constraint says that each data node can only have one corresponding model node; that is, the matching must be one-to-one.

Additionally, the minimal total link length constraint attempts to minimize the total distance between all links used in the matching; the optimal part proportions constraint attempts to find matches closest to the proportions defined by their minimize statements, and the maximal part count attempts to avoid assigning model variables to null, minimizing the number of unmatched optional limbs.

They created two implementations; one used the clp(FD) library in SICStus Prolog, while the other used a heuristic state-space framework, with each state representing an assignment of CSP variables. Both results were roughly equally effective.

### 3.2 Pathfinding and Reachability

#### 3.2.1 Quesada et al

Quesada et al. [33] approached the problem of constrained pathfinding, in which the goal is to find a path between two nodes subject to certain constraints, such as making it mandatory for the path to visit certain nodes. They represent the problem with a graph variable \( g \) (a set variable whose domain is, similar to Dooms', all subsets of the available nodes); a source node source, chosen from \( g \); a destination node \( i \), likewise within \( g \); a set of nodes in \( g \) reachable from \( i \); and the sets of 'bridge' edges \( be \) and nodes \( cn \) appearing in all possible
paths going from source to i. More formally, for a graph set g whose nodes are a subset of N, they define their reachability constraint from a source node source within g as follows:

\begin{align*}
\text{rn}(i) &= \text{Reach}(g,i) \\
\text{Reachability}(g, \text{source}, \text{rn}, \text{cn}, \text{be}) &\equiv \forall i \in N \cdot \text{cn}(i) = \text{CutNodes}(g, \text{source}, i) \land \\
&\quad \text{be}(i) = \text{Bridges}(g, \text{source}, i)
\end{align*}

\(\text{rn}(i)\) is the set of nodes reachable from node i, \(\text{cn}(i)\) is the set of nodes on all paths from source to i, and \(\text{be}(i)\) is the set of edges on all paths from source to i. These are, as described above, defined formally using \text{Reach}, \text{Paths}, \text{CutNodes}, and \text{Bridges}:

\begin{align*}
\text{\text{rn}}(i) &\equiv \text{Reach}(g,i) \\
\text{p} \in \text{Paths}(g,i,j) &\iff p = \langle k_1, \ldots, k_h \rangle \in \text{nodes}(g)^h \land k_1 = i \land k_h = j \land \\
&\quad \forall 1 \leq f < h : \langle k_f, k_f + 1 \rangle \in \text{edges}(g) \\
\text{k} \in \text{CutNodes}(g,i,j) &\iff \forall p \in \text{Paths}(g,i,j) : k \in \text{nodes}(p) \\
\text{e} \in \text{CutNodes}(g,i,j) &\iff \forall e \in \text{Paths}(g,i,j) : e \in \text{edges}(p)
\end{align*}

They note that their pathfinding approach could be implemented with the \text{Path}(p, s, d, \text{maxlength}) constraint of Dooms’ CP(graph), but that they have not yet compared the respective performance of these methods.

They then demonstrate the applicability of their \text{Reachability} constraint by using it to solve Simple Path with mandatory nodes, that is, to find paths in a directed graph that visit every mandatory node and which visits each node at most once. More formally, they define the Simple Path with mandatory nodes problem as:

\begin{align*}
\text{SPMN}(g, \text{source}, \text{dest}, \text{mandnodes}, p) &\iff \text{p} \in \text{Paths}(g, \text{source}, \text{dest}) \land \\
&\quad \text{NoCycle}(p) \land \\
&\quad \text{mandnodes} \subseteq \text{nodes}(p)
\end{align*}

\text{NoCycle}(p)\) means that p is a path which visits each node at most once. Based on this, they can say:

\begin{align*}
\text{Reachability}(\text{p}, \text{source}, \text{rn}, \text{cn}, \text{be}) &\land \text{dest} \in \text{rn}(\text{source}) \land \text{cn}(\text{dest}) \supseteq \text{mandnodes}
\end{align*}

They propose using \text{Reachability}(\text{p}, \text{source}, \text{rn}, \text{cn}, \text{be}), \text{dest} \in \text{rn}(\text{source}), and \text{cn}(\text{dest}) \supseteq \text{mandnodes} as additional constraints on top of \text{SPMN} in order to improve performance; \text{dest} \in \text{rn}(\text{source}) says that the destination is reachable
from the source, while $cn(dest) \supseteq mandnodes$ says that the mandatory nodes are required to be on the path from the source to the destination.

