Certified Constraint Programming

Matthew McIlree

2nd WHOOPS/EuroProofNet Workshop on Automated Reasoning and Proof Logging,

13th September 2025

Constraint Programming (CP)

Constraint Programming (CP)

Variables

Constraint Programming (CP)

Variables

Domains

Constraint Programming (CP)

Variables	Domains	Constraints

Constraint Programming (CP)

Variables	Domains	Constraints
X		
Y		
Z		
W		

Constraint Programming (CP)

Variables	Domains	Constraints
X	$dom(X)$	
Y	$dom(Y)$	
Z	$dom(Z)$	
W	$dom(W)$	

Constraint Programming (CP)

Variables	Domains	Constraints
X	$dom(X)$	$C(X, Y)$
Y	$dom(Y)$	$B(Z, W)$
Z	$dom(Z)$	$D(X, W, Z)$
W	$dom(W)$	$E(X, Y, Z, W)$

Constraint Programming (CP)

Variables	Domains	Constraints
X	$\{0,1\}$	$\neg X \vee Y$
Y	$\{0,1\}$	$Z \vee W$
Z	$\{0,1\}$	$X \vee \neg W \vee Z$
W	$\{0,1\}$	$\neg X \vee \neg Y \vee Z \vee W$

Constraint Programming (CP)

Variables	Domains	Constraints
X	$\mathbb{R}$	$X + 3Y \geq 1$
Y	$\mathbb{R}$	$Z - W \leq 0$
Z	$\mathbb{R}$	$X + W + Z = 2$
W	$\mathbb{Z}$	$2X + 5Y$
		$-Z + 3W > 4$

Constraint Programming (CP)

Variables	Domains	Constraints
X	$[1..5]$	$X \neq 3Y$
Y	$[-3..7]$	$Z \times W = 5$
Z	$[2..6]$	AllDifferent(X, W, Z)
W	$[-2..6]$	$2X + 5Y$
		$-Z + 3W > 4$

Constraint Programming (CP)

Variables	Domains	Constraints
X	$[1..5]$	$X \neq 3Y$
Y	$[-3..7]$	$Z \times W = 5$
Z	$[2..6]$	AllDifferent(X, W, Z)
W	$[-2..6]$	$2X + 5Y$
		$-Z + 3W > 4$

Objective Variable or Function $\max Z$

Inference

Inference

Search

Inference

$$X = Y + 2$$

Search

Inference

Search

$$X = Y + 2$$

$$Y \in \{2, 4, 5, 7\}$$

Inference

$$X = Y + 2$$

$$Y \in \{2, 4, 5, 7\}$$

Domain Consistency
= "Poking Holes"

Search

Inference

$$X = Y + 2$$

$$Y \in \{2, 4, 5, 7\}$$

Domain Consistency
= "Poking Holes"

$$X \in \{3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

Search

Inference

$$X = Y + 2$$

$$Y \in \{2, 4, 5, 7\}$$

Domain Consistency
= "Poking Holes"

$$X \in \{3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

Search

Inference

$X = Y + 2$

$Y \in \{2, 4, 5, 7\}$

Domain Consistency
= "Poking Holes"

$X \in \{3, 4, 5, 6, 7, 8, 9, 10, 11\}$

Bounds Consistency
= "Narrowing Min/Max"

Search

Inference

$$X = Y + 2$$

$$Y \in \{2, 4, 5, 7\}$$

Domain Consistency
= "Poking Holes"

$$X \in \{3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

Bounds Consistency
= "Narrowing Min/Max"

$$X \in \{3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

Search

Inference

$$X = Y + 2$$

$$Y \in \{2, 4, 5, 7\}$$

Domain Consistency
= "Poking Holes"

$$X \in \{3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

Bounds Consistency
= "Narrowing Min/Max"

$$X \in \{3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

Search

='Enforcing consistency'/
'Propagating Constraints'

Search

$$X = Y + 2$$

$$Y \in \{2, 4, 5, 7\}$$

Domain Consistency
= "Poking Holes"

$$X \in \{3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

Bounds Consistency
= "Narrowing Min/Max"

$$X \in \{3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

='Enforcing consistency'/
'Propagating Constraints'

$$X = Y + 2$$

$$Y \in \{2, 4, 5, 7\}$$

Domain Consistency
= "Poking Holes"

$$X \in \{3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

Bounds Consistency
= "Narrowing Min/Max"

$$X \in \{3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

Search

Backtracking Search

='Enforcing consistency'/
'Propagating Constraints'

$$X = Y + 2$$

$$Y \in \{2, 4, 5, 7\}$$

Domain Consistency
= "Poking Holes"

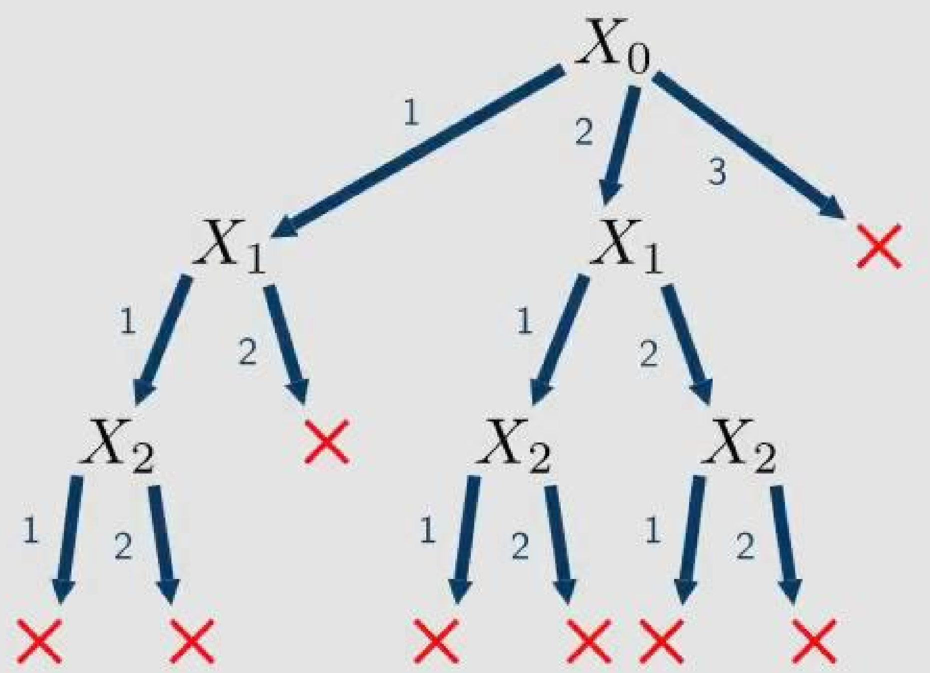
$$X \in \{3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

Bounds Consistency
= "Narrowing Min/Max"

$$X \in \{3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

Search

Backtracking Search



=‘Enforcing consistency’/
‘Propagating Constraints’

$$X = Y + 2$$

$$Y \in \{2, 4, 5, 7\}$$

Domain Consistency
= "Poking Holes"

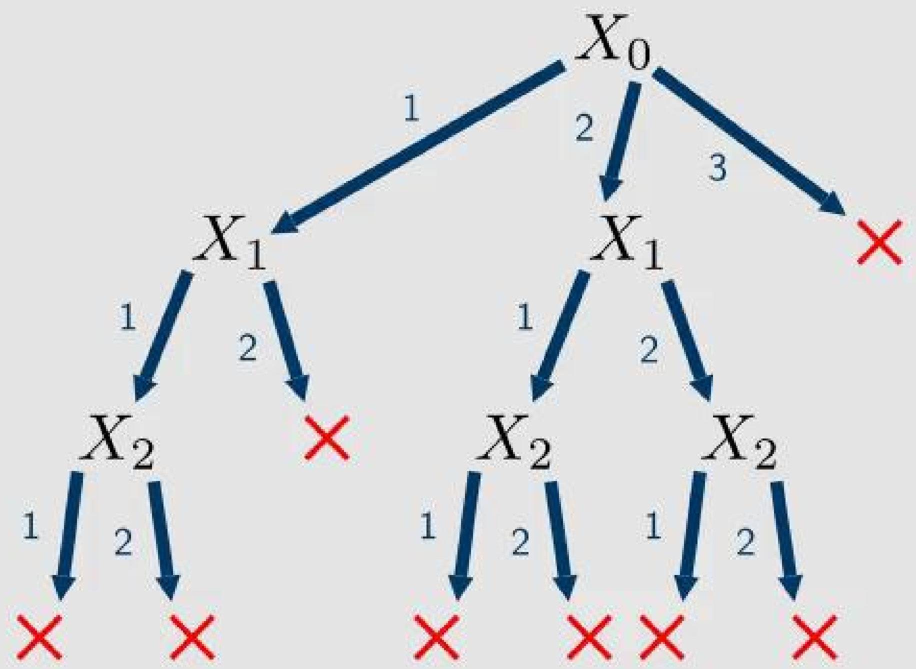
$$X \in \{3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

Bounds Consistency
= "Narrowing Min/Max"

$$X \in \{3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

Search

Backtracking Search



(Conflict-Driven Search)

='Enforcing consistency'/
'Propagating Constraints'

$$X = Y + 2$$

$$Y \in \{2, 4, 5, 7\}$$

Domain Consistency
= "Poking Holes"

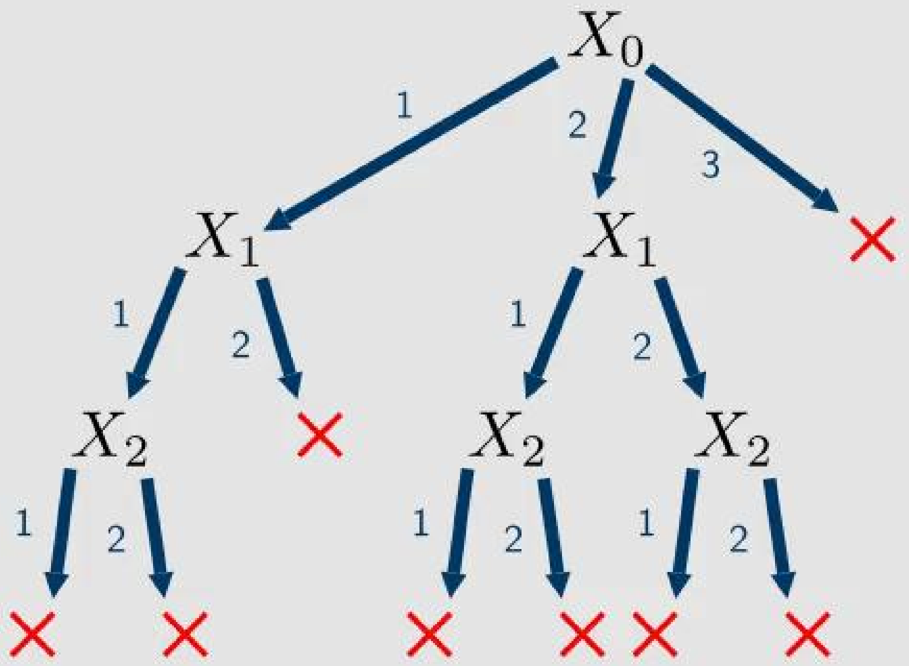
$$X \in \{3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

Bounds Consistency
= "Narrowing Min/Max"

$$X \in \{3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

Search

Backtracking Search



(Conflict-Driven Search)

(Local Search)

='Enforcing consistency'/
'Propagating Constraints'

$$X = Y + 2$$

$$Y \in \{2, 4, 5, 7\}$$

Domain Consistency
= "Poking Holes"

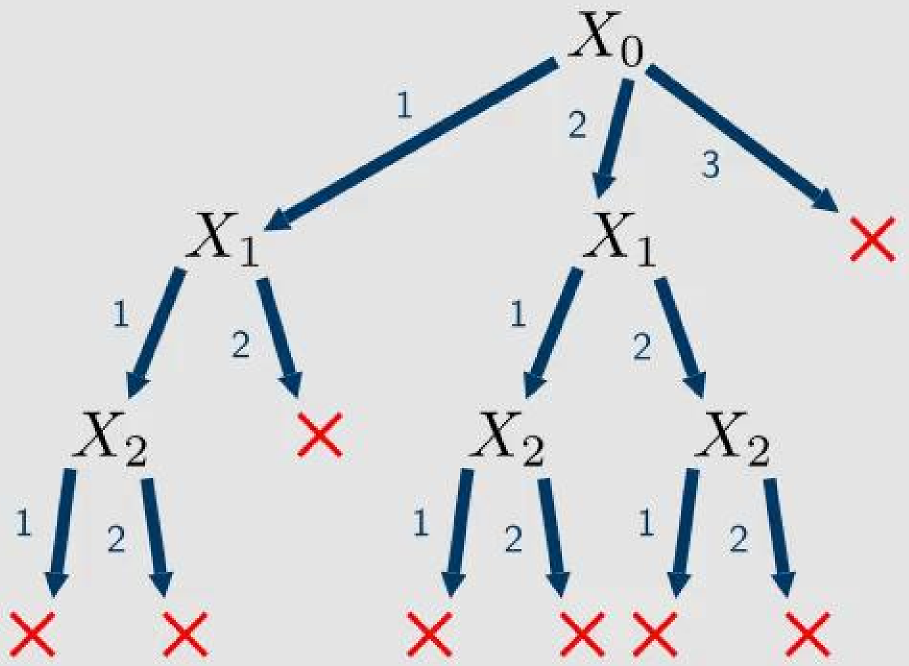
$$X \in \{3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

Bounds Consistency
= "Narrowing Min/Max"

$$X \in \{3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

Search

Backtracking Search



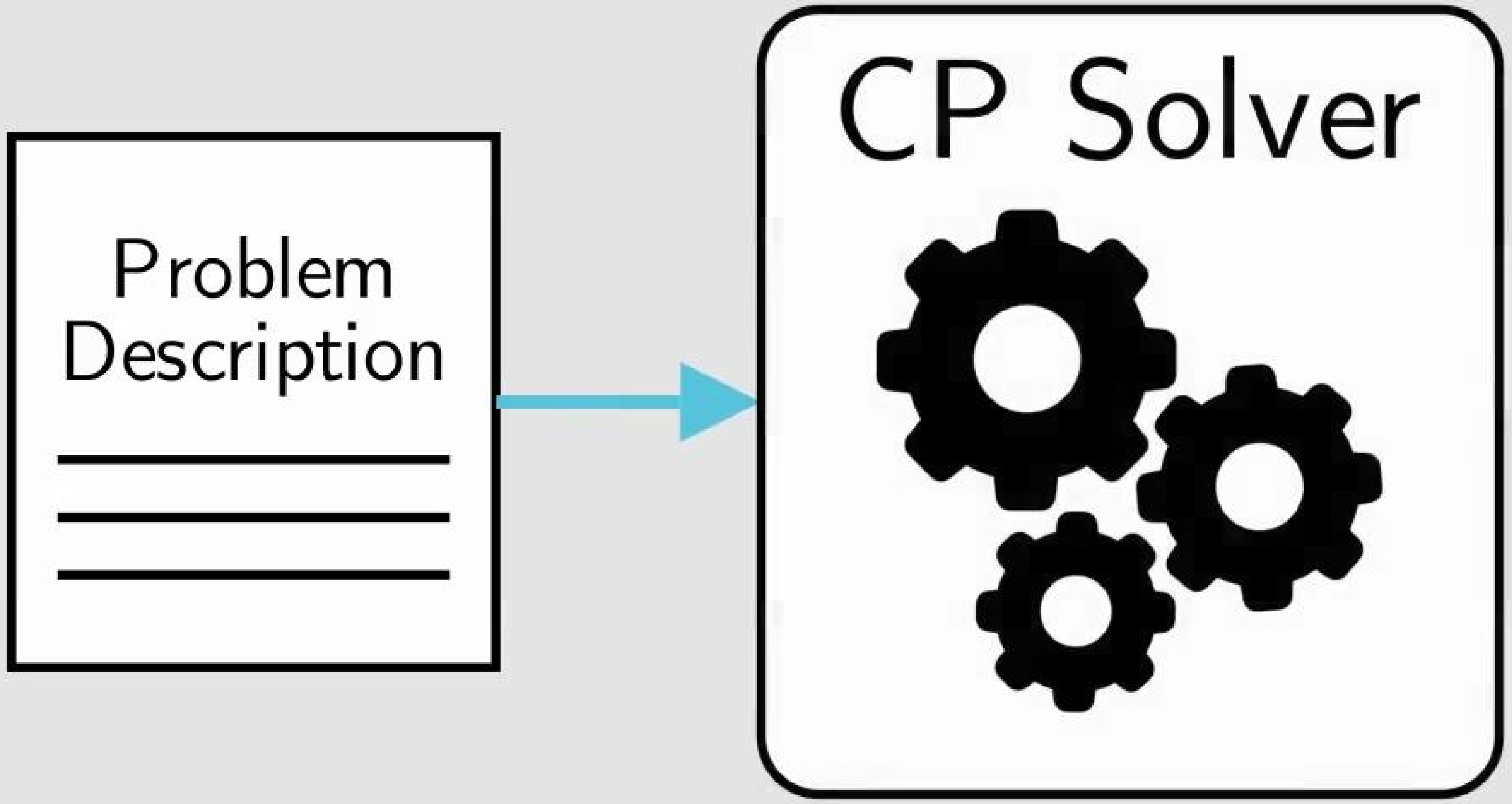
(Conflict-Driven Search)

(Local Search)

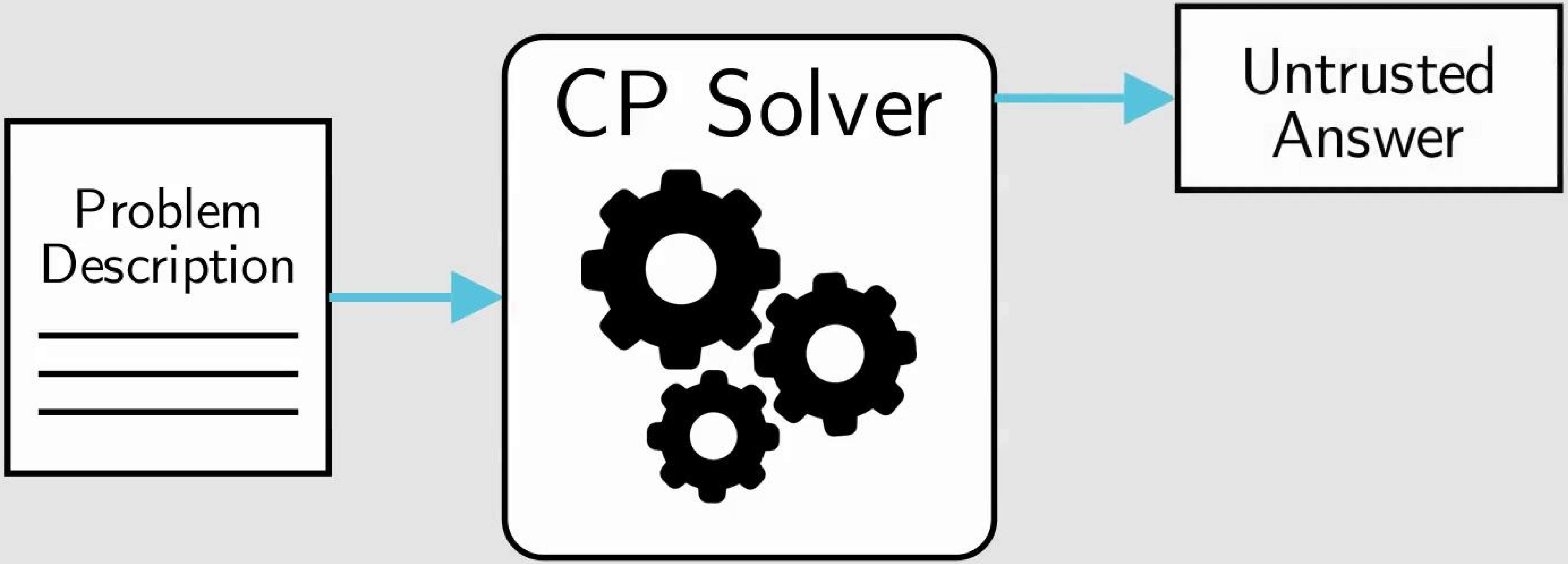
What we want:

What we want:

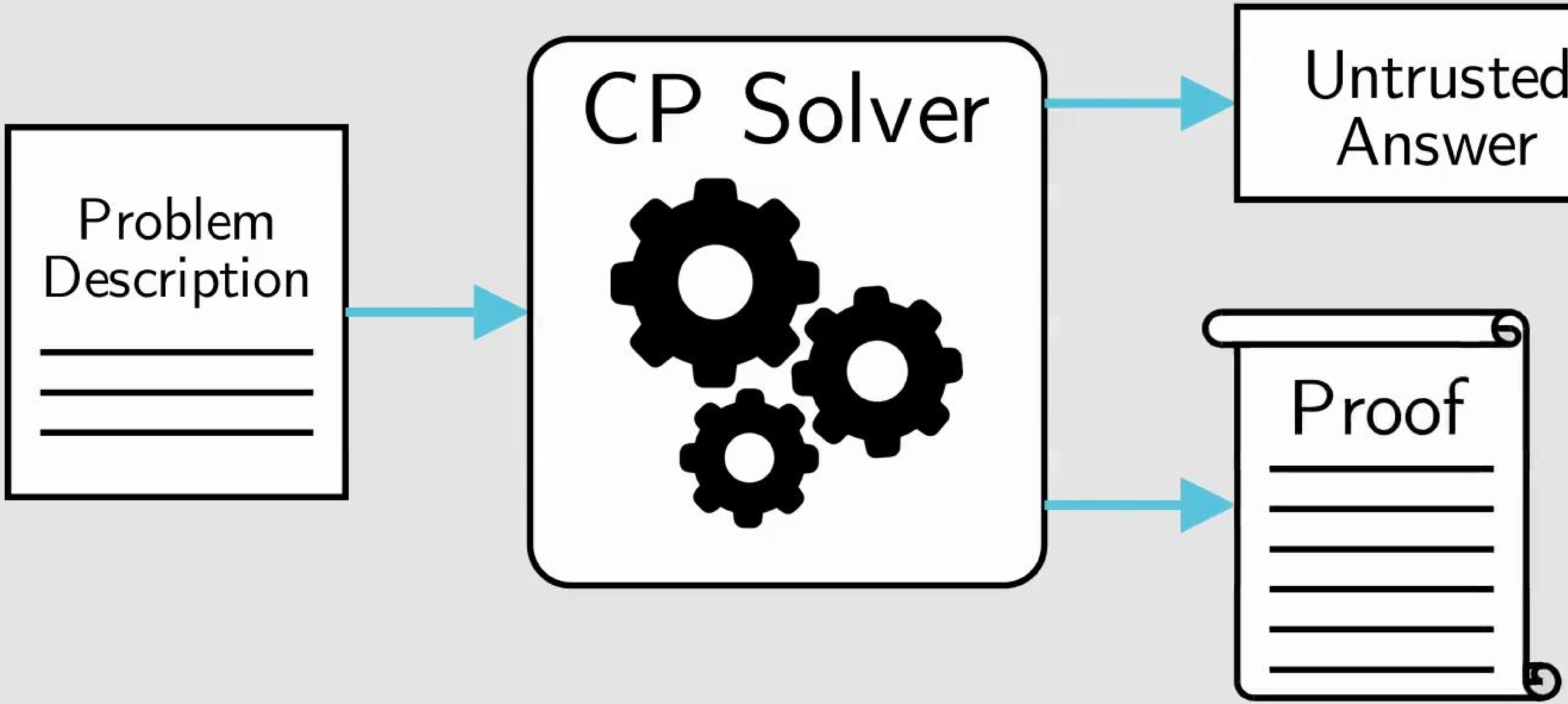
What we want:



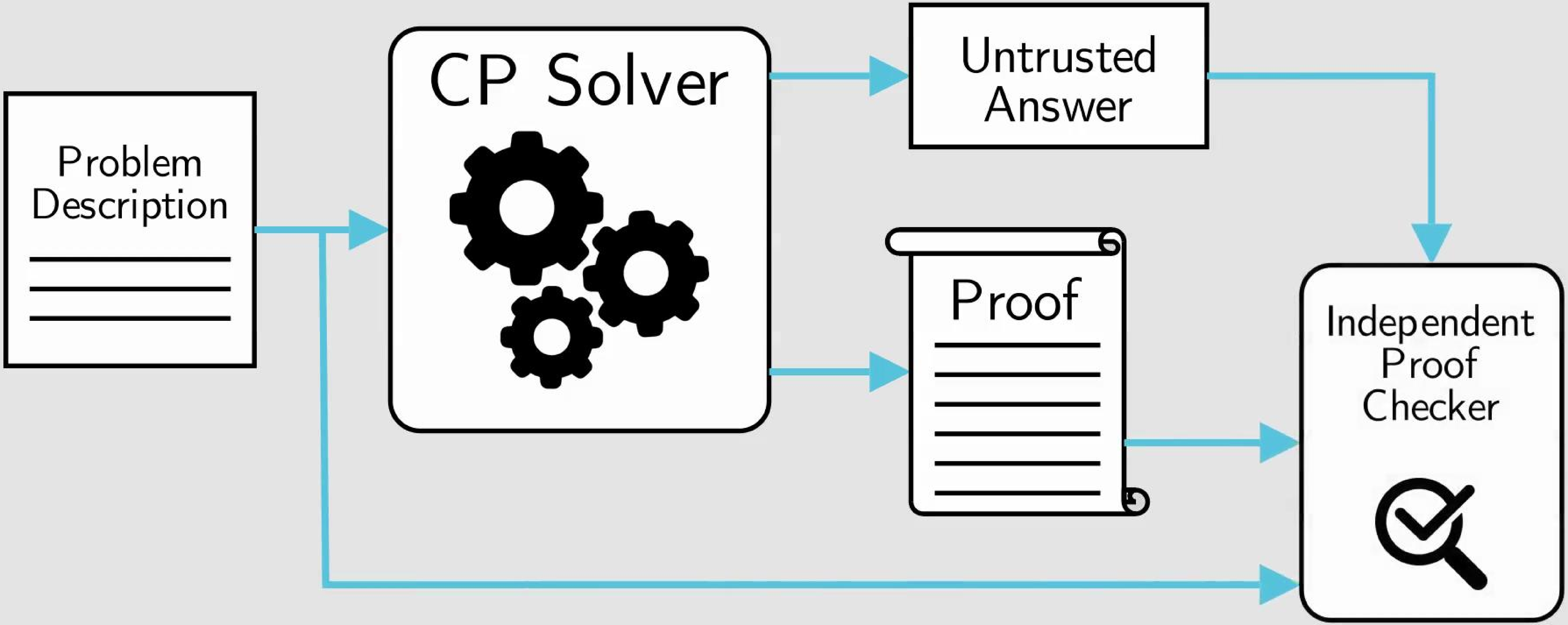
What we want:



What we want:



What we want:


Problem Description
CP Solver
Untrusted Answer
Proof
Independent Proof Checker

Related Work

Related Work

A Proof-Producing CSP Solver

Michael Veksler and **Ofer Strichman**
mveksler@tx.technion.ac.il ofers@ie.technion.ac.il
Information systems Engineering, IE, Technion, Haifa, Israel

Abstract

PCS is a CSP solver that can produce a machine-checkable deductive proof in case it decides that the input problem is unsatisfiable. The roots of the proof may be nonclausal constraints, whereas the rest of the proof is based on resolution of signed clauses, ending with the empty clause. PCS uses parameterized, constraint-specific inference rules in order to bridge between the nonclausal and the clausal parts of the proof. The consequent of each such rule is a signed clause that is 1) logically implied by the nonclausal premise, and 2) strong enough to be the premise of the consecutive proof steps. The resolution process itself is integrated in the learning mechanism, and can be seen as a generalization to CSP of a similar solution that is adopted by competitive SAT solvers.

1 Introduction

Many problems in planning, scheduling, automatic test-generation, configuration and more, can be naturally modeled as Constraint Satisfaction Problems (CSP) (Dechter 2003), and solved with one of the many publicly available CSP solvers. The common definition of this problem refers to a set of variables over finite and discrete domains, and arbitrary constraints over these variables. The goal is to decide whether there is an assignment to the variables from their respective domains, which satisfies all the constraints. If the answer is positive the assignment that is emitted by the CSP solver can be verified easily. On the other hand a negative answer is harder to verify, since current CSP solvers do not produce a deductive proof of unsatisfiability.

In contrast, most modern CNF-based SAT solvers accompany an unsatisfiability result with a deductive proof that can be checked automatically. Specifically, they produce a *resolution proof*, which is a sequence of application of a single inference rule, namely the binary *resolution rule*. In the case of SAT the proof has uses other than just the ability to independently validate an unsatisfiability result. For example, there is a successful SAT-based model-checking algorithm which is based on deriving interpolants from the resolution proof (Henzinger et al. 2004).

Unlike SAT solvers, CSP solvers do not have the luxury of handling clausal constraints. They need to handle constraints such as $a < b + 5$, $allDifferent(x, y, z)$, $a \neq b$, and so on. However, we argue that the effect of a constraint in a given state can always be replicated with a *signed clause*, which can then be part of a resolution proof. A signed clause is a disjunction between *signed literals*. A signed literal is a unary constraint, constraining a variable to a domain of values. For example, the signed clause $(x_1 \in \{1,2\} \vee x_2 \notin \{3\})$ constrains[1] x_1 to be in the range $[1,2]$ or x_2 to be anything but 3. A conjunction of signed clauses is called *signed CNF*, and the problem of solving signed CNF is called *signed SAT*[2], a problem which attracted extensive theoretical research and development of tools (Liu, Kuehlmann, and Moskewicz 2003; Beckert, Hähnle, and Manyá 2000b).

In this article we describe how our arc-consistency-based CSP solver PCS (for a "Proof-producing Constraint Solver") produces deductive proofs when the formula is unsatisfiable. In order to account for propagations by general constraints it uses constraint-specific parametric inference rules. Each such rule has a constraint as a premise and a signed clause as a consequent. These consequents, which are generated during conflict analysis, are called *explanation clauses*. These clauses are logically implied by the premise, but are also strong enough to imply the same literal that the premise implies at the current state. The emitted proof is a sequence of inferences of such clauses and application of special resolution rules that are tailored for signed clauses.

Like in the case of SAT, the signed clauses that are learned as a result of analyzing conflicts serve as 'milestone' atoms in the proof, although they are not the only ones. They are generated by a repeated application of the resolution rule. The intermediate clauses that are generated in this process are discarded and hence have no effect on the solving process itself. In case the learned clause eventually participates in the proof PCS reconstructs them, by using information that it saves during the learning process. We will describe this conflict-analysis mechanism in detail in Section 3 and 4, and compare it to alternatives such as 1-UIP (Zhang et al. 2001), MVS (Liu, Kuehlmann, and Moskewicz 2003) and EFC (Katsirelos and Bacchus 2005) in Section 5. We begin, however, by describing several preliminaries such as CSP

[1] Alternative notations such as $\{1,2\}{:}x_1$ and $x_1^{\{1,2\}}$ are used in the literature to denote a signed literal $x_1 \in \{1,2\}$.

[2] Signed SAT is also called MV-SAT (i.e. Many Valued SAT).

Certifying Optimality in Constraint Programming

GRAEME GANGE, Monash University
GEOFFREY CHU, Data61, CSIRO
PETER J. STUCKEY, Monash University

Discrete optimization problems are one of the most challenging class of problems to solve, they are typically NP-hard. Complete solving approaches to these problems, such as integer programming or constraint programming, are able to prove optimal solutions. Since complete solvers are highly complex software objects, when a solver returns that it has proved optimality, how confident can we be in this result? The short answer is *not very*. Constraint programming (CP) solvers can hide difficult to observe bugs because they rely on complex state maintenance over backtracking.

In this paper we develop a strategy for validating unsatisfiability and optimality results. We extend a lazy clause generation CP solver with proof-generating capabilities, which is paired with an external, formally certified proof checking procedure. From this, we derive several proof checkers, which establish different compromises between trust base and performance. We validate the practicality of this approach by verifying the correctness of alleged unsatisfiability and optimality results from the 2016 MiniZinc challenge.

CCS Concepts: • **Theory of computation → Constraint and logic programming**; **Discrete optimization**; • **Software and its engineering → Software verification**; • **Computing methodologies** → *Theorem proving algorithms*;

Additional Key Words and Phrases: constraint programming, certified code, verification, Boolean satisfiability

ACM Reference Format:
Graeme Gange, Geoffrey Chu, and Peter J. Stuckey. 2023. Certifying Optimality in Constraint Programming. 1, 1 (September 2023), 39 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Discrete optimization problems arise in a vast range of applications: scheduling, rostering, routing, and management decision. These problems frequently arise in mission critical applications; ambulance dispatch [40], E-commerce [28] and disaster recovery [47], amongst others – situations where mistakes can have disastrous consequences. Since the results of the optimization problems are critical to the industry to which they belong, when we use optimization technology to create solutions we wish to be able to trust the results we obtain. Optimization tools are also seeing increasing use in combinatorics, where an incorrect result fundamentally undermines the entire endeavor.

Two kinds of error can occur:

- a "solution" returned by the solver does not satisfy the problem
- a claimed optimal solution returned by the solver is not in fact optimal

Authors' addresses: Graeme Gange, Faculty of Information Technology, Monash University, graeme.gange@monash.edu; Geoffrey Chu, Data61, CSIRO, chu.geoffrey@gmail.com; Peter J. Stuckey, Faculty of Information Technology, Monash University, peter.stuckey@monash.edu.

XXXX-XXXX/2023/9-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

, Vol. 1, No. 1, Article . Publication date: September 2023.

Related Work

A Proof-Producing CSP Solver

Michael Veksler and **Ofer Strichman**
mveksler@tx.technion.ac.il ofers@ie.technion.ac.il
Information systems Engineering, IE, Technion, Haifa, Israel

Abstract

PCS is a CSP solver that can produce a machine-checkable deductive proof in case it decides that the input problem is unsatisfiable. The roots of the proof may be nonclausal constraints, whereas the rest of the proof is based on resolution of signed clauses, ending with the empty clause. PCS uses parameterized, constraint-specific inference rules in order to bridge between the nonclausal and the clausal parts of the proof. The consequent of each such rule is a signed clause that is 1) logically implied by the nonclausal premise, and 2) strong enough to be the premise of the consecutive proof steps. The resolution process itself is integrated in the learning mechanism, and can be seen as a generalization to CSP of a similar solution that is adopted by competitive SAT solvers.

1 Introduction

Many problems in planning, scheduling, automatic test-generation, configuration and more, can be naturally modeled as Constraint Satisfaction Problems (CSP) (Dechter 2003), and solved with one of the many publicly available CSP solvers. The common definition of this problem refers to a set of variables over finite and discrete domains, and arbitrary constraints over these variables. The goal is to decide whether there is an assignment to the variables from their respective domains, which satisfies all the constraints. If the answer is positive the assignment that is emitted by the CSP solver can be verified easily. On the other hand a negative answer is harder to verify, since current CSP solvers do not produce a deductive proof of unsatisfiability.

In contrast, most modern CNF-based SAT solvers accompany an unsatisfiability result with a deductive proof that can be checked automatically. Specifically, they produce a *resolution proof*, which is a sequence of application of a single inference rule, namely the binary *resolution rule*. In the case of SAT the proof has uses other than just the ability to independently validate an unsatisfiability result. For example, there is a successful SAT-based model-checking algorithm which is based on deriving interpolants from the resolution proof (Henzinger et al. 2004).

Unlike SAT solvers, CSP solvers do not have the luxury of handling clausal constraints. They need to handle constraints such as $a < b+5$, *allDifferent*(x,y,z), $a \neq b$, and so on. However, we argue that the effect of a constraint in a given state can always be replicated with a *signed clause*, which can then be part of a resolution proof. A signed clause is a disjunction between *signed literals*. A signed literal is a unary constraint, constraining a variable to a domain of values. For example, the signed clause $(x_1 \in \{1,2\} \vee x_2 \notin \{3\})$ constrains[1] x_1 to be in the range $[1,2]$ or x_2 to be anything but 3. A conjunction of signed clauses is called *signed CNF*, and the problem of solving signed CNF is called *signed SAT*[2], a problem which attracted extensive theoretical research and development of tools (Liu, Kuehlmann, and Moskewicz 2003; Beckert, Hähnle, and Manyá 2000b).

In this article we describe how our arc-consistency-based CSP solver PCS (for a "Proof-producing Constraint Solver") produces deductive proofs when the formula is unsatisfiable. In order to account for propagations by general constraints it uses constraint-specific parametric inference rules. Each such rule has a constraint as a premise and a signed clause as a consequent. These consequents, which are generated during conflict analysis, are called *explanation clauses*. These clauses are logically implied by the premise, but are also strong enough to imply the same literal that the premise implies at the current state. The emitted proof is a sequence of inferences of such clauses and application of special resolution rules that are tailored for signed clauses.

Like in the case of SAT, the signed clauses that are learned as a result of analyzing conflicts serve as 'milestone' atoms in the proof, although they are not the only ones. They are generated by a repeated application of the resolution rule. The intermediate clauses that are generated in this process are discarded and hence have no effect on the solving process itself. In case the learned clause eventually participates in the proof PCS reconstructs them, by using information that it saves during the learning process. We will describe this conflict-analysis mechanism in detail in Section 3 and 4, and compare it to alternatives such as 1-UIP (Zhang et al. 2001), MVS (Liu, Kuehlmann, and Moskewicz 2003) and EFC (Katsirelos and Bacchus 2005) in Section 5. We begin, however, by describing several preliminaries such as CSP

[1] Alternative notations such as $\{1,2\}{:}x_1$ and $x_1^{\{1,2\}}$ are used in the literature to denote a signed literal $x_1 \in \{1,2\}$.

[2] Signed SAT is also called MV-SAT (i.e. Many Valued SAT).

Certifying Optimality in Constraint Programming

GRAEME GANGE, Monash University
GEOFFREY CHU, Data61, CSIRO
PETER J. STUCKEY, Monash University

Discrete optimization problems are one of the most challenging class of problems to solve, they are typically NP-hard. Complete solving approaches to these problems, such as integer programming or constraint programming, are able to prove optimal solutions. Since complete solvers are highly complex software objects, when a solver returns that it has proved optimality, how confident can we be in this result? The short answer is *not very*. Constraint programming (CP) solvers can hide difficult to observe bugs because they rely on complex state maintenance over backtracking.

In this paper we develop a strategy for validating unsatisfiability and optimality results. We extend a lazy clause generation CP solver with proof-generating capabilities, which is paired with an external, formally certified proof checking procedure. From this, we derive several proof checkers, which establish different compromises between trust base and performance. We validate the practicality of this approach by verifying the correctness of alleged unsatisfiability and optimality results from the 2016 MiniZinc challenge.

