Proof Logging for Some Interesting Propagation Algorithms

Speaker: Matthew McIlree, University of Glasgow
Venue: WHOOPS, Copenhagen, 23rd May 2024

(These are brief notes in lieu of slides as the animated presentation I used in the talk is difficult to export to pdf)

Introduction

Assuming you are happy with... :-)

- Proof logging being a useful thing
- VeriPB proof rules (particularly RUP)
- Reifying PB constraints
- Basic Ideas of Constraint Programming (Search, Propagation)
- Encoding CP Variables for proofs (Binary and Direct Encoding)
- The general idea for CP proof logging (RUP on backtrack, propagators log justifications)

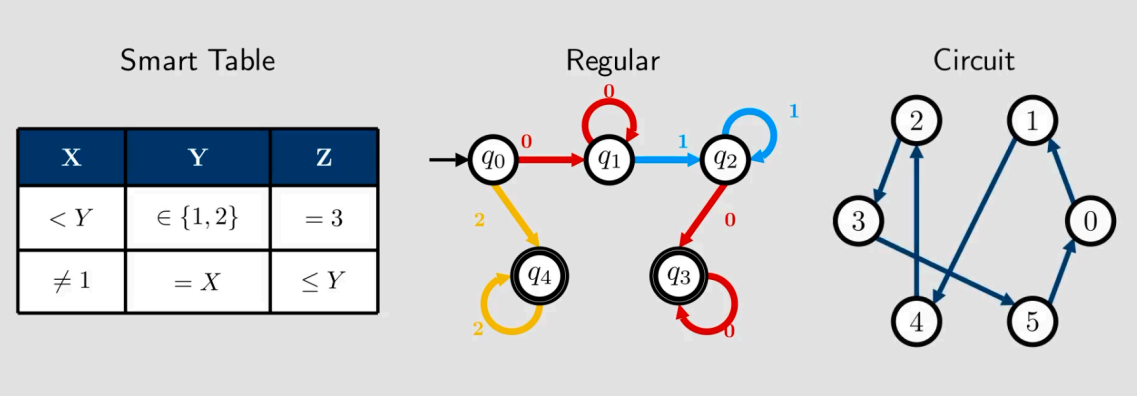
The Main Challenge

- There are *many many* propagation algorithms in CP (>400)
- Each might need their own specialised justification procedure.

```
atleast_nvector
atmost
atmost1
atmost_nvalue
atmost_nvector
balance
balance_cycle
balance_interval
balance_modulo
balance_partition
balance_path
```

A first step

- Focus on constraints that can be used to encode other constraints (Smart Table, Regular)
- Or just ones that have non-trivial propagation algorithms (Circuit)
- Prove people who say "you can't proof log `<x constraint>`" wrong.



The Smart Table Constraint

"Dumb" Table Constraints

- Fundamental to CP
- Just list the allowed combinations of values, e.g.

$$(X, Y, Z) \in \{(1, 2, 3), (3, 3, 2), (1, 2, 2)\}$$

- Pros: Allows for efficient propagation
- Cons: May require exponentially many tuples (AllDifferent)

What are Smart Tables?

- Happy medium between listing all solutions and having a more compact representation
- Allow the entries of the table to be simple binary or unary constraints, so we have a disjunction of conjunctions of constraints (like DNF)
- For example the SmartTable represented above says that either we have
  - $X < Y$, AND $Y \in \{1, 2\}$ AND $Z = 3$
    OR we have
  - $X \neq 1$ AND $Y = X$ AND $Z < Y$

Technical requirement for efficient propagation

- The entries in a tuple can overlap, but with a critical restriction:
- **There must not be a cyclic dependency among the variables within a single tuple.**
- For example: $X < Y \quad Y < Z \quad Z < X$ would not be allowed.

