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 presentation will focus on providing an overview of constraint logic programming and efforts to apply it to graphs and graph problems, as well as an overview of some alternative logic programming approaches to graph problems.
Goal: Find the solution to a problem, subject to a set of constraints.

Problems are divided up into:

- Variables, which must be assigned values to reach a solution.
- Domains for the variables, which their values are taken from,
- Constraints on the values that the variables can be assigned.
A constraint satisfaction problem is defined by sets <X, D, C>:

- A set of variables, {X₁…Xₙ}
- A set of corresponding domains for each variable, {D₁…Dₙ}
- A set of constraints C, each of which applies to a subset of the variables and implies a relation between their domains.

A solution therefore assigns values to all variables from their domains, without violating any of the constraints.
- Variables $A, B, C$
- Domains of $[1, 2, 3]$ for each.
- Constraints:
  - $A \neq B$
  - $A \neq C$
  - $B > C$
CONSTRAINT NETWORKS

- Constraints are represented by an undirected graph
  - Nodes for each variable.
  - Edges representing the constraints between them.
- The constraint network for the \([A, B, C]\) example:
We can show the range of possible assignments in a directed graph for our search space.

- Nodes are possible sets of assignments
- Edges show assignments that can be reached by assigning values to unassigned variables.
Notice that the search space for even a tiny problem can become very large very quickly. Additionally, notice that starting with an assignment of \{C=3\}, on the far right, can never lead to a solution.

Possible satisfying assignments:

- A=3, C=1, B=2
- A=2, C=1, B=3
- A=1, C=2, B=3
VARIABLE AND DOMAIN TYPES

- Integer variables: Values from a finite domain of integer values.
- Real variables: Values from a range of real numbers.
- Both allow for constraints using mathematical comparisons (eg. less than, greater than, or more complicated algebraic expressions.)
- Set variables: Domains based on sets (eg. ‘all subsets of this set’ as possible values)
  - Constraints can take the form of things like subset and superset relations.
Unary constraints apply to a single variable.
- Can be applied in a single pass.

Constraints that apply to two or more variables are more complex.
- We apply these constraints and reduce the domains of a variable in the filtering step,
- Domain reduction then affects the domains of every variable linked to that one via constraints.
BACKTRACKING

- Repeatedly try different combinations.
- Backtrack on hitting a point where it's no longer possible to make an assignment.
- But this 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.
We resolve this with the concept of *local consistency*.

- Keep every variable consistent with its neighbors in the constraint network.
- This allows constraints to *propagate* through the entire network until the whole thing is consistent.

Mackworth, 1977
- Reduce the domains of adjacent variables to remove values that won’t be part of a valid solution.
- Types of *local consistency* this enforces:
  - *Node consistency:* Every variable is consistent with its unary constraints.
  - *Arc consistency:* All variables connected by multi-variable constraints (‘arcs’) are consistent with those constraints.
  - *Path consistency:* For a given pair of nodes, there exists assignments for them which satisfy their constraints while also satisfying their constraints with any third variable.

Mackworth, 1977
For nodes $i$ and $j$, unary predicates $P_x$ and binary predicates $P_{x,y}$:

- 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 $(i_0, i_1, \ldots, i_m)$ 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 $P_{i_0i_m}(x, y)$ all hold, there is a sequence of values $z_1 \in D_i, \ldots, z_{m-1} \in D_{i_{m-1}}$ such that, first, $P_{i_1}(z_1)$ and ... and $P_{i_{m-1}}(z_{m-1})$ holds; and second, $P_{i_0i_1}(x, z_1)$ and $P_{i_1i_2}(z_1, z_2)$ and ... and $P_{i_{m-1}i_m}(z_{m-1}, y)$ holds.

Mackworth, 1977
ARC CONSISTENCY: AC-1

- AC-1: 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 a domain was reduced, iterate over all of them again.

Mackworth, 1977
Iterating over all constraints every time wastes a great deal of effort re-checking binary constraints whose variables were not affected by the removal.