CCS Concepts: • **Theory of computation → Constraint and logic programming**; **Discrete optimization**; • **Software and its engineering → Software verification**; • **Computing methodologies →** *Theorem proving algorithms*;

Additional Key Words and Phrases: constraint programming, certified code, verification, Boolean satisfiability

ACM Reference Format:
Graeme Gange, Geoffrey Chu, and Peter J. Stuckey. 2023. Certifying Optimality in Constraint Programming. 1, 1 (September 2023), 39 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Discrete optimization problems arise in a vast range of applications: scheduling, rostering, routing, and management decision. These problems frequently arise in mission critical applications; ambulance dispatch [40], E-commerce [28] and disaster recovery [47], amongst others – situations where mistakes can have disastrous consequences. Since the results of the optimization problems are critical to the industry to which they belong, when we use optimization technology to create solutions we wish to be able to trust the results we obtain. Optimization tools are also seeing increasing use in combinatorics, where an incorrect result fundamentally undermines the entire endeavor.

Two kinds of error can occur:

- a "solution" returned by the solver does not satisfy the problem
- a claimed optimal solution returned by the solver is not in fact optimal

Authors' addresses: Graeme Gange, Faculty of Information Technology, Monash University, graeme.gange@monash.edu; Geoffrey Chu, Data61, CSIRO, chu.geoffrey@gmail.com; Peter J. Stuckey, Faculty of Information Technology, Monash University, peter.stuckey@monash.edu.

XXXX-XXXX/2023/9-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

, Vol. 1, No. 1, Article . Publication date: September 2023.

Do we need trusted inference checkers for every constraint?

vec_eq_tuple

visible

weighted_partial_alldiff

xor

zero_or_not_zero

zero_or_not_zero_vectors

Simple enough to be easy to verify

Simple enough to be easy to verify

Expressive enough for CP reasoning

Simple enough to be easy to verify

Expressive enough for CP reasoning

pseudo-Boolean proofs!

VeriPB

Pseudo-Boolean constraints
are very expressive

VeriPB

Pseudo-Boolean constraints
are very expressive

VeriPB

Cutting planes is a
powerful proof system

Pseudo-Boolean constraints are very expressive

Working proof checker implementation (+ formally verified checker)

VeriPB

Cutting planes is a powerful proof system

Pseudo-Boolean constraints are very expressive

Working proof checker implementation (+ formally verified checker)

VeriPB

Cutting planes is a powerful proof system

- SAT
- MaxSAT
- PB
- Graphs
- ...CP!

PB Variable

PB Variable

$$x_i \in \{0, 1\}$$

PB Literal

$$\ell_i := \ x_i \in \{0, 1\}$$

$$\text{or } \bar{x}_i = 1 - x$$

PB Constraint

$$C_j := \sum_i a_{ij}\ell_i \geq b_j \quad a_{ij}, b_j \in \mathbb{Z}$$

PB Formula/Model

$$\left\{ C_j := \sum_i a_{ij}\ell_i \geq b_j \right\}_j$$

PB Formula/Model

$$\left\{ C_j := \sum_i a_{ij} \ell_i \geq b_j \right\}_j$$

$$(\min \sum_i c_i \ell_i) \quad a_{ij}, b_j, c_i, \in \mathbb{Z}$$

PB Formula/Model

$$\left\{ C_j := \sum_i a_{ij} \ell_i \geq b_j \right\}_j$$

$$(\min \sum_i c_i \ell_i) \quad a_{ij}, b_j, c_i, \in \mathbb{Z}$$

PB Proof

PB Formula/Model

$$\left\{ C_j := \sum_i a_{ij}\ell_i \geq b_j \right\}_j$$

$$(\min \sum_i c_i \ell_i) \quad a_{ij}, b_j, c_i, \in \mathbb{Z}$$

PB Proof

PB Formula/Model

$$\left\{ C_j := \sum_i a_{ij}\ell_i \geq b_j \right\}_j$$

$$(\min \sum_i c_i \ell_i) \quad a_{ij}, b_j, c_i, \in \mathbb{Z}$$

(load formula)

(rule) $\sum_i a_{im+1}\ell_i \geq b_{j+1}$

PB Proof

PB Formula/Model

$$\left\{ C_j := \sum_i a_{ij} \ell_i \geq b_j \right\}_j$$

$$(\min \sum_i c_i \ell_i) \quad a_{ij}, b_j, c_i, \in \mathbb{Z}$$

(load formula)

(rule) $\sum_i a_{im+1} \ell_i \geq b_{j+1}$

(rule) $\sum_i a_{im+2} \ell_i \geq b_{j+2}$

PB Proof

PB Formula/Model

$$\left\{ C_j := \sum_i a_{ij}\ell_i \geq b_j \right\}_j$$

$$(\min \sum_i c_i\ell_i) \quad a_{ij}, b_j, c_i, \in \mathbb{Z}$$

(load formula)

(rule) $\sum_i a_{im+1}\ell_i \geq b_{j+1}$

(rule) $\sum_i a_{im+2}\ell_i \geq b_{j+2}$

$\vdots$

PB Proof

PB Formula/Model

$$\left\{ C_j := \sum_i a_{ij}\ell_i \geq b_j \right\}_j$$

$$(\min \sum_i c_i\ell_i) \quad a_{ij}, b_j, c_i, \in \mathbb{Z}$$

(load formula)

(rule) $\sum_i a_{im+1}\ell_i \geq b_{j+1}$

(rule) $\sum_i a_{im+2}\ell_i \geq b_{j+2}$

⋮

⋮

PB Proof

PB Formula/Model

$$\left\{ C_j := \sum_i a_{ij}\ell_i \geq b_j \right\}_j$$

$$(\min \sum_i c_i\ell_i) \quad a_{ij}, b_j, c_i, \in \mathbb{Z}$$

(load formula)

(rule) $\sum_i a_{im+1}\ell_i \geq b_{j+1}$

(rule) $\sum_i a_{im+2}\ell_i \geq b_{j+2}$

$\vdots$

$\vdots$

(rule) $0 \geq 1$ $\square$

PB Proof

PB Formula/Model

$$\left\{ C_j := \sum_i a_{ij}\ell_i \geq b_j \right\}_j$$

$$(\min \sum_i c_i \ell_i) \quad a_{ij}, b_j, c_i, \in \mathbb{Z}$$

(load formula)

(rule) $\sum_i a_{im+1}\ell_i \geq b_{j+1}$

(rule) $\sum_i a_{im+2}\ell_i \geq b_{j+2}$

$\vdots$

$\vdots$

(rule) $\sum_i c_i\ell_i \geq o_i$ $\square$

PB Proof

PB Formula/Model

$$\left\{ C_j := \sum_i a_{ij}\ell_i \geq b_j \right\}_j$$

$$(\min \sum_i c_i\ell_i) \quad a_{ij}, b_j, c_i, \in \mathbb{Z}$$

(load formula)

(rule) $\sum_i a_{im+1}\ell_i \geq b_{j+1}$

(rule) $\sum_i a_{im+2}\ell_i \geq b_{j+2}$

$\vdots$

(rule) $-\sum_i c_i\ell_i \geq -o_i$

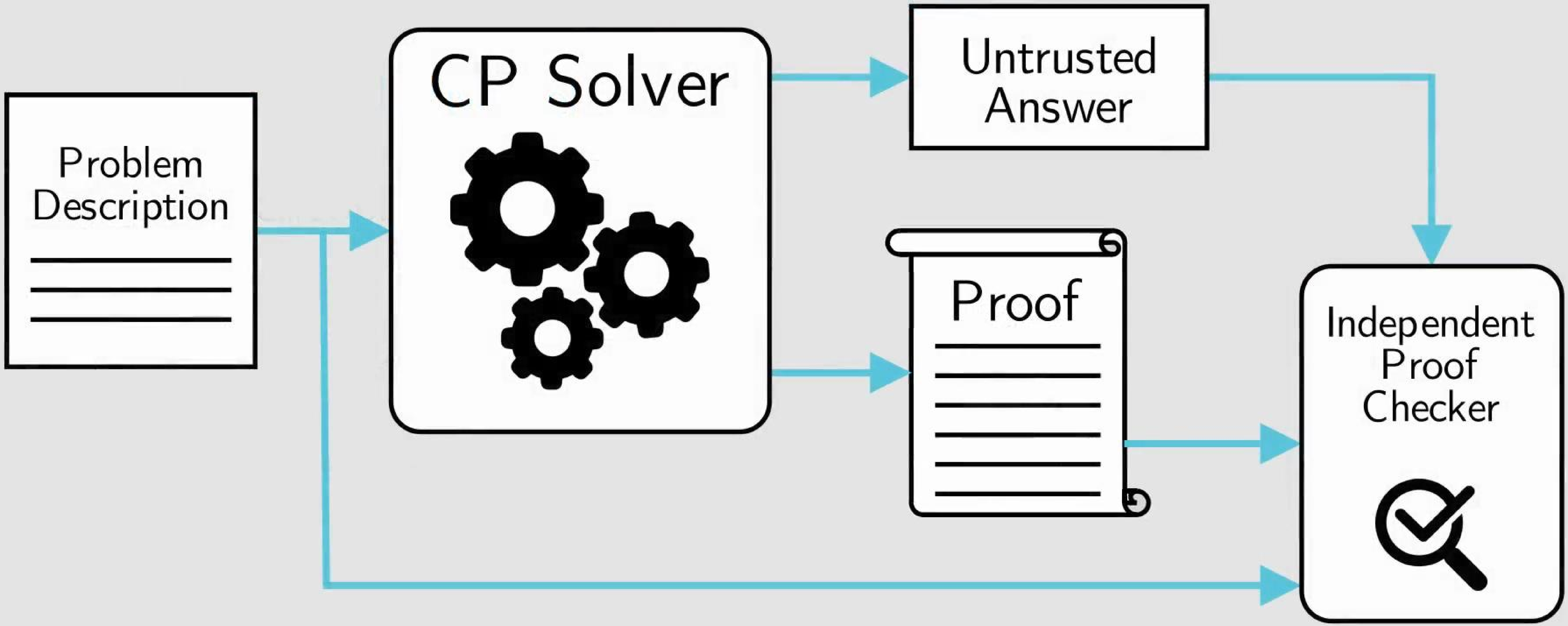
(rule) $\sum_i c_i\ell_i \geq o_i$ $\square$

PB Proof

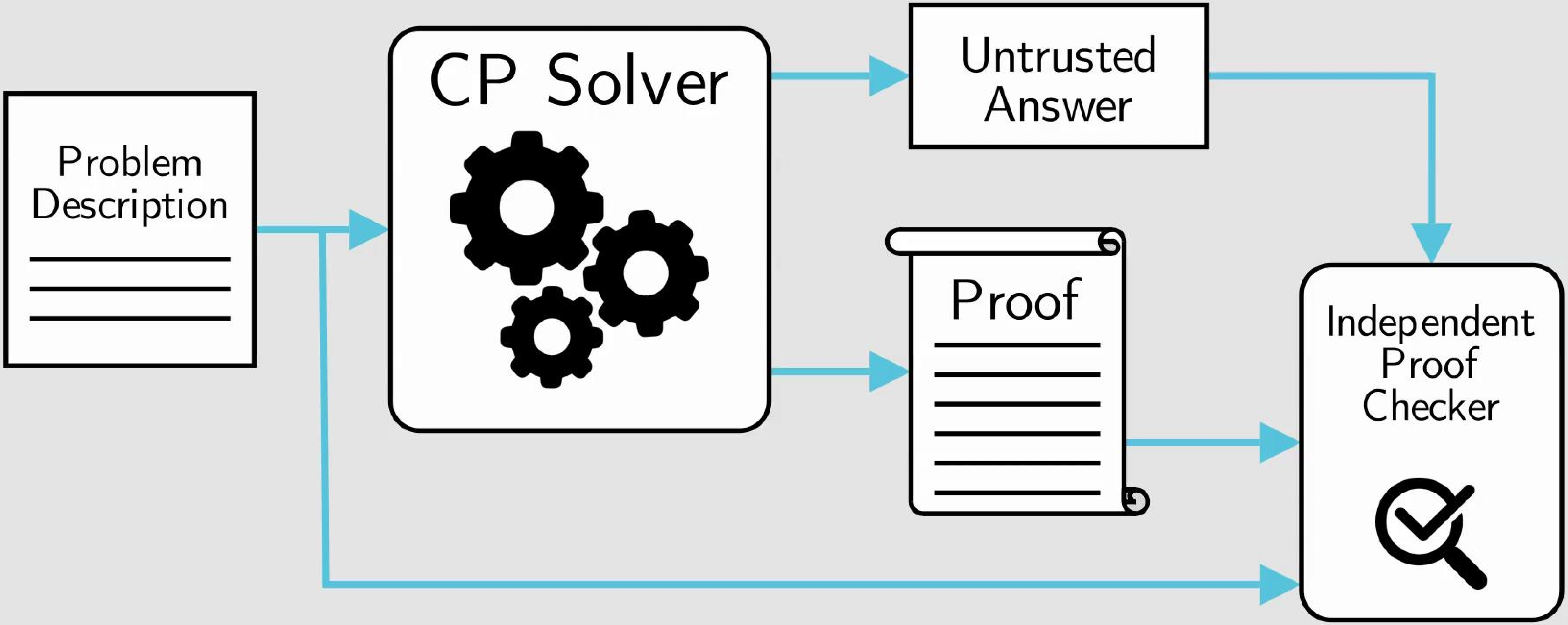
PB Formula/Model

```
% my_problem.opb

3 x1 4 x2 5 ~x3 >= 1 ;
5 x4 2 ~x1 3 ~x2 -1 x1 >= 4 ;
3 x1 -2 x2 >= -1 ;
-1 x1 -2 ~x4 >= -1 ;
```

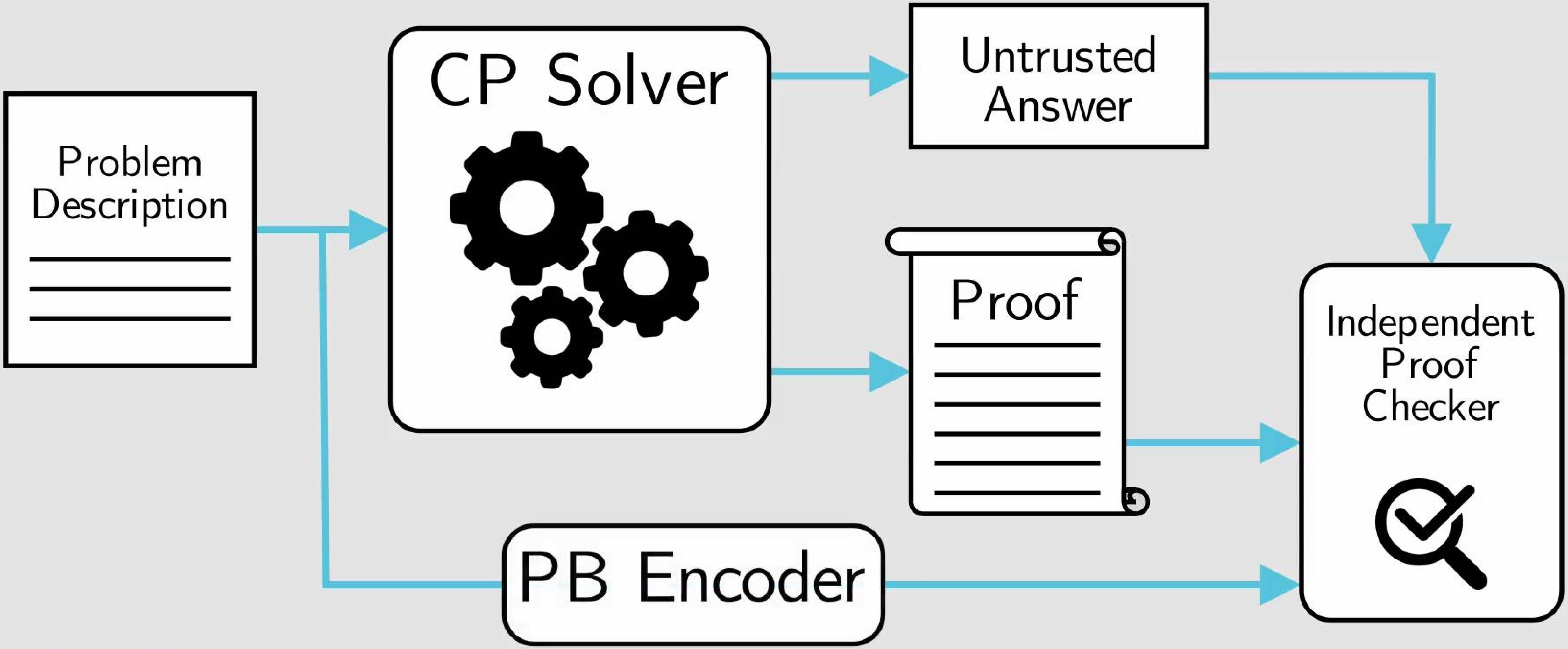
```
% my_proof.pbp

pseudo-Boolean proof version 3.0
f 4 ;
rup 1 x1 1 ~x2 >= 1 ;
rup 1 ~x3 2 ~x4 4 ~x5 >= 5 ;
pol 1 2 + ;
ia 1 x1 5 ~x4 >= 5 ;
u >= 1 ;

output NONE ;
conclusion UNSAT;
end pseudo-Boolean proof ;
```


Problem Description
CP Solver
Untrusted Answer
Proof
Independent Proof Checker

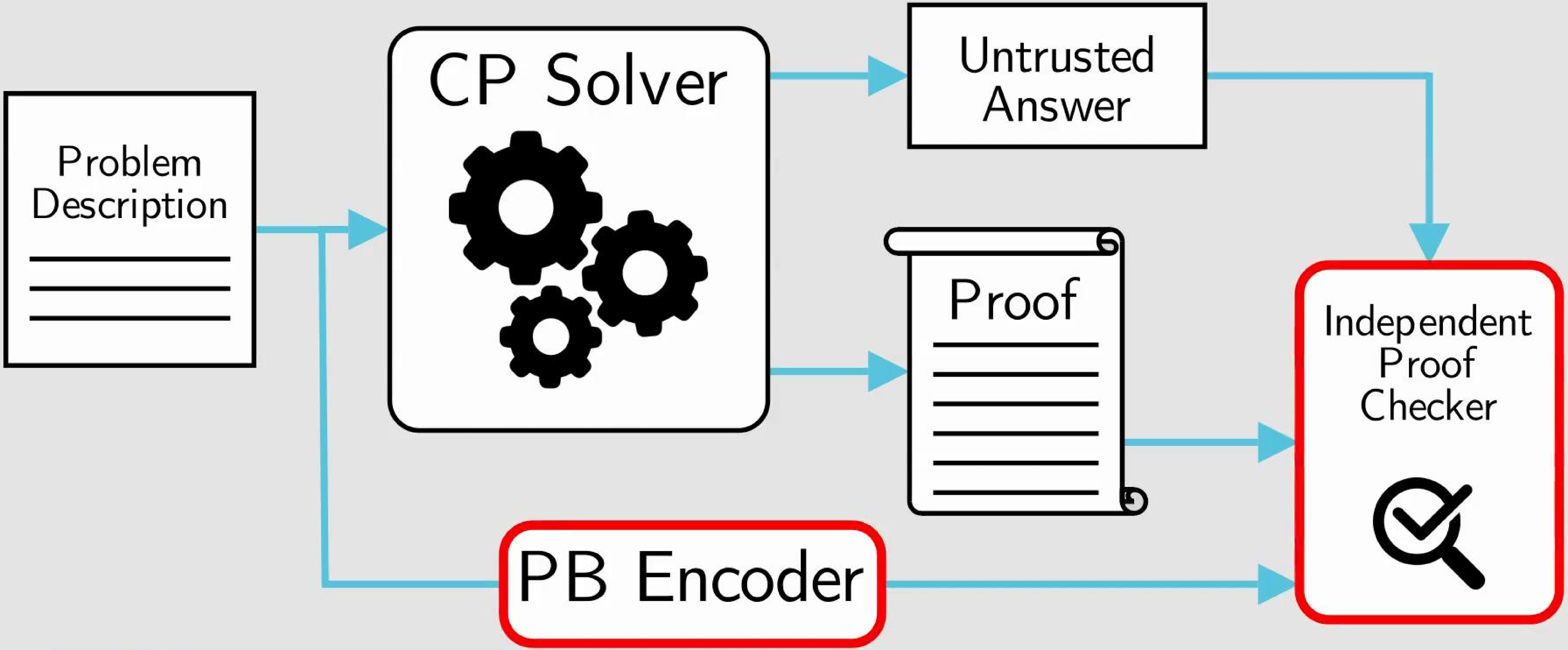
Slightly Modifying the Basic Proof Logging Idea



Slightly Modifying the Basic Proof Logging Idea



Slightly Modifying the Basic Proof Logging Idea



Binary Variable Encoding

Binary Variable Encoding

$$X \in [3...10]$$

Binary Variable Encoding

$$8x_{b3} + 4x_{b2} + 2x_{b1} + x_{b0} \geq 3$$
$$-8x_{b3} - 4x_{b2} - 2x_{b1} - x_{b0} \geq -10$$

Binary Variable Encoding

$$X \in [-12 \ldots 10]$$

Binary Variable Encoding

$$-16x_{b4} + 8x_{b3} + 4x_{b2} + 2x_{b1} + x_{b0} \geq -12$$
$$16x_{b4} - 8x_{b3} - 4x_{b2} - 2x_{b1} - x_{b0} \geq -10$$

Binary Variable Encoding

$$bits(X) \geq -12$$

$$-bits(X) \geq 10$$

Binary Variable Encoding

$$X + 2Y - 4Z \geq 11$$

Binary Variable Encoding

$$X + 2Y - 4Z \geq 11$$

$$\downarrow$$

$$bits(X) + 2bits(Y) - 4bits(Z) \geq 11$$

Reifying PB Constraints

Reifying PB Constraints

$$8x_1 - 4x_2 + 6x_3 - 10x_4 \geq 6$$

Reifying PB Constraints

$$y \Rightarrow 8x_1 - 4x_2 + 6x_3 - 10x_4 \geq 6$$

Reifying PB Constraints

$$20\bar{y} + 8x_1 - 4x_2 + 6x_3 - 10x_4 \geq 6$$

Reifying PB Constraints

$$20 \cdot 1 + 8x_1 - 4x_2 + 6x_3 - 10x_4 \geq 6$$

Reifying PB Constraints

$$8x_1 - 4x_2 + 6x_3 - 10x_4 \geq -14$$

Reifying PB Constraints

$$20\bar{y} + 8x_1 - 4x_2 + 6x_3 - 10x_4 \geq 6$$

Reifying PB Constraints

$$20 \cdot 0 + 8x_1 - 4x_2 + 6x_3 - 10x_4 \geq 6$$

Reifying PB Constraints

$$8x_1 - 4x_2 + 6x_3 - 10x_4 \geq 6$$

Reifying PB Constraints

$$y \Leftrightarrow 8x_1 - 4x_2 + 6x_3 - 10x_4 \geq 6$$

Reifying PB Constraints

$$y \Rightarrow 8x_1 - 4x_2 + 6x_3 - 10x_4 \geq 6$$

$$\bar{y} \Rightarrow -8x_1 + -4x_2 - 6x_3 + 10x_4 \geq -5$$

Reifying PB Constraints

$$y_1 \wedge y_2 \ldots \wedge y_k \Rightarrow$$

$$8x_1 - 4x_2 + 6x_3 - 10x_4 \geq 6$$

Reifying PB Constraints

$$y_1 \Rightarrow (y_2 \Rightarrow (\ldots \Rightarrow (y_k \Rightarrow 8x_1 - 4x_2 + 6x_3 - 10x_4 \geq 6)\ldots))$$

Reifying PB Constraints

$$20\bar{y}_1 + 20\bar{y}_2 + \cdots + 20\bar{y}_k$$

$$8x_1 - 4x_2 + 6x_3 - 10x_4 \geq 6)...))$$

Reifying PB Constraints

$$C \qquad \neg C$$

$$y \Rightarrow C \qquad \bar{y} \Rightarrow \neg C$$

$$y \Leftrightarrow C \qquad y_1 \wedge \cdots \wedge y_k \Rightarrow C$$

$$X \neq Y$$
$$X \notin \{3, 5, 7\}$$

$$X \neq Y$$

$$X \notin \{3, 5, 7\}$$

$$X \neq Y$$

$$X \notin \{3, 5, 7\}$$

$$f \Rightarrow bits(X) - bits(Y) \geq 1$$

$$\bar{f} \Rightarrow bits(Y) - bits(X) \geq 1$$

$$X \neq Y$$
$$X \notin \{3, 5, 7\}$$

$$f \Rightarrow bits(X) - bits(Y) \geq 1$$
$$\bar{f} \Rightarrow bits(Y) - bits(X) \geq 1$$
$$x_{\geq 3} \Leftrightarrow bits(X) \geq 3$$

$$X \neq Y$$
$$X \notin \{3, 5, 7\}$$

$$f \Rightarrow bits(X) - bits(Y) \geq 1$$
$$\bar{f} \Rightarrow bits(Y) - bits(X) \geq 1$$
$$x_{\geq 3} \Leftrightarrow bits(X) \geq 3$$
$$x_{\leq 3} \Leftrightarrow -bits(X) \geq -3$$

$$X \neq Y$$
$$X \notin \{3, 5, 7\}$$

$$f \Rightarrow bits(X) - bits(Y) \geq 1$$
$$\bar{f} \Rightarrow bits(Y) - bits(X) \geq 1$$
$$x_{\geq 3} \Leftrightarrow bits(X) \geq 3$$
$$x_{\leq 3} \Leftrightarrow -bits(X) \geq -3$$
$$x_{=3} \Leftrightarrow x_{\geq 3} + x_{\leq 3} \geq 2$$

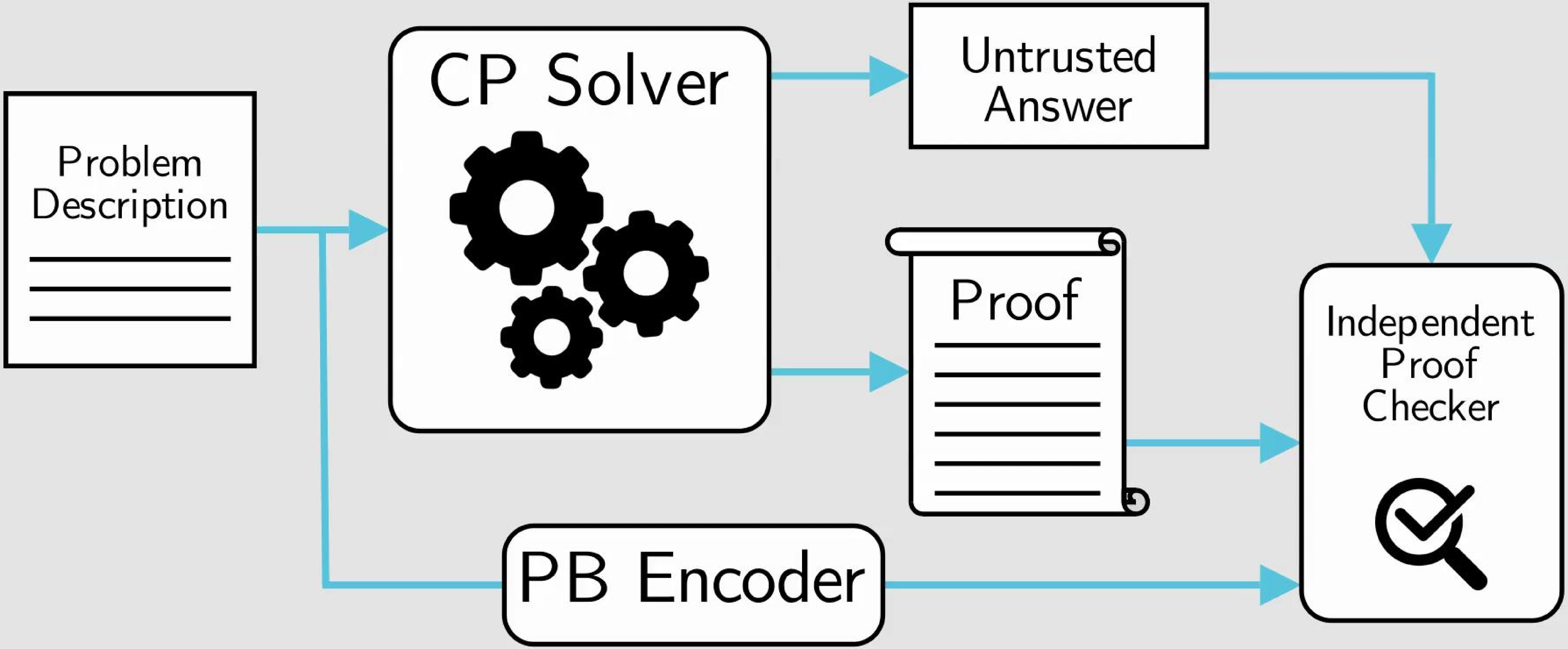
$$X \neq Y$$
$$X \notin \{3, 5, 7\}$$

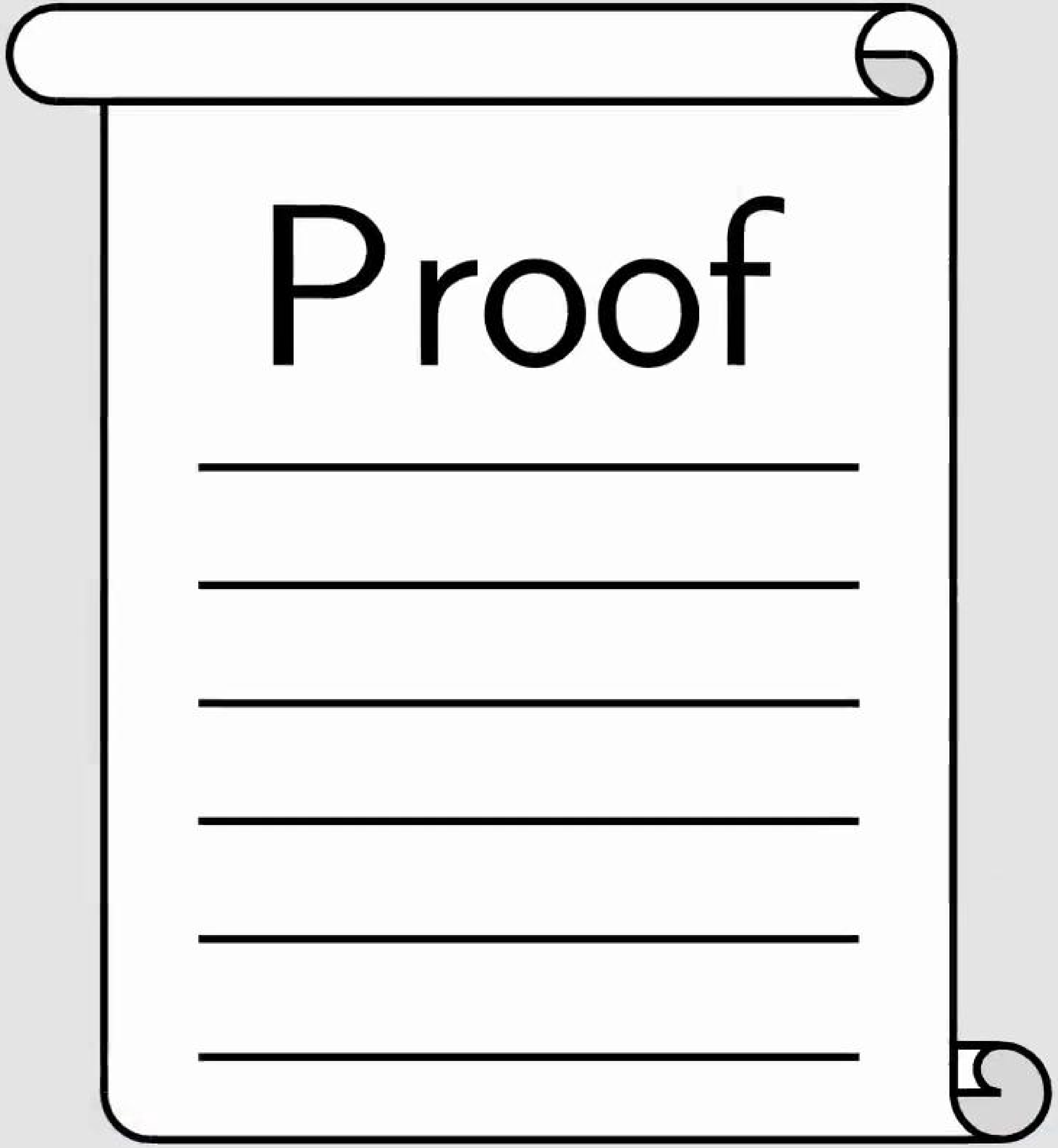
$$f \Rightarrow bits(X) - bits(Y) \geq 1$$
$$\bar{f} \Rightarrow bits(Y) - bits(X) \geq 1$$
$$x_{\geq 3} \Leftrightarrow bits(X) \geq 3$$
$$x_{\leq 3} \Leftrightarrow -bits(X) \geq -3$$
$$x_{=3} \Leftrightarrow x_{\geq 3} + x_{\leq 3} \geq 2$$
$$\dots$$

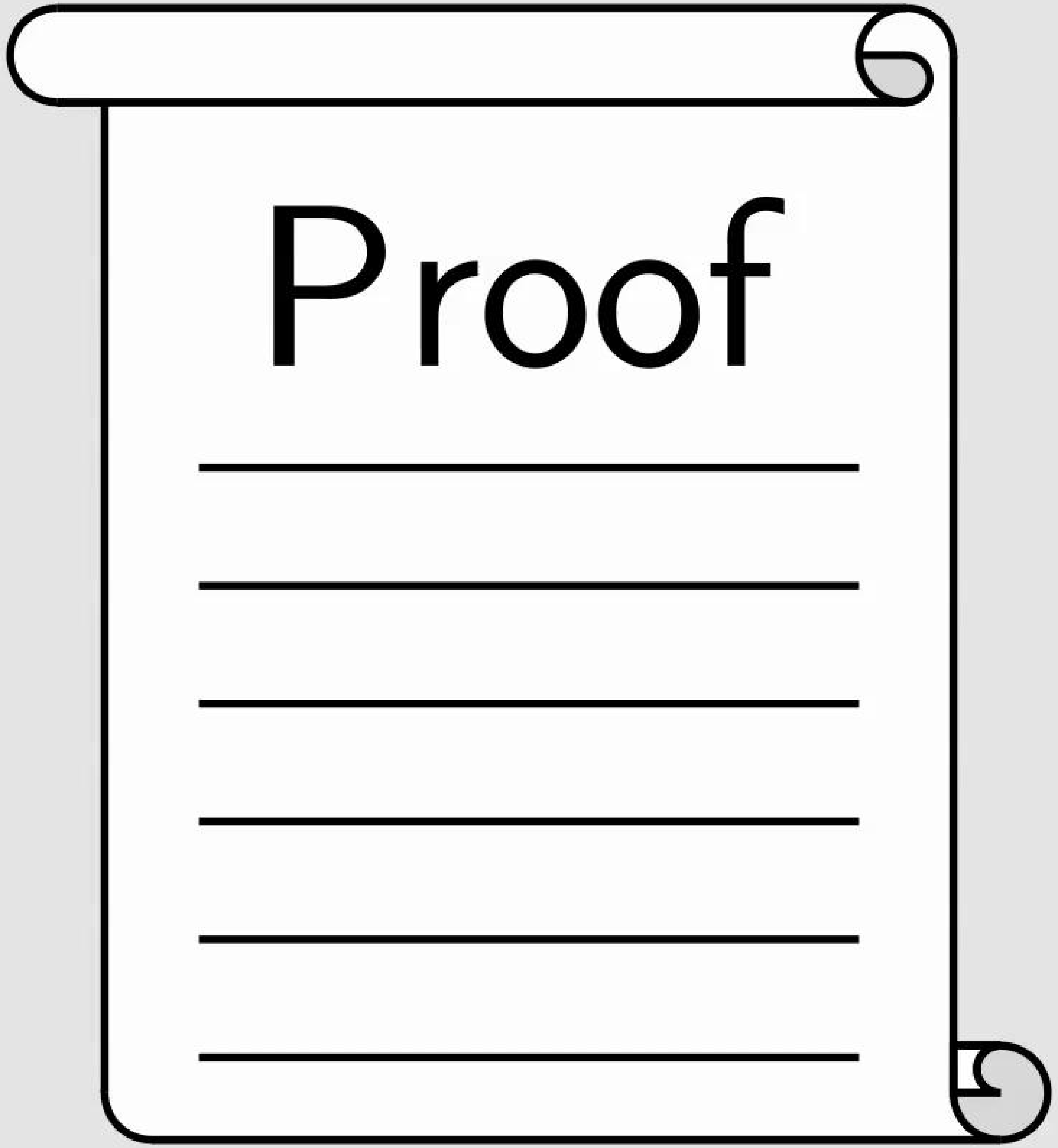
$$X \neq Y$$
$$X \notin \{3, 5, 7\}$$

$$f \Rightarrow bits(X) - bits(Y) \geq 1$$
$$\bar{f} \Rightarrow bits(Y) - bits(X) \geq 1$$
$$x_{\geq 3} \Leftrightarrow bits(X) \geq 3$$
$$x_{\leq 3} \Leftrightarrow -bits(X) \geq -3$$
$$x_{=3} \Leftrightarrow x_{\geq 3} + x_{\leq 3} \geq 2$$
$$\cdots$$
$$\bar{x}_{=3} + \bar{x}_{=5} + \bar{x}_{=7} \geq 3$$

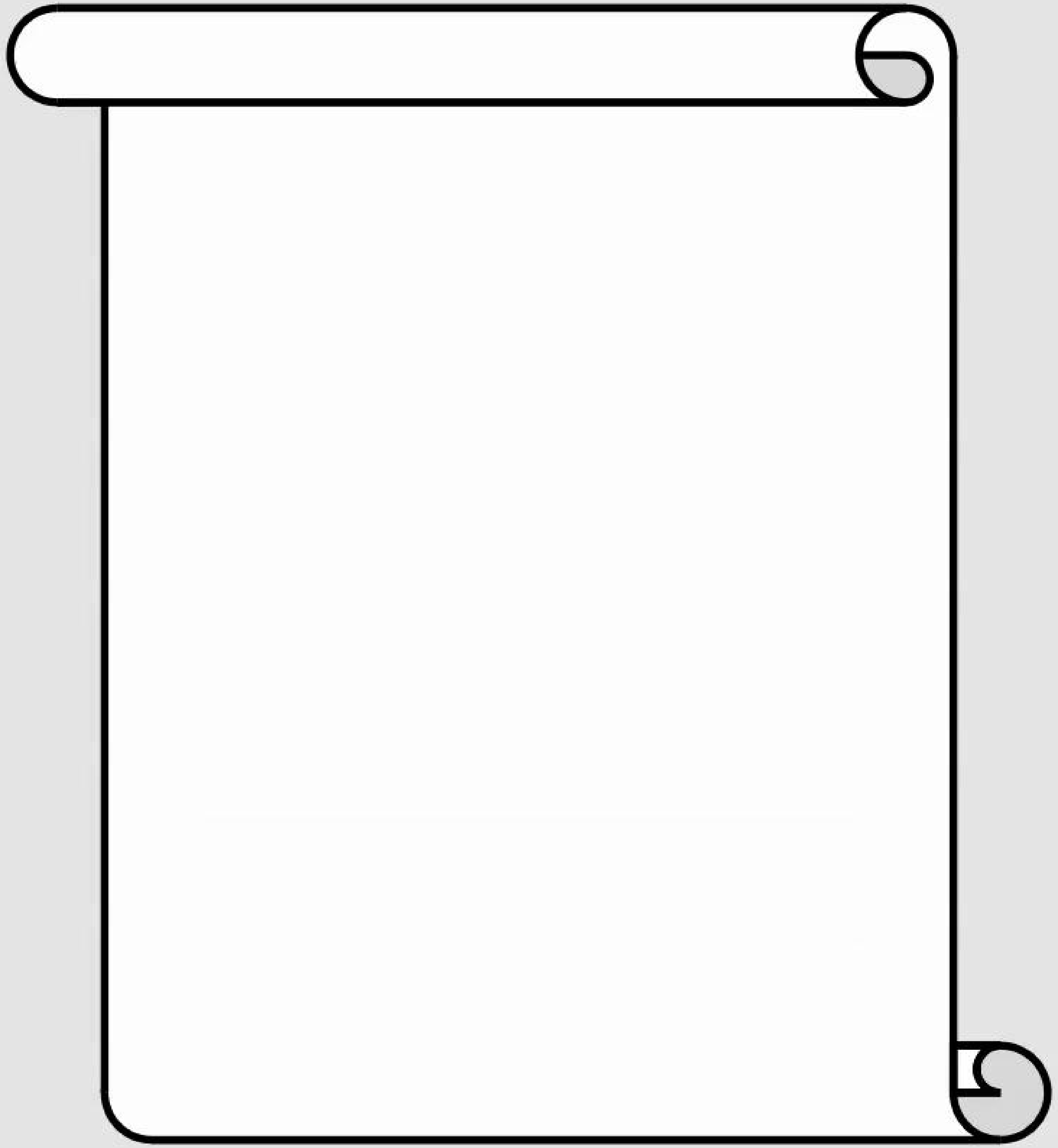
Slightly more convinced?