Encoding Smart Table Constraints to PB

- Encode each atomic constraint and reify with a PB flag

$$e_{X<Y} \iff -x_{b0} - 2x_{b1} - 4x_{b2} + y_{b0} + 2y_{b1} + 4y_{b2} \geq 1$$

$$e_{Y\in\{1,2\}} \iff \dots$$

$$e_{Z=3} \iff \dots$$

$$e_{X\neq 1} \iff \dots$$

$$e_{Y=X} \iff \dots$$

$$e_{Z\leq Y} \iff \dots$$

- Encode tuples as conjunctions of these flags and reify with another flag (selector)
- Finally, have a disjunction over the selector

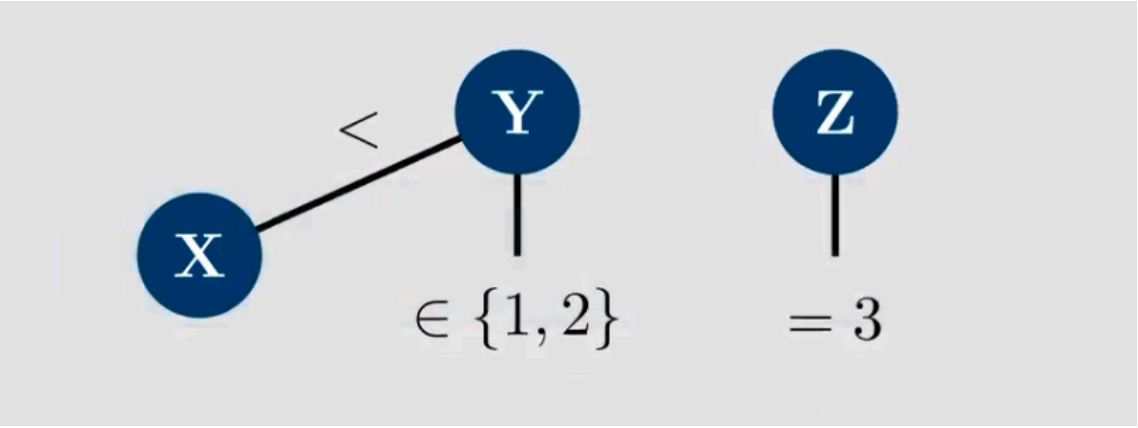
$$s_1 \iff e_{X<Y} + e_{Y\in\{1,2\}} + e_{Z=3} \geq 3$$

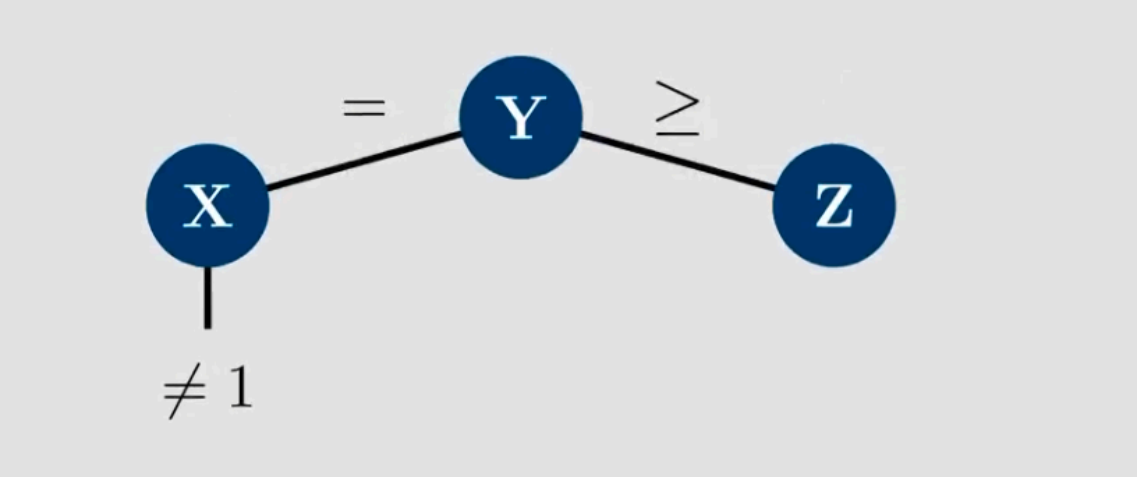
$$s_2 \iff e_{X\neq 1} + e_{Y=X} + e_{Z\leq Y} \geq 3$$

$$s_1 + s_2 \geq 1$$

Propagation for Smart Table Constraints

- Propagation of smart table constraints works similarly to table constraints:
  1. For a given point in the search tree, collect all the values supported by each tuple.
  2. Find unsupported values for the variables and remove them.
- The propagation process treats each tuple as a small sub-CSP (constraint satisfaction problem).
- Since the tuple is acyclic, we can achieve *global consistency*—removing all unsupported values across the constraints.
- This is done using a two-pass filtering over the constraint graphrs represented as trees





Logging the filtering process

- Simply RUP each inference, but conditionally on the tuple selector

$$\text{RUP} \quad s_1 \implies \bar{y}_{=1};$$
$$\text{RUP} \quad s_1 \implies \bar{y}_{=3};$$
$$\text{RUP} \quad s_1 \implies x_{=1};$$
$$\text{RUP} \quad s_1 \implies z_{=3};$$
$$\text{RUP} \quad s_2 \implies \bar{x}_{=1};$$
$$\text{RUP} \quad s_2 \implies \bar{y}_{=1};$$

- This ensures we can eventually RUP any unsupported values.
- Relies on an implicit theorem about the nature of PB proofs.