AC-2 fixes this by only re-examining binary constraints that include the variable which had its domain reduced, rather than all of them.

This requires a queue to track the nodes we want to examine.

When examining a node $i$, we note the nodes in the constraint network one step away from it, and enqueue them all if the domain of $i$ is revised.

Mackworth, 1977
- AC-2 can sometimes have a variable in the queue multiple times at once.
- To fix this, AC-3 retains a queue of arcs rather than nodes. When a variable from an arc that has already been examined has its domain reduced, that arc is re-inserted into the queue.
- AC-3 has a worst-case time of $O(ea^3)$; its queues give it a space complexity of $O(ea^2)$.

Mackworth, 1977
AC-4 works by tracking the support a value has.

Potential values are removed from a domain when they have no way of satisfying a particular constraint.

For each constraint, the values in linked variables that allow a given value to satisfy that constraint are called supports for it on that constraint.

The number of supports a value has are tracked, and every time a value is removed, all the other values it supported have their supports decremented by one.

When a value has no support on a given constraint, that value is removed from its domain.
With constraints $A < B$ and $B < C$, and domains of $[1, 2, 3]$.

Unsupported nodes are in red.

Mohr and Henderson, 1986
After removing unsupported values once.

Mohr and Henderson, 1986
After removing unsupported values again.

Mohr and Henderson, 1986
AC-5 refines AC-3 by having each entry in the queue consist of not just 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.

Hentenryck et al, 1992
AC-6 functions similarly to AC-4, but it tracks only a single support for each constraint rather than all of them.

Values are only removed when they run out of constraints; therefore, it can just track one support per value, and search for another only if that one is removed.

If no other supports are found, the value is removed from its domain.

Later refined into AC-7, which exploits the bi-directionality of constraints.

Bessiere, 1994
Sometimes, you have two variables with values in their domains that satisfy all binary constraints between them, but which fail to satisfy a ‘path’ between them through the constraint network.

Establishing consistency for paths of length 2 is sufficient to establish larger path consistency.
- Iterate over the entire set of length-2 paths, repeatedly removing invalid combinations until it is path-consistent.
- Comparable to AC-1.

Montanari, 1974
PATH CONSISTENCY: PC-2

- Maintain a queue of length-2 paths that must be examined, and re-enqueue any paths that were potentially invalidated by a removal if they are not already in the queue.
- Comparable to AC-3.

Montanari, 1974
- **Equivalence class**: Multiple solutions that represent the same basic concept with a trivial difference (for example, a different ordering.)
- This can cause us to repeatedly find trivial permutations of the same answer.
- We only want one solution from each equivalence class.
- Preventing such symmetric solutions from coming up again and again is called *symmetry breaking*. 
CLP ON GRAPHS

- Graph Matching & Graph Isomorphism
- Pathfinding and Reachability
- Trees
- Graph Labeling
GRAPH MATCHING & GRAPH ISOMORPHISM

- Finding similarities between graphs.
- Retrieving subgraphs that match a specific pattern.
Dooms et al wrote CP(Graph), a general-purpose computation domain for graphs focused on finding subgraphs using constraints.

Broad applications, but primarily intended for biochemical network analysis.

Graph domain variables use finite sets, with graphs represented by sets of nodes, arcs, and arcnodes (connections between nodes and arcs.)

Weights are represented by constraints rather than variables.

Designed for directed graphs, but can also support undirected graphs.

Dooms et al, 2004
Graphs are defined as a set of nodes, SN, and a set of edges, SA.

- The edges are defined from pairs of node to node within SN:
  - $SA \subseteq SN \times SN$

Graphs have a partial ordering based on graph inclusion:

- A graph is a subset of another graph if its nodes and edges are both subsets of that graph’s nodes and edges.
- 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 $sa_1 \subseteq sa_2$.
Graphs have a domain bounded by the greatest lower bound and least upper bound within this ordering.