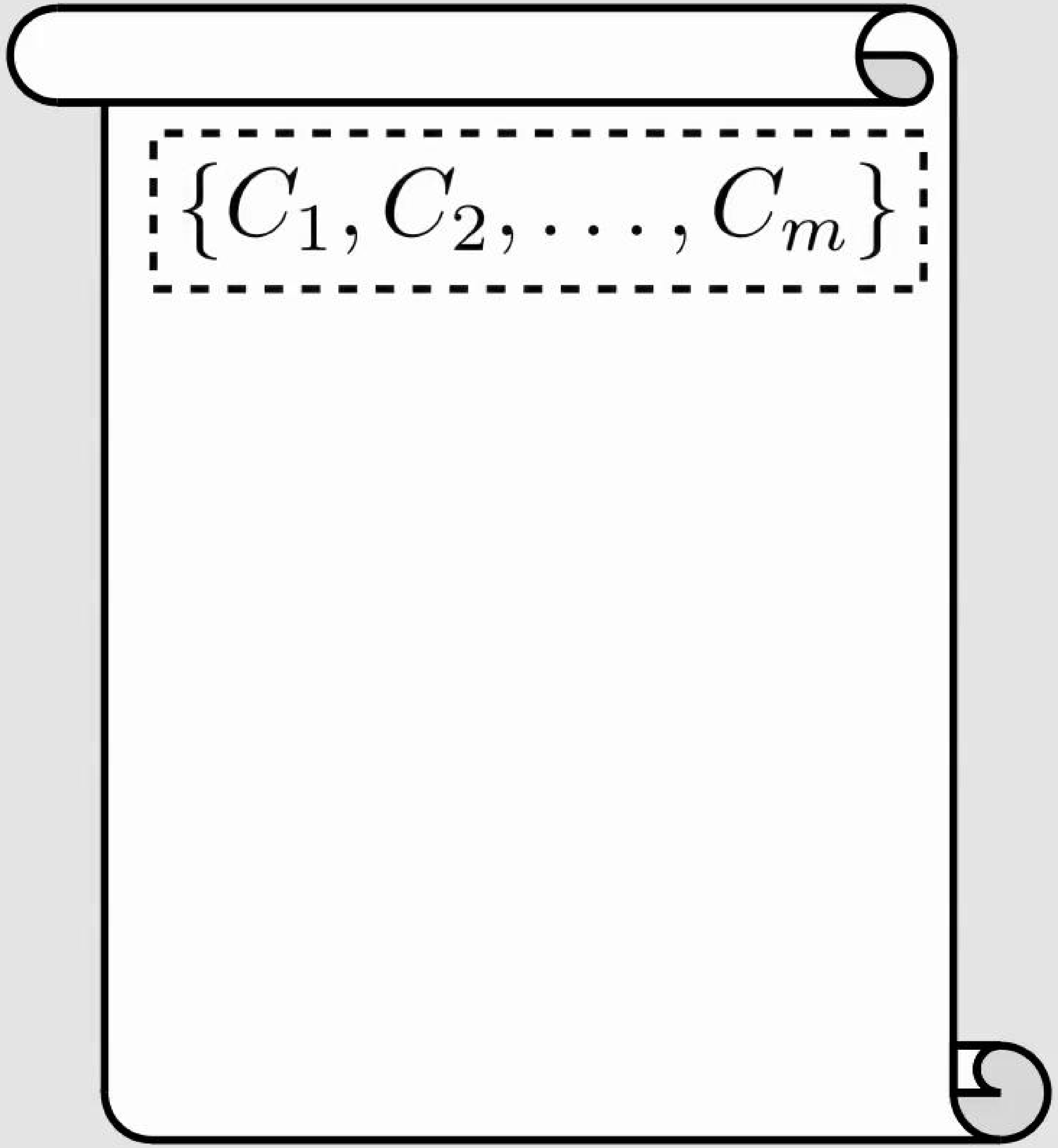

Proof


Proof

RUP



RUP



RUP

$$\boxed{\{C_1, C_2, \ldots, C_m\}}$$

(rule) D_1

(rule) D_2

$\vdots$

(rule) D_{i-1}

RUP

$$\{C_1, C_2, \ldots, C_m\}$$

(rule) D_1

(rule) D_2

$\vdots$

(rule) D_{i-1}

(RUP) D_i

$\{C_1, C_2, \ldots, C_m\}$

(rule) D_1

(rule) D_2

$\vdots$

(rule) D_{i-1}

(RUP) D_i

Checking Process:

$C_1 \wedge \ldots, \wedge C_m \wedge D_1, \ldots, D_m$

$$\{C_1, C_2, \ldots, C_m\}$$

(rule) D_1

(rule) D_2

$\vdots$

(rule) D_{i-1}

(RUP) D_i

Checking Process:

$$C_1 \wedge \ldots, \wedge C_m \wedge D_1, \ldots, D_m, \wedge \neg D_i$$

$\{C_1, C_2, \ldots, C_m\}$

(rule) D_1

(rule) D_2

$\vdots$

(rule) D_{i-1}

(RUP) D_i

Checking Process:

$$C_1 \wedge \ldots, \wedge C_m \wedge D_1, \ldots, D_m, \wedge \neg D_i$$

'Unit Propagation'

Contradiction

$\boxed{\{C_1, C_2, \ldots, C_m\}}$

(rule) D_1

(rule) D_2

$\vdots$

(rule) D_{i-1}

(RUP) D_i

Checking Process:

$C_1 \wedge \ldots, \wedge C_m \wedge D_1, \ldots, D_m, \wedge \neg D_i$

~~'Unit Propagation'~~

Contradiction

$\boxed{\{C_1, C_2, \ldots, C_m\}}$

(rule) D_1

(rule) D_2

$\vdots$

(rule) D_{i-1}

(RUP) D_i

Checking Process:

$$C_1 \wedge \ldots, \wedge C_m \wedge D_1, \ldots, D_m, \wedge \neg D_i$$

~~'Unit Propagation'~~

'Simple, Dumb Reasoning'

Contradiction

RUP

RUP

$$F \vdash C$$

RUP

$$(F \wedge \neg C) \vdash \bot$$

RUP

$$(F \wedge \neg C)$$

is always false

RUP

$$(\neg F \vee C)$$

is always true

RUP

$$F \implies C$$

The "Reverse Unit Propagation" Rule

$$F \implies C$$

A thing that is RUP:

A thing that is RUP:

$$x_{b0} + 2x_{b1} + 4x_{b2} \geq 6$$

A thing that is RUP:

$$x_{b0} + 2x_{b1} + 4x_{b2} \geq 6$$

$$\text{(RUP)}\ x_{b0} + 2x_{b1} + 4x_{b2} \geq 3$$

A thing that is RUP:

$$x_{b0} + 2x_{b1} + 4x_{b2} \geq 6$$

(RUP) $x_{b0} + 2x_{b1} + 4x_{b2} \geq 3$

Checking Process:

$$x_{b0} + 2x_{b1} + 4x_{b2} \geq 6$$

$$-x_{b0} - 2x_{b1} - 4x_{b2} \geq -2$$

A thing that is RUP:

$$x_{b0} + 2x_{b1} + 4x_{b2} \geq 6$$

(RUP) $x_{b0} + 2x_{b1} + 4x_{b2} \geq 3$

Checking Process:

$$x_{b0} + 2x_{b1} \geq 2$$

$$-x_{b0} - 2x_{b1} - 4x_{b2} \geq -2$$

A thing that is RUP:

$$x_{b0} + 2x_{b1} + 4x_{b2} \geq 6$$

$$\text{(RUP)}\ x_{b0} + 2x_{b1} + 4x_{b2} \geq 3$$

Checking Process:

$$x_{b0} + 2x_{b1} \geq 2$$

$$-x_{b0} - 2x_{b1} \geq 2$$

A thing that is RUP:

$$x_{b0} + 2x_{b1} + 4x_{b2} \geq 6$$

(RUP) $x_{b0} + 2x_{b1} + 4x_{b2} \geq 3$

Checking Process:

$$x_{b0} + 2x_{b1} \geq 2$$

$$-x_{b0} - 2x_{b1} \geq 2$$

A thing that is RUP:

$$x_{b0} + 2x_{b1} + 4x_{b2} \geq 6$$

$$\text{(RUP)}\ x_{b0} + 2x_{b1} + 4x_{b2} \geq 3 \quad ✓$$

Checking Process:

$$x_{b0} + 2x_{b1} \geq 2$$

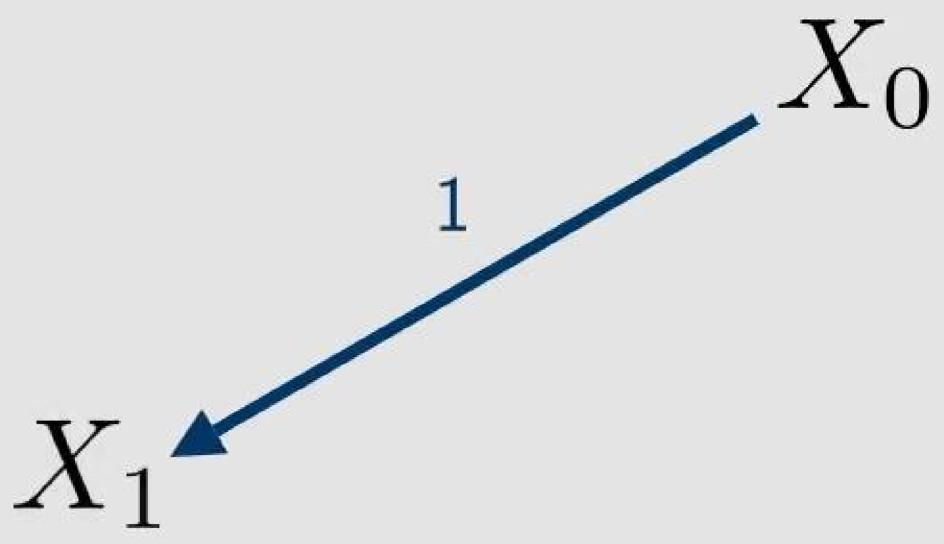
$$-x_{b0} - 2x_{b1} \geq 2$$

Proof Logging Backtracking Search

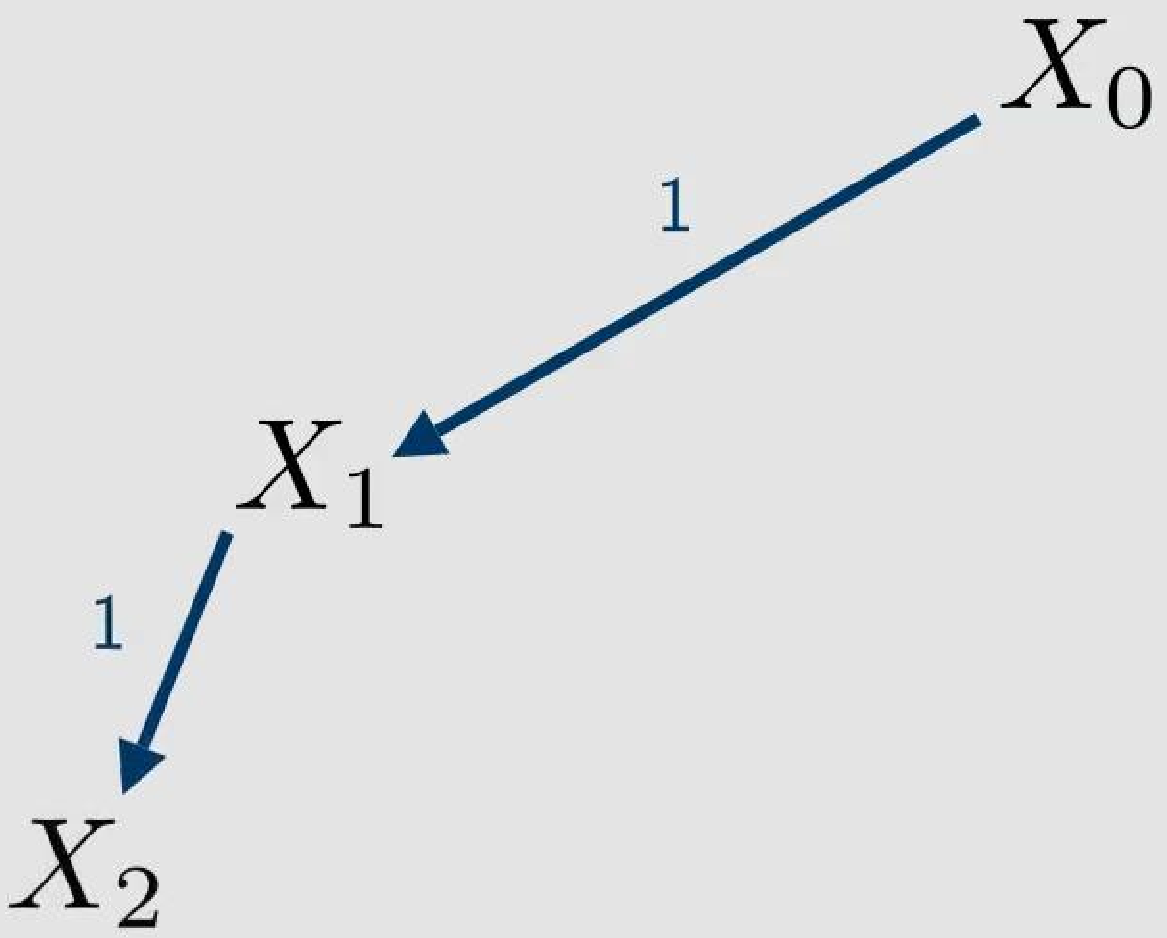
Proof Logging Backtracking Search

$$X_0$$

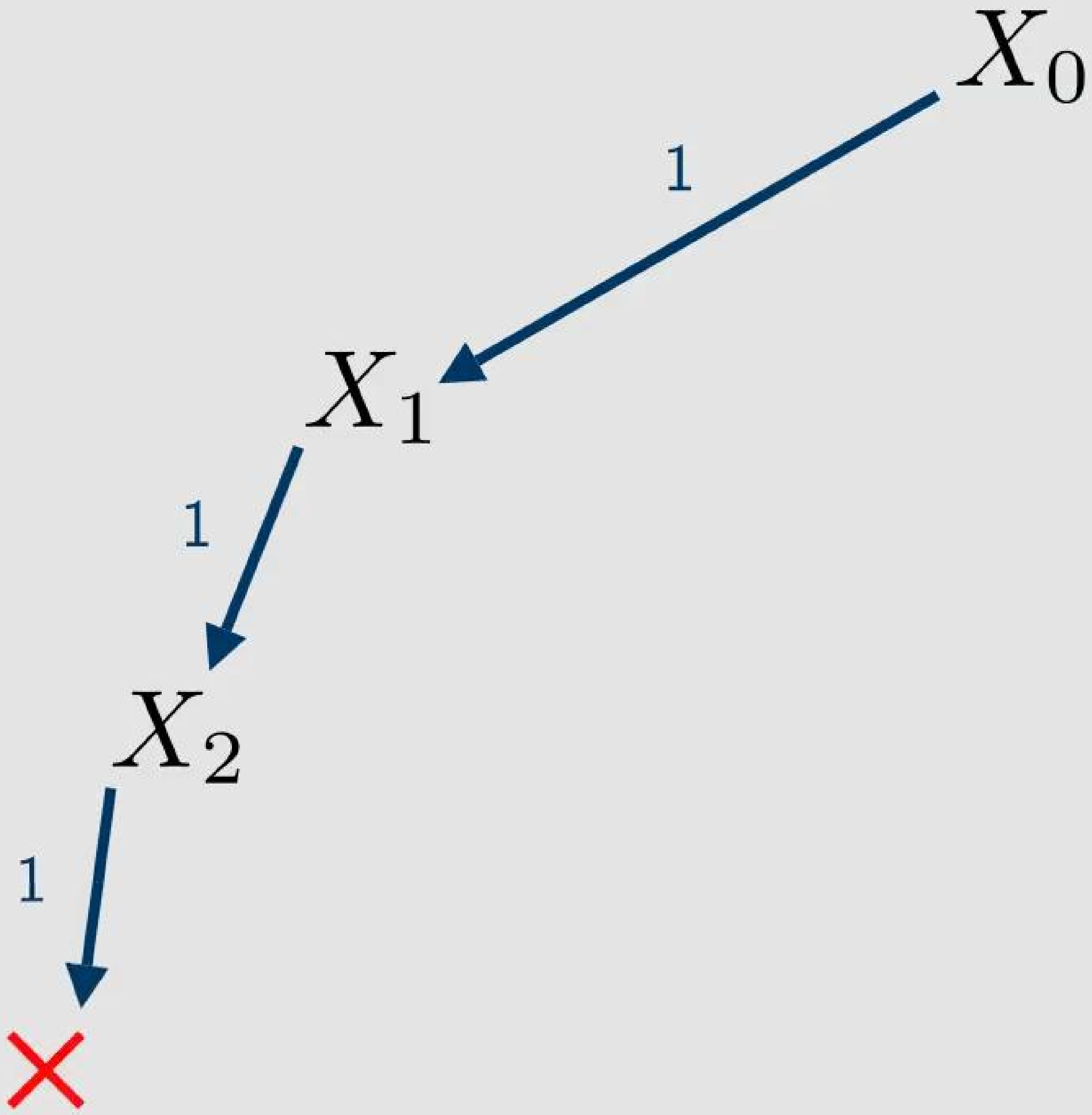
Proof Logging Backtracking Search



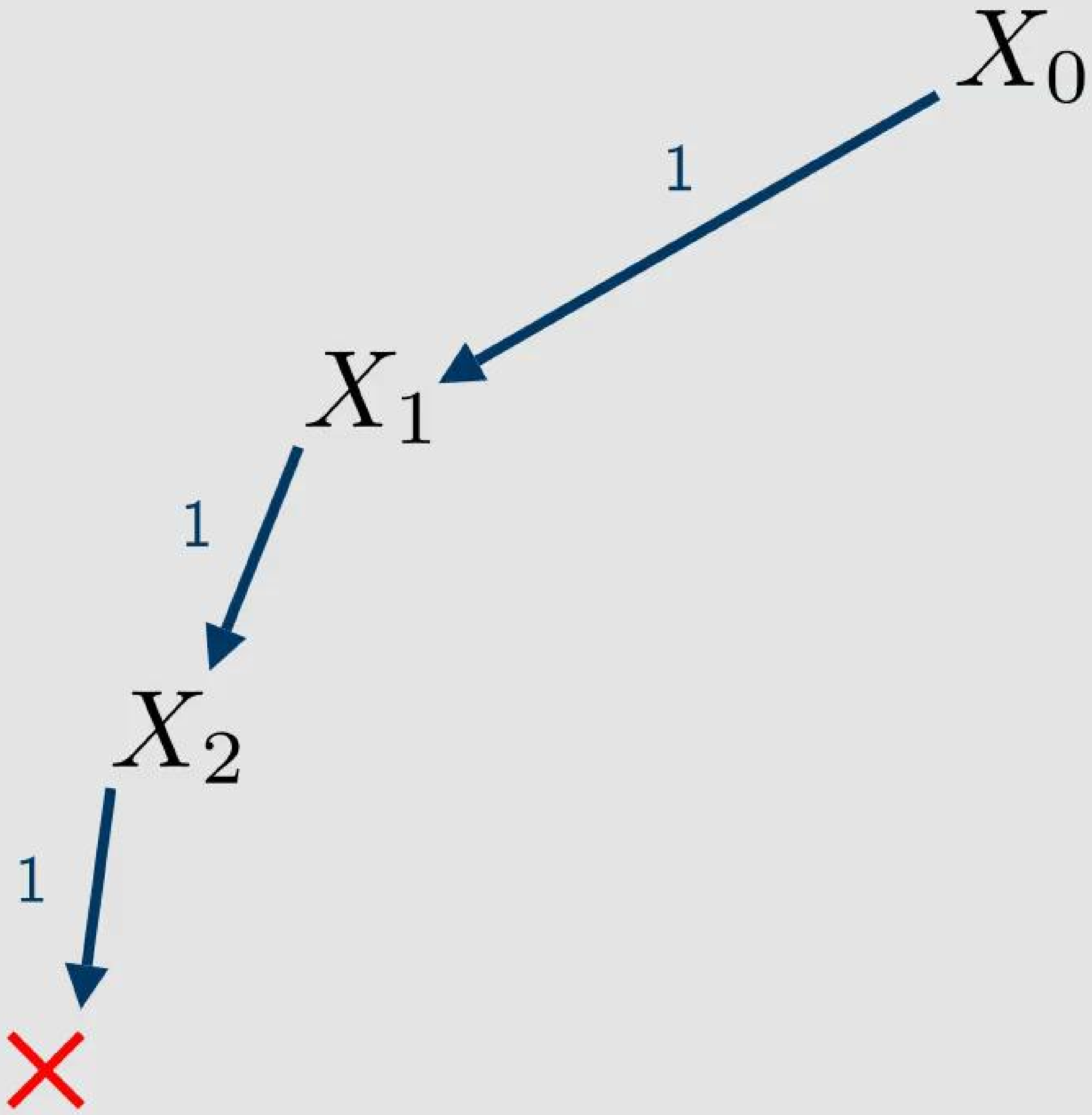
Proof Logging Backtracking Search

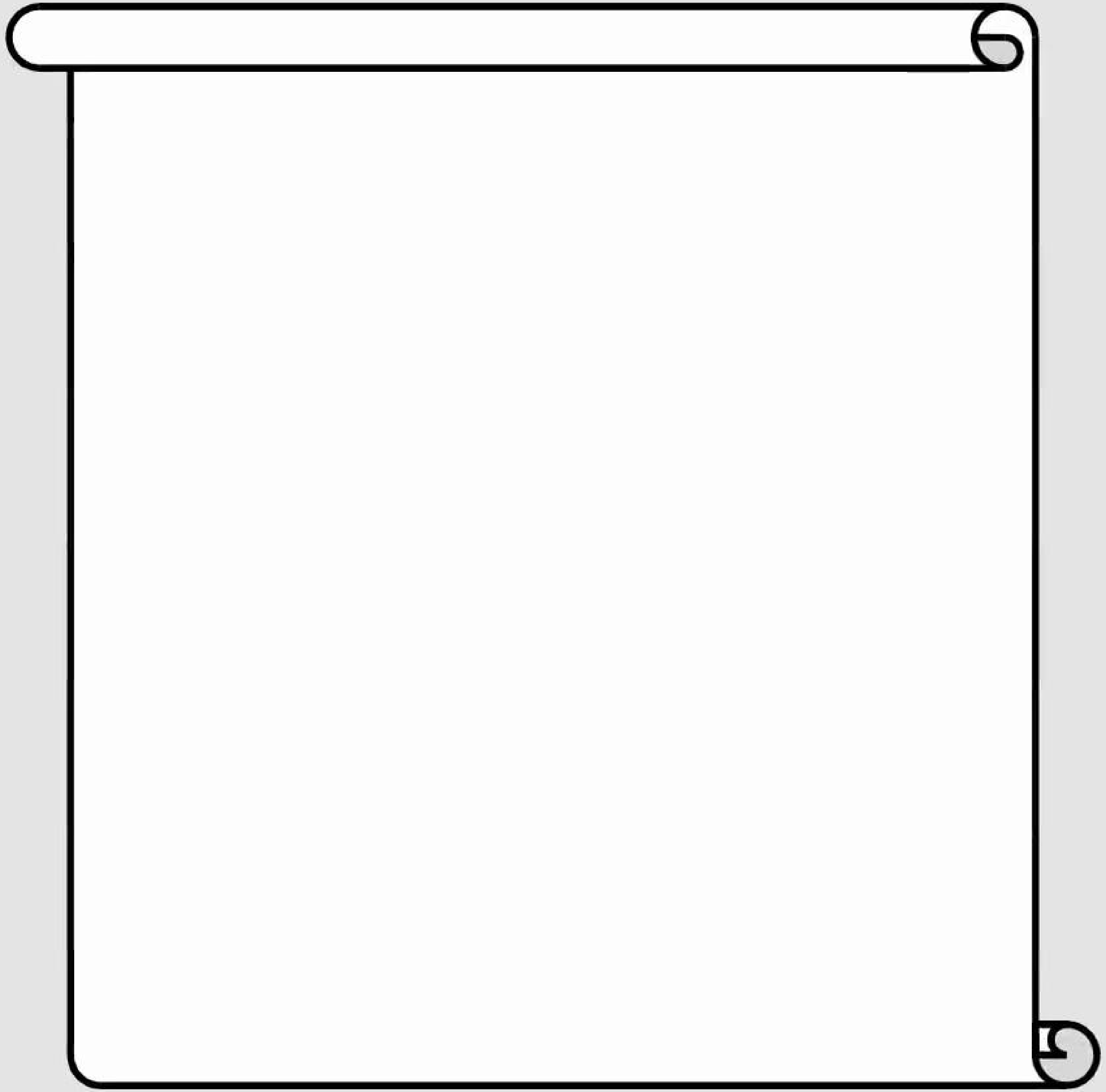


Proof Logging Backtracking Search

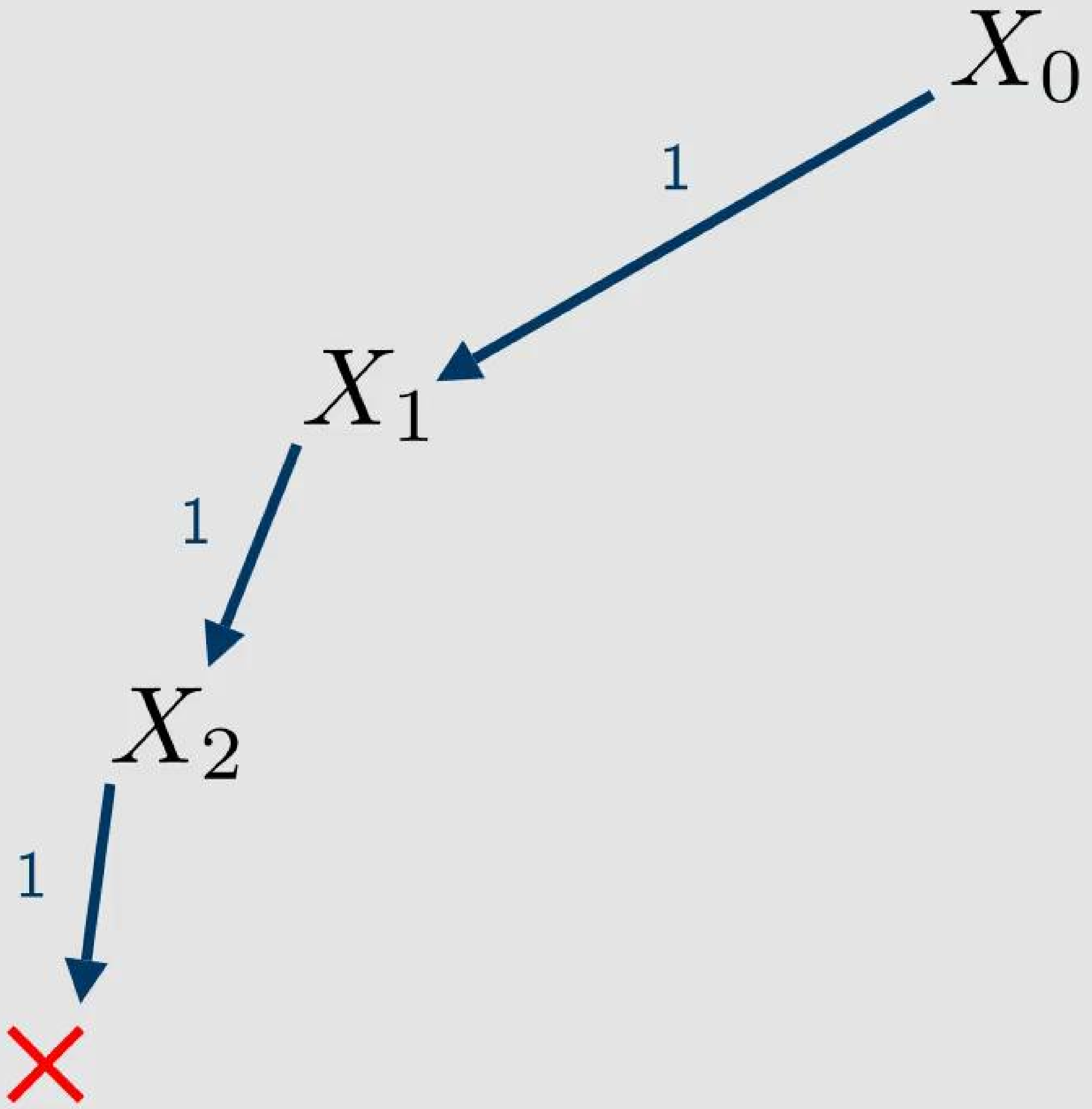


Proof Logging Backtracking Search



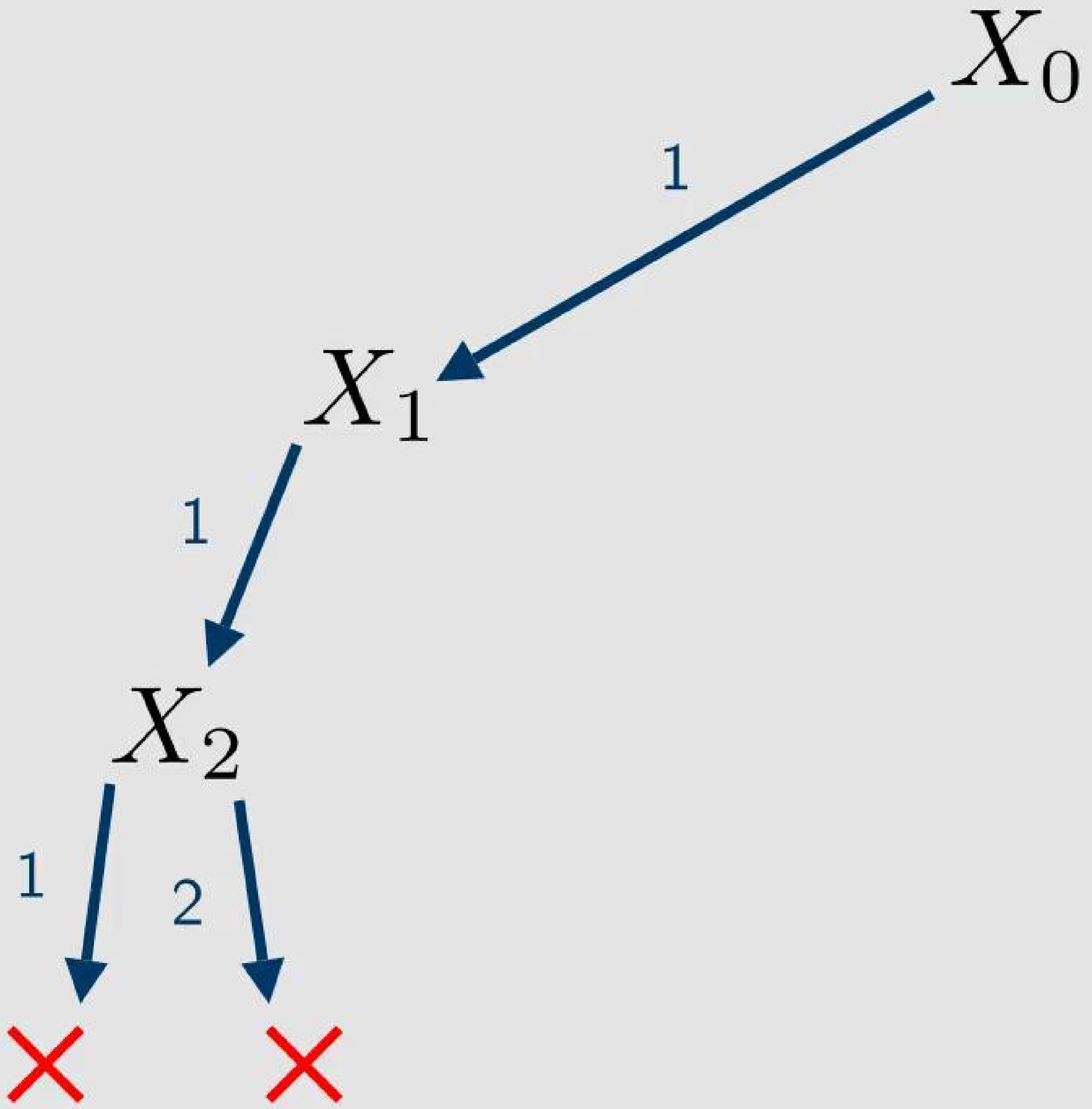


Proof Logging Backtracking Search



(RUP) $\bar{x}_{0=1} + \bar{x}_{1=1} + \bar{x}_{2=1} \geq 1$

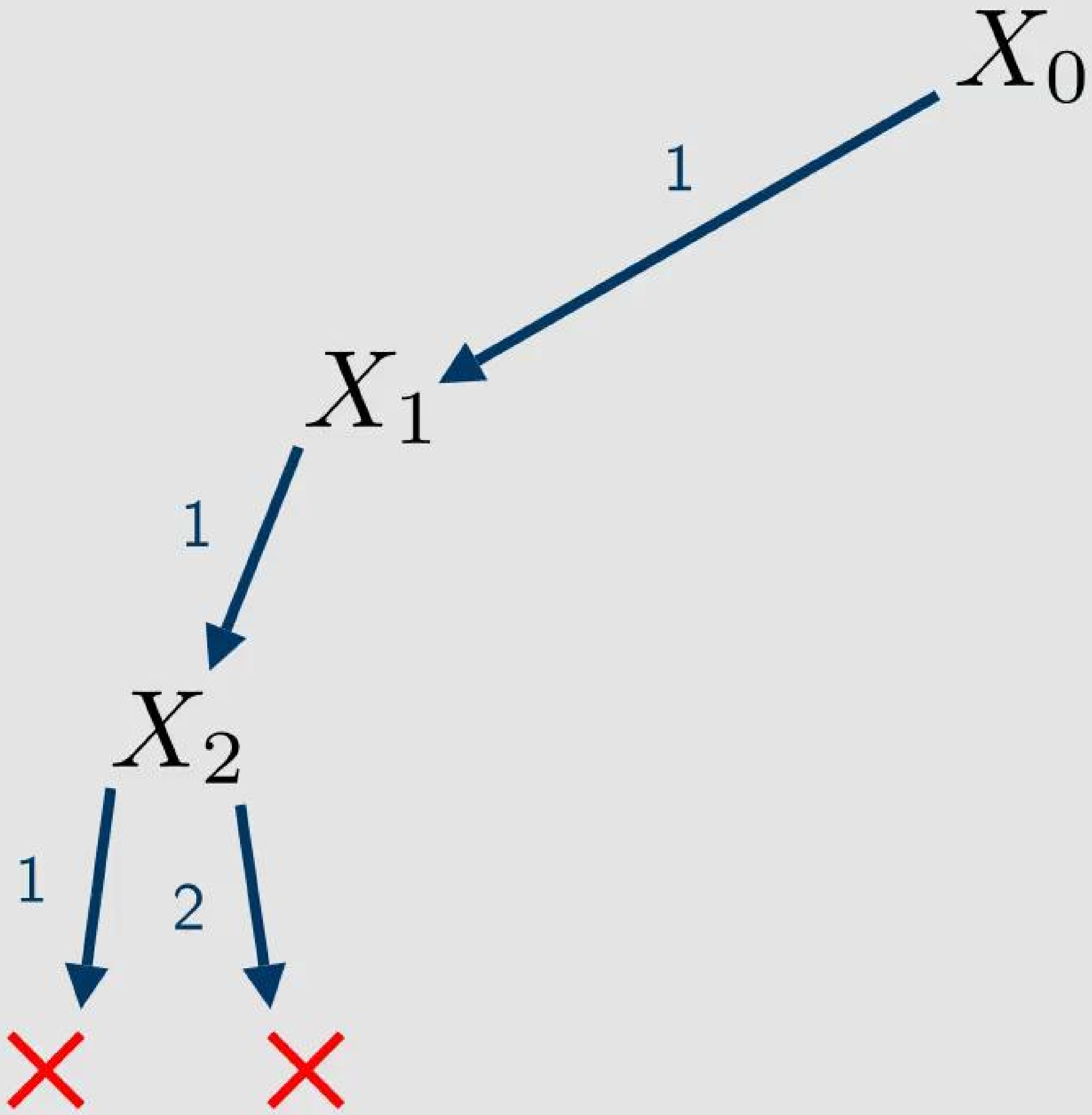
Proof Logging Backtracking Search



(RUP) $\bar{x}_{0=1} + \bar{x}_{1=1} + \bar{x}_{2=1} \geq 1$

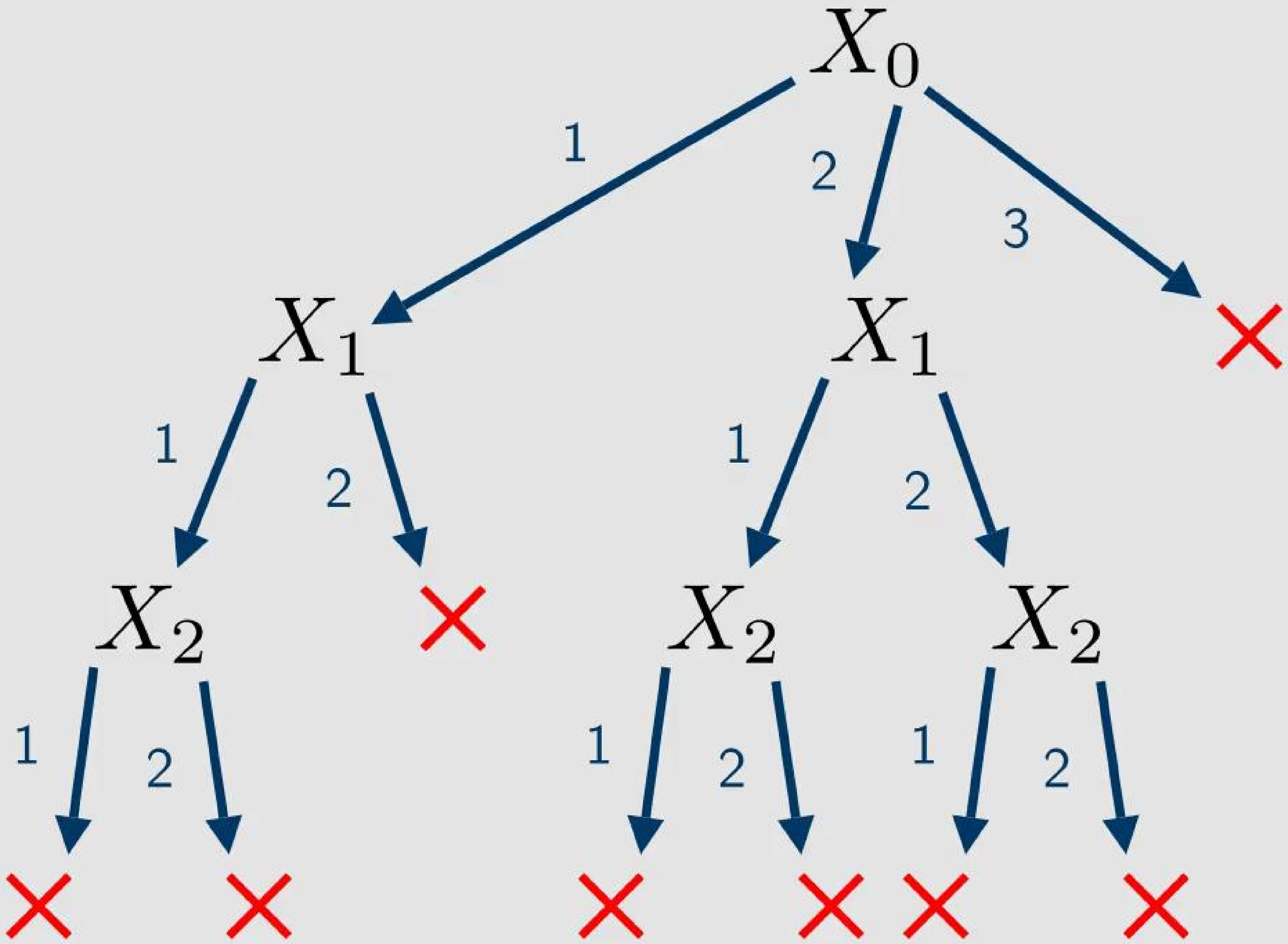
(RUP) $\bar{x}_{0=1} + \bar{x}_{1=1} + \bar{x}_{2=2} \geq 1$