ⓘ Theorem: Proofs under restrictions

Let F be a formula, ρ be a partial assignment and suppose that from $F \restriction_\rho$ we can derive a constraint D using a cutting planes and RUP derivation of length L. Then we can construct a derivation of length $O(n \cdot L)$ from F of the constraint

$$\bigwedge_{\ell \in \rho} \ell \implies D$$

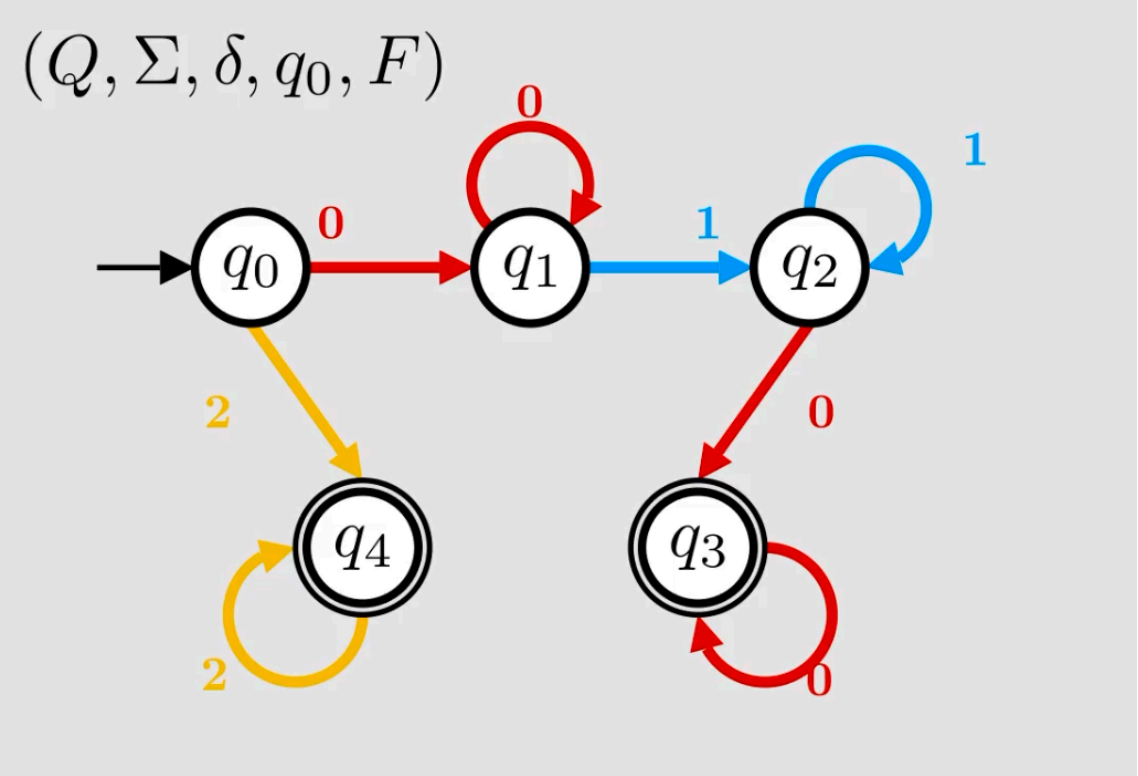
The Regular Language Membership Constraint

What are Regular Language Constraints?

- Takes the idea of listing all solutions even further
- For a sequence of variables, the sequence of values they take must belong to a regular language.
- We specify the regular language by explicitly providing the DFA, namely:

- A set of states
- A set of transitions
- A starting state
- Final (accepting) states
- (The alphabet is implicitly assumed to be the union of domains of variables.)