- Least upper bound: All nodes and edges that could possibly be part of the graph.
- Greatest lower bound: All nodes and edges that must be part of the graph.
CP(GRAPH): KERNEL CONSTRAINTS

- ‘kernel’ constraints: Define the structure of a graph by constraining the sets that define it.
  - Can be composed with finite domain and finite set constraints to make more complex constraints.
- Arcs(G, SA): Defines SA as the set of edges for graph G.
- Nodes(G, SN): Defines SN as the set of nodes for G.
- ArcNode(A, N₁, N₂): Defines A as an edge from nodes N₁ to N₂.

Dooms et al, 2004
The kernel constraints can be used to construct other constraints.

For instance, the subgraph constraint:

“A graph’s nodes and arcs are subsets of the nodes and arcs of another graph.”

Subgraph$(G_1, G_2)$: $G_1$ is a subset of $G_2$:

\[ \text{Subgraph}(G_1, G_2) \equiv \text{Nodes}(G_1) \subseteq \text{Nodes}(G_2), \text{Arcs}(G_1) \subseteq \text{Arcs}(G_2) \]
A constraint-based modeling language for graph matching.

The goal is to match nodes in one graph to nodes in a second graph.

Nodes in the first graph are assigned integer variables indicating which graph-two node they’re paired with.

- Alternatively, it can be set up to allow nodes in the first graph to pair up with groups of nodes in the second.
- In this case, the domains are sets of integers.

Solnon et al, 2009
Graphs are sets of nodes, edges, and labels:
- $G_1 = (N_1, E_1, L_1)$ and $G_2 = (N_2, E_2, L_2)$

Edges are defined according to the nodes they connect:
- $E \subseteq N \times N$

Labels assign integers to nodes and edges:
- $L: N \cup \rightarrow \mathbb{N}$

The goal is to find a labeling that matches every node of $G_1$ to a node or set of nodes in $G_2$. 

Solnon et al, 2009
Their first set of constraints \((\text{MinMatch}, \text{MaxMatch})\) define the maximum and minimum number of nodes a given node (or set of nodes) can be matched to in a matching.

They use an \textit{Injective} constraint to ensure that nodes in a given set are matched to distinct nodes in the other graph.

Their third set of constraints is for pairs of nodes which must be matched to nodes connected by an edge; \text{MatchedToSomeEdges} ensures that there exists a pair of matched nodes that have an edge between them, while \text{Matched-ToAllEdges} ensures that all pairs of matched nodes have edges between them.
Gay et al described a system for finding subgraph isomorphisms intended for use in studying biochemical reactions. It turns an 'antecedent' graph into a reduced version via the application of transformation rules. The goal graph is represented by *morphism* variables.

- One variable per edge or node in the goal graph.
- Domain: Sets of edges or nodes, accordingly.
- These represent the edges or nodes from the antecedent graph combined into it.

The antecedent graph is a graph to be transformed.

- One variable per edge or node in the antecedent graph.
- Domain are edges or nodes in the goal graph respectively.
- Values here indicate which goal-graph object corresponds to this one.
- Not one-to-one; multiple antecedent nodes / edges can be paired to one in the goal graph.
• Constraints between the antecedent and goal graphs define the valid transformations.

• Nodes can be deleted or merged according to the rules of biochemical reactions.
  • For instance, some nodes can only be merged with nodes of the same type.
  • Some nodes can be deleted, and others can’t.

• In the above graph, \( d \) and \( F \) are deleted, and \( c \) is merged with \( p \) to produce \( r \). (\( E \) becomes the node labeled \( C \).)
Fromherz and Mahoney applied constraint-based graph matching to image analysis.

Their algorithm analyzed images of stick figures and converted it into graphs representing the lines that formed the figure and the connections between them.
The sloppy stick figure is parsed into a data graph to a model graph representing an ideal stick figure, composed of limb statements (some of which are optional) and combined with linked statements.

The variables are the nodes in the model graph, with values from the domain of the data graph.

Model variables marked as optional can be assigned to null, indicating that that part isn’t in the data graph.

Limbs consist of a pair of nodes linked by a bond.