Proof Logging Backtracking Search



(RUP) $\bar{x}_{0=1} + \bar{x}_{1=1} + \bar{x}_{2=1} \geq 1$
(RUP) $\bar{x}_{0=1} + \bar{x}_{1=1} + \bar{x}_{2=2} \geq 1$
(RUP) $\bar{x}_{0=1} + \bar{x}_{1=1} \geq 1$

Proof Logging Backtracking Search



(RUP) $\bar{x}_{0=1} + \bar{x}_{1=1} + \bar{x}_{2=1} \geq 1$

(RUP) $\bar{x}_{0=1} + \bar{x}_{1=1} + \bar{x}_{2=2} \geq 1$

(RUP) $\bar{x}_{0=1} + \bar{x}_{1=1} \geq 1$

(RUP) $\bar{x}_{0=1} + \bar{x}_{1=2} \geq 1$

(RUP) $\bar{x}_{0=1} \geq 1$

(RUP) $\bar{x}_{0=2} + \bar{x}_{1=1} + \bar{x}_{2=1} \geq 1$

(RUP) $\bar{x}_{0=2} + \bar{x}_{1=1} + \bar{x}_{2=2} \geq 1$

(RUP) $\bar{x}_{0=2} + \bar{x}_{1=1} \geq 1$

(RUP) $\bar{x}_{0=2} + \bar{x}_{1=2} + \bar{x}_{2=1} \geq 1$

(RUP) $\bar{x}_{0=2} + \bar{x}_{1=2} + \bar{x}_{2=2} \geq 1$

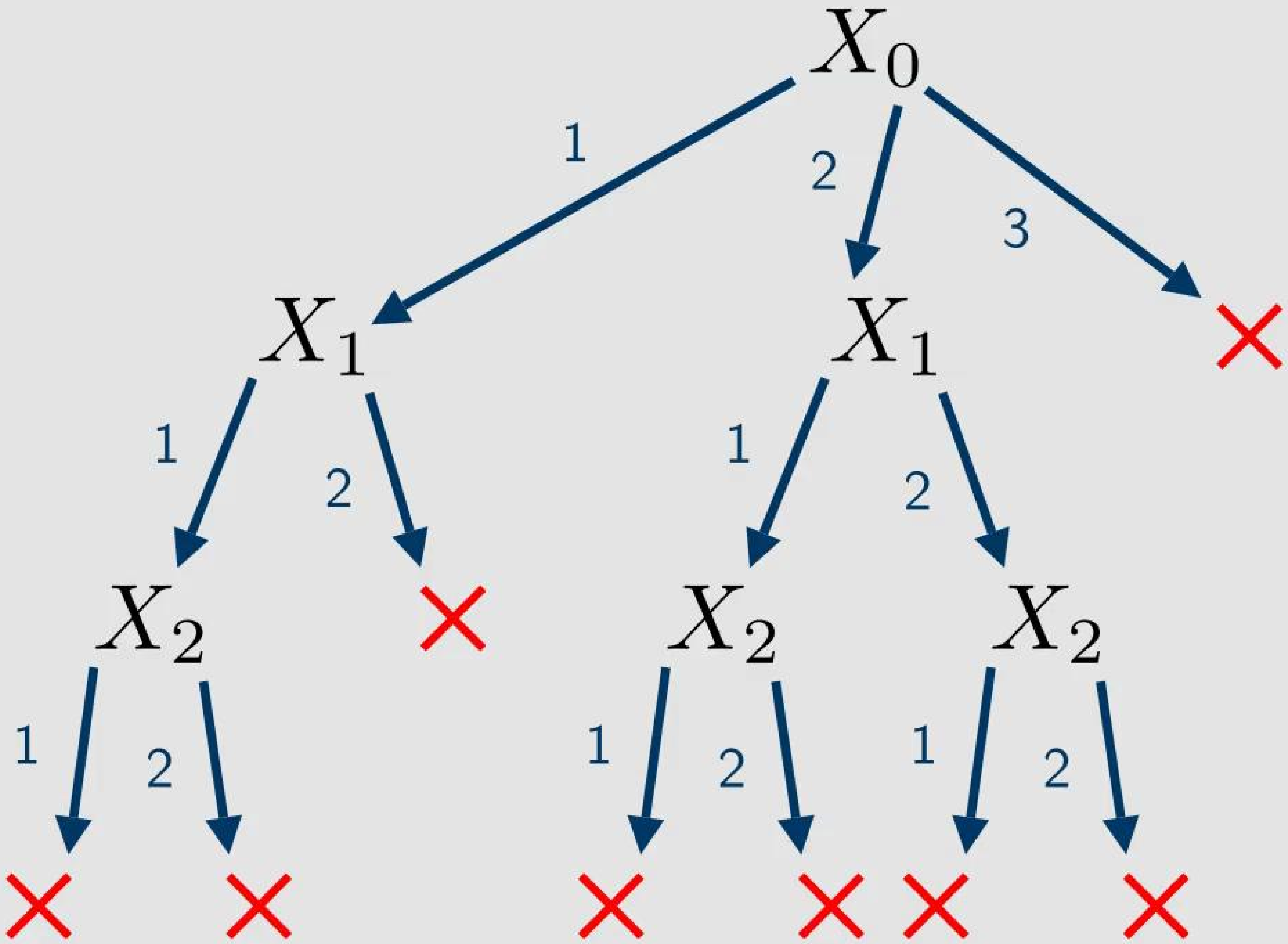
(RUP) $\bar{x}_{0=2} + \bar{x}_{1=2} \geq 1$

(RUP) $\bar{x}_{0=2} \geq 1$

(RUP) $\bar{x}_{0=3} \geq 1$

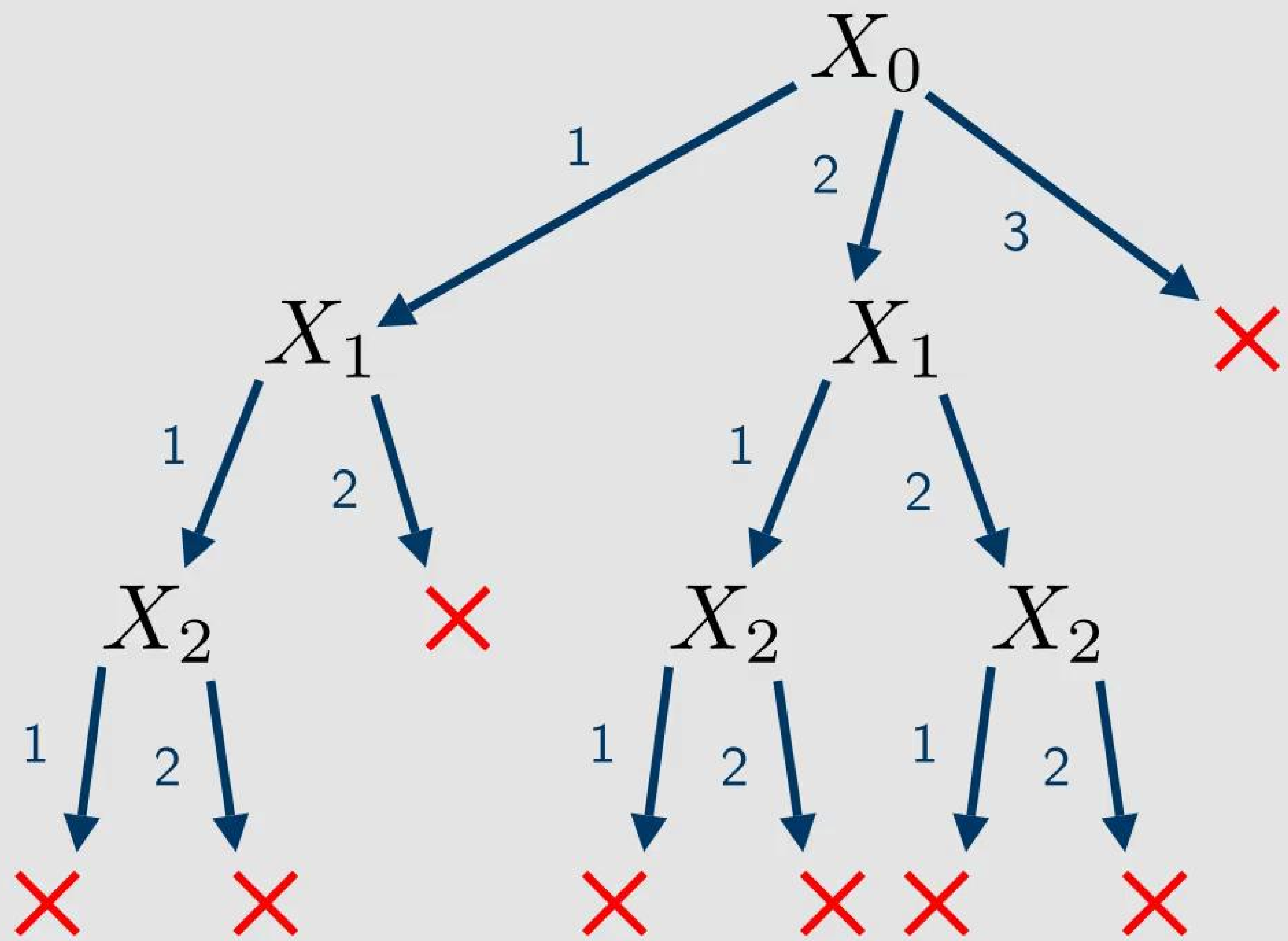
(RUP) $\perp$

Proof Logging Backtracking Search




```
rup 1 x0e1 + 1 x1e1 + 1 x2e1 >= 1 ;
rup 1 x0e1 + 1 x1e1 + 1 x2e2 >= 1 ;
rup 1 x0e1 + 1 x1e1 >= 1 ;
rup 1 x0e1 + 1 x1e2 >= 1 ;
rup 1 x0e1 >= 1 ;
rup 1 x0e2 + 1 x1e1 + 1 x2e1 >= 1 ;
rup 1 x0e2 + 1 x1e1 + 1 x2e2 >= 1 ;
rup 1 x0e2 + 1 x1e1 >= 1 ;
rup 1 x0e2 + 1 x1e2 + 1 x2e1 >= 1 ;
rup 1 x0e2 + 1 x1e2 + 1 x2e2 >= 1 ;
rup 1 x0e2 + 1 x1e2 >= 1 ;
rup 1 x0e2 >= 1 ;
rup 1 x0e3 >= 1 ;
rup 0 >= 1 ;
```

Proof Logging Backtracking Search




```
...?
rup 1 x0e1 + 1 x1e1 + 1 x2e1 >= 1 ;
...?
rup 1 x0e1 + 1 x1e1 + 1 x2e2 >= 1 ;
...?
rup 1 x0e1 + 1 x1e1 >= 1 ;
...?
rup 1 x0e1 + 1 x1e2 >= 1 ;
...?
rup 1 x0e1 >= 1 ;
...?
rup 1 x0e2 + 1 x1e1 + 1 x2e1 >= 1 ;
...?
rup 1 x0e2 + 1 x1e1 + 1 x2e2 >= 1 ;
...?
rup 1 x0e2 + 1 x1e1 >= 1 ;
...?
rup 1 x0e2 + 1 x1e2 + 1 x2e1 >= 1 ;
...?
rup 1 x0e2 + 1 x1e2 + 1 x2e2 >= 1 ;
...?
rup 1 x0e2 + 1 x1e2 >= 1 ;
...?
rup 1 x0e2 >= 1 ;
...?
rup 1 x0e3 >= 1 ;
...?
rup 0 >= 1 ;
...?
```

How do we define all those variables?

How do we define all those variables?

$$x_{\geq 3} \Leftrightarrow bits(X) \geq 3$$

$$x_{\leq 3} \Leftrightarrow -bits(X) \geq -3$$

$$x_{=3} \Leftrightarrow x_{\geq 3} + x_{\leq 3} \geq 2$$

How do we define all those variables?

$$\text{(RED)} \quad x_{\geq 3} \Leftrightarrow bits(X) \geq 3$$

$$\text{(RED)} \quad x_{\leq 3} \Leftrightarrow -bits(X) \geq -3$$

$$\text{(RED)} \quad x_{=3} \Leftrightarrow x_{\geq 3} + x_{\leq 3} \geq 2$$

How do we define all those variables?

$$
\begin{aligned}
&(\text{RED}) \quad x_{\geq 3} \Leftrightarrow bits(X) \geq 3 \\
&(\text{RED}) \quad x_{\leq 3} \Leftrightarrow -bits(X) \geq -3 \\
&(\text{RED}) \quad x_{=3} \Leftrightarrow x_{\geq 3} + x_{\leq 3} \geq 2
\end{aligned}
$$

Redundance-Based Strengthening

How do we define all those variables?

$$\begin{array}{ll} \text{(RED)} & x_{\geq 3} \Leftrightarrow bits(X) \geq 3 \\ \text{(RED)} & x_{\leq 3} \Leftrightarrow -bits(X) \geq -3 \\ \text{(RED)} & x_{=3} \Leftrightarrow x_{\geq 3} + x_{\leq 3} \geq 2 \end{array}$$

~~Redundance-Based Strengthening~~

How do we define all those variables?

$$\text{(RED)} \quad x_{\geq 3} \Leftrightarrow bits(X) \geq 3$$

$$\text{(RED)} \quad x_{\leq 3} \Leftrightarrow -bits(X) \geq -3$$

$$\text{(RED)} \quad x_{=3} \Leftrightarrow x_{\geq 3} + x_{\leq 3} \geq 2$$

~~Redundance-Based Strengthening~~

Rule that lets us introduce reified constraints on fresh variables :-)

We need to log justifications every time we infer something.

We need to log justifications every time we infer something.

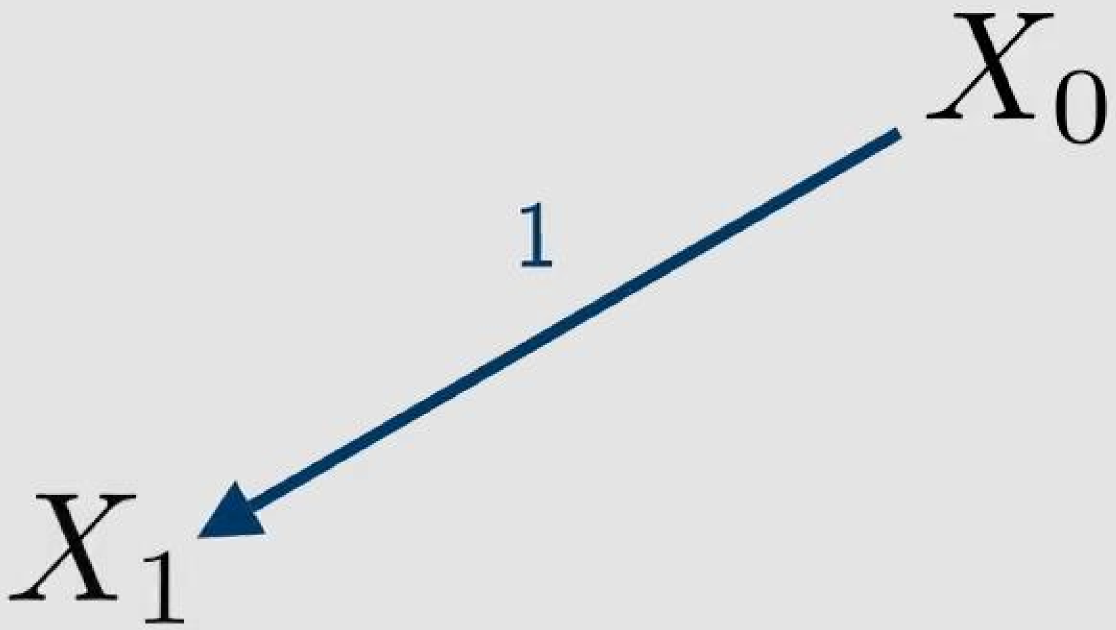
$$\text{reason} \Rightarrow \text{inference}$$

We need to log justifications every time we infer something.

$$\text{reason} \Rightarrow \text{inference}$$

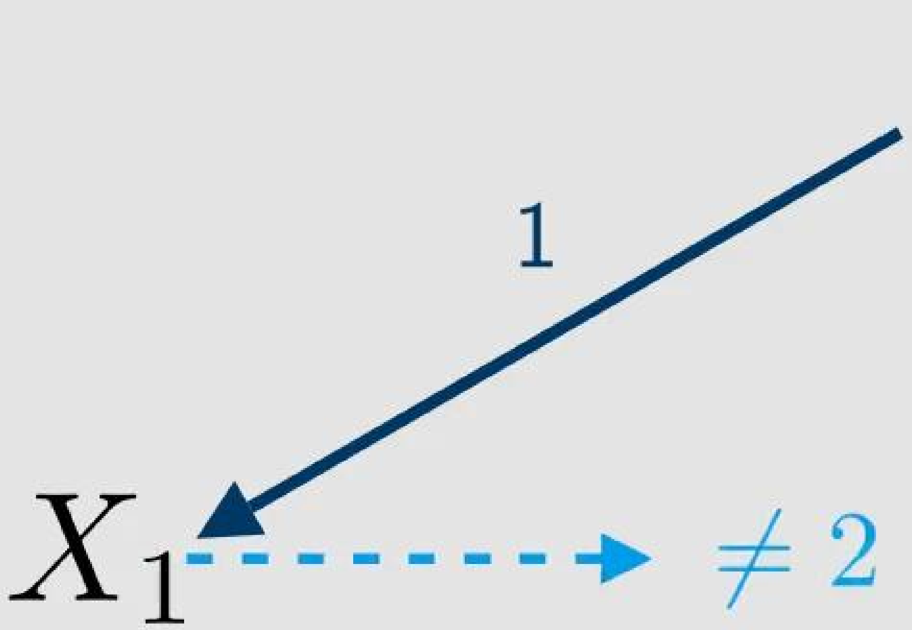
We need to log justifications every time we infer something.

$$\text{reason} \Rightarrow \text{inference}$$



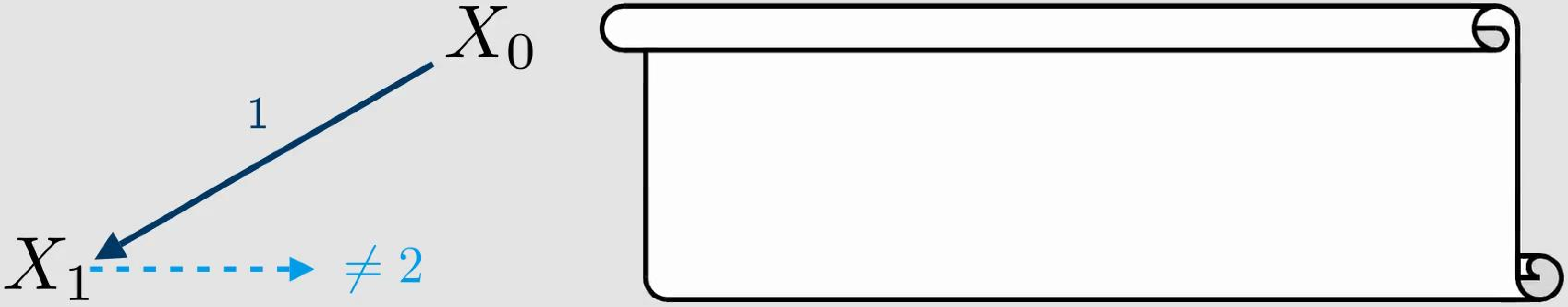
We need to log justifications every time we infer something.

$$\text{reason} \Rightarrow \text{inference}$$



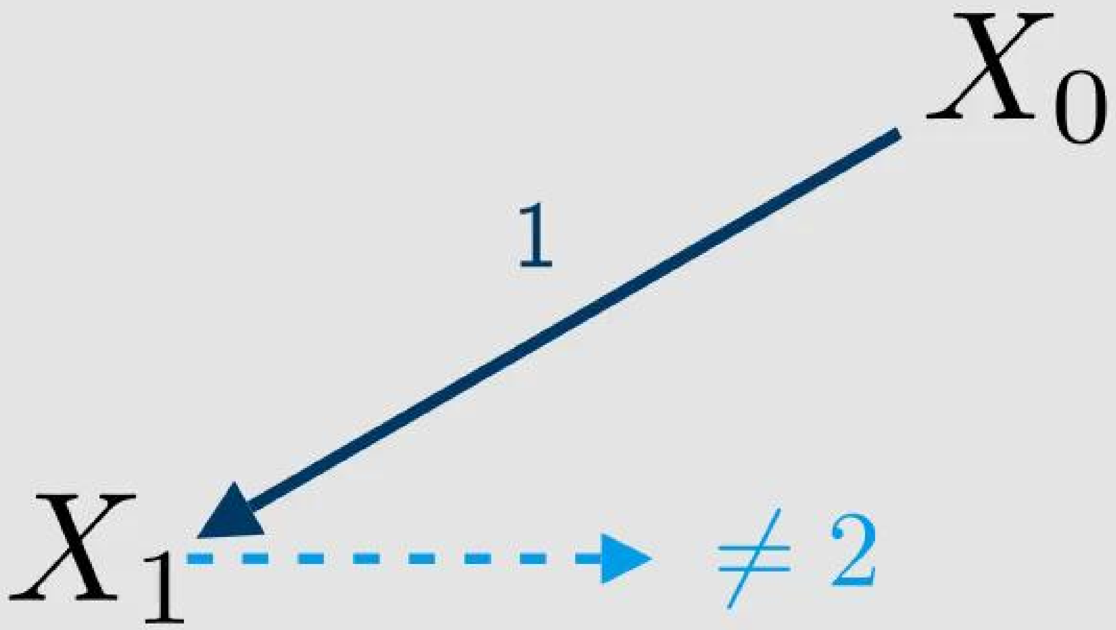
We need to log justifications every time we infer something.

$$\text{reason} \Rightarrow \text{inference}$$



We need to log justifications every time we infer something.

$$\text{reason} \Rightarrow \text{inference}$$



Want to derive:

$$x_{0=1} \Rightarrow \bar{x}_{1=2} \geq 1$$

Up to this point (the 'rules of the game')

Up to this point (the 'rules of the game')

- Use RED/reification to introduce CP literals

Up to this point (the 'rules of the game')

- Use RED/reification to introduce CP literals
- Write a RUP step at every backtrack

Up to this point (the 'rules of the game')

- Use RED/reification to introduce CP literals
- Write a RUP step at every backtrack
- (Also log solutions/bounds if proving optimality)

Up to this point (the 'rules of the game')

- Use RED/reification to introduce CP literals
- Write a RUP step at every backtrack
- (Also log solutions/bounds if proving optimality)
- Interleave derived 'justifications' to account for constraint propagation

Up to this point (the 'rules of the game')

- Use RED/reification to introduce CP literals
- Write a RUP step at every backtrack
- (Also log solutions/bounds if proving optimality)
- Interleave derived 'justifications' to account for constraint propagation

Some constraints are really easy

Some constraints are really easy

e.g. $X \neq Y, \quad X \in \{1, \ldots, 15\}, Y \in \{5\}$

Some constraints are really easy

e.g. $X \neq Y, \quad X \in \{1, \ldots, 15\}, Y \in \{5\}$

..hence $X \neq 5$

Some constraints are really easy

$$\text{e.g. } X \neq Y, \quad X \in \{1, \ldots, 15\}, Y \in \{5\}$$

$$\text{..hence } X \neq 5$$

(justify?) $y_{=5} \Rightarrow \bar{x}_{=5} \geq 1$

Some constraints are really easy

e.g. $X \neq Y, \quad X \in \{1, \ldots, 15\}, Y \in \{5\}$

..hence $X \neq 5$

(RED) $y_{\geq 5} \Leftrightarrow y_{b0} + 2y_{b1} + 4y_{b2} \geq 5$

(RED) $y_{\leq 5} \Leftrightarrow -y_{b0} - 2y_{b1} - 4y_{b2} \geq -5$

(RED) $y_{=5} \Leftrightarrow y_{\geq 5} + y_{\leq 5} \geq 2$

(justify?) $y_{=5} \Rightarrow \bar{x}_{=5} \geq 1$

Some constraints are really easy

e.g. $X \neq Y, \quad X \in \{1, \ldots, 15\}, Y \in \{5\}$

..hence $X \neq 5$

(RED) $y_{\geq 5} \Leftrightarrow y_{b0} + 2y_{b1} + 4y_{b2} \geq 5$

(RED) $y_{\leq 5} \Leftrightarrow -y_{b0} - 2y_{b1} - 4y_{b2} \geq -5$

(RED) $y_{=5} \Leftrightarrow y_{\geq 5} + y_{\leq 5} \geq 2$

(RED) $x_{\geq 5} \Leftrightarrow x_{b0} + 2x_{b1} + 4x_{b2} \geq 5$

(RED) $x_{\leq 5} \Leftrightarrow -x_{b0} - 2x_{b1} - 4x_{b2} \geq -5$

(RED) $x_{=5} \Leftrightarrow x_{\geq 5} + x_{\leq 5} \geq 2$

(justify?) $y_{=5} \Rightarrow \bar{x}_{=5} \geq 1$

Some constraints are really easy

e.g. $X \neq Y, \quad X \in \{1, \ldots, 15\}, Y \in \{5\}$

..hence $X \neq 5$

(RED) $y_{\geq 5} \Leftrightarrow y_{b0} + 2y_{b1} + 4y_{b2} \geq 5$

(RED) $y_{\leq 5} \Leftrightarrow -y_{b0} - 2y_{b1} - 4y_{b2} \geq -5$

(RED) $y_{=5} \Leftrightarrow y_{\geq 5} + y_{\leq 5} \geq 2$

(RED) $x_{\geq 5} \Leftrightarrow x_{b0} + 2x_{b1} + 4x_{b2} \geq 5$

(RED) $x_{\leq 5} \Leftrightarrow -x_{b0} - 2x_{b1} - 4x_{b2} \geq -5$

(RED) $x_{=5} \Leftrightarrow x_{\geq 5} + x_{\leq 5} \geq 2$

(RUP) $y_{=5} \Rightarrow \bar{x}_{=5} \geq 1$

Other constraints will need more than just RUP

Other constraints will need more than just RUP

$$2X + 3Y + 4Z \leq 42$$

Other constraints will need more than just RUP

$$2X + 3Y + 4Z \leq 42$$

$$X \geq 5 \land Z \geq 3 \Rightarrow Y \leq 6$$

Other constraints will need more than just RUP

$$2X + 3Y + 4Z \leq 42$$

$$X \geq 5 \land Z \geq 3 \Rightarrow Y \leq 6$$

Other constraints will need more than just RUP

$$2X + 3Y + 4Z \leq 42$$

$$X \geq 5 \wedge Z \geq 3 \Rightarrow Y \leq 6$$

(RED)

Other constraints will need more than just RUP

$$2X + 3Y + 4Z \leq 42$$

$$X \geq 5 \land Z \geq 3 \Rightarrow Y \leq 6$$

(RED) $\quad x_{\geq 5} \Rightarrow x_{b0} + 2x_{b1} + 4x_{b2} + 8x_{b3} \geq 5$

Other constraints will need more than just RUP

$$2X + 3Y + 4Z \leq 42$$

$$X \geq 5 \land Z \geq 3 \Rightarrow Y \leq 6$$

(RED) $x_{\geq 5} \Rightarrow x_{b0} + 2x_{b1} + 4x_{b2} + 8x_{b3} \geq 5$

(RED)

Other constraints will need more than just RUP

$$2X + 3Y + 4Z \leq 42$$

$$X \geq 5 \wedge Z \geq 3 \Rightarrow Y \leq 6$$

(RED) $x_{\geq 5} \Rightarrow x_{b0} + 2x_{b1} + 4x_{b2} + 8x_{b3} \geq 5$

(RED) $z_{\geq 3} \Rightarrow z_{b0} + 2z_{b1} + 4z_{b2} + 8z_{b3} \geq 3$

Other constraints will need more than just RUP

$$2X + 3Y + 4Z \leq 42$$

$$X \geq 5 \wedge Z \geq 3 \Rightarrow Y \leq 6$$

(RED) $x_{\geq 5} \Rightarrow x_{b0} + 2x_{b1} + 4x_{b2} + 8x_{b3} \geq 5$

(RED) $z_{\geq 3} \Rightarrow z_{b0} + 2z_{b1} + 4z_{b2} + 8z_{b3} \geq 3$

(RED)

Other constraints will need more than just RUP

$$2X + 3Y + 4Z \leq 42$$

$$X \geq 5 \land Z \geq 3 \Rightarrow Y \leq 6$$

(RED) $x_{\geq 5} \Rightarrow x_{b0} + 2x_{b1} + 4x_{b2} + 8x_{b3} \geq 5$

(RED) $z_{\geq 3} \Rightarrow z_{b0} + 2z_{b1} + 4z_{b2} + 8z_{b3} \geq 3$

(RED) $\overline{y_{\leq 6}} \Rightarrow y_{b0} + 2y_{b1} + 4x_{b2} + 8x_{b3} \geq 7$

Other constraints will need more than just RUP

$$2X + 3Y + 4Z \leq 42$$

$$X \geq 5 \land Z \geq 3 \Rightarrow Y \leq 6$$

(RED) $x_{\geq 5} \Rightarrow x_{b0} + 2x_{b1} + 4x_{b2} + 8x_{b3} \geq 5$

(RED) $z_{\geq 3} \Rightarrow z_{b0} + 2z_{b1} + 4z_{b2} + 8z_{b3} \geq 3$

(RED) $\overline{y_{\leq 6}} \Rightarrow y_{b0} + 2y_{b1} + 4x_{b2} + 8x_{b3} \geq 7$

(Axiom)

Other constraints will need more than just RUP

$$2X + 3Y + 4Z \leq 42$$

$$X \geq 5 \wedge Z \geq 3 \Rightarrow Y \leq 6$$

$$\begin{array}{ll}
\text{(RED)} & x_{\geq 5} \Rightarrow x_{b0} + 2x_{b1} + 4x_{b2} + 8x_{b3} \geq 5 \\
\text{(RED)} & z_{\geq 3} \Rightarrow z_{b0} + 2z_{b1} + 4z_{b2} + 8z_{b3} \geq 3 \\
\text{(RED)} & \overline{y_{\leq 6}} \Rightarrow y_{b0} + 2y_{b1} + 4x_{b2} + 8x_{b3} \geq 7 \\
 & -2x_{b0} - 4x_{b1} - 8x_{b2} - 16x_{b3} \\
\text{(Axiom)} & -3y_{b0} - 6y_{b1} - 12y_{b2} - 24y_{b3} \\
 & -4z_{b0} - 8z_{b1} - 16z_{b2} - 32z_{b3} \geq -42
\end{array}$$

Other constraints will need more than just RUP

$$2X + 3Y + 4Z \leq 42$$

$$X \geq 5 \land Z \geq 3 \Rightarrow Y \leq 6$$

Recall: Cutting planes allows us to derive linear combinations of constraints.

$$\begin{array}{ll}
\text{(RED)} & x_{\geq 5} \Rightarrow x_{b0} + 2x_{b1} + 4x_{b2} + 8x_{b3} \geq 5 \\
\text{(RED)} & z_{\geq 3} \Rightarrow z_{b0} + 2z_{b1} + 4z_{b2} + 8z_{b3} \geq 3 \\
\text{(RED)} & \overline{y_{\leq 6}} \Rightarrow y_{b0} + 2y_{b1} + 4x_{b2} + 8x_{b3} \geq 7 \\
 & -2x_{b0} - 4x_{b1} - 8x_{b2} - 16x_{b3} \\
\text{(Axiom)} & -3y_{b0} - 6y_{b1} - 12y_{b2} - 24y_{b3} \\
 & -4z_{b0} - 8z_{b1} - 16z_{b2} - 32z_{b3} \geq -42
\end{array}$$

Other constraints will need more than just RUP

$$2X + 3Y + 4Z \leq 42$$

$$X \geq 5 \land Z \geq 3 \Rightarrow Y \leq 6$$

$$\begin{array}{lll}
2\times & \text{(RED)} & x_{\geq 5} \Rightarrow x_{b0} + 2x_{b1} + 4x_{b2} + 8x_{b3} \geq 5 \\
3\times & \text{(RED)} & z_{\geq 3} \Rightarrow z_{b0} + 2z_{b1} + 4z_{b2} + 8z_{b3} \geq 3 \\
4\times & \text{(RED)} & \overline{y_{\leq 6}} \Rightarrow y_{b0} + 2y_{b1} + 4x_{b2} + 8x_{b3} \geq 7 \\
 & & -2x_{b0} - 4x_{b1} - 8x_{b2} - 16x_{b3} \\
 & \text{(Axiom)} & -3y_{b0} - 6y_{b1} - 12y_{b2} - 24y_{b3} \\
 & & -4z_{b0} - 8z_{b1} - 16z_{b2} - 32z_{b3} \geq -42
\end{array}$$

Recall: Cutting planes allows us to derive linear combinations of constraints.

Other constraints will need more than just RUP

$$2X + 3Y + 4Z \leq 42$$

$$X \geq 5 \land Z \geq 3 \Rightarrow Y \leq 6$$

$$\begin{array}{lll}
2\times & \text{(RED)} & x_{\geq 5} \Rightarrow x_{b0} + 2x_{b1} + 4x_{b2} + 8x_{b3} \geq 5 \\
3\times & \text{(RED)} & z_{\geq 3} \Rightarrow z_{b0} + 2z_{b1} + 4z_{b2} + 8z_{b3} \geq 3 \\
4\times & \text{(RED)} & \overline{y_{\leq 6}} \Rightarrow y_{b0} + 2y_{b1} + 4x_{b2} + 8x_{b3} \geq 7 \\
 & & -2x_{b0} - 4x_{b1} - 8x_{b2} - 16x_{b3} \\
 & \text{(Axiom)} & -3y_{b0} - 6y_{b1} - 12y_{b2} - 24y_{b3} \\
 & & -4z_{b0} - 8z_{b1} - 16z_{b2} - 32z_{b3} \geq -42 \\
\end{array}$$

$$\text{(Sum)} \quad 10\overline{x_{\geq 5}} + 12\overline{z_{\geq 3}} + 21y_{\geq 6} \geq 1$$

Recall: Cutting planes allows us to derive linear combinations of constraints.

Other constraints will need more than just RUP

$$2X + 3Y + 4Z \leq 42$$

$$X \geq 5 \land Z \geq 3 \Rightarrow Y \leq 6$$

$$
\begin{array}{lll}
2\times & \text{(RED)} & x_{\geq 5} \Rightarrow x_{b0} + 2x_{b1} + 4x_{b2} + 8x_{b3} \geq 5 \\
3\times & \text{(RED)} & z_{\geq 3} \Rightarrow z_{b0} + 2z_{b1} + 4z_{b2} + 8z_{b3} \geq 3 \\
4\times & \text{(RED)} & \overline{y_{\leq 6}} \Rightarrow y_{b0} + 2y_{b1} + 4x_{b2} + 8x_{b3} \geq 7 \\
 & & -2x_{b0} - 4x_{b1} - 8x_{b2} - 16x_{b3} \\
 & \text{(Axiom)} & -3y_{b0} - 6y_{b1} - 12y_{b2} - 24y_{b3} \\
 & & -4z_{b0} - 8z_{b1} - 16z_{b2} - 32z_{b3} \geq -42 \\
\end{array}
$$

$$\text{(Sum)} \quad x_{\geq 5} \land z_{\geq 3} \Rightarrow y_{\geq 6}$$

Recall: Cutting planes allows us to derive linear combinations of constraints.