Encoding Regular Constraints to PB



- Define extension variable $s_{i=j}$ which means "the state after processing i variables in the regular expression is j"
- So $s_{0=0}$ (initial state) and either $s_{5=3}$ or $s_{5=4}$ (final states)
- We then encode the transitions using the values of the variables and the transition function

$$s_i := \text{The state after processing } i \text{ variables}$$

$$s_{0=0} \geq 1$$

$$s_{5=3} + s_{5=4} \geq 1$$

$$\text{For each } X_i, j \in \text{dom}(X_i), \text{ and } q \in Q:$$

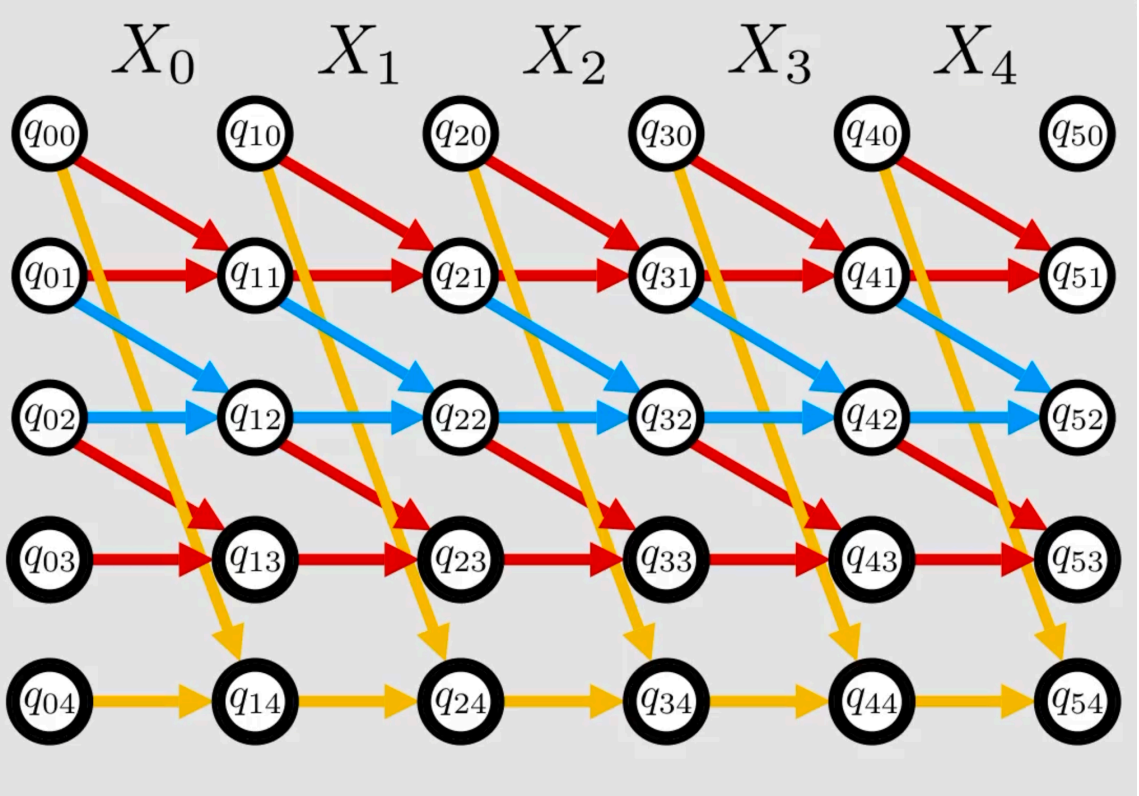
$$x_{i=j} \wedge s_{i=q} \implies s_{i+1=\delta(q,j)}$$