A node can only be bonded to one other node; if two limbs are connected, they have endpoints close together (or on top of each other) connected by linked statement.

Fromherz and Mahoney, 2001
- **Link support**: If two model nodes have a link or bond between them, then any two data nodes they are matched to must share that connection.
- **Unique interpretation**: Matching must be one-to-one.
- **Minimal total link length**: Minimizes the total distance between all links used in the matching.
- **Optimal part proportions**: Constrains matches closest to the proportions defined by their minimize statements.
- **Maximal part count**: Minimizes number of optional model variables assigned to null, and therefore the number of unmatched optional limbs.

Fromherz and Mahoney, 2001
Quesada et al 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.

Work based on Dooms.
The graph variable $g$ is based on Dooms'; domains are all subsets of a given set of nodes.

They also define variables for:

- A source node and a destination node in $g$.
- A set of nodes in $g$ reachable from the source node.
- A set of bridge edges and cut nodes appearing in all possible paths from the source to the destination.
All paths from 1 to 9 go through 5.

Therefore, node 5 must be part of the graph, 5 must be reachable from 1, and 9 must be reachable from 5.
SELLMANN ET AL:
FINDING THE SHORTEST PATH VIA GAP-CLOSING

- Applied a similar focus on bridges to finding the shortest path.
- Directed, weighted, acyclic graph
- Uses a gap-closing approach: Begin with upper and lower bounds and narrow them by finding improved solutions.
- This refines the best-known or incumbent solution.
- Works through a minimization / optimization constraint on the path length
- Starting with one destination, repeatedly determine which edges cannot be part of the shortest path, and which edges serve as bridges that must be part of the shortest path.

Sellmann et al, 2003
Lorca et al 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.
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.

A door node is a node with an edge leading out of a strongly-connected component.
Each node has an associated variable for its successor; since these are *anti-arboresences* (trees with edges pointing to the root), each non-root node has one successor, and a root has a null successor.
Lorca et al. (2011) present a method for determining root nodes in a graph. The number of roots $L$ is crucial in this analysis:

- **Lower bound** equal to the number of sink components.
- **Upper bound** equal to the number of potential roots.
- Note that this also represents the number of trees needed to divide the graph.

Additionally:
- Each sink component must always contain a potential root.
- When $L$’s domain is empty, it indicates an unsolvable assignment.
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 there they define three rules:

- If a sink component contains only one potential root, then all outgoing non-loop edges from that root must be removed. Roots cannot have outgoing edges.

- 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; in other words, we must go ‘through’ the door’.

- Finally, for each strong articulation point, any outgoing edges from that point that lead to a subgraph whose only paths to a winner go through that point 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.
Smith applied constraint programming to determining if graphs have a graceful labeling and finding one if it exists.

A graceful labeling is one where each node has a unique integer label from 0 to $q$, and where each edge can be uniquely labeled with the absolute difference between its two node labels, such that the label edges are the set 1 to $q$.

Smith produced two CP models for this, a traditional one and one based on the edge-label model to refine it.
Each node has an associated variable for its label, with a domain from 0 to \(q\).

Each edge is likewise given a variable for its label, with a domain 1 to \(q\).

The edge labels are constrained to be the absolute value of the difference between their associated node labels.

Two potential sources of symmetry that need to be broken:

- The graph itself may be symmetric (such as a path, where the order of the node labels can be reversed.)
  - Broken by imposing an ordering on the variables representing symmetric parts of the graph.

- Each node variable can have its label \(v\) replaced with its complement, \(|v - q|\).
  - Broken by taking any node that is not symmetrically equivalent to another node and constraining its value to less than the 'halfway' point from 0 to \(q\).
In this model, variables are assigned to the edge labels \{1…q\}

These variables have values equal to the smaller of the two nodes that that label connects.

For example, if we assign edge label 1 to node 3, we know that it connects nodes 3 and 3 + 1 = 4, because an edge is labeled with the absolute value of the nodes it connects.