Justifying AllDifferent

Justifying AllDifferent

$$
\begin{array}{rcl}
V & \in & \{\ 1 \quad\quad\quad\quad 4 \quad 5\ \} \\
W & \in & \{\ 1 \quad 2 \quad 3 \quad\quad\quad\quad\ \} \\
X & \in & \{\ \quad\ \ 2 \quad 3 \quad\quad\quad\quad\ \} \\
Y & \in & \{\ 1 \quad\quad\ \ 3 \quad\quad\quad\quad\ \} \\
Z & \in & \{\ 1 \quad\quad\ \ 3 \quad\quad\quad\quad\ \}
\end{array}
$$

Justifying AllDifferent

$$
\begin{array}{lcllllll}
V & \in & \{ & 1 & & & 4 & 5 & \} \\
W & \in & \{ & 1 & 2 & 3 & & & \} \\
X & \in & \{ & & 2 & 3 & & & \} \\
Y & \in & \{ & 1 & & 3 & & & \} \\
Z & \in & \{ & 1 & & 3 & & & \}
\end{array}
$$

Justifying AllDifferent

$$
\begin{array}{rcl}
V & \in & \{\ 1 \qquad\quad 4\ \ 5\ \} \\
W & \in & \{\ 1\ \ 2\ \ 3 \qquad \} \\
X & \in & \{\ \ \ \ 2\ \ 3 \qquad \} \\
Y & \in & \{\ 1 \quad 3 \qquad \} \\
Z & \in & \{\ 1 \quad 3 \qquad \}
\end{array}
$$

$$\mathcal{R} := w_{\geq 1} \wedge w_{\leq 3}$$
$$\wedge x_{\geq 2} \wedge x_{\leq 3} \wedge y_{\geq 1} \wedge y_{\leq 3}$$
$$\wedge \bar{y}_{=2} \wedge z_{\geq 1} \wedge \bar{z}_{=2} \wedge z_{\leq 3}$$

Justifying AllDifferent

$$
\begin{array}{ccccccccc}
V & \in & \{ & 1 & & & 4 & 5 & \} \\
W & \in & \{ & 1 & 2 & 3 & & & \} \\
X & \in & \{ & & 2 & 3 & & & \} \\
Y & \in & \{ & 1 & & 3 & & & \} \\
Z & \in & \{ & 1 & & 3 & & & \}
\end{array}
$$

$$
\begin{aligned}
\mathcal{R} := w_{\geq 1} \wedge w_{\leq 3} \\
\wedge x_{\geq 2} \wedge x_{\leq 3} \wedge y_{\geq 1} \wedge y_{\leq 3} \\
\wedge \bar{y}_{=2} \wedge z_{\geq 1} \wedge \bar{z}_{=2} \wedge z_{\leq 3}
\end{aligned}
$$

Justifying AllDifferent

$$\begin{array}{ccccccccc} V & \in & \{ & 1 & & & 4 & 5 & \} \\ \boxed{W} & \in & \{ & 1 & 2 & 3 & & & \} \\ \boxed{X} & \in & \{ & & 2 & 3 & & & \} \\ \boxed{Y} & \in & \{ & 1 & & 3 & & & \} \\ \boxed{Z} & \in & \{ & 1 & & 3 & & & \} \end{array}$$

$$\mathcal{R} := w_{\geq 1} \wedge w_{\leq 3}$$
$$\wedge x_{\geq 2} \wedge x_{\leq 3} \wedge y_{\geq 1} \wedge y_{\leq 3}$$
$$\wedge \bar{y}_{=2} \wedge z_{\geq 1} \wedge \bar{z}_{=2} \wedge z_{\leq 3}$$

(RUP) $\mathcal{R} \Rightarrow w_{=1} + w_{=2} + w_{=3} \geq 1$

(RUP) $\mathcal{R} \Rightarrow x_{=2} + x_{=3} \geq 1$

(RUP) $\mathcal{R} \Rightarrow y_{=1} \quad y_{=3} \geq 1$

(RUP) $\mathcal{R} \Rightarrow z_{=1} \quad z_{=3} \geq 1$

Justifying AllDifferent

$$
\begin{array}{l}
V \in \{\ 1 \quad \quad \quad 4 \quad 5\ \} \\
\boxed{W} \in \{\ 1 \quad 2 \quad 3 \ \} \\
\boxed{X} \in \{\ \quad 2 \quad 3 \ \} \\
\boxed{Y} \in \{\ 1 \quad \quad 3 \ \} \\
\boxed{Z} \in \{\ 1 \quad \quad 3 \ \}
\end{array}
$$

$$
\mathcal{R} := w_{\geq 1} \wedge w_{\leq 3} \wedge x_{\geq 2} \wedge x_{\leq 3} \wedge y_{\geq 1} \wedge y_{\leq 3} \wedge \bar{y}_{=2} \wedge z_{\geq 1} \wedge \bar{z}_{=2} \wedge z_{\leq 3}
$$

$$
\begin{array}{ll}
\text{(RUP)} & \mathcal{R} \Rightarrow w_{=1} + w_{=2} + w_{=3} \geq 1 \\
\text{(RUP)} & \mathcal{R} \Rightarrow x_{=2} + x_{=3} \geq 1 \\
\text{(RUP)} & \mathcal{R} \Rightarrow y_{=1} \quad y_{=3} \geq 1 \\
\text{(RUP)} & \mathcal{R} \Rightarrow z_{=1} \quad z_{=3} \geq 1 \\
\text{(RUP)} & \mathcal{R} \Rightarrow -v_{=1} + -w_{=1} + -y_{=1} + -z_{=1} \geq -1 \\
\text{(RUP)} & \mathcal{R} \Rightarrow -w_{=2} + -x_{=2} \geq -1 \\
\text{(RUP)} & \mathcal{R} \Rightarrow -w_{=3} + -x_{=3} + -y_{=3} + -z_{=3} \geq -1
\end{array}
$$

Justifying AllDifferent

$$
\begin{array}{lllllll}
V \in \{ & 1 & & & 4 & 5 & \} \\
\boxed{W} \in \{ & 1 & 2 & 3 & & & \} \\
\boxed{X} \in \{ & & 2 & 3 & & & \} \\
\boxed{Y} \in \{ & 1 & & 3 & & & \} \\
\boxed{Z} \in \{ & 1 & & 3 & & & \}
\end{array}
$$

$$\mathcal{R} := w_{\geq 1} \wedge w_{\leq 3}$$
$$\wedge x_{\geq 2} \wedge x_{\leq 3} \wedge y_{\geq 1} \wedge y_{\leq 3}$$
$$\wedge \bar{y}_{=2} \wedge z_{\geq 1} \wedge \bar{z}_{=2} \wedge z_{\leq 3}$$

(RUP) $\mathcal{R} \Rightarrow w_{=1} + w_{=2} + w_{=3} \geq 1$

(RUP) $\mathcal{R} \Rightarrow x_{=2} + x_{=3} \geq 1$

(RUP) $\mathcal{R} \Rightarrow y_{=1} \quad y_{=3} \geq 1$

(RUP) $\mathcal{R} \Rightarrow z_{=1} \quad z_{=3} \geq 1$

(RUP) $\mathcal{R} \Rightarrow -v_{=1} + -w_{=1} + - y_{=1} + -z_{=1} \geq -1$

(RUP) $\mathcal{R} \Rightarrow - w_{=2} + -x_{=2} \geq -1$

(RUP) $\mathcal{R} \Rightarrow - w_{=3} + -x_{=3} + -y_{=3} + -z_{=3} \geq -1$

(Sum all of the above:) $\mathcal{R} \Rightarrow -v_{=1} \geq 1$

Justifying AllDifferent

$$
\begin{array}{lcccccccl}
V & \in \{ & 1 & & & 4 & 5 & \} \\
\boxed{W} & \in \{ & 1 & 2 & 3 & & & \} \\
\boxed{X} & \in \{ & & 2 & 3 & & & \} \\
\boxed{Y} & \in \{ & 1 & & 3 & & & \} \\
\boxed{Z} & \in \{ & 1 & & 3 & & & \}
\end{array}
$$

$$
\begin{aligned}
\mathcal{R} := {} & w_{\geq 1} \wedge w_{\leq 3} \\
& \wedge x_{\geq 2} \wedge x_{\leq 3} \wedge y_{\geq 1} \wedge y_{\leq 3} \\
& \wedge \bar{y}_{=2} \wedge z_{\geq 1} \wedge \bar{z}_{=2} \wedge z_{\leq 3}
\end{aligned}
$$

(RUP) $\mathcal{R} \Rightarrow w_{=1} + w_{=2} + w_{=3} \geq 1$

(RUP) $\mathcal{R} \Rightarrow x_{=2} + x_{=3} \geq 1$

(RUP) $\mathcal{R} \Rightarrow y_{=1} \quad y_{=3} \geq 1$

(RUP) $\mathcal{R} \Rightarrow z_{=1} \quad z_{=3} \geq 1$

(RUP) $\mathcal{R} \Rightarrow -v_{=1} + -w_{=1} + \quad - y_{=1} + -z_{=1} \geq -1$

(RUP) $\mathcal{R} \Rightarrow - w_{=2} + -x_{=2} \geq -1$

(RUP) $\mathcal{R} \Rightarrow - w_{=3} + -x_{=3} + -y_{=3} + -z_{=3} \geq -1$

(Sum all of the above:) $\mathcal{R} \Rightarrow -v_{=1} \geq 1$

(Literal axiom:) $v_{=1} \geq 0$

Justifying AllDifferent

$$
\begin{array}{lcl}
V & \in & \{\ 1 \quad\quad\quad\quad 4 \quad 5\ \} \\
\boxed{W} & \in & \{\ 1 \quad 2 \quad 3 \quad\quad\quad\ \} \\
\boxed{X} & \in & \{\ \quad\ 2 \quad 3 \quad\quad\quad\ \} \\
\boxed{Y} & \in & \{\ 1 \quad\quad\ 3 \quad\quad\quad\ \} \\
\boxed{Z} & \in & \{\ 1 \quad\quad\ 3 \quad\quad\quad\ \}
\end{array}
$$

$$\mathcal{R} := w_{\geq 1} \wedge w_{\leq 3}$$
$$\wedge x_{\geq 2} \wedge x_{\leq 3} \wedge y_{\geq 1} \wedge y_{\leq 3}$$
$$\wedge \bar{y}_{=2} \wedge z_{\geq 1} \wedge \bar{z}_{=2} \wedge z_{\leq 3}$$

(RUP) $\mathcal{R} \Rightarrow w_{=1} + w_{=2} + w_{=3} \geq 1$

(RUP) $\mathcal{R} \Rightarrow x_{=2} + x_{=3} \geq 1$

(RUP) $\mathcal{R} \Rightarrow y_{=1} + y_{=3} \geq 1$

(RUP) $\mathcal{R} \Rightarrow z_{=1} + z_{=3} \geq 1$

(RUP) $\mathcal{R} \Rightarrow -v_{=1} + -w_{=1} + - y_{=1} + -z_{=1} \geq -1$

(RUP) $\mathcal{R} \Rightarrow - w_{=2} + -x_{=2} \geq -1$

(RUP) $\mathcal{R} \Rightarrow - w_{=3} + -x_{=3} + -y_{=3} + -z_{=3} \geq -1$

(Sum all of the above:) $\mathcal{R} \Rightarrow -v_{=1} \geq 1$

(Literal axiom:) $v_{=1} \geq 0$

(Add:) $\mathcal{R} \Rightarrow 0 \geq 1$

The Circuit constraint

The Circuit constraint

$$X_0, \ldots, X_{n-1}$$

$$\{0, \ldots, n-1\}$$

The Circuit constraint

$$\text{Circuit}(X_0, \ldots, X_{n-1})$$

$$\{0, \ldots, n-1\}$$

The Circuit constraint

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

$$\{0, \ldots, n-1\}$$

The Circuit constraint

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

$$\{0, 1, 2, 3, 4, 5\}$$

The Circuit constraint

X_0

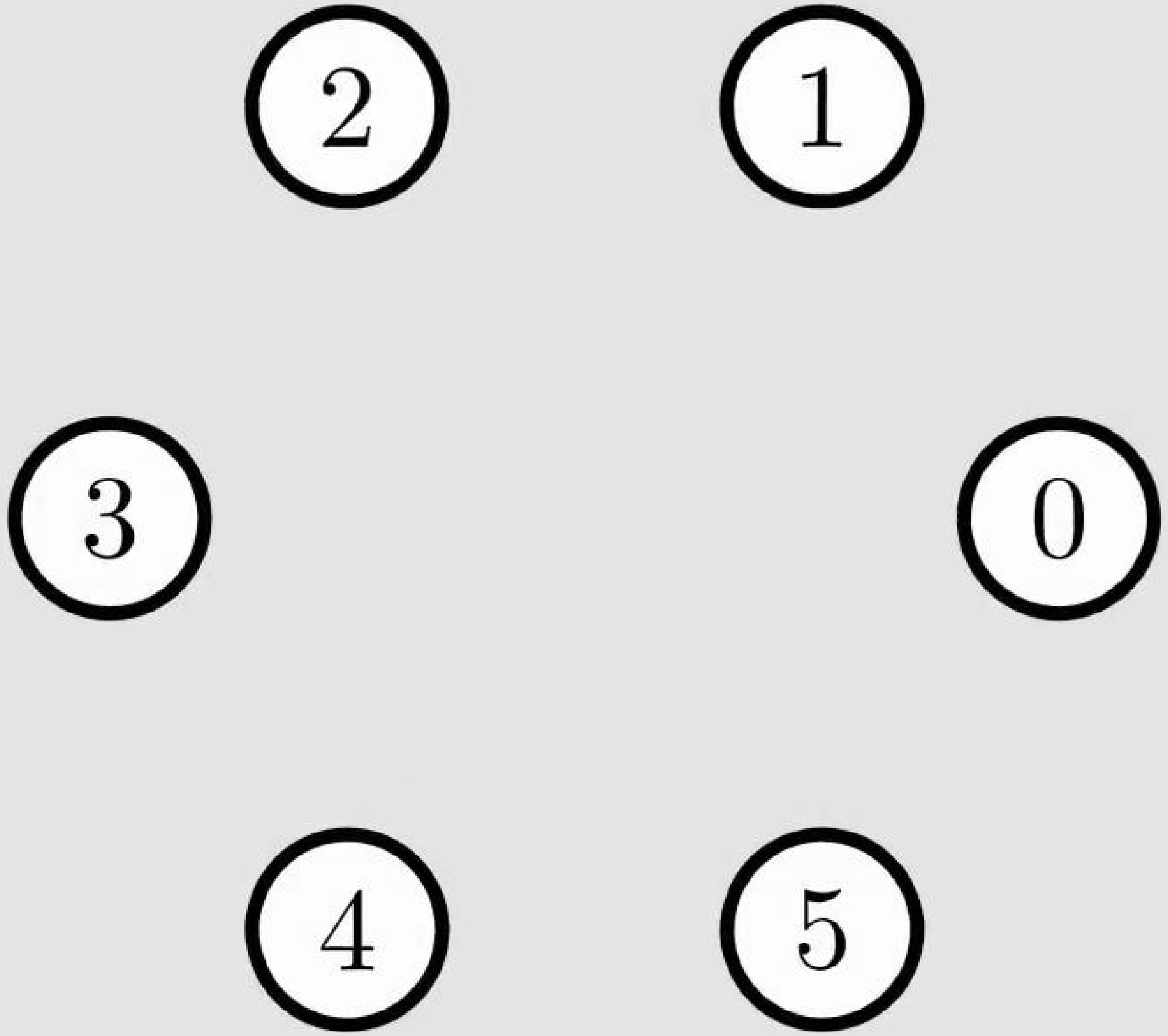
X_1

X_2

X_3

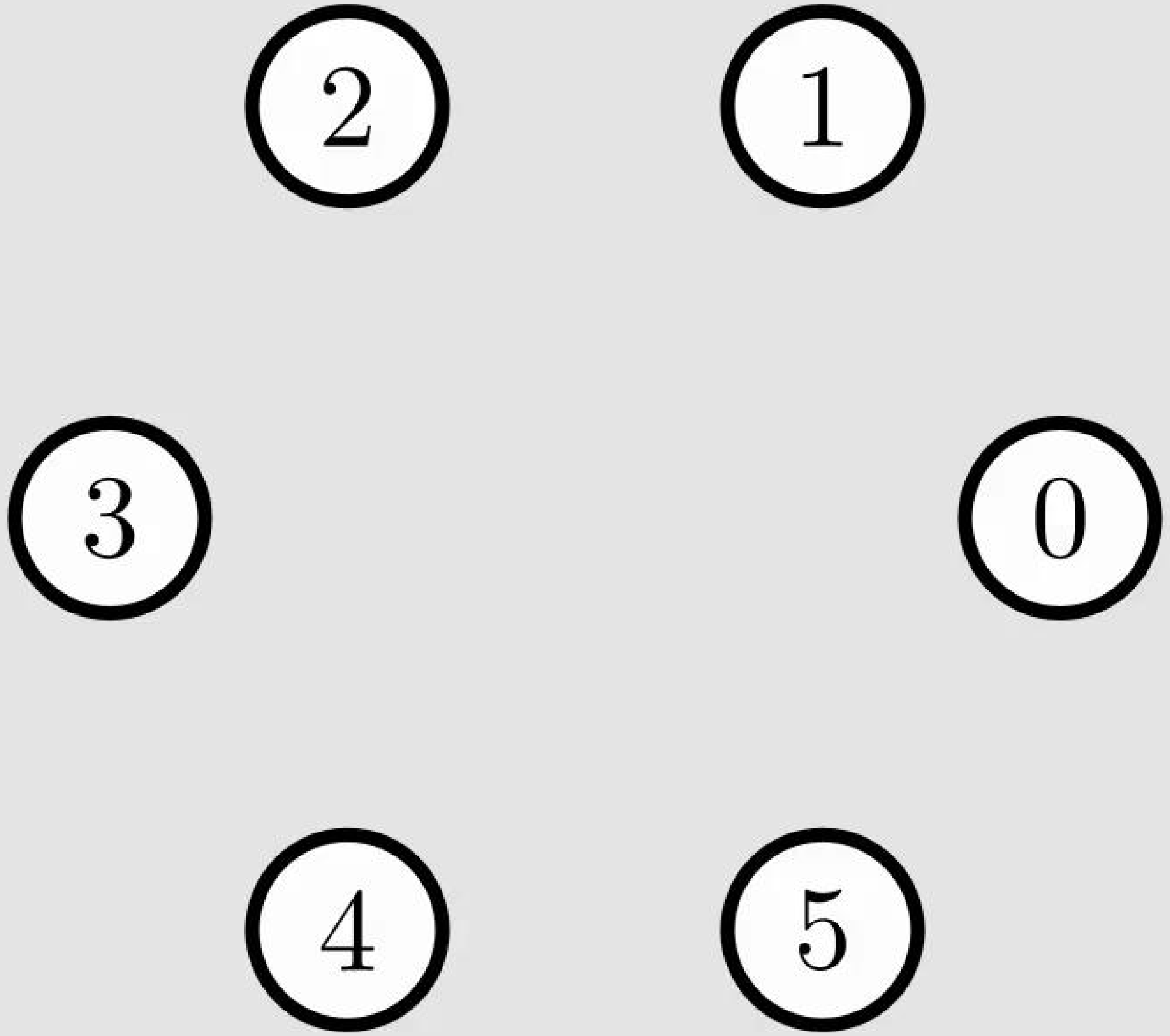
X_4

X_5



The Circuit constraint

$$X_0$$
$$X_1$$
$$X_2 = 5$$
$$X_3$$
$$X_4$$
$$X_5$$



The Circuit constraint

X_0

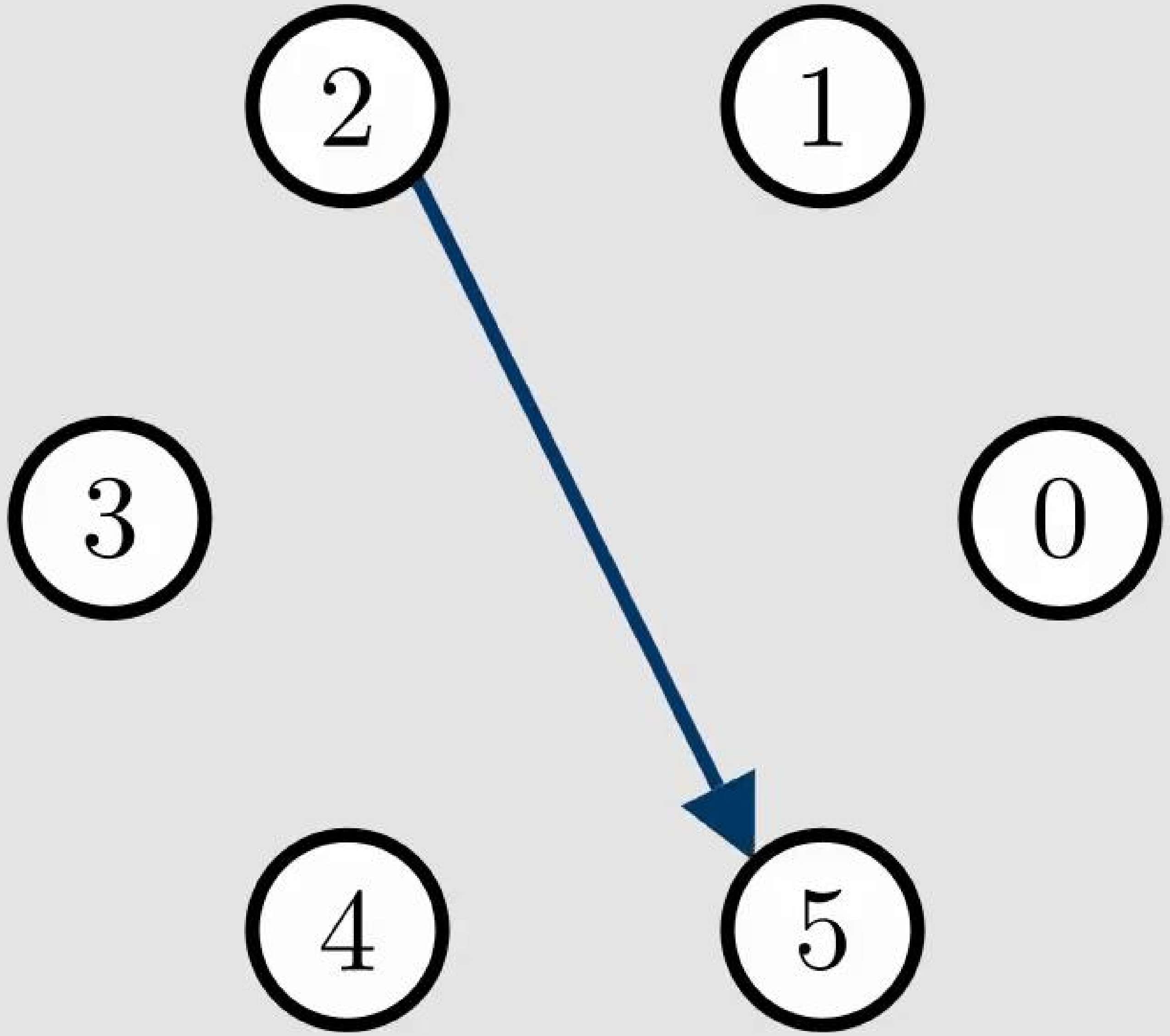
X_1

$X_2 = 5$

X_3

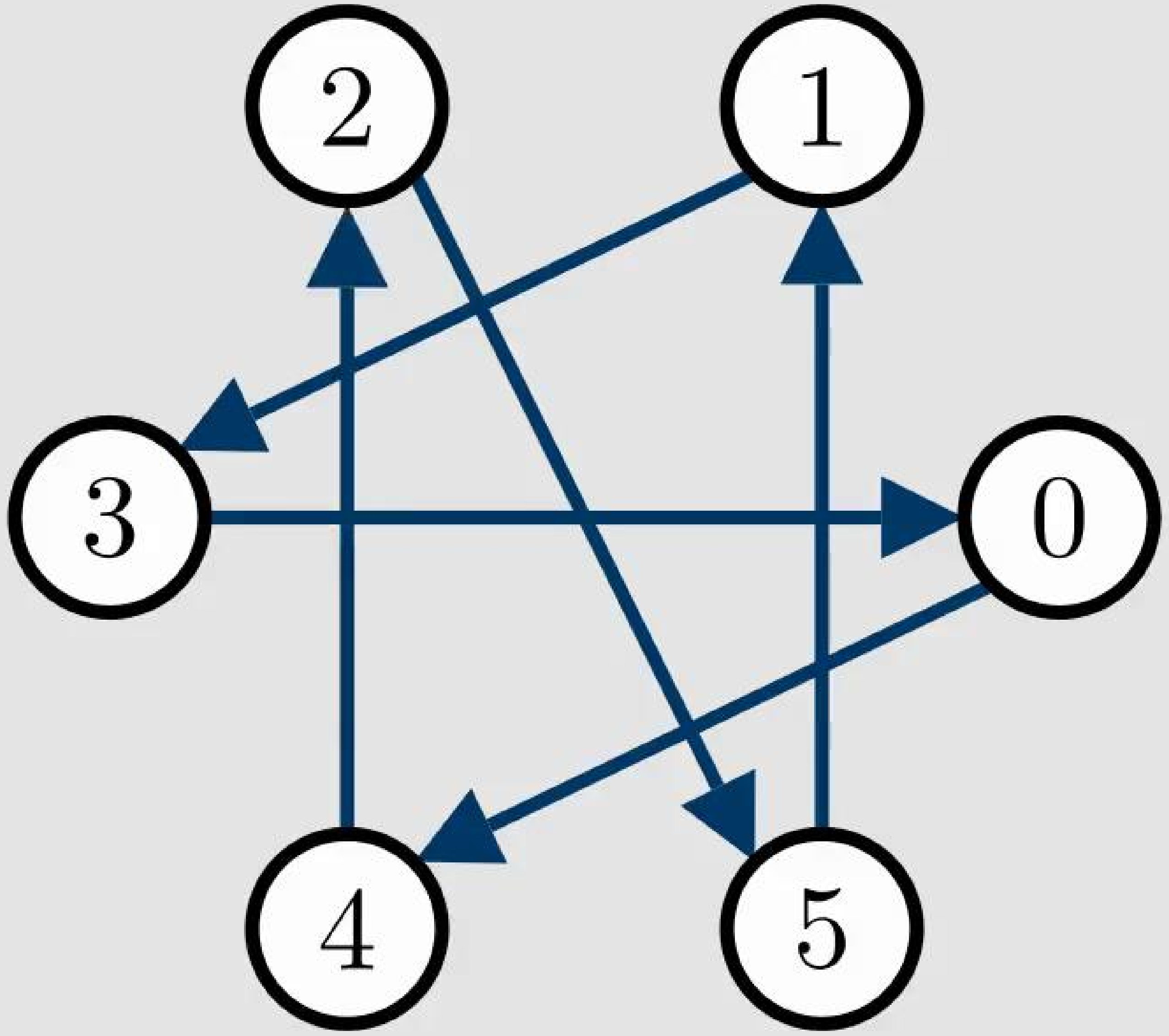
X_4

X_5



The Circuit constraint

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



Enforcing Circuit:

X_0

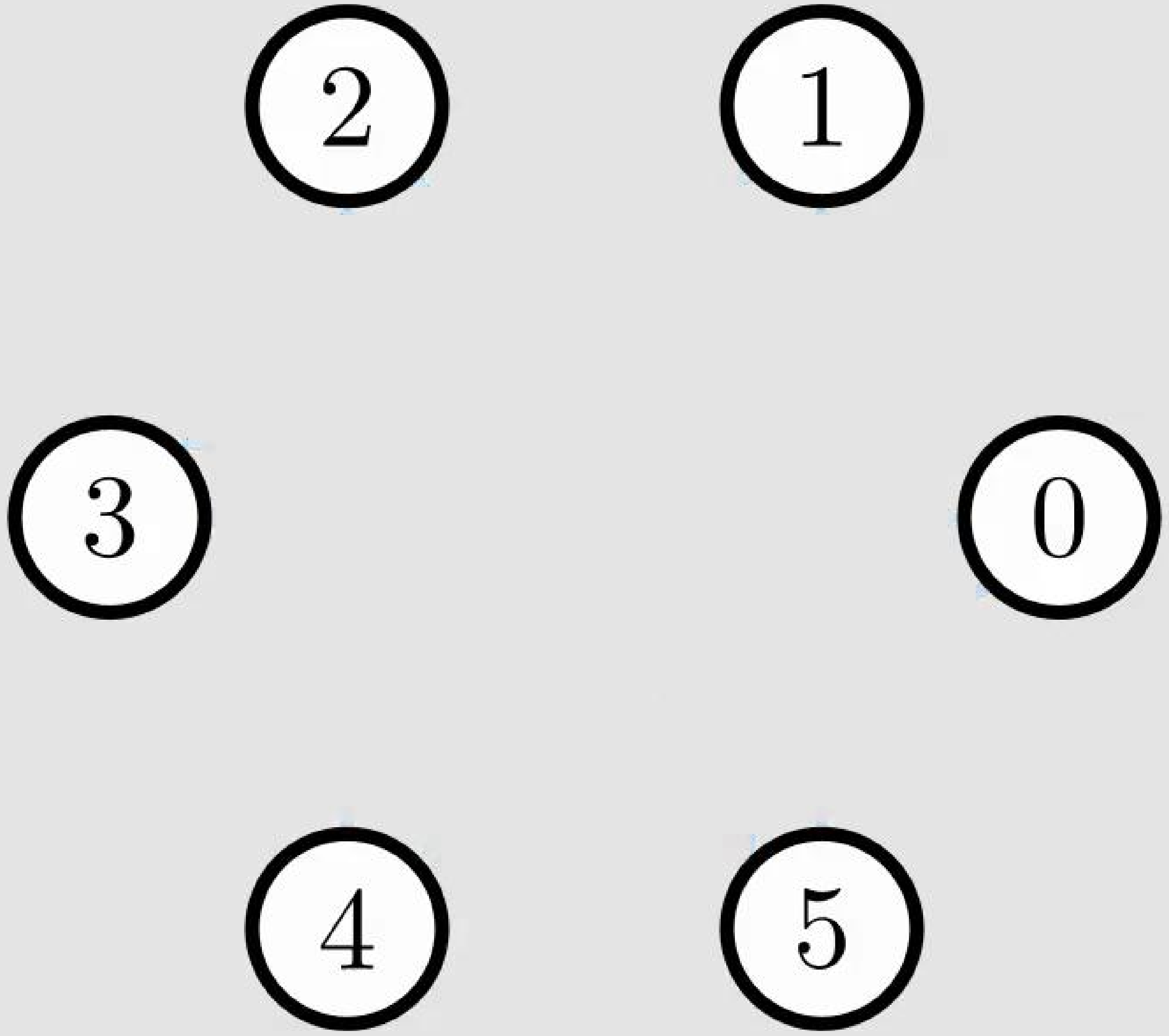
X_1

X_2

X_3

X_4

X_5



Enforcing Circuit:

X_0

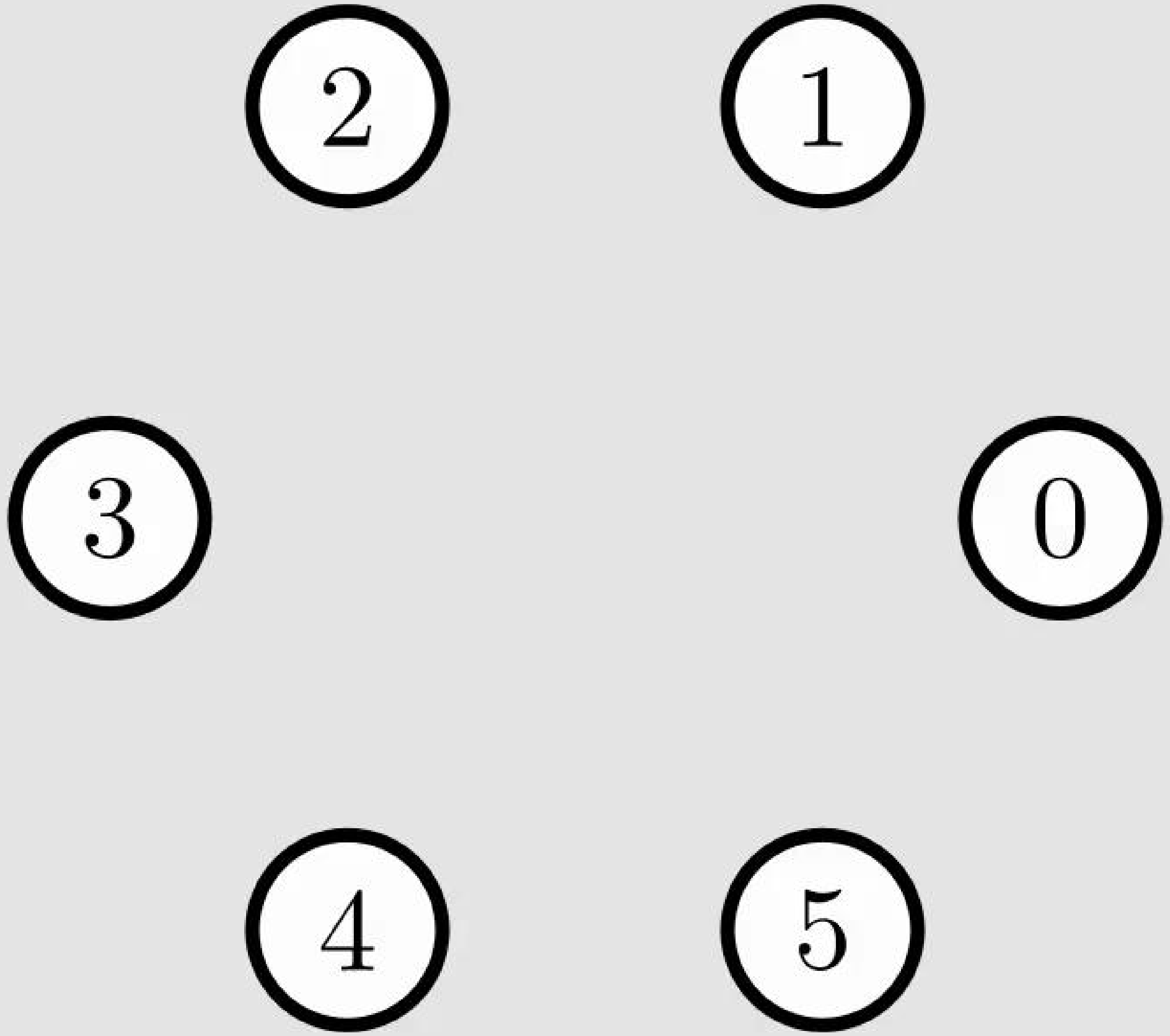
X_1

X_2

X_3

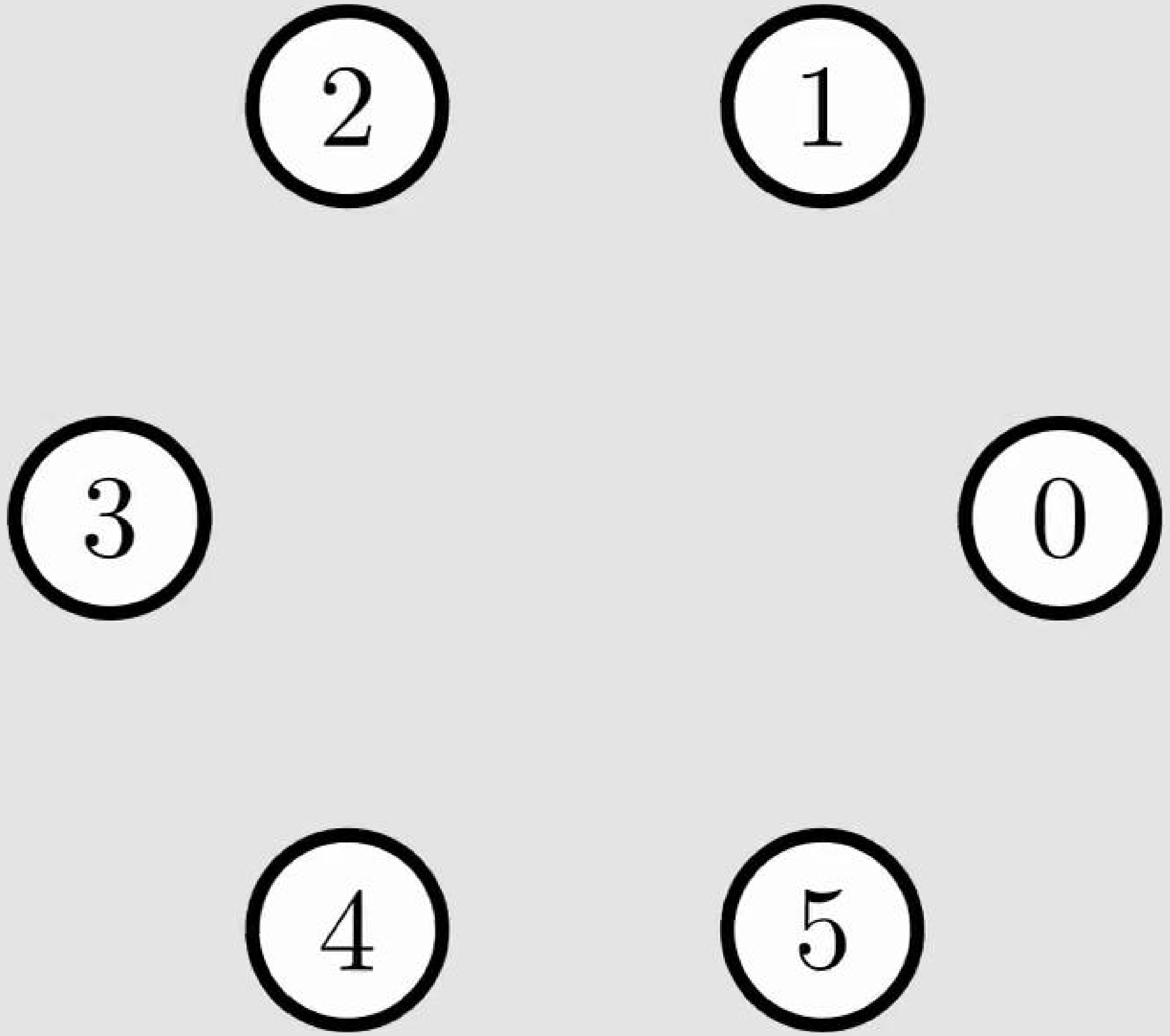
X_4

X_5



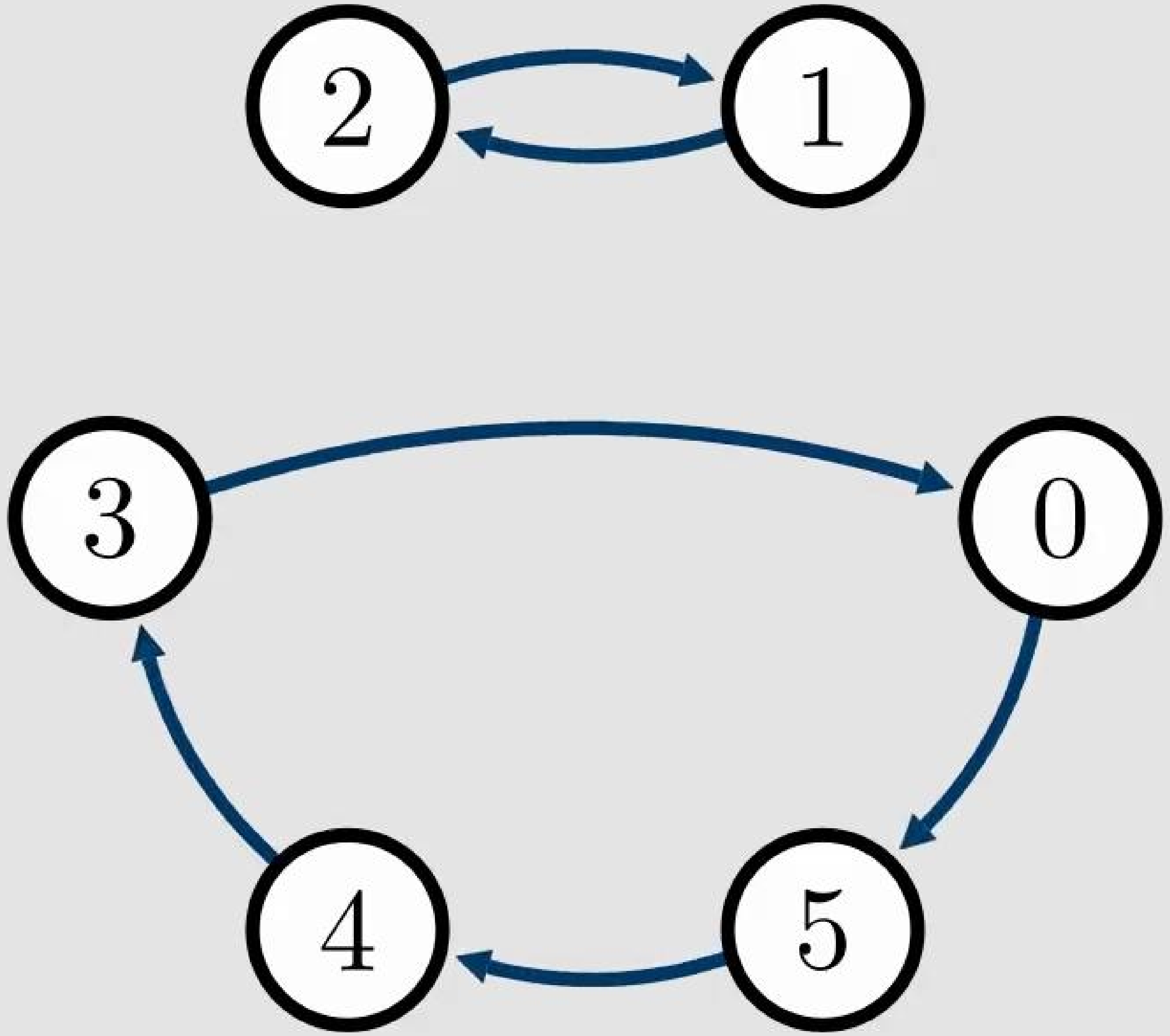
Enforcing Circuit:

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



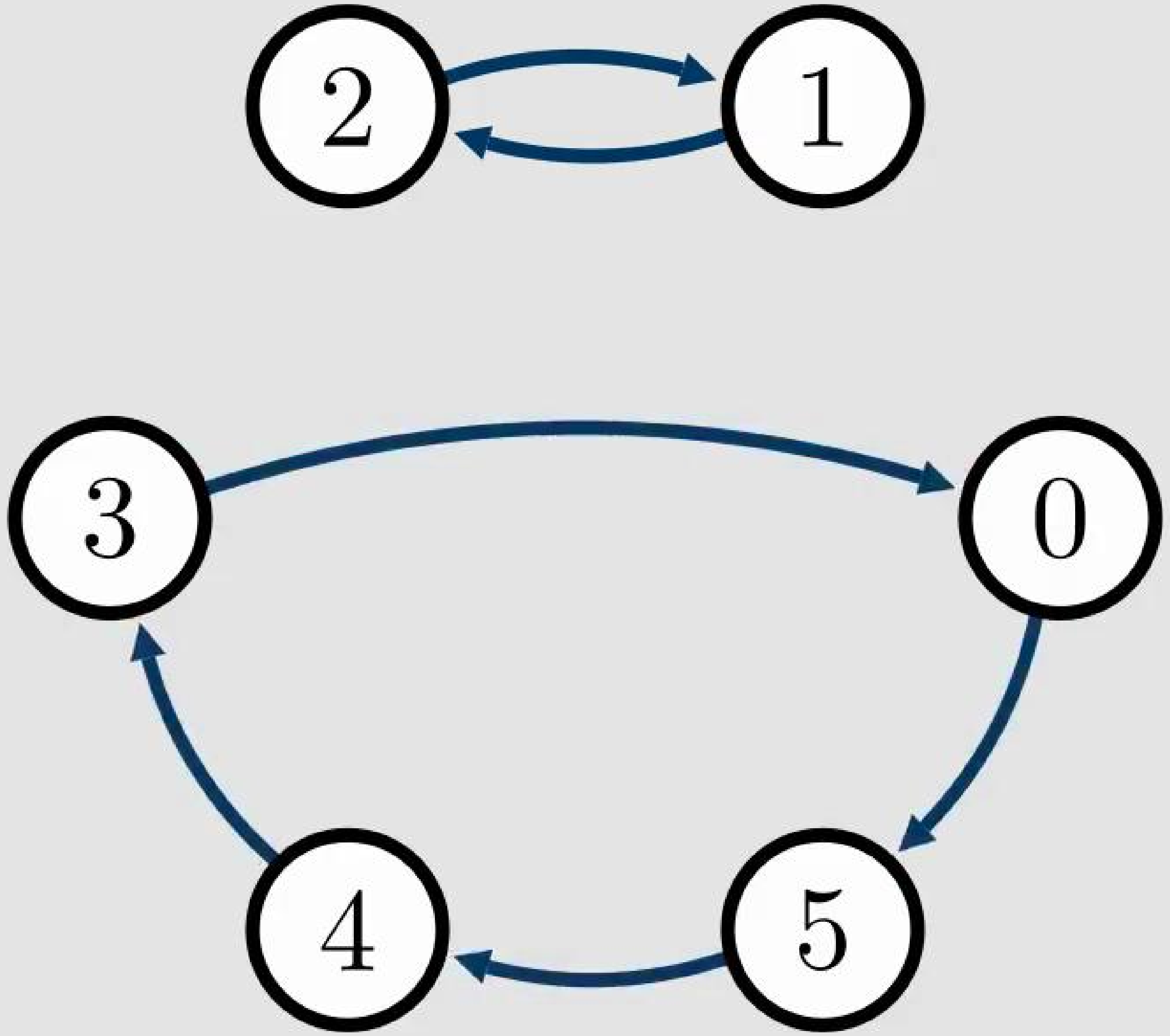
Enforcing Circuit:

AllDiff$(X_0, X_1, X_2, X_3, X_4, X_5)$



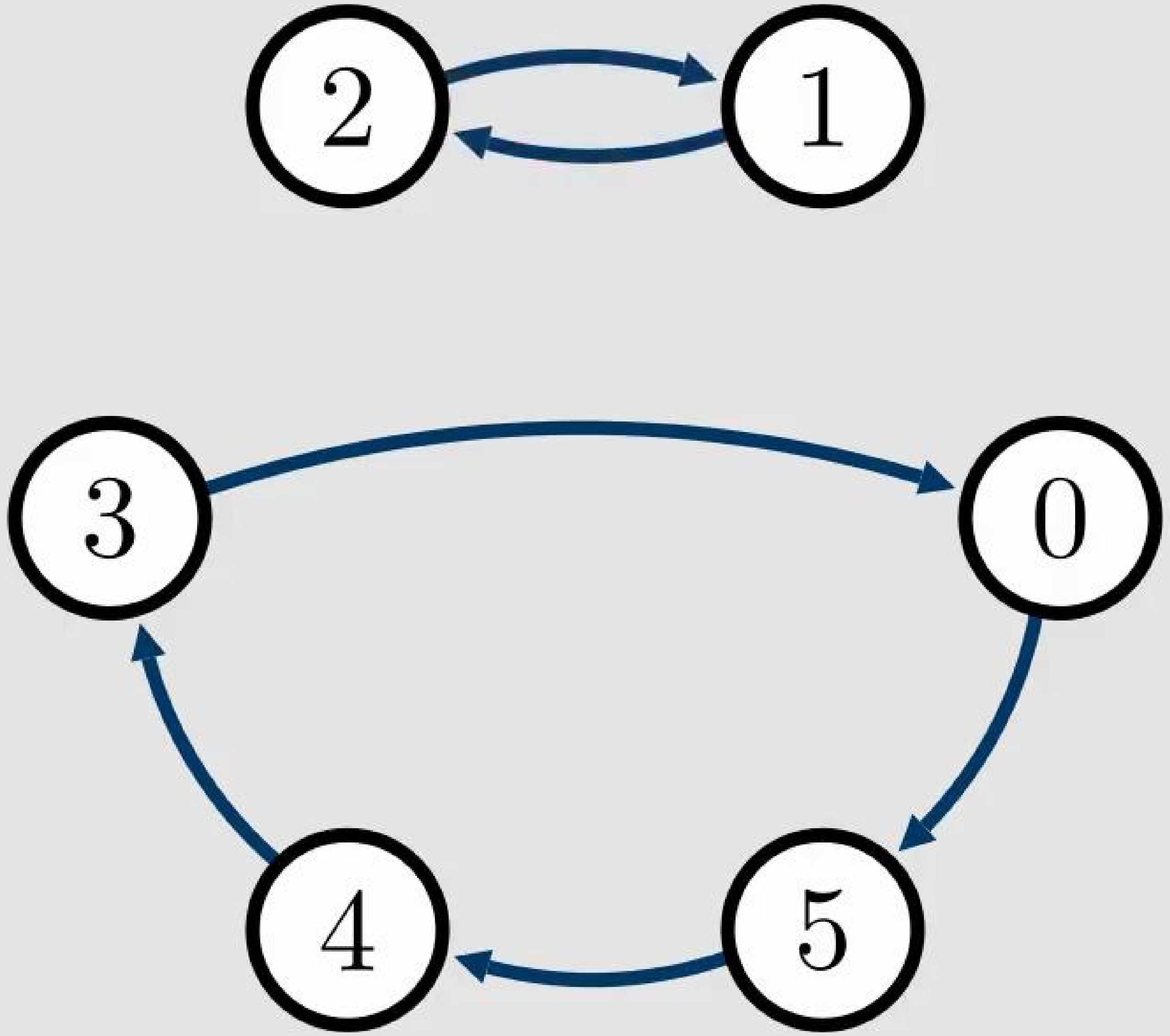
Enforcing Circuit:

AllDiff$(X_0, X_1, X_2, X_3, X_4, X_5)$



Enforcing Circuit:

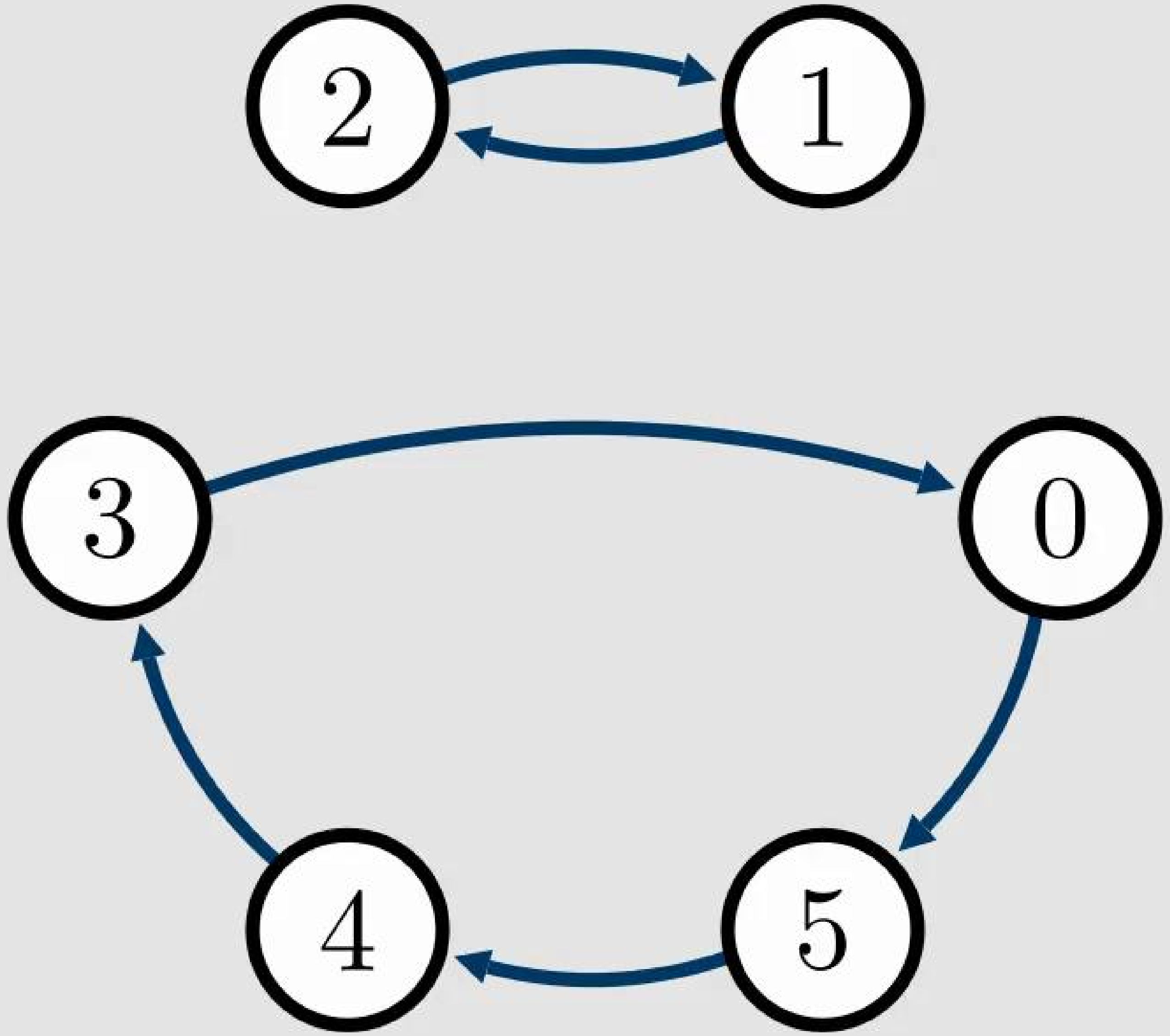
AllDiff$(X_0, X_1, X_2, X_3, X_4, X_5)$



Enforcing Circuit:

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

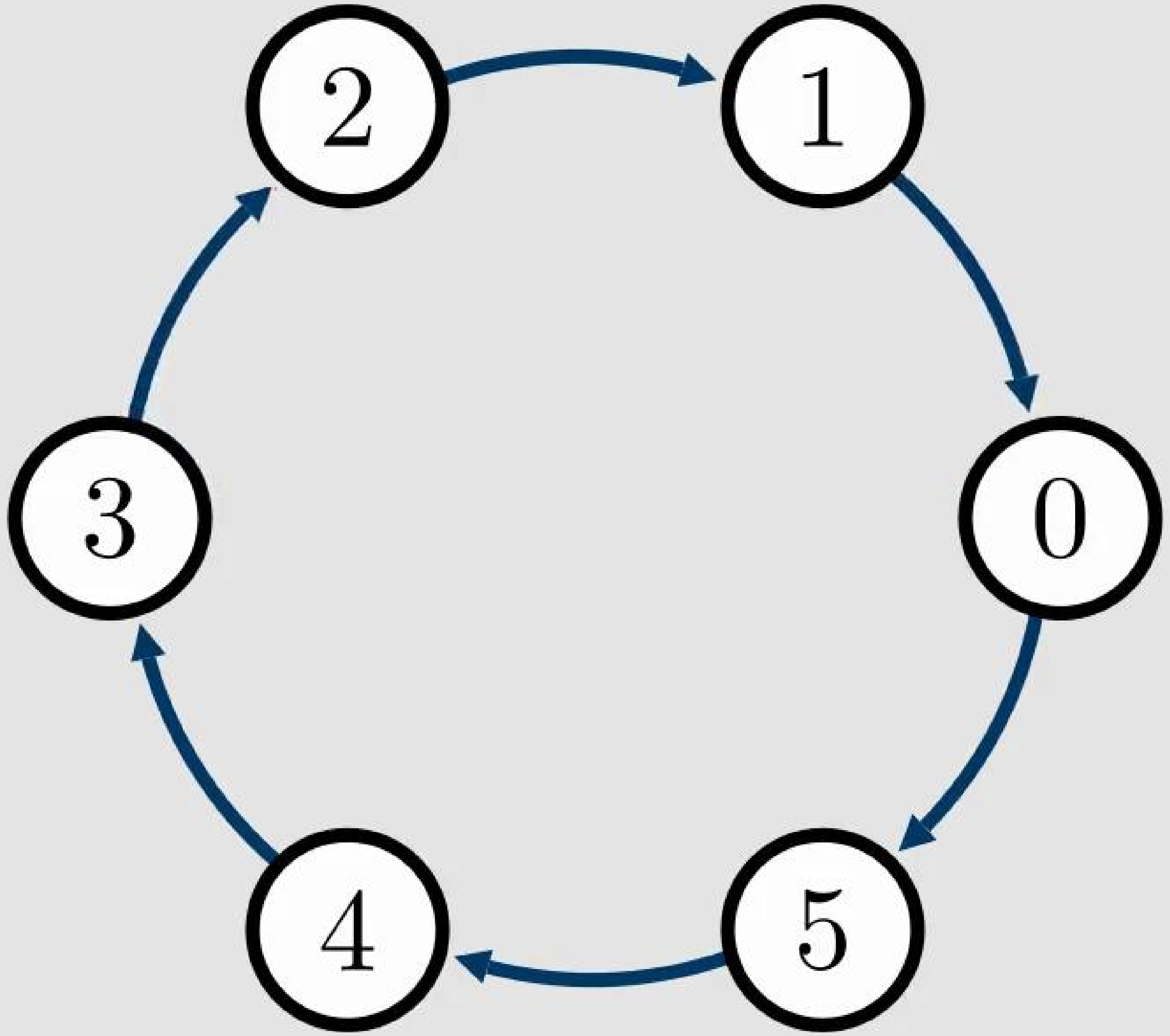
$\text{NoCycle}(X_0, X_1, X_2, X_3, X_4, X_5)$



Enforcing Circuit:

AllDiff$(X_0, X_1, X_2, X_3, X_4, X_5)$

NoCycle$(X_0, X_1, X_2, X_3, X_4, X_5)$



Consistency for Circuit:

X_0

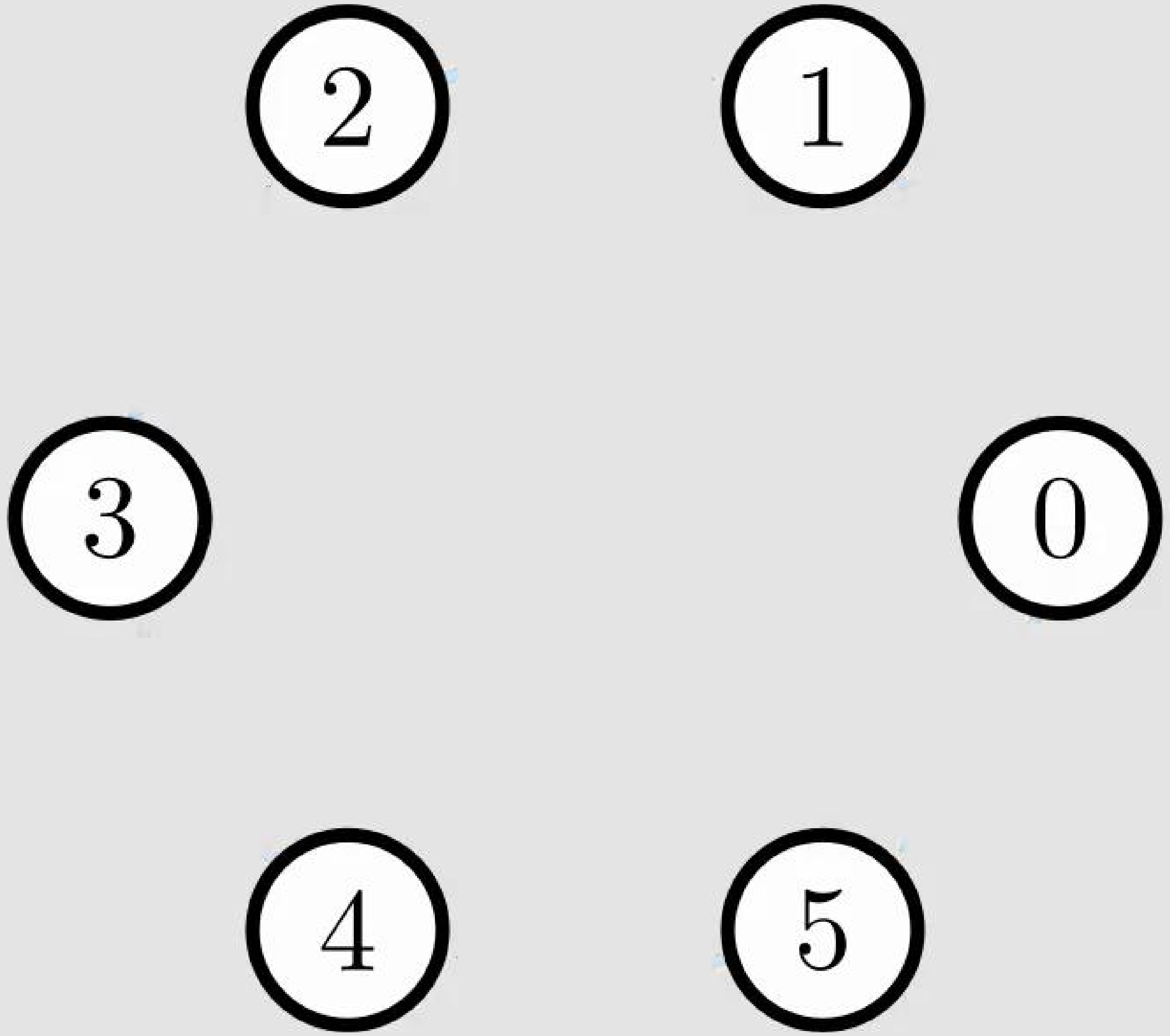
X_1

X_2

X_3

X_4

X_5



Consistency for Circuit:

X_0

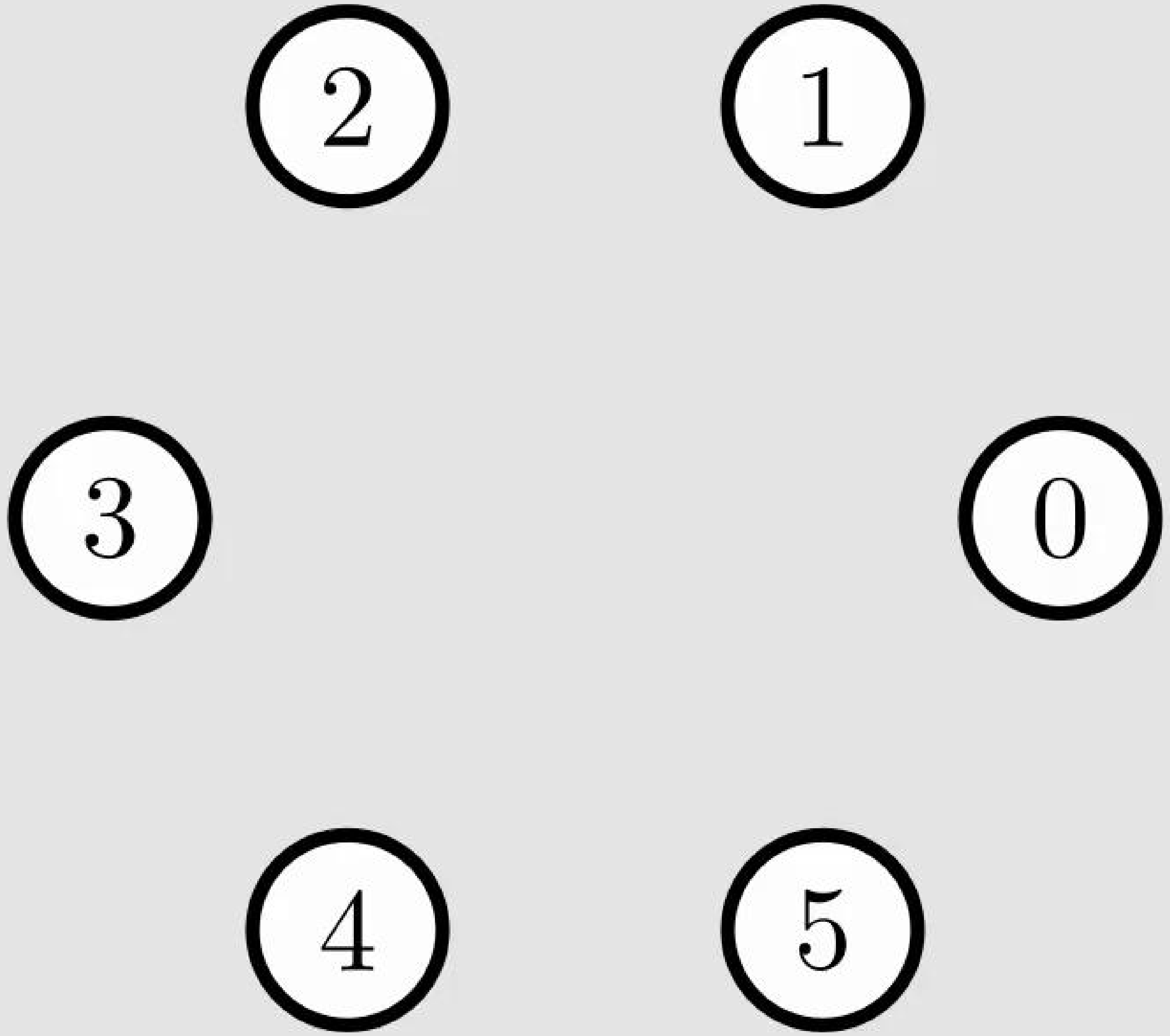
X_1

X_2

X_3

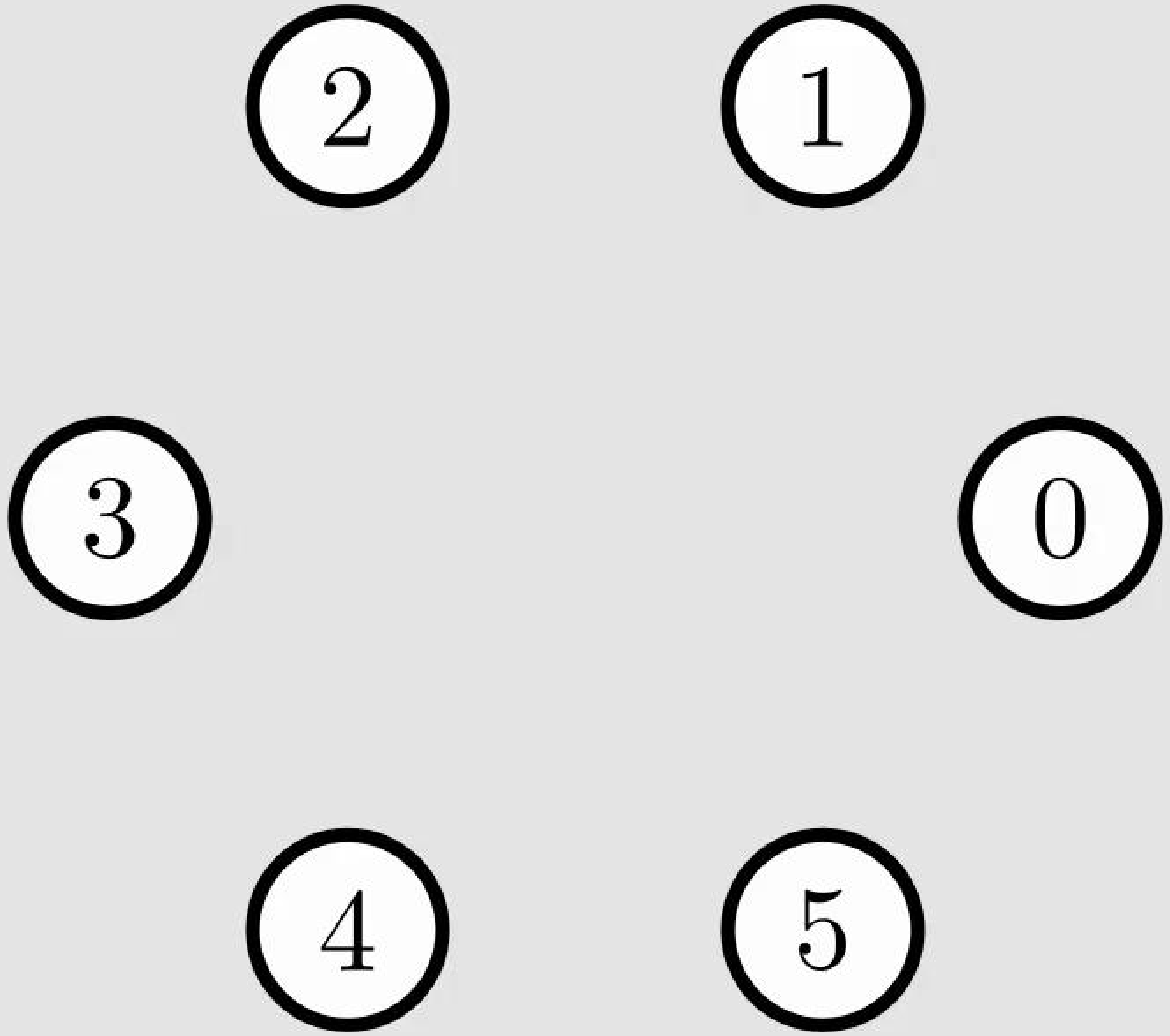
X_4

X_5



Consistency for Circuit:

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



Consistency for Circuit:

$$X_0 \in \{0, 1, 2, 5\}$$

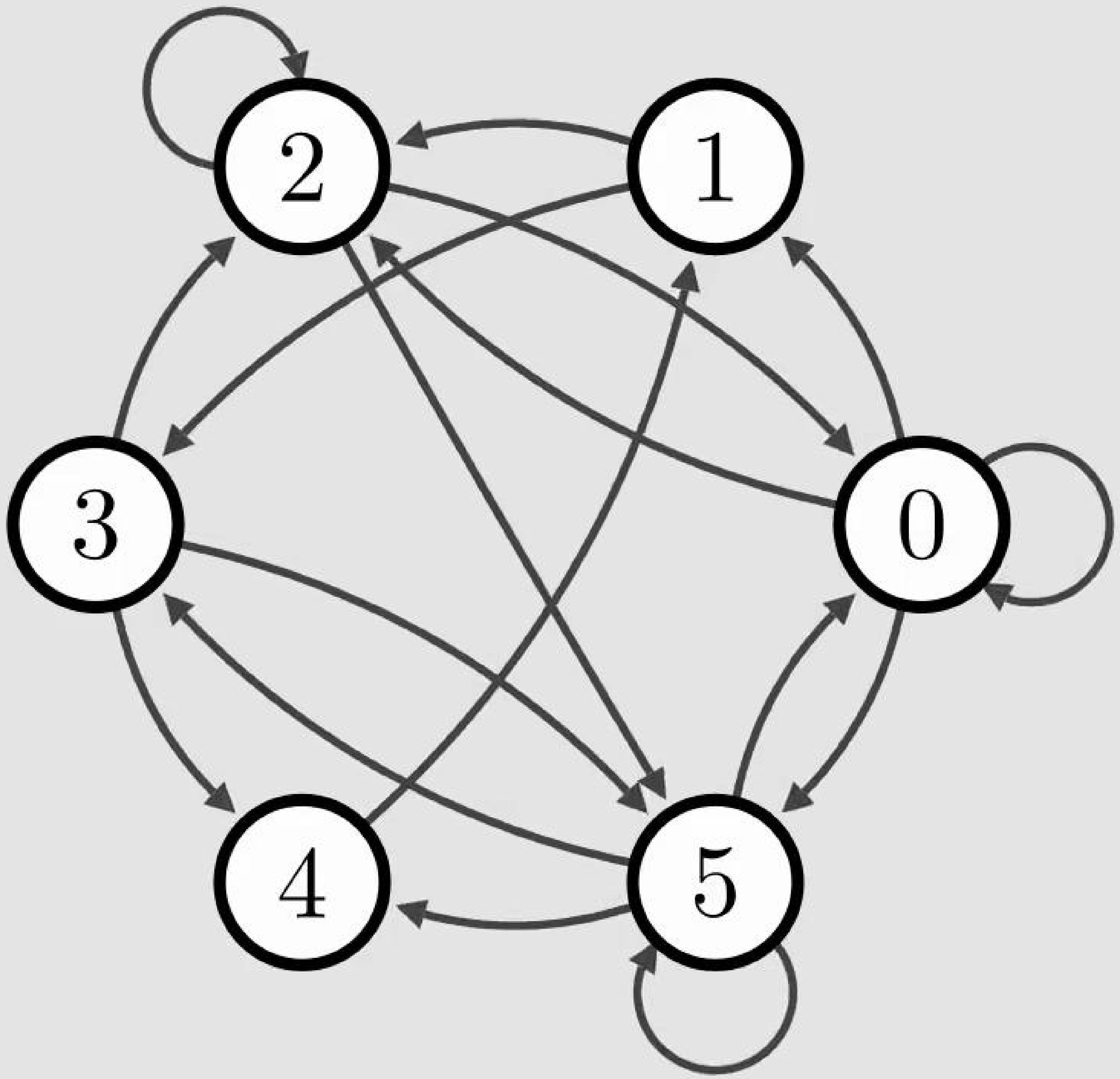
$$X_1 \in \{2, 3\}$$

$$X_2 \in \{0, 2, 5\}$$

$$X_3 \in \{2, 4, 5\}$$

$$X_4 \in \{1\}$$

$$X_5 \in \{0, 3, 4, 5\}$$



Consistency for Circuit:

$$X_0 \in \{0, 1, 2, 5\}$$

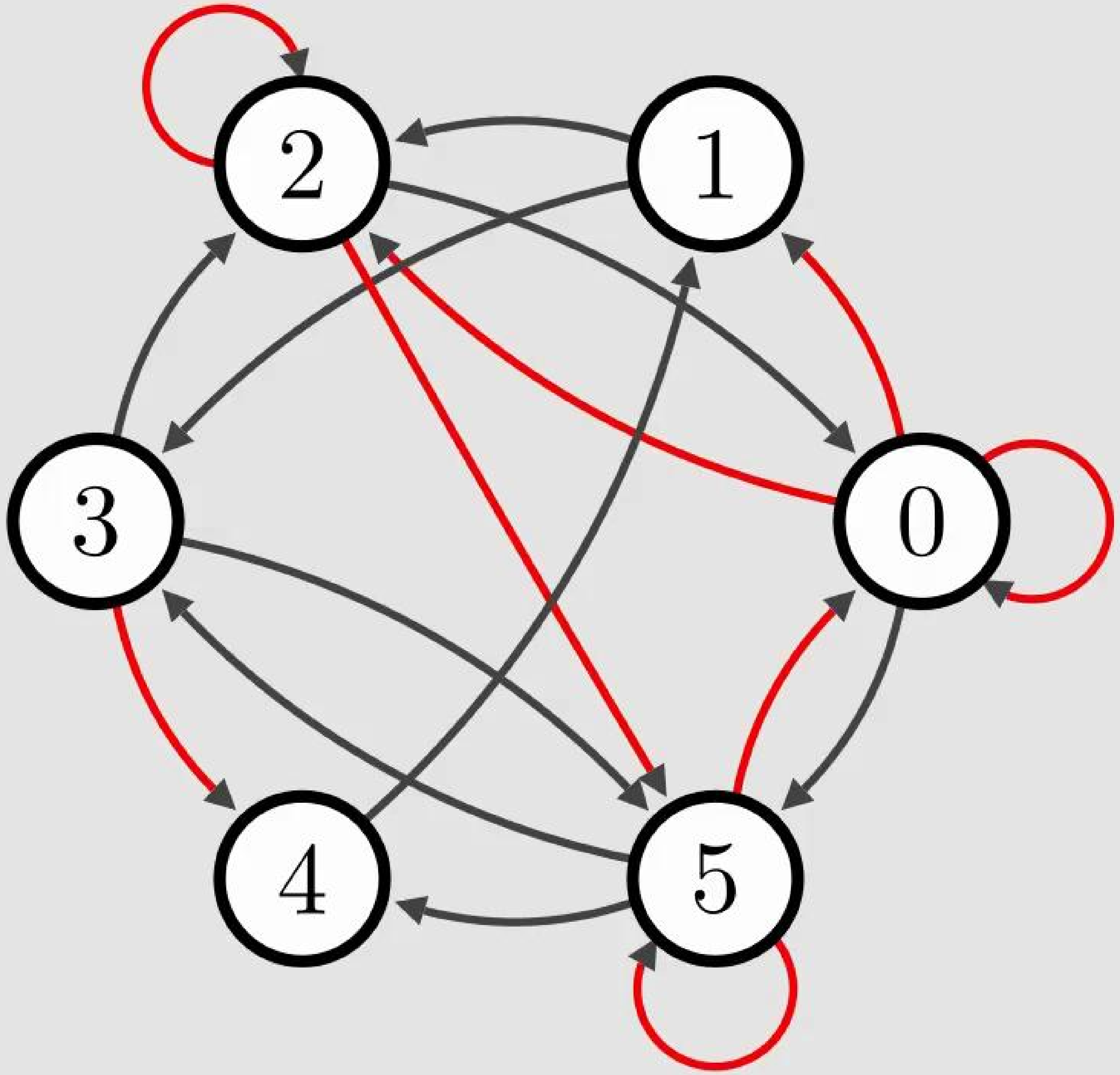
$$X_1 \in \{2, 3\}$$

$$X_2 \in \{0, 2, 5\}$$

$$X_3 \in \{2, 4, 5\}$$

$$X_4 \in \{1\}$$

$$X_5 \in \{0, 3, 4, 5\}$$



Consistency for Circuit:

$$X_0 \in \{5\}$$

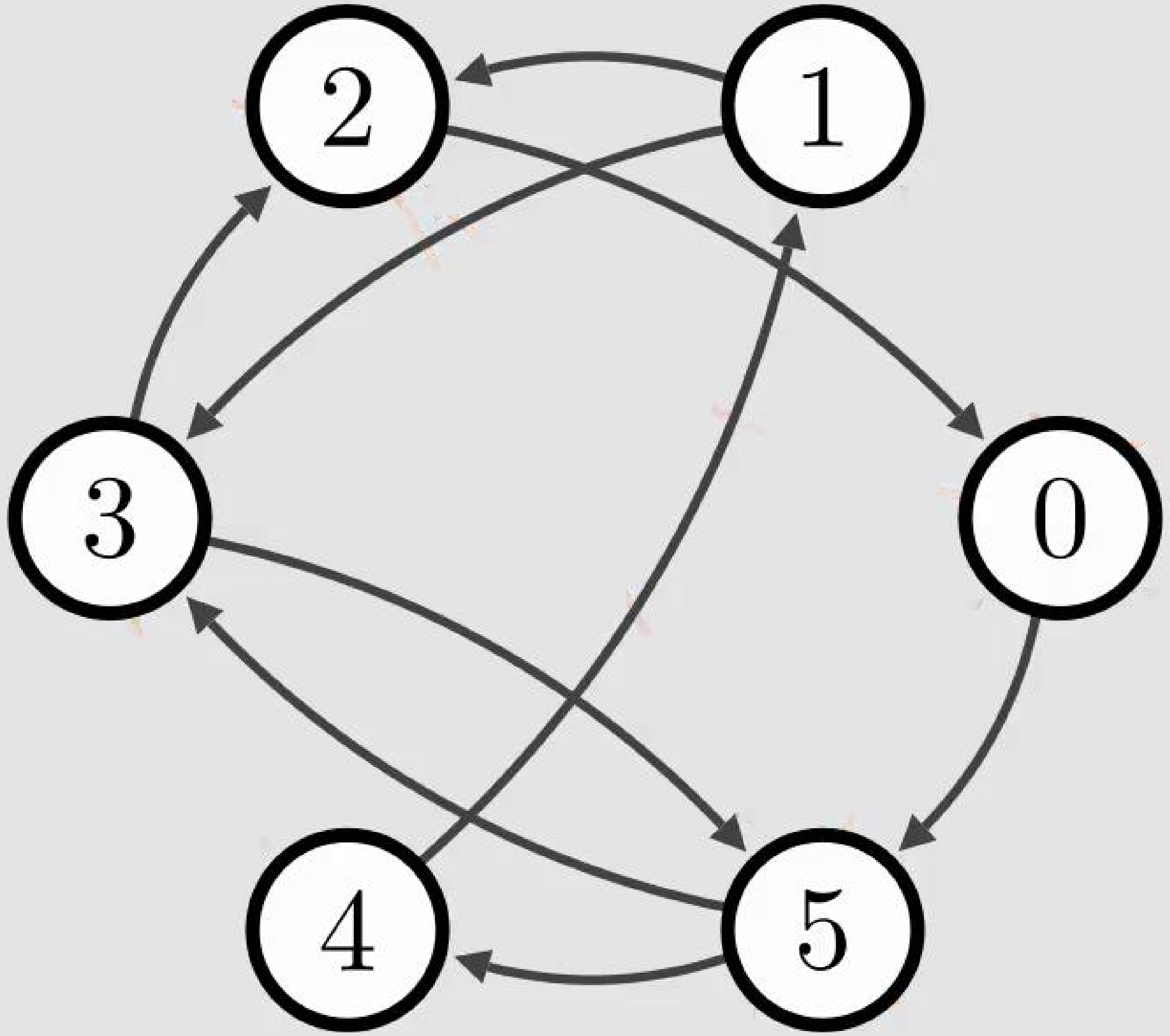
$$X_1 \in \{2, 3\}$$

$$X_2 \in \{0\}$$

$$X_3 \in \{2, 5\}$$

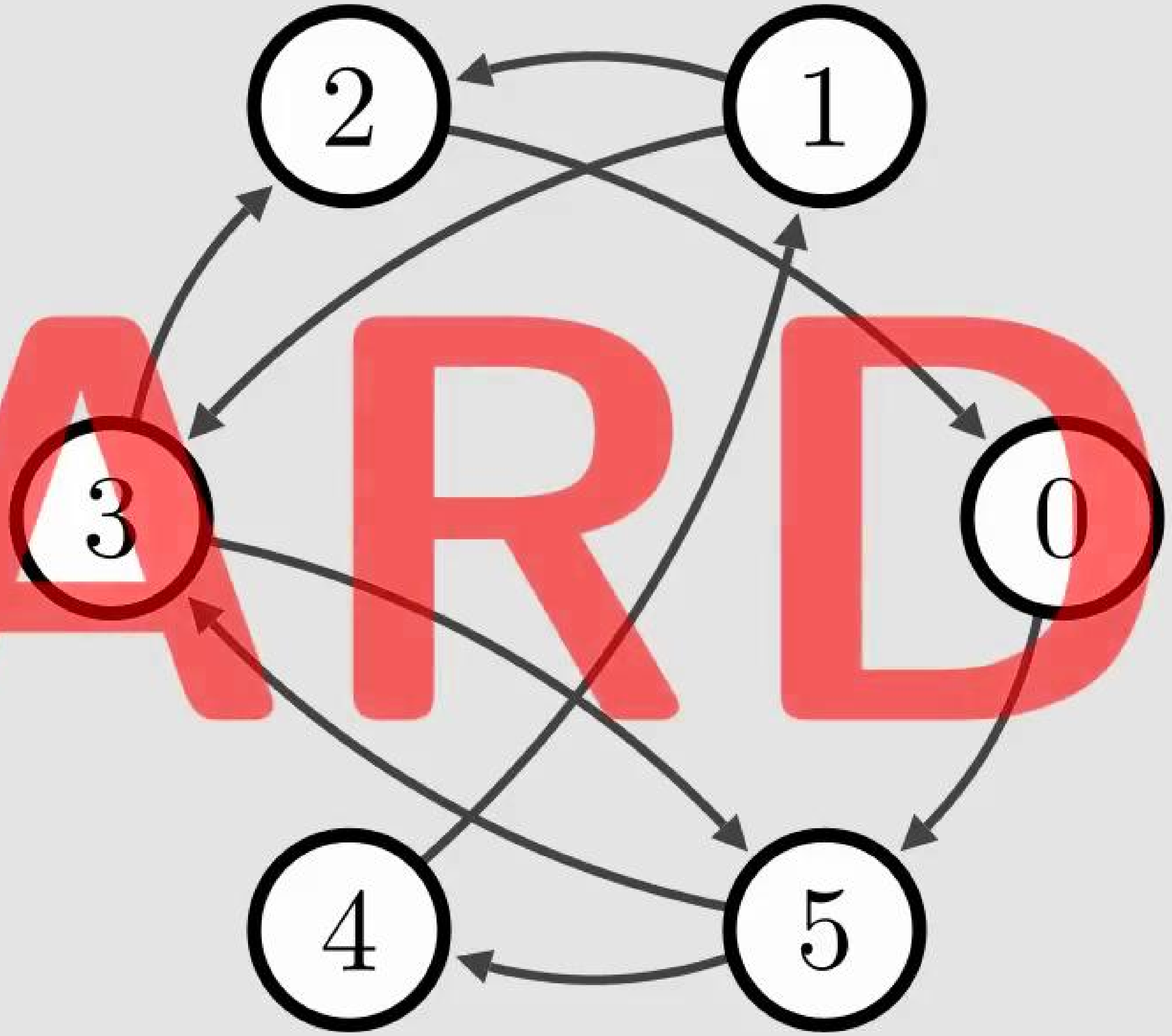
$$X_4 \in \{1\}$$

$$X_5 \in \{3, 4\}$$



Consistency for Circuit:

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



NP HARD

(Partial) Consistency for Circuit

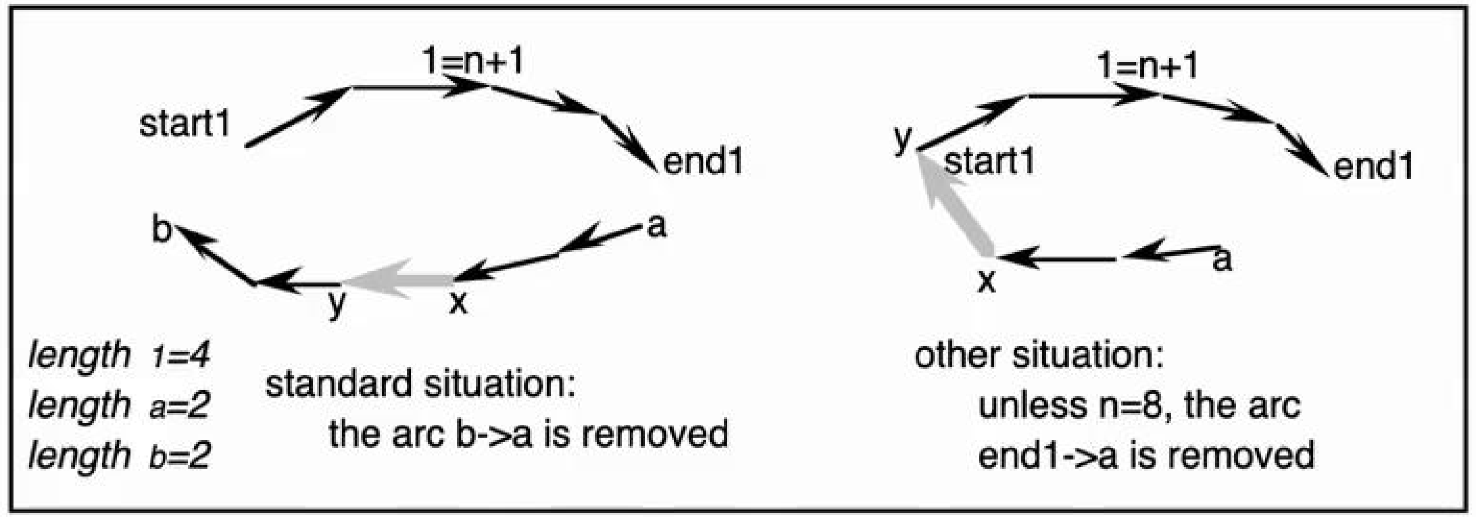


Figure 1: Propagation of the nocycle constraint

- If $x{=}end_1$ and $length_1{+}length_b{<}n\text{-}2$ we infer $Next(b) \neq start_1$.
- If $y{=}start_1$ and $length_1{+}length_a{<}n\text{-}2$ we infer $Next(end_1) \neq a$
- Otherwise, we infer $Next(b) \neq a$.

Caseau, Y. and Laburthe, F., 1997, July.
Solving Small TSPs with Constraints. In ICLP (Vol. 97, p. 104).

16 Constraints (2014) 19:1–29

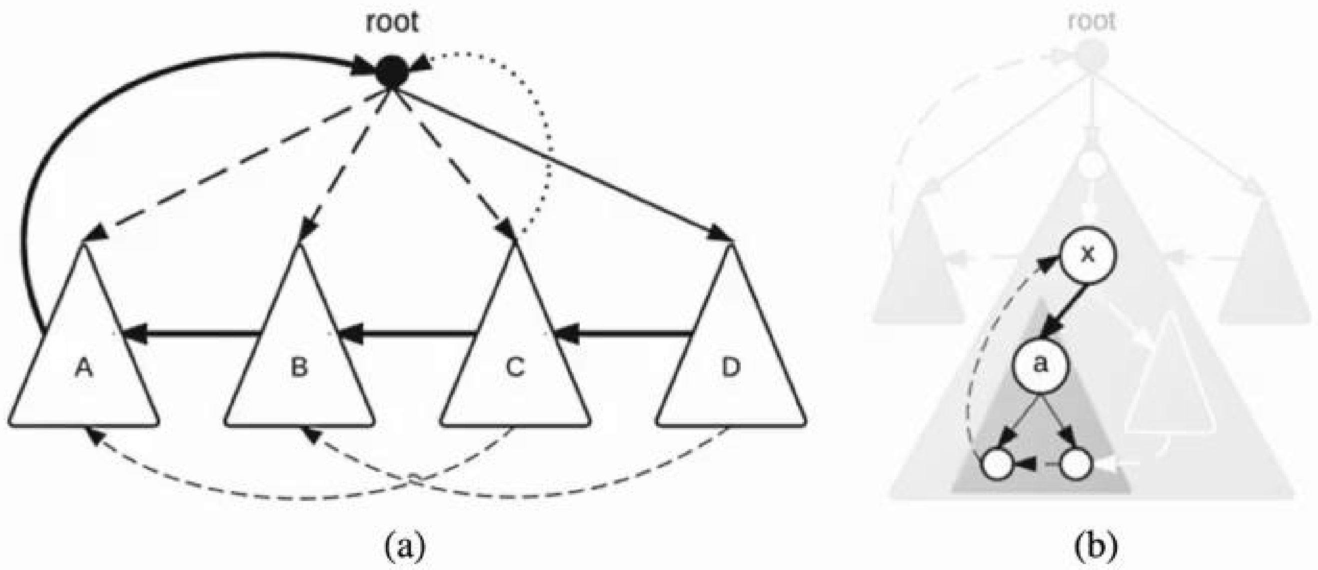
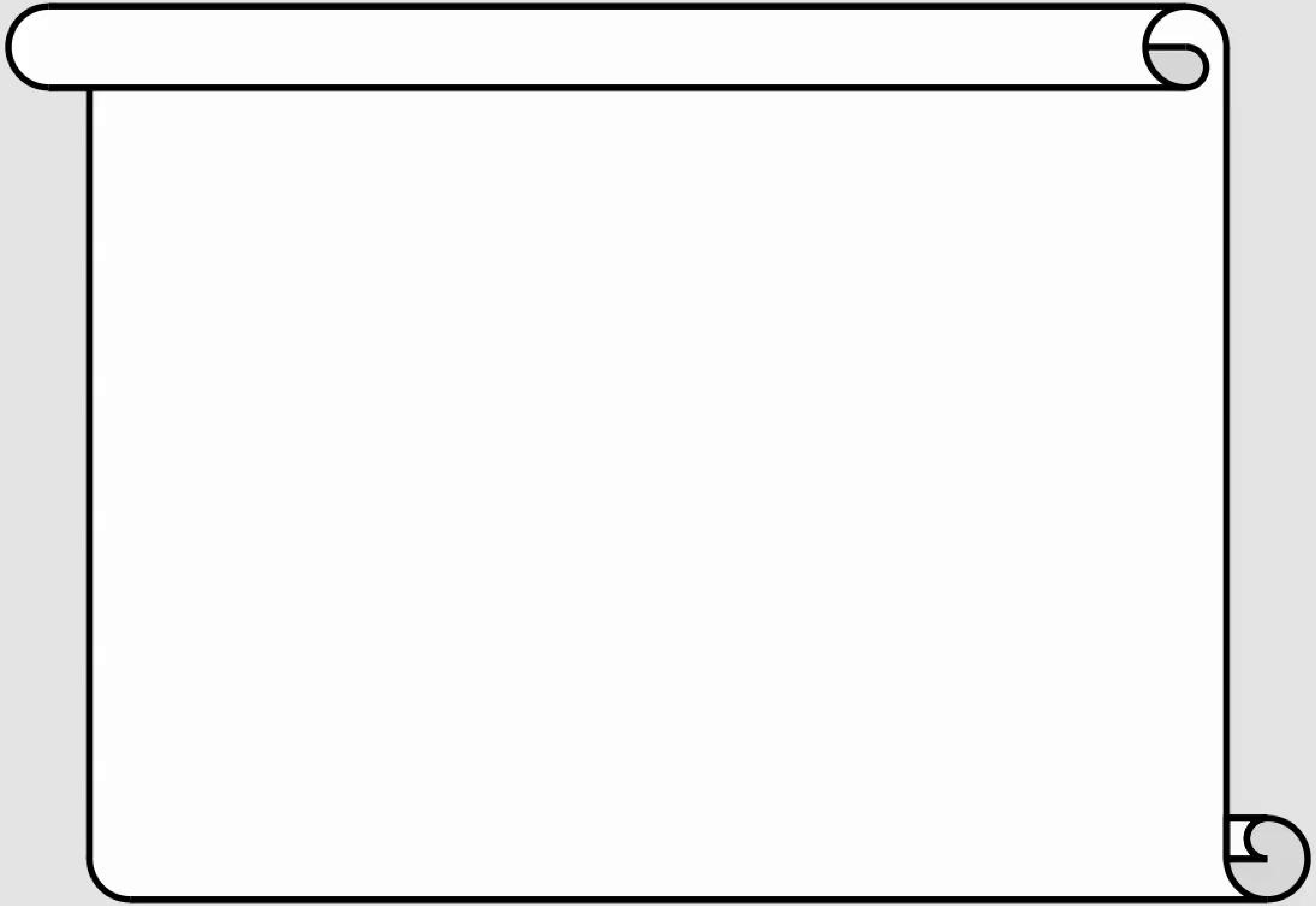


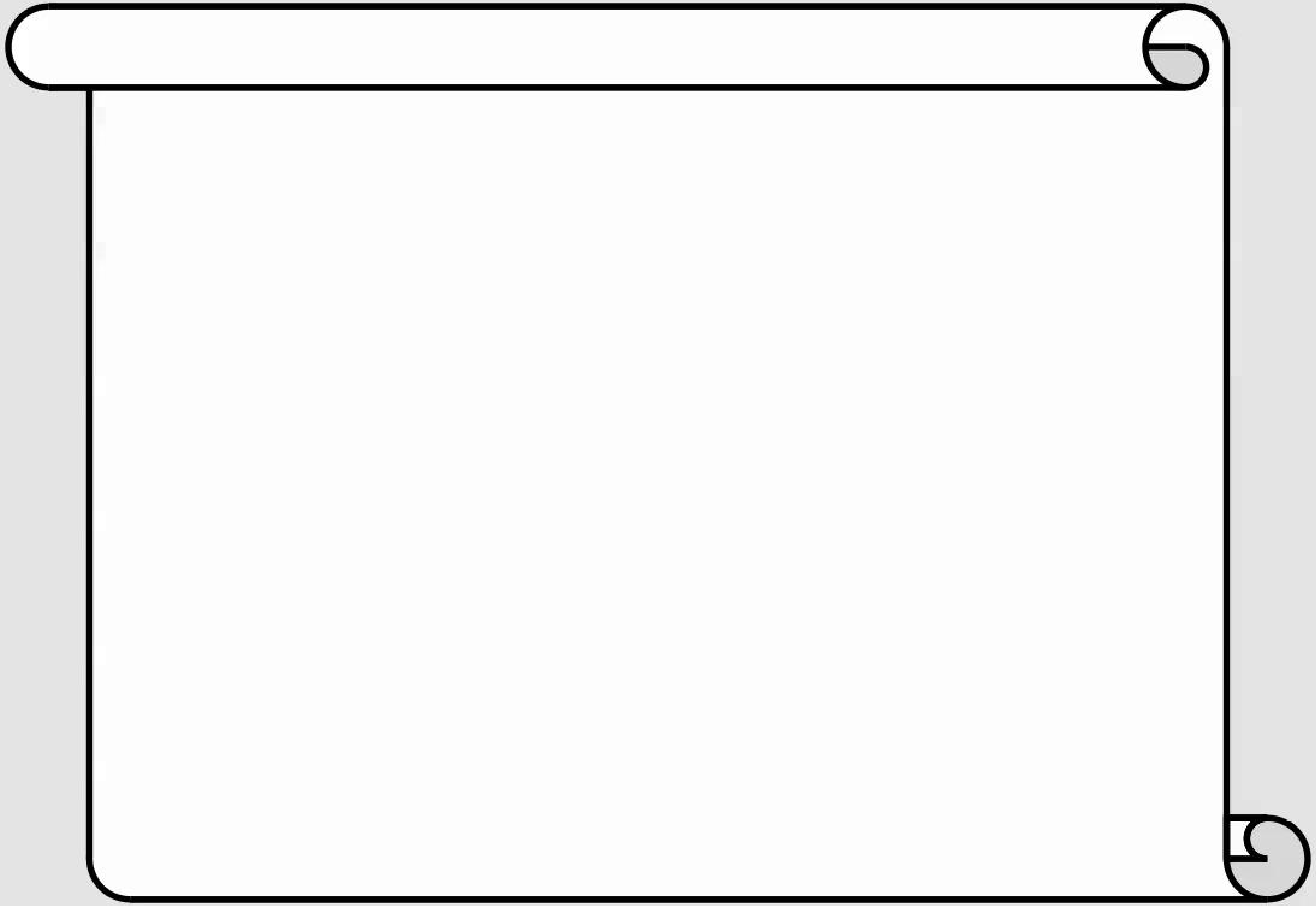
Fig. 5 **a** The SCC exploration graph for *circuit* starting from root. At least one (*thick*) edge from A to the root, from D to C, C to B, and B to A must exist (rule 1). Backwards (*dotted*) edges to the root from B, C or D cannot be used (rule 1). The (*thin-dashed*) edges from C to A and D to B cannot be used (rule 2). The (*thick-dashed*) edges leading from root to A, B and C cannot be used (rule 3). **b** Illustration of *prune-within* (rule 4). The edge from x to a cannot be used otherwise we cannot escape the subtree rooted at a (*dark grey*). We need to enter the subtree from elsewhere

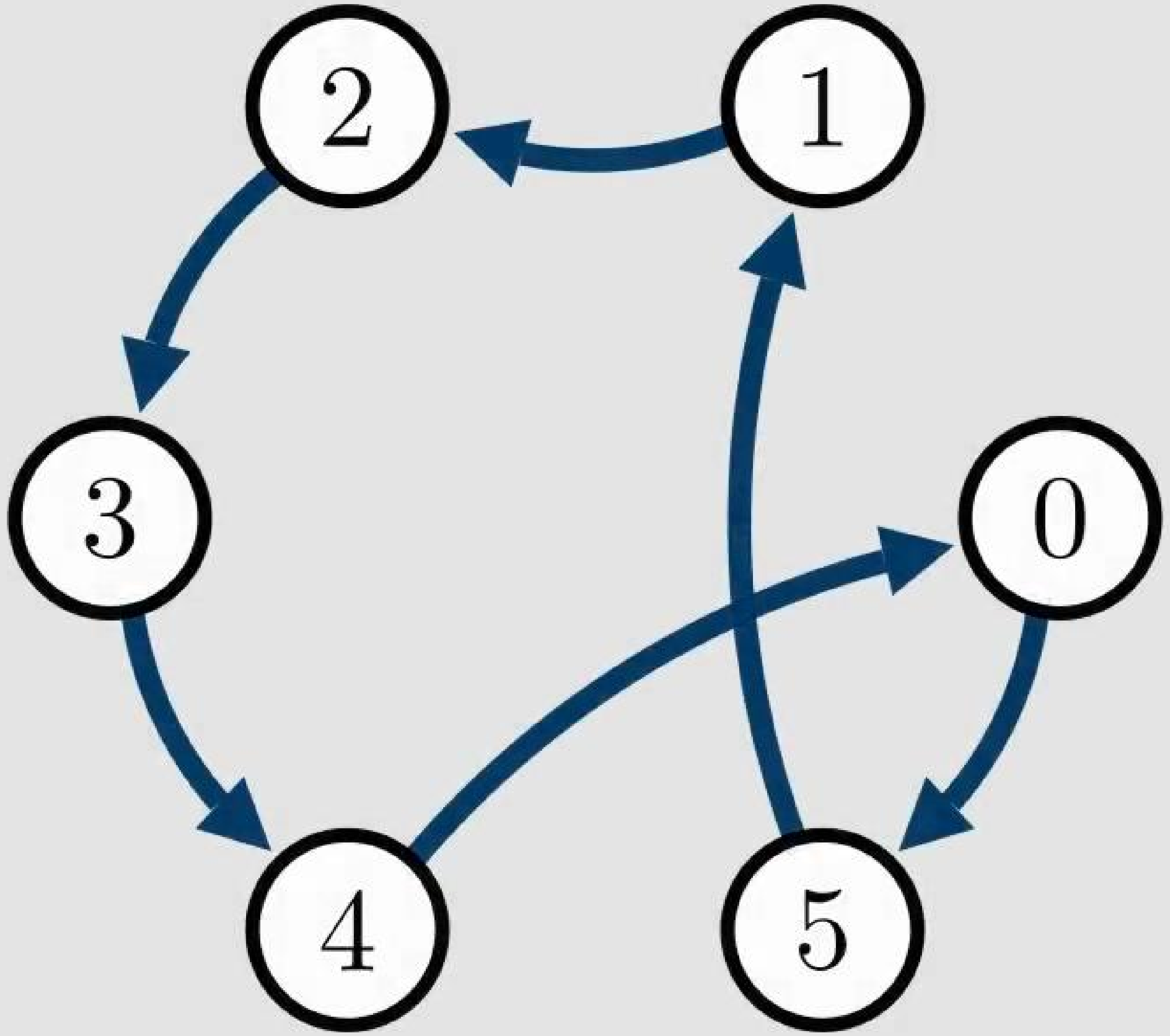
Francis, K.G. and Stuckey, P.J., 2014.
Explaining circuit propagation. Constraints, 19, pp.1-29.