- We also need to disallow non-existent transitions

Propagating and Justifying Regular

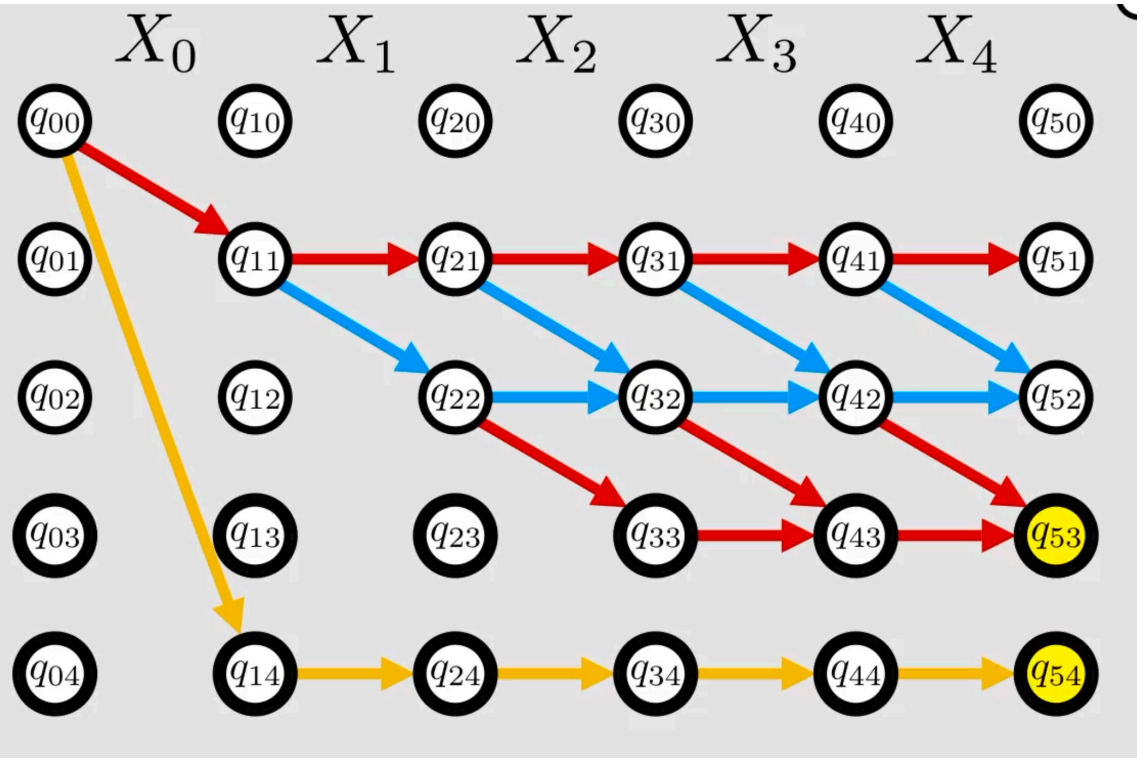
- Regular constraints are propagated efficiently by leveraging a layered directed multi-graph.

- Each layer corresponds to processing i variables, each node in a layer corresponds to a possible state after processing i variables
- Edges between layers are possible transitions at that point i.e. possible values each variable can take