Nodes labels have similar variables, with the value representing the node assigned to that label, or n + 1 for labels that are not used.

Smith, 2006
Complement symmetry can be broken by examining the edge label, \( q-I \); it must have a value of either 0 or 1, so we arbitrarily reduce it to 1.

Graph symmetry can be broken by mapping the node label variables to the node variables from the first model and applying the constraints described there.
OTHER APPROACHES
The foundation of modern SAT checking is the DPLL (Davis-Putnam-Logemann-Loveland) algorithm.

This operates using what is now called the splitting rule:

- For a formula in conjunctive normal form with clauses joined by \(^\lor\) 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 only if one of these two subproblems is solvable.
- **Pure literal rule:** If a variable appears only as $X$ or $\neg X$, all clauses containing it can be made true by assigning it the necessary value, and therefore all clauses containing it can immediately be removed.

- **Unit clause rule / Unit propagation:**
  - For any clause containing just one literal to be true, that literal's value must be true.
  - Conversely, for any clause containing just the negation of one literal to be true, that literal's value must be false.
  - We can immediately assign the necessary values to such literals on encountering them.
- *Shortest clause rule*: Selects variables from clauses with the fewest unassigned literals.
- Other approaches:
  - Eliminating smaller clauses earlier
  - Making inferences early in the search
  - Eliminating areas of the search space as soon as possible
Conflict-Driven Clause Learning (CDCL) SAT solvers have proven highly effective at implementing many of these improvements and have therefore seen considerable practical use.

They are based on DPLL, but refine it in several ways, including:

- Adding new clauses based on conflicts they encounter,
- Exploiting the structure of conflicts,
- Lazy data structures, and
- Random restarts.
Gay applied an SAT model to their subgraph epimorphism problems.

They expressed their subgraph epimorphism problem as a partial surjective function; that is, one where every result in the target graph is paired with exactly one result in the source graph.

In their tests, 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.

Gay et al., 2013
Answer Set Programming is a form of declarative programming that relies on what is called answer set or stable model semantics.

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.

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.
APPLICATIONS FOR CLP GRAPH ANALYSIS

- Dooms et al: Biochemical network analysis, graphs of genes, molecules, reactions, and controls
  - Using known reactions in a cell to map out additional ones, finding them as paths in a directed graph.
  - Can be modeled as a constrained search for shortest paths.
- Solnon et al: Pattern recognition in images
  - Convert points of interest in an image into a graph.
  - Search the resulting graph for subgraph isomorphisms.
- Fromherz and Mahoney: Sloppy stick figure analysis, another form of image pattern-recognition.
  - Produced an 'ideal' graph of the image they were looking for.
  - Used this to search for specific patterns in images (stick figures, in this case).
Mamoulis et al 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.
APPLICATIONS:
OPTIMIZING ELECTRIC VEHICLE TRAVEL

- Monreal et al 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.
- Graph represents the network over which vehicles travel
  - Directed and double-weighted, with weights for both time and energy consumption.
- Variables: Sets of paths, with each being a sequence of nodes, plus a total time and energy cost for that path
- Domain: The set of all possible paths.
- Constraints enforce the road network and sets of appointments the vehicles must keep, as well as defining the optimizations necessary for the best path.
Catta et al used CLP(FD) to determine optimal valve placement in water distribution network.

Valves: Disposable maintenance tools attached to a network of pipes
- Each valve has a cost; the goal is to minimize this
- It is also important to be able to disable specific areas for maintenance while disabling as few other areas as possible.

This is an optimization problem where they seek to minimize the number of valves and the maximum possible disruption.
Applications: Optimal Valve Placement (Cont.)

- Variables: A list of booleans for each possible valve location, indicating whether a valve is present or not;
- Solutions are tested as a two-player game, with one player trying to maximize disruption and the other trying to minimize it.
  - Player 1 attempts to assign an optimal valve placement
  - Player 2 attempts to maximize the disruption caused by broken pipes
  - Player 1 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 at each step.
- 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.
BIBLIOGRAPHY