In their tests, they determined that using Reachability was significantly more effective than using SPMN alone for problems with optional nodes; they found that computing bridges for be was costly, but that it could pay off in most cases, though it required also computing $cn$ to use it properly.

### 3.2.2 Sellmann et al

Sellmann et al [35] applied a similar focus on bridges to finding the shortest path in a directed, weighted, acyclic graph, using a gap-closing approach where they would begin with upper and lower bounds and steadily narrow them by finding improved solutions, repeatedly refining the best-known or incumbent solution.

This was done through a minimization or optimization constraint on the path length; starting with one destination, they repeatedly determine which edge cannot be part of the shortest path, and which edges serve as bridges that must be part of the shortest path.

They define their minimization constraint using the following definitions for variables, domains, constraints, and their objective function: $n \in \mathbb{N}, X_1, ..., X_n$ are variables with finite domains $D_1 := D(X_1), ..., D_n := D(X_n)$. Given a constraint $\zeta : D_1 \times ... \times D_n \rightarrow 0, 1$ and an objective function $Z : D_1 \times ... \times D_n \rightarrow \mathbb{Q}$, let $x_i \in D_i \forall i \leq n$.

From that, they define their minimization constraint as follows: Let $B \in \mathbb{Q}$ denote an upper bound on the objective $Z$ to be minimized. A function $\vartheta_{\zeta,Z}[B] : D_1 \times ... \times D_n \rightarrow 0, 1$ with $\vartheta_{\zeta,Z}[B](x_1,...,x_n) = 1$ iff $\zeta(x_1,...,x_n)$ and $Z(x_1,...,x_n) < B$ is called a minimization or, more generally, an optimization constraint.

...or, in other words, minimization constraints find values for variables that meet the list of constraints while producing a value for the objective function below the upper bound. This can then be used to filter out assignments that cannot improve the solution.

To apply this effectively to NP-hard problems, they produce a relaxation of their minimization constraint, which finds the lowest valid minimization of some subset of the problem.

### 3.3 Trees

Lorca et al [1] [13] applied constraint logic programming to partitioning a directed graph into anti-arborescences, directed trees with no overlapping nodes and all edges directed towards the root node.

Using these definitions: A connected component is a maximal subgraph where all nodes have a chain of edges which, ignoring direction, connect them to another node in the component.

A strongly connected component is a maximal subgraph where all nodes have a directed path to each other node in the component.

A strong articulation point is a node that would break a strongly connected component in two if removed.
A sink component is a strongly-connected component with no edges leading out of it.

They represent their problem as a graph $G = (V, E)$ of order $n$. Since they are examining anti-arborescences, each non-root node has a single successor node $i$ represented by a variable $v_i$ with the domain $[1; n]$. The tree-height of each node is represented as a variable $r_i$ of domain $[0; n - 1]$, and is used to prevent circuits; and finally, each node has a boolean variable $b_i$, representing whether it is a root. Therefore, for a partition with $L$ trees, they define their problem as:

$$v_i = j \land i \neq j \Rightarrow r_i > r_j, \forall \in [1; n]$$

$$b_i \Leftrightarrow v_i = i \forall \in [1; n]$$

$$L = \sum_{i=1}^{n} i^2$$

To determine feasibility, they constrain $L$ to a value between the number of sink components and the number of potential roots. For a number of sink components $\ell$ and a number of potential roots $\ell'$, $L = [\ell, \ell']$; additionally, each sink component must contain at least one potential root.