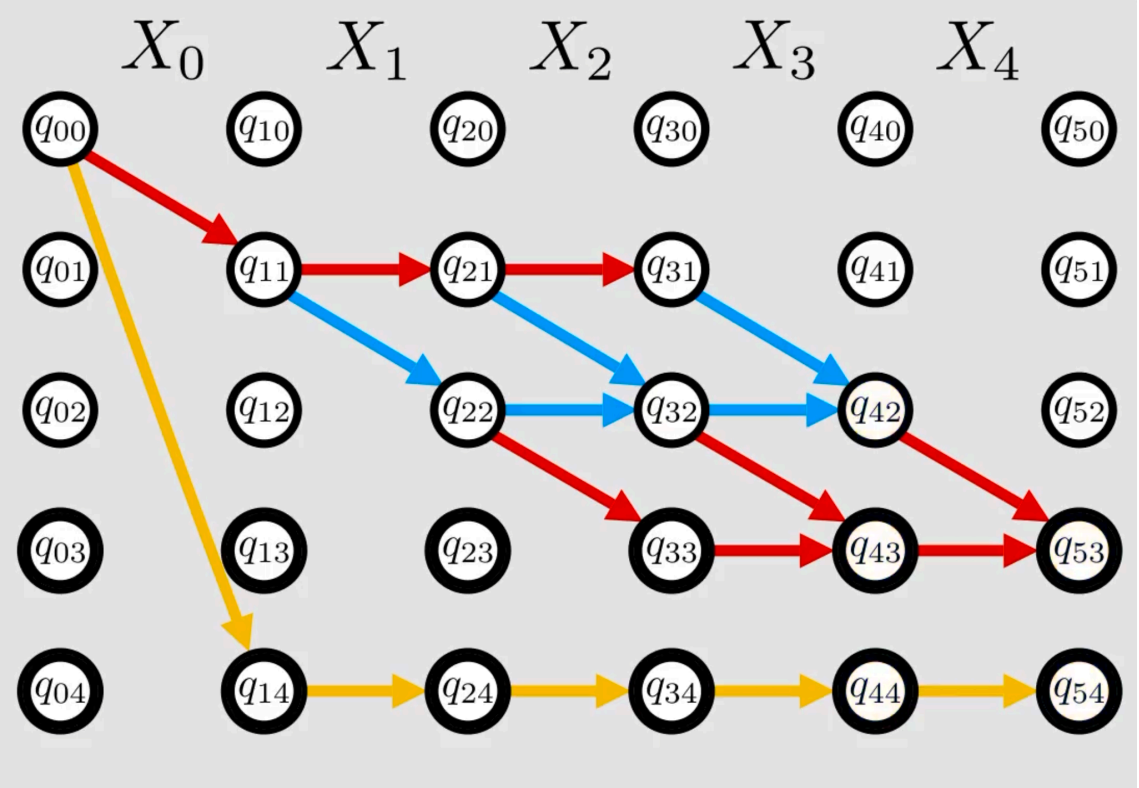


- A possible solution is therefore a path in the graph starting at the start state and ending in one of the final states.
- Filtering of inconsistent edges can be done in a forwards, backwards pass.

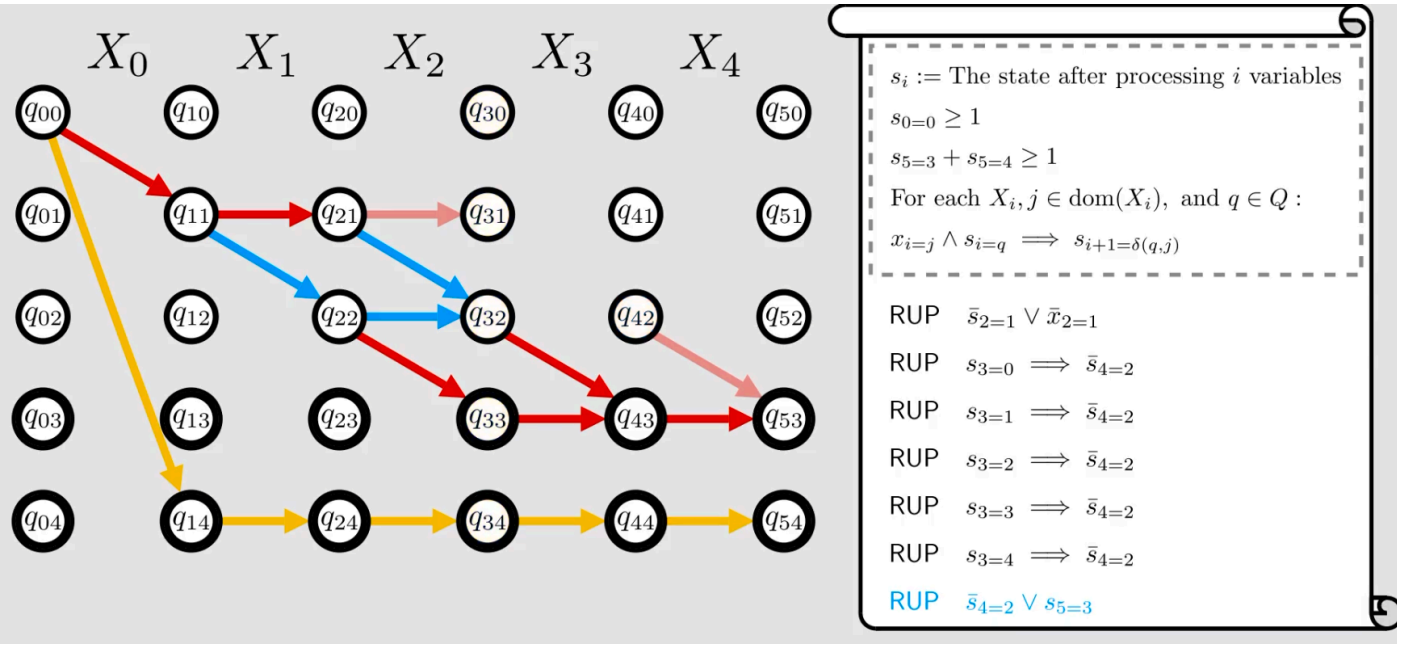
Forwards:



Backwards:



- Consistency can then be maintained on variable domain update, by filtering outward from the edge removal.
- To log this in the proof we log each edge deletion in the form $\bar{s}_{ij} \vee \bar{s}_{jk}$ - these are all RUP



The Circuit Constraint

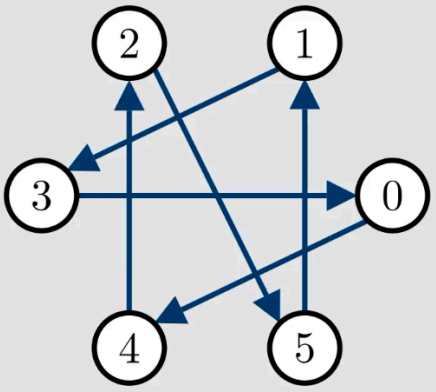
What is the Circuit Constraints

- Hamiltonian Circuits in a graph - not logical circuits
- Uses a successor representation to constraint variables and make their values correspond to vertices in a directed graph
- $X_i = j$ means there is an edge (i, j) in the graph

- Constrains that any assignment to these variables must represent a Hamiltonian Circuit

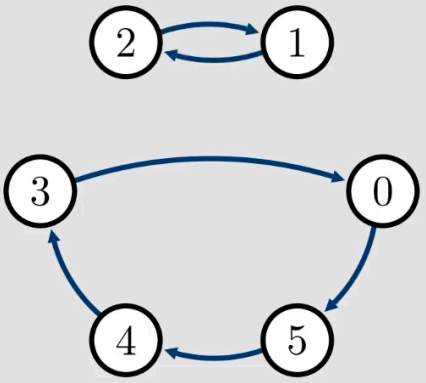
The Circuit constraint

$$X_0 = 4$$
$$X_1 = 3$$
$$X_2 = 5$$
$$X_3 = 0$$
$$X_4 = 2$$
$$X_5 = 1$$



- Essentially it is a compound constraint: AllDifferent, and NoSubcycles
- All Different ensures we have a permutation, and then we have to disallow subtours

$\text{AllDiff}(X_0, X_1, X_2, X_3, X_4, X_5)$

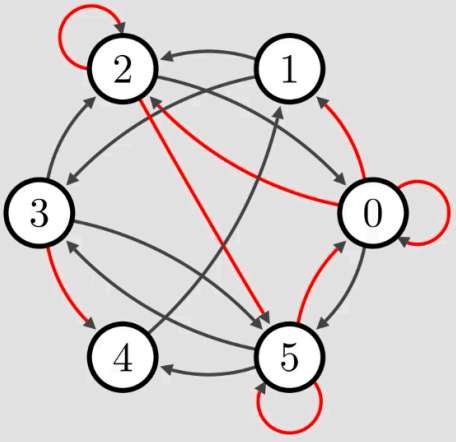


Propagating Circuit

- Full domain consistency (maximum pruning) is NP-Hard

Consistency for Circuit:

$$X_0 \in \{0, 1, 2, 5\}$$
$$X_1 \in \{2, 3\}$$
$$X_2 \in \{0, 2, 5\}$$
$$X_3 \in \{2, 4, 5\}$$
$$X_4 \in \{1\}$$
$$X_5 \in \{0, 3, 4, 5\}$$



- Since it would allow us to decide whether a Hamiltonian circuit exists in polynomial time
- So instead we use ad-hoc pruning rules

Circuit PB Encoding

- Associate an auxiliary variable P_i with each vertex i.
- Constrain that $P_i = j$ if and only if vertex i appears exactly j positions after vertex 0 in the tour.

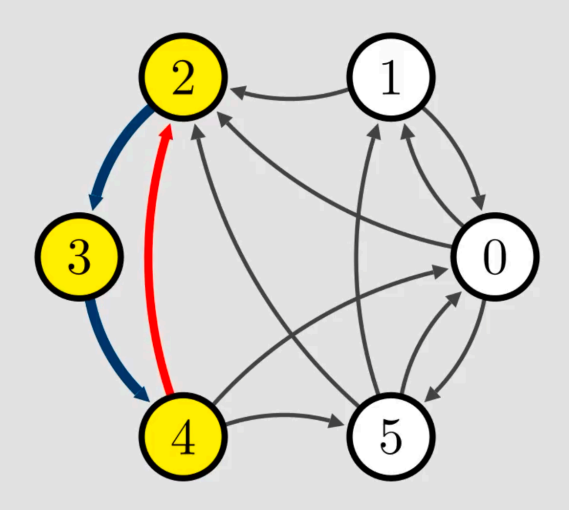
$$P_i := \text{Position of vertex } i \text{ relative to } 0$$

$$\text{For each } X_i, j \in dom(X_i)\ j \neq 0:$$

$$x_{i=j} \implies P_j = P_i + 1$$

Logging a Simple Lookahead Propagator

- Find chains of assigned variables $X_{t_1} = t_2, X_{t_2} = t_3, \ldots, X_{t_{n-1}} = t_n$
- Ensure $X_{t_n} \neq t_1$ unless we have seen all the variables, since this would complete a small cycle



- To prove this, simply add up constraints from the encoding!