These limits are enforced by propagators to filter out unusable values. The bound filtering propagator removes values from the domain of $L$ that go beyond the feasibility limits of $[\ell, \ell']$, narrowing in on the correct number of trees to partition the graph into. The bound filtering propagator also removes options that become infeasible when the value of $L$ is instantiated to its limits of $\ell$ or $\ell'$. Specifically, when the number of trees, $L$, is equal to the number of sink components, $\ell$, then any root not in a sink component is infeasible because the only way to put $L$ sink components into $L$ trees is to use a root inside of each sink component; picking a root outside of a sink component would leave one inaccessible. Similarly, when the number of trees is equal to the number of candidate roots, then we must use all of those roots; since they will all definitely be roots and therefore cannot have any edges to other other nodes, all of their non-loop outgoing edges must be removed from consideration.

The structural filtering propagator removes edges that cannot be part of any tree. It is defined using door nodes, which have an edge leading to a different strongly-connected component, and winner nodes, which are either a potential root or a door. From this they define three rules: First, if a sink component contains only one potential root, then all outgoing non-loop edges from that root must be removed. Second, if a strongly connected component has no potential root and just one door, then all edges leading from that door into the component must be removed. Finally, for each strong articulation point $p$, any outgoing edges from $p$ that leads to a subgraph whose only paths to a winner go through $p$ must be removed; this is because including such an edge would deny that subgraph any path to a winner, turning it to a subgraph with no potential roots.
3.4 Graph Labeling

Smith [37] applied constraint programming to determining if graphs have a graceful labeling and finding one if it exists. A graceful labeling $f$ is one where each node has a unique integer label $0, 1, ..., q$, and each edge $(x, y)$ can be uniquely labeled with $|f(x) - f(y)|$, the absolute difference between its two node's labels; this means that the label edges are the set $1, ..., q$.

Smith produced two CP models for this; the first is a straightforward representation of the problem through variables and constraints (with some additional constraints for symmetry breaking), while the second, which she calls the edge-label model, serves as a refinement on this.

In her first model, each node is given a variable for its label, $x_1, x_2, ..., x_n$ with a domain $\{0, 1, ..., q\}$; and each edge is likewise given a variable for its label, $d_1, d_2, ..., d_q$, with a domain $1, ..., q$. The edge labels are constrained to be the absolute value of the difference between their node labels, with an edge $k$ between nodes $i$ and $j$ constrained to $d_k = |x_i - x_j|$. Additionally, the edge labels and node labels are each constrained to be all different.

Smith identifies two potential sources of symmetry that need to be broken to improve this approach. First, the graph itself may be symmetric (such as a path, where the order of the node labels can be reversed); and second, each node variable $x_i$ with a value $v$ can be replaced with its complement $q - v$. The first symmetry she eliminates by imposing an ordering on the variables representing symmetric parts of the graph. The complement symmetry can be broken by taking any node that is not symmetrically equivalent to another node and constraining its value to $\leq q/2$, provided $q$ is odd.

Her second model assigns variables to the edge labels; each edge label $i$ has a variable $e_i$, equal to the smaller node label on the nodes that edge connects. Since an edge is labeled with $|f(x) - f(y)|$, if $e_i = j$, then we know that the edge with label $i$ connects the nodes labeled $j$ and $j + i$. This allows us to give $e_i$ a domain of $0, 1, ..., q - i$.

Node labels have similar variables, $l_j$, with a value representing the node assigned that label, or $n + 1$ for labels that are not used.

These variables are connected with the constraints $e_i = j$ iff $l_j$ and $l_{j+1}$'s labels correspond to adjacent nodes ($1 \leq iq; 0 \leq j \leq q - i$); and, of course, the node variables must be all different, aside from the $n + 1$ value for unused labels.

Complement symmetry can be broken by taking the edge-label $e_{q-1}$, whose domain is $0, 1$ and arbitrarily reducing its domain to $1$. Graph symmetry can be broken by mapping the node label variables to the node variables $x_1, ..., x_n$ from the first model and then applying the constraints described there.

Smith implemented this in ILOG Solver 6.0, finding that the edge-label method was a far more efficient algorithm for determining if a graph has a graceful labeling and finding one if it existed.
4 Other Approaches

4.1 SAT

The foundation of modern SAT checking is the DPLL (Davis-Putnam-Logemann-Loveland) algorithm. It operates using what is now called the splitting rule: For a formula in conjunctive normal form with clauses joined by $\land$ containing a given variable $\lambda$, we can split the process of solving it into two subproblems, one in which the positive-polarity $\lambda$ is removed from all clauses that contained it, and one in which $\neg \lambda$ is removed from all clauses that contain it. The formula is solvable if and only if one of these two subproblems is solvable. Or, in other words, we can split some $[(N_1 \lor X) \land (N_2 \lor X), (N_3 \lor \neg X) \land (N_4 \lor \neg X)]$ into two branches, $[((N_1) \land (N_2))]$ and $[((N_3) \land (N_4))]$. Naturally, situations where we can use this to simplify the problem without branching are ideal; in our example, this would be if our variable appeared only as $X$ or $\neg X$. Applying this logic is called the pure literal rule: If a literal appears only with one polarity, all clauses containing it can be made true by assigning it the necessary value, and therefore all clauses containing it can immediately be removed. Another way of simplifying the process is known as the unit clause rule or unit propagation; essentially, we know that for any clause containing just one literal to be true, that literal’s value must be true (or, conversely, for any clause containing just the negation of one literal to be true, that literal’s value must be false), and, therefore, it’s possible to immediately assign the necessary values to such literals on encountering them, reducing the amount of necessary backtracking [9] [8] [10].

These basic approaches have been improved on in a number of ways. The unit clause rule led to the an assignment-ordering rule known as the shortest clause rule, which selected variables from clauses with the fewest unassigned literals. Other approaches to improve on it were based on eliminating smaller clauses earlier, on making inferences early in the search, and eliminating areas of the search space as soon as possible [4].

Conflict-Driven Clause Learning (CDCL) SAT solvers have proven highly effective at implementing many of these improvements and have therefore seen considerable practical use. As they are based on DPLL, CDCL solvers use a similar backtracking search, but they have refined it in several ways, including adding new clauses based on conflicts they encounter, exploiting the structure of conflicts, lazy data structures, and random restarts. Their structure includes an antecedent for each variable; when a variable is assigned an value through implication via the unit clause rule, this antecedent points to the clause whose assignment caused the given implication. This in turn produces an implication graph which can be analyzed to determine the cause of conflicts. Randomly restarting the search takes advantage of this approach; since DPLL uses a depth-first search, it is easy to get stuck deep in an assignment that may take exponential amounts of time to complete. Restarting at random allows some of the benefits of a broader search while retaining the knowledge gained from analyzing the source of conflicts in previous searches. Lazy data structures reduce the overhead of assigning variables using a method comparable to the
way CLP’s AC6 tracks only a single support; with normal assignments, each clause must track the status of each variable that appears in it (that is, each adjacent variable) in order to determine its state, which requires a large number of operations for each variable assignment as these adjacent clauses are checked. Since we only need to know when a clause becomes unit or unsatisfied, we can save time by having each clause track the status of two variables, rather than all of them; whenever one of these two is assigned a value of 0, we move the tracking reference to another variable. This means that we will sometimes not recognize immediately when a clause is satisfied (hence the ‘lazy’), due to both tracking references pointing to unassigned variables when another in the clause has been assigned a value of ‘1’; but that does not affect the correctness of the algorithm. [4]

4.1.1 SAT on Graphs

There have been several efforts to apply SAT to graph analysis in a manner similar to the CLP approaches described above, including some that directly compare it to a CLP method. In addition to their aforementioned CLP approach, Gay et al [16] applied an SAT model to their subgraph epimorphism problems; to review, their goal is, for a pair of graphs \((G, G')\), to determine the deletions and merges necessary to transform \(G\) into \(G'\).

They expressed their subgraph epimorphism problem as a partial surjective function on the vertex sets, \(V\) and \(V'\); that is, one where every vertex in the target graph is paired with exactly one vertex in the source graph. They define these partial surjective problems formally as follows. For a binary relation \(m \subseteq E \times E'\):

\[
\forall x \in E, x_1' \in E', x_2' \in E', ((x, x_1') \in m \land (x, x_2') \in m) \Rightarrow x_1' = x_2'
\]

\[
\forall x' \in E', \exists x \in E, (x, x') \in m
\]

This can be used to find a pairing that describes a transformation from the source graph to the target graph. Notice that the function is partial because deletions are possible; nodes in the source graph that are not paired with a node in the target graph are ones which this transformation deletes.