From encoding:

$$x_{2=3} \implies P_3 = P_2 + 1$$

$$x_{3=4} \implies P_4 = P_3 + 1$$

$$x_{4=2} \implies P_2 = P_4 + 1$$

Cutting planes addition:

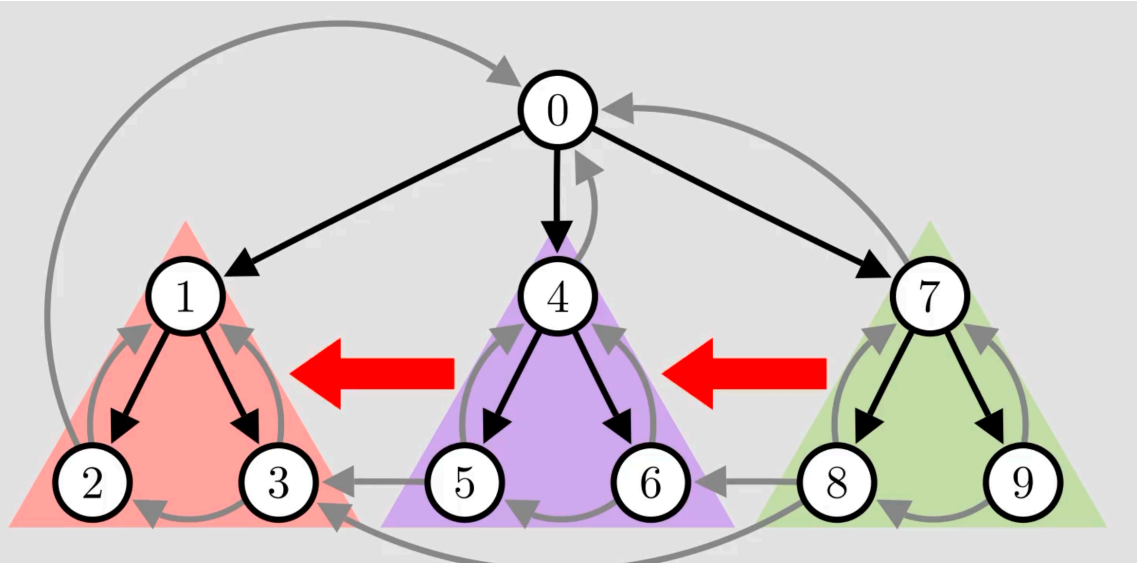
$$x_{2=3} \wedge x_{3=4} \wedge x_{4=2} \implies P_3 - P_2 + P_4 - P_3 + P_2 - P_4 = 1 + 1 + 1$$

- Which is the same as

$$x_{2=3} \wedge x_{3=4} \implies \overline{x_{4=2}}$$

Logging the Strongly Connected Components Propagator

- Basic observation: If AllDifferent is enforced then
  - No Subcycles $\iff$
  - All vertices part of one cycle $\iff$
  - Every vertex reachable from every other vertex $\iff$
  - There is only one strong connected to component (SCC)
- We can identify the SCCs in linear time (Tarjan's algorithm), so if we find more than one, contradict.
- The idea for proof logging is to build up sets of possible P variables starting from a vertex P_v that can take each value until we have more values than variables.
- Won't go in to detail, but call this procedure `ReachTooSmall(v)`
- Further propagation rules are based on examining the DFS spanning tree created by Tarjan and reasoning about which edges cannot possibly exist



- See CPAIOR paper for full details

Conclusion

- Lots of complex reasoning is easy to capture with VeriPB
- Proofs under implications / assumptions are quite powerful.
- Not restricted by the kind of consistency enforced by CP propagators
- Can confirm the power of proof logging as a debugging tool.

Future Work

- Many more propagators to do :-D
- Regular -> Cost Regular, MDD
- Circuit -> Subcircuit, Path
- Also other kinds of consistency: can chat about bounds-consistent multiplication.