They tested their approaches on some System Biology models from the BioModels repository. [30] To do so, they implemented their CLP approach in GNU Prolog and their SAT approach in Glucose. With a timeout of 20 minutes, they found that the set of solutions they would find for most types of problems using CLP was a subset of what they would find with SAT, and that both would often find the solution within 10 seconds.

4.2 ASP

Answer Set Programming is a form of declarative programming that relies on what is called answer set or stable model semantics [18].
The goal in Answer Set Programming is to find the answer set or stable model for some variables and constraints - the set of minimal assignments to those variables that match the given constraints. A possible satisfying assignment is called a Herbrand model, and a minimal Herbrand model is one such that none of its subsets are valid Herbrand models. [19]

This model adds default statements, which are treated as true unless their negation is true; and negations, which remove such statements from the stable model (that is, the answer set.) This focus on removing statements from the stable model allows answer set programming to employ strong negation and disjunction in a Prolog-like environment [20].

Answer Set problems are described by Gelfond as \( \{ \sigma, \Pi \} \), with \( \sigma \) being a sorted classical logic signature and \( \Pi \) being a set of logic programming rules for \( \sigma \). The rules can contain negation and or; the goal is therefore to find a minimal answer set \( S \) which satisfies the constraints of \( \Pi \) [17].

5 Applications

Dooms et al [12] applied their CLP(g) to biochemical network analysis, exploring graphs of genes, molecules, reactions, and controls; this deals with using known reactions in a cell to map out additional ones, finding them as paths in a directed graph. This can be modeled as a constrained search for shortest paths.

Solnon et al [7] applied their constraint-based graph matching technique to pattern recognition in images; they approached it by converting points of interest in an image into a graph, and searching the resulting graph for subgraph isomorphisms.

This is conceptually similar to the sloppy stick figure analysis used by Fromherz and Mahoney [14], although they produced an ‘ideal’ graph of the image they were looking for and used this to search for specific patterns in images (stick figures, in this case) rather than looking for patterns more generally.

Mamoulis et al [26] applied constraint satisfaction to analyzing XML; they converted XML to graphs, then searched it using constraint satisfaction. Generating large graphs at random, which they termed rooted node-labeled graphs, they used intermediate nodes to represent the set of possible labels, and leaf nodes to store text.

Monreal et al [28] applied soft constraint logic programming to the problem of optimizing electric vehicle travel; in soft CLP, each constraint is given a cost for violations, and if a solution that satisfies all constraints cannot be found, solutions that violate the progressively lower-ranked constraints are used instead. Their graphs, representing the network over which vehicles traveled, are both directed and double-weighted, with weights for both time and energy consumption. Their variables are sets of paths, with each being a sequence of nodes, plus a total time and energy cost for that path; their domain is the set of all possible paths, while their constraints establish the road network and sets of appointments the vehicles must keep, as well as defining the optimizations necessary for the best path.
Cattafi et al [5] used CLP(FD) to determine optimal valve placement in a water distribution network. In this problem, valves are disposable maintenance tools attached to a network of pipes; each valve has a cost, so the goal is to place as few as possible, but it is also important to be able to disable specific areas for maintenance while disabling as few other areas as possible. Therefore, this is an optimization problem where they seek to minimize the number of valves and the maximum possible disruption. The variables are a list of booleans for each possible valve location, indicating whether a valve is present or not; from here, they modeled the test of a particular solution as a two-player game, with one player trying to maximize disruption and the other trying to minimize it.

The first player attempts to assign an optimal valve placement; the second player attempts to maximize the disruption caused by broken pipes; and then the first player tries to close valves to handle this by closing valves in a way that minimizes disruption. Constraints determine the pipe graph and impose the minimization and maximization constraints for each player. They implemented their algorithm in ECLiPSe, testing graphs with five to thirteen valves, and found that computing time grows sub-exponentially with the number of valves.

References