Circuit PB Encoding



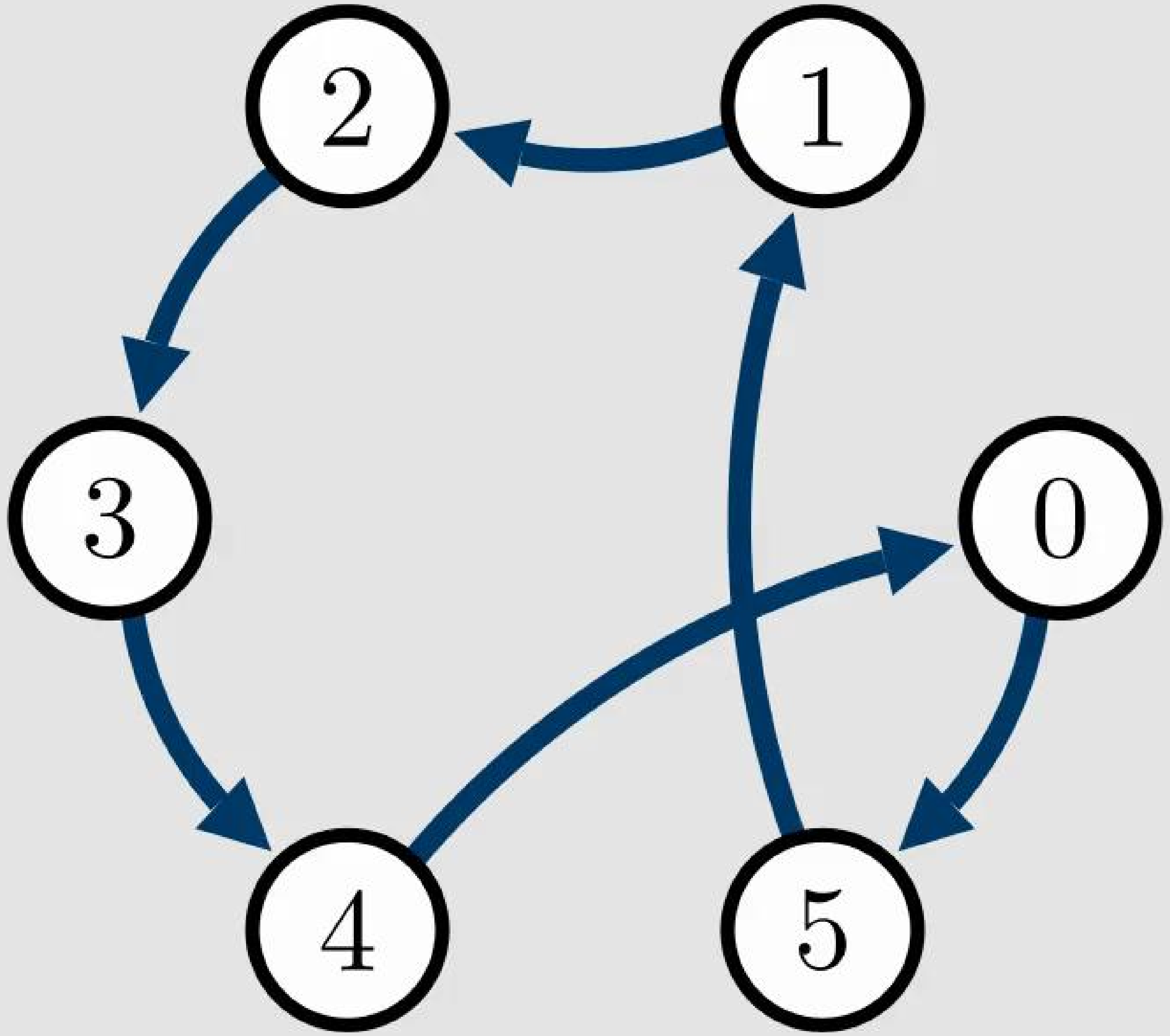
Circuit PB Encoding





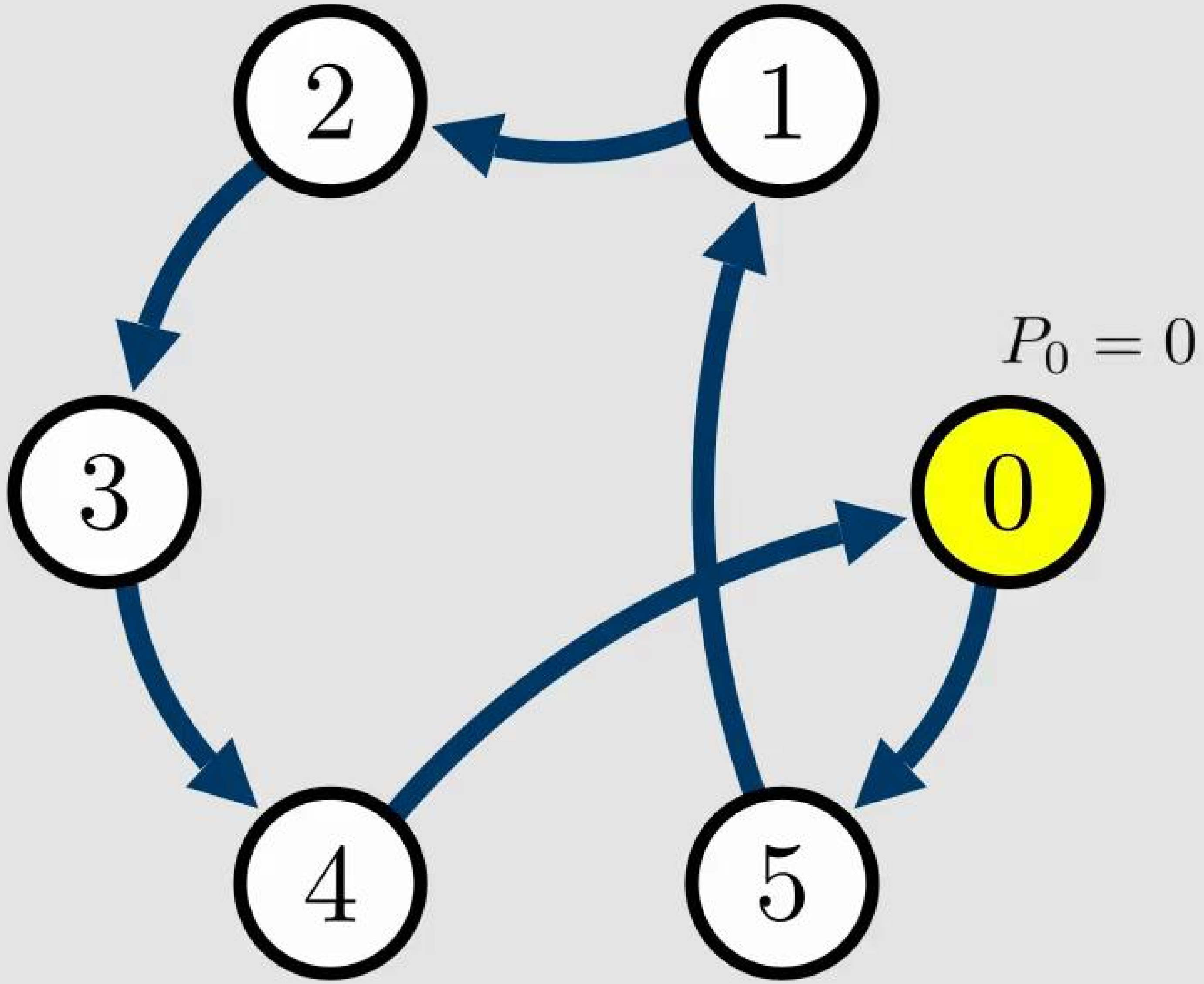
Circuit PB Encoding

$bits(P_i) :=$ Position of vertex i relative to 0



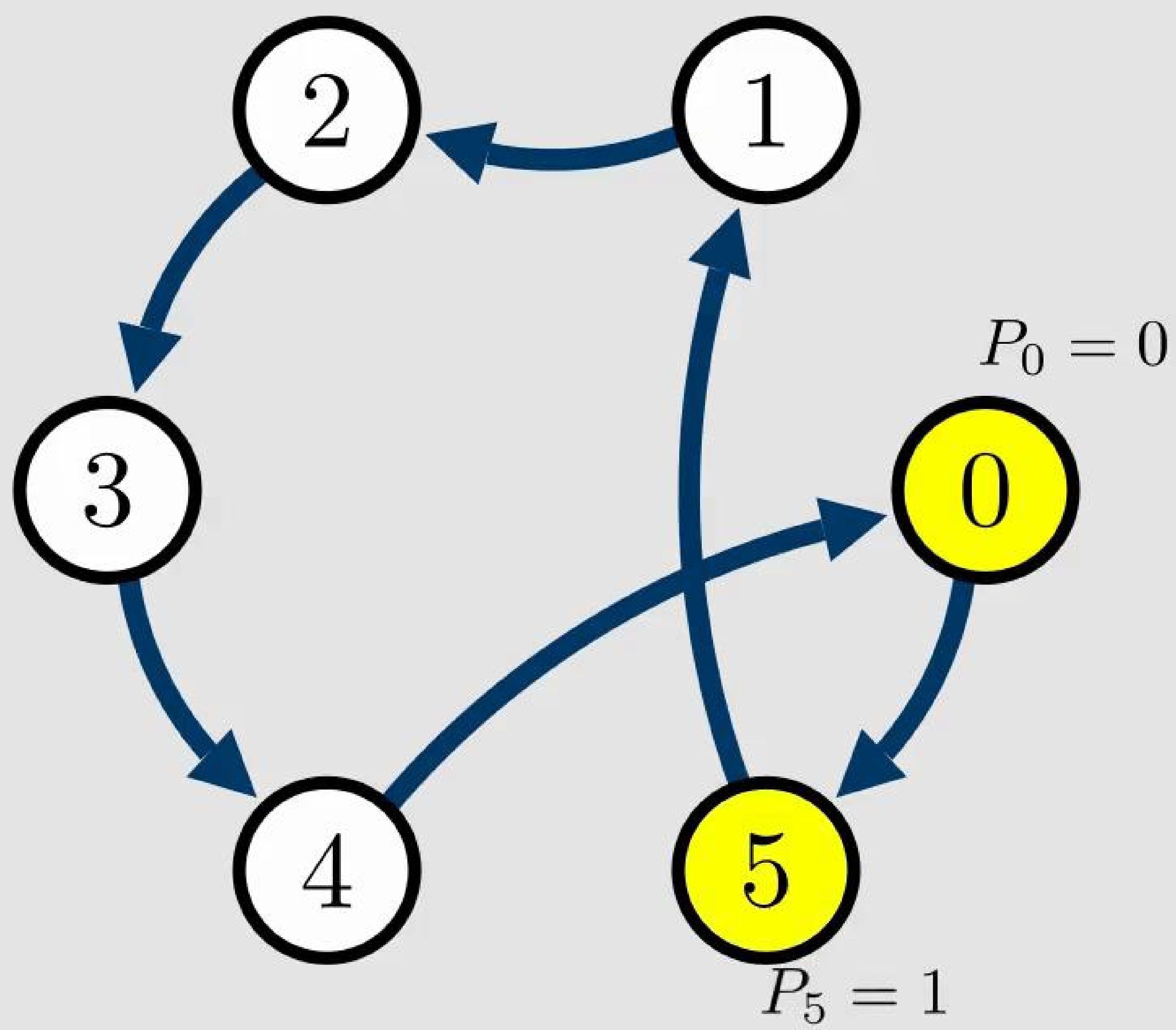
Circuit PB Encoding

$bits(P_i) :=$ Position of vertex i relative to 0



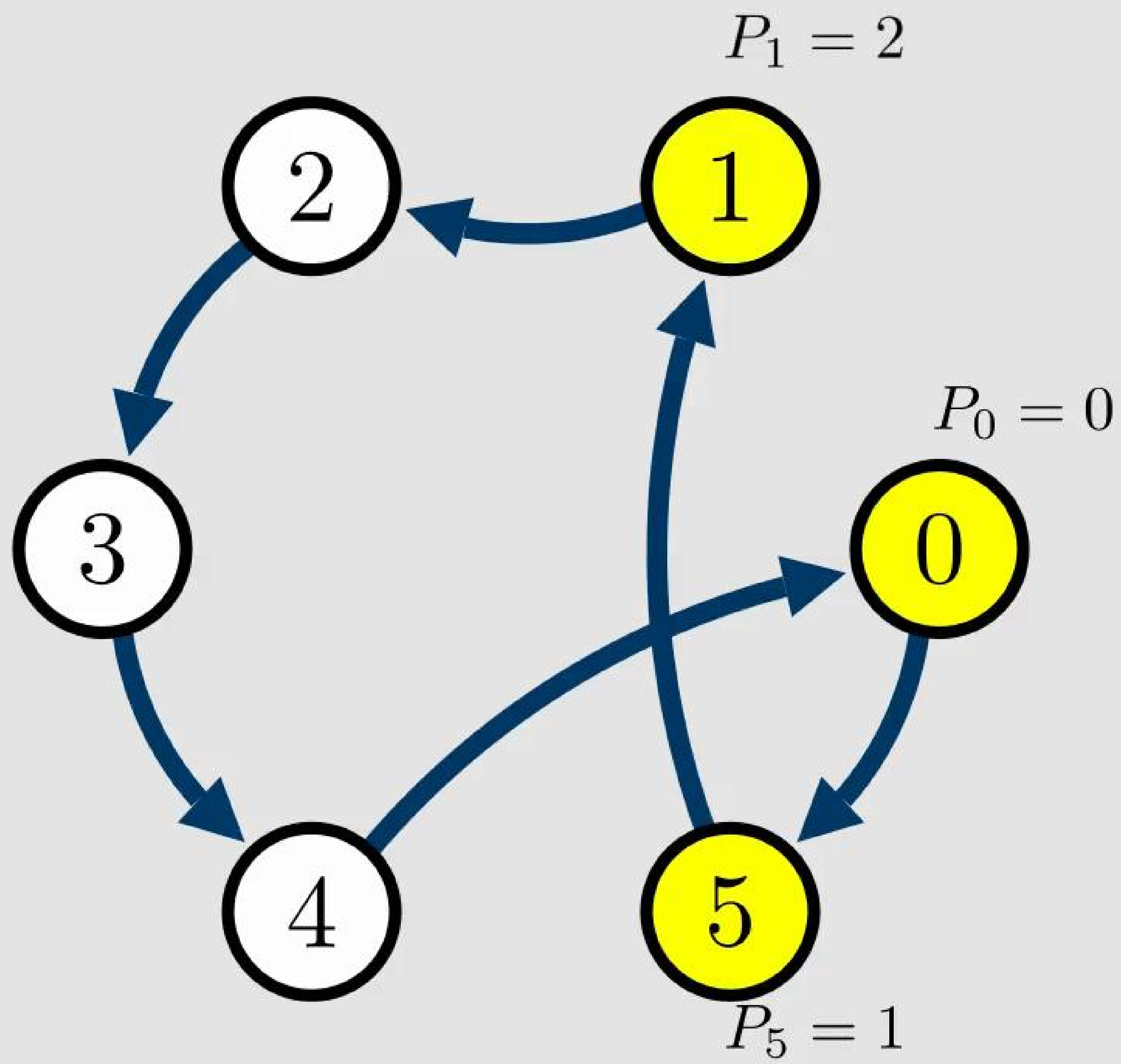
Circuit PB Encoding

$bits(P_i) :=$ Position of vertex i relative to 0



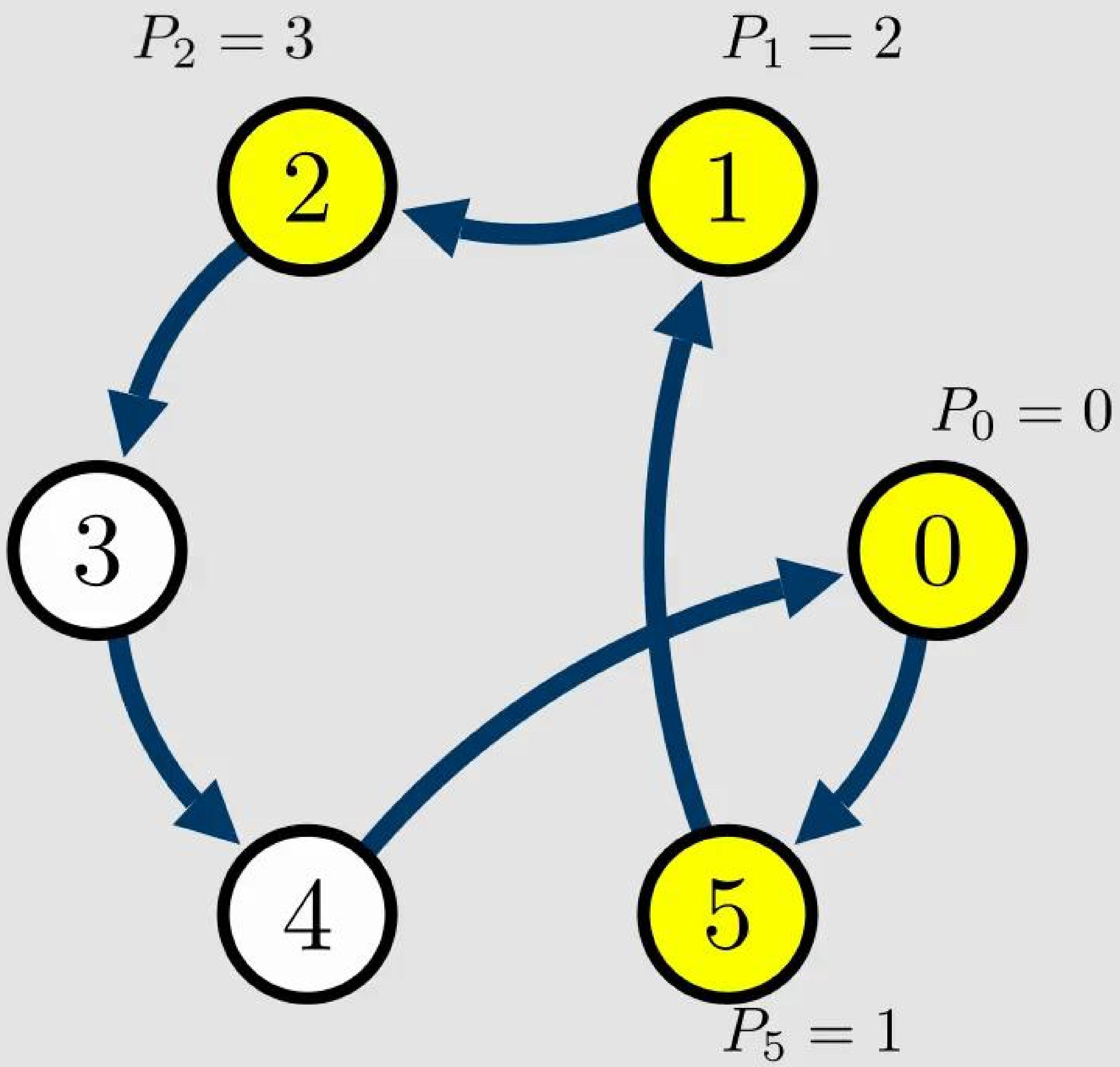
Circuit PB Encoding

$bits(P_i) :=$ Position of vertex i relative to 0



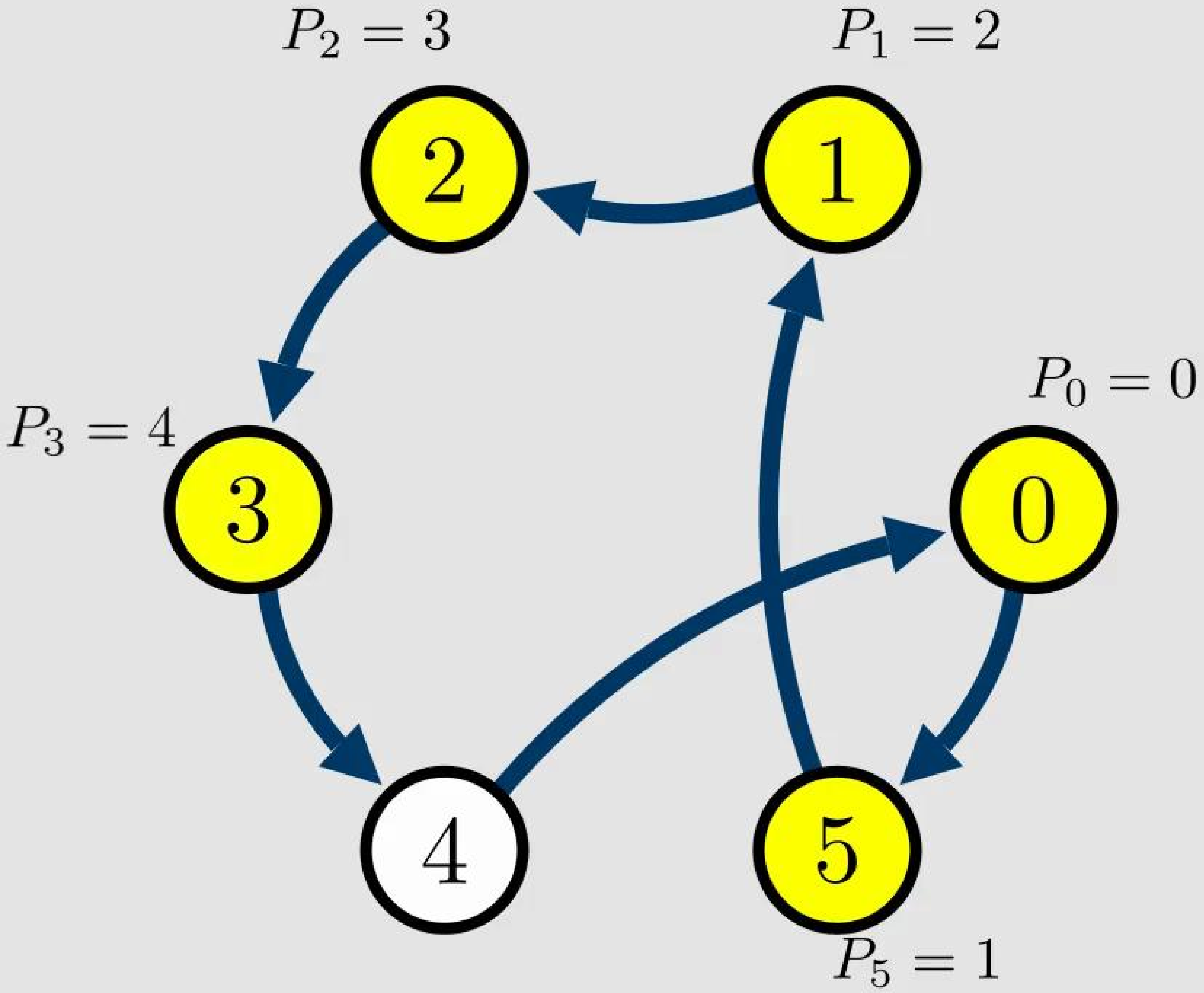
Circuit PB Encoding

$bits(P_i) :=$ Position of vertex i relative to 0



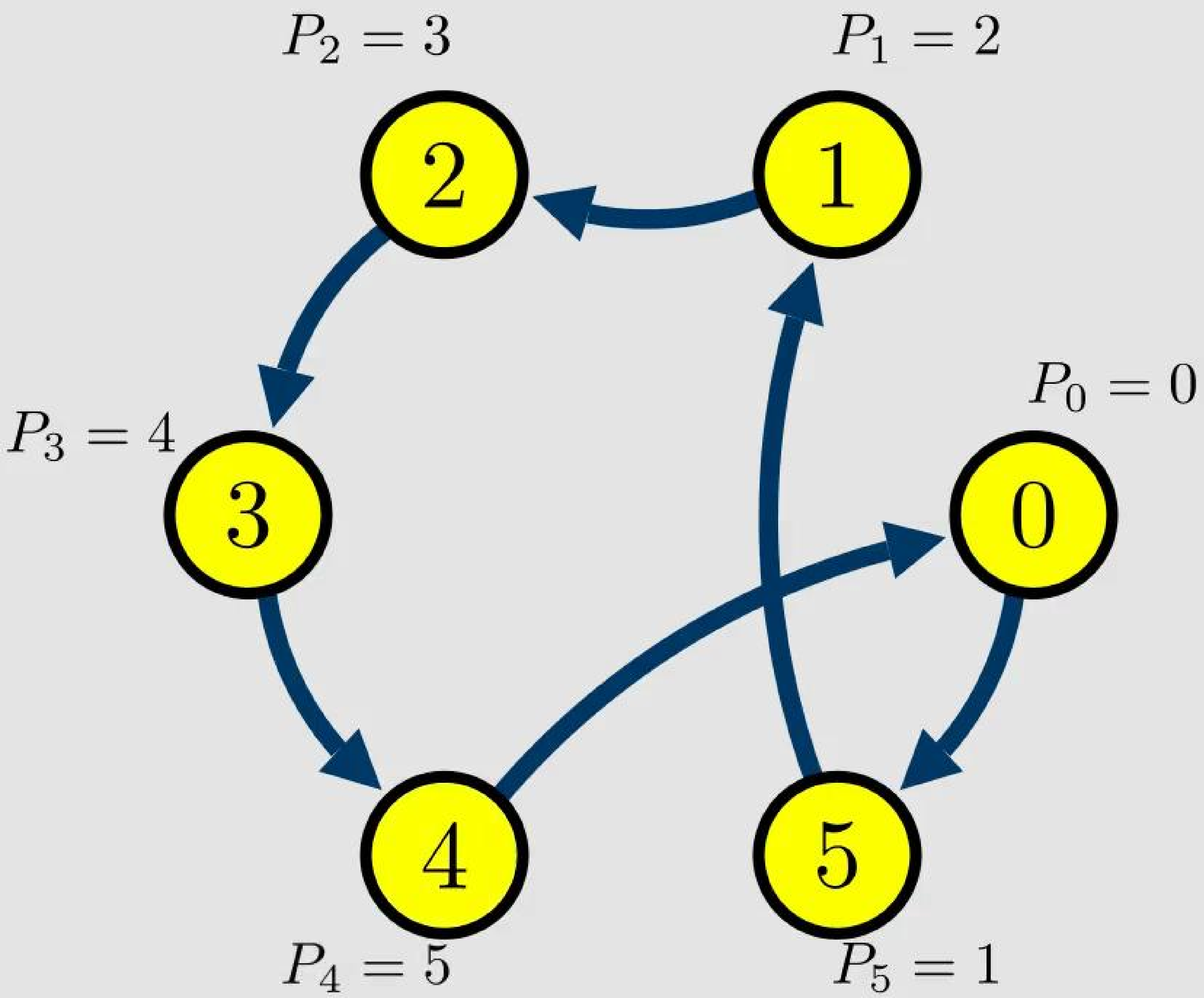
Circuit PB Encoding

$bits(P_i) :=$ Position of vertex i relative to 0



Circuit PB Encoding

$bits(P_i)$:= Position of vertex i relative to 0

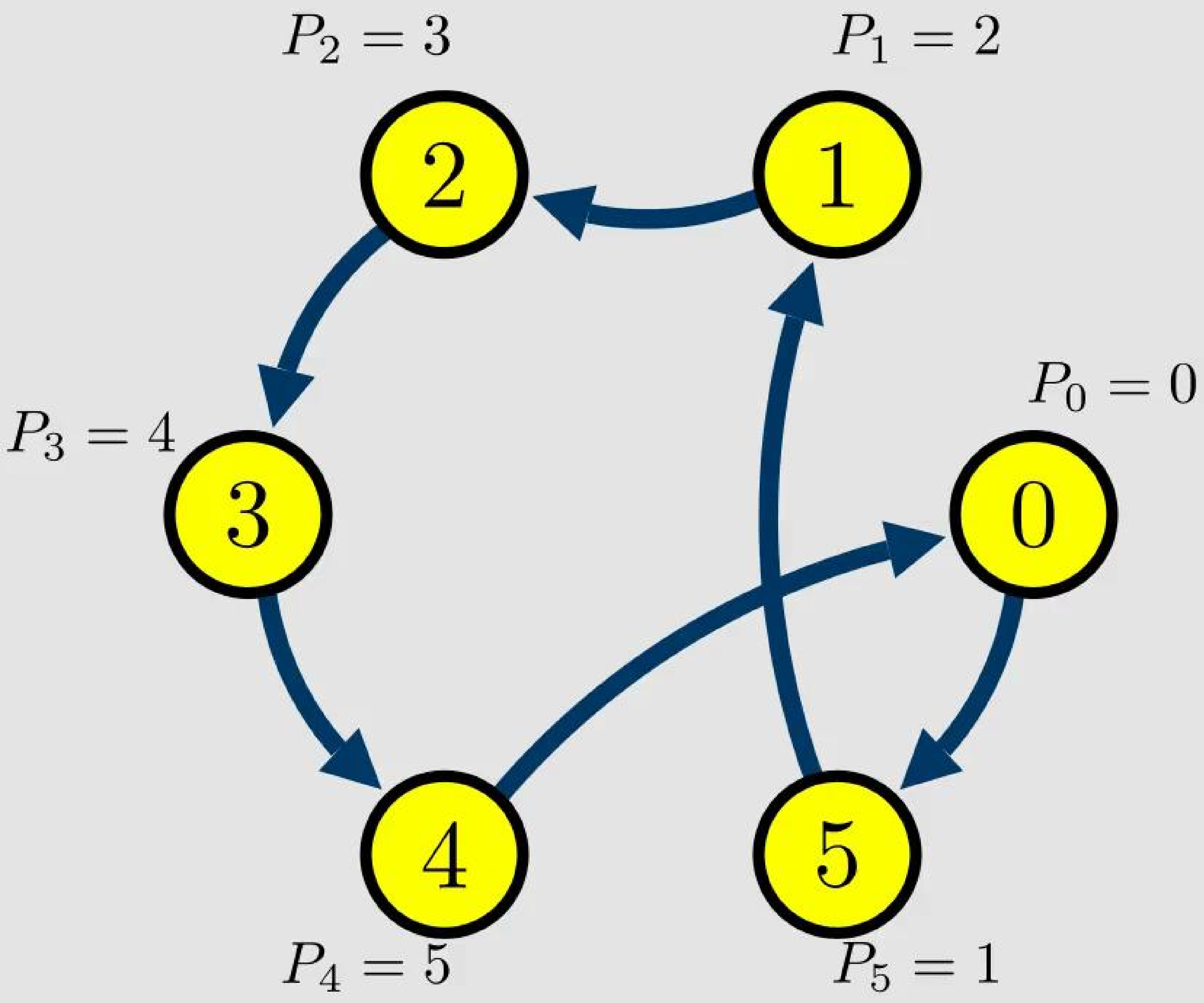


Circuit PB Encoding

$bits(P_i) :=$ Position of vertex i relative to 0

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

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

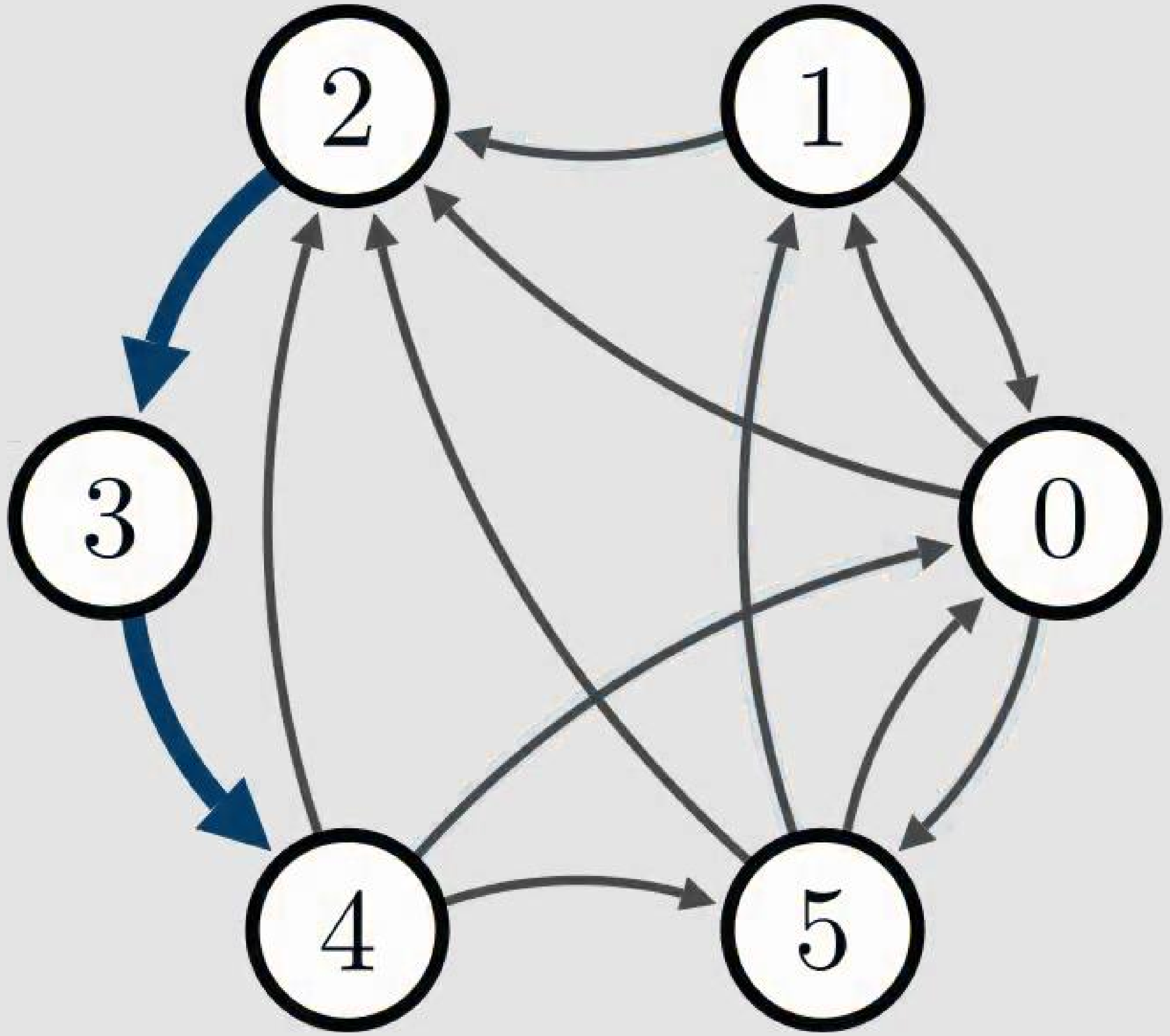


Circuit PB Encoding

$bits(P_i) :=$ Position of vertex i relative to 0

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

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

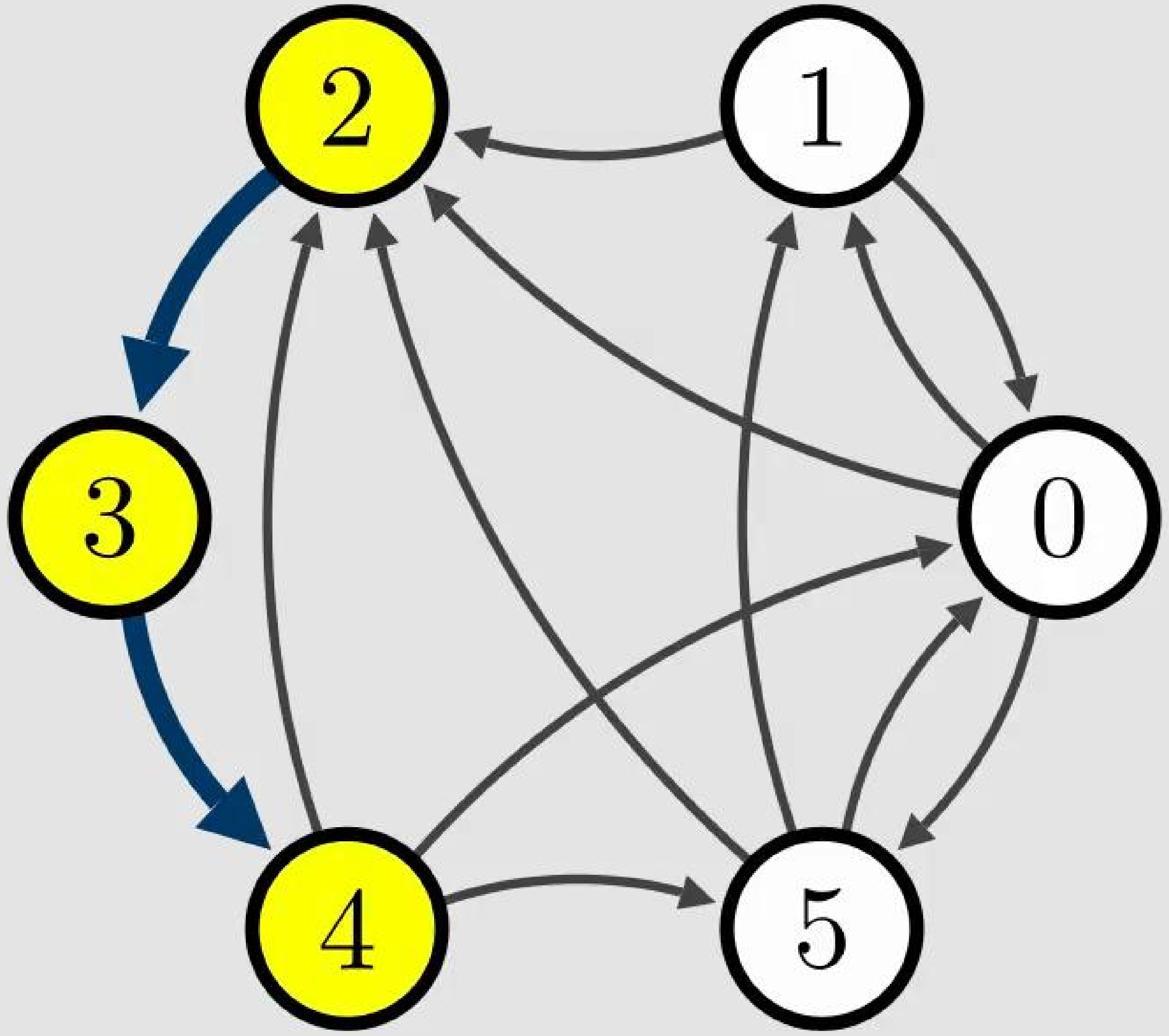


Circuit PB Encoding

$bits(P_i)$:= Position of vertex i relative to 0

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

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

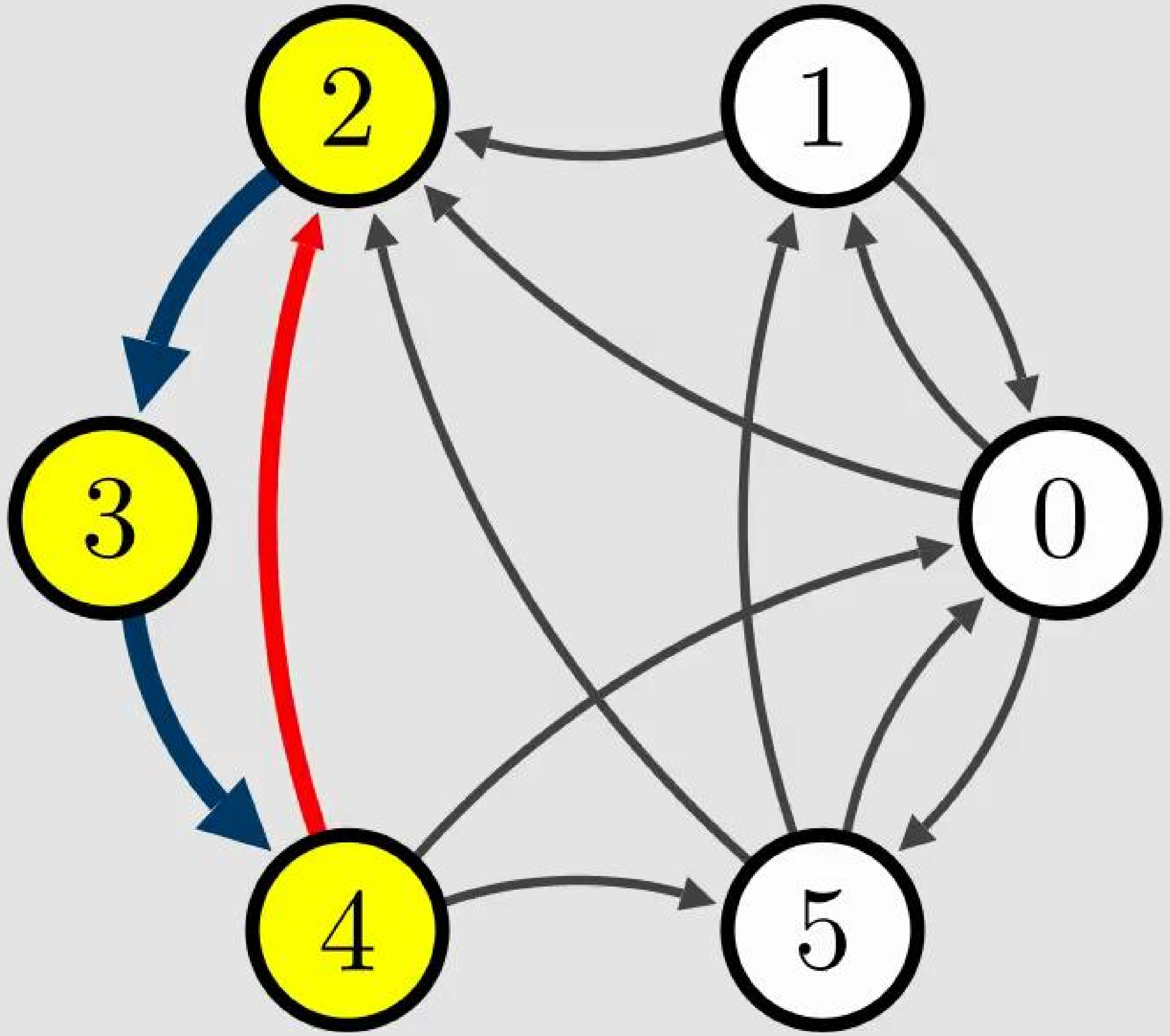


Circuit PB Encoding

$bits(P_i)$:= Position of vertex i relative to 0

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

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

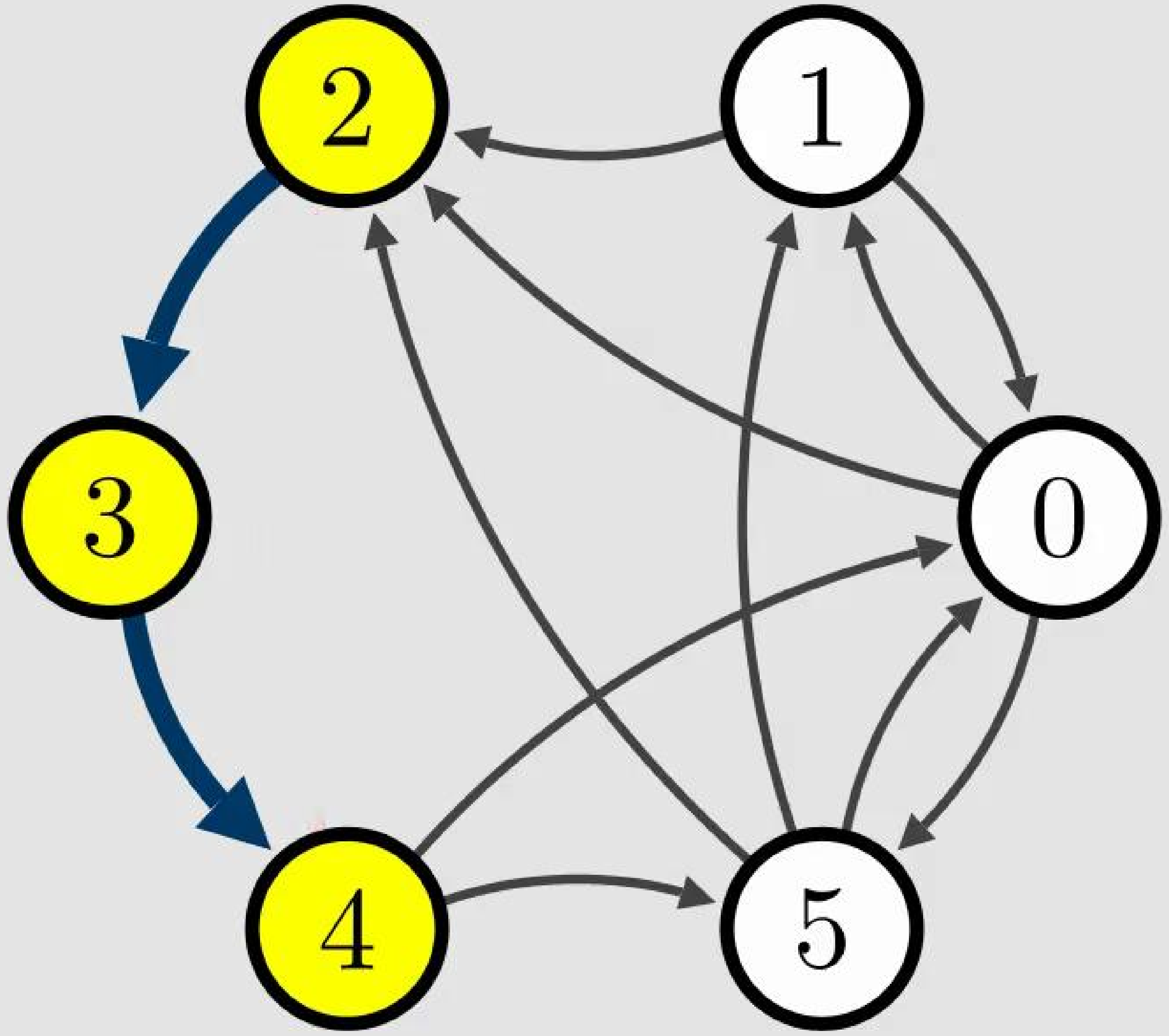


Circuit PB Encoding

$bits(P_i) :=$ Position of vertex i relative to 0

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

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



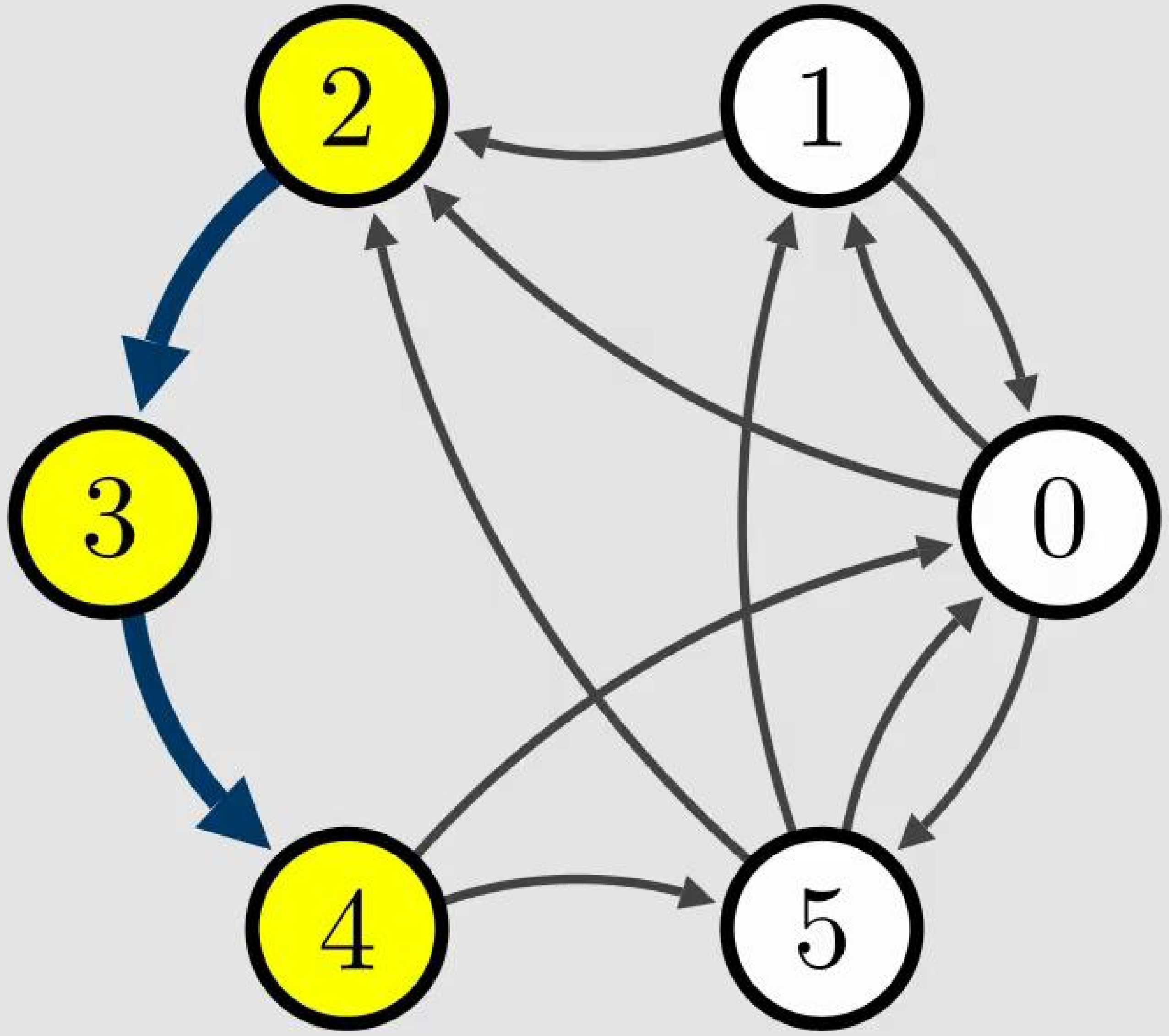
Circuit PB Encoding

$bits(P_i)$:= Position of vertex i relative to 0

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

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

From encoding:



Circuit PB Encoding

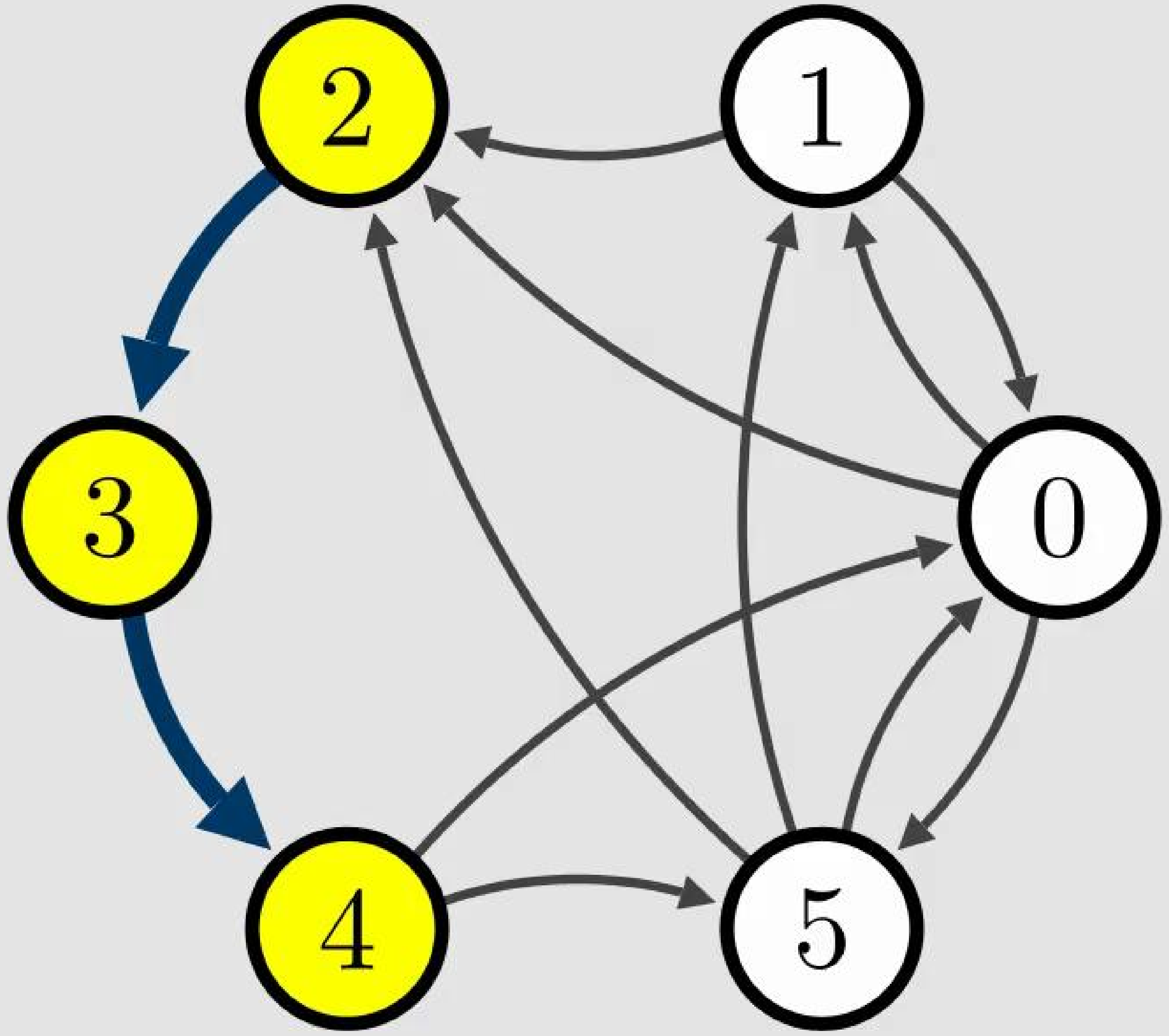
$bits(P_i)$:= Position of vertex i relative to 0

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

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

From encoding:

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



Circuit PB Encoding

$bits(P_i)$:= Position of vertex i relative to 0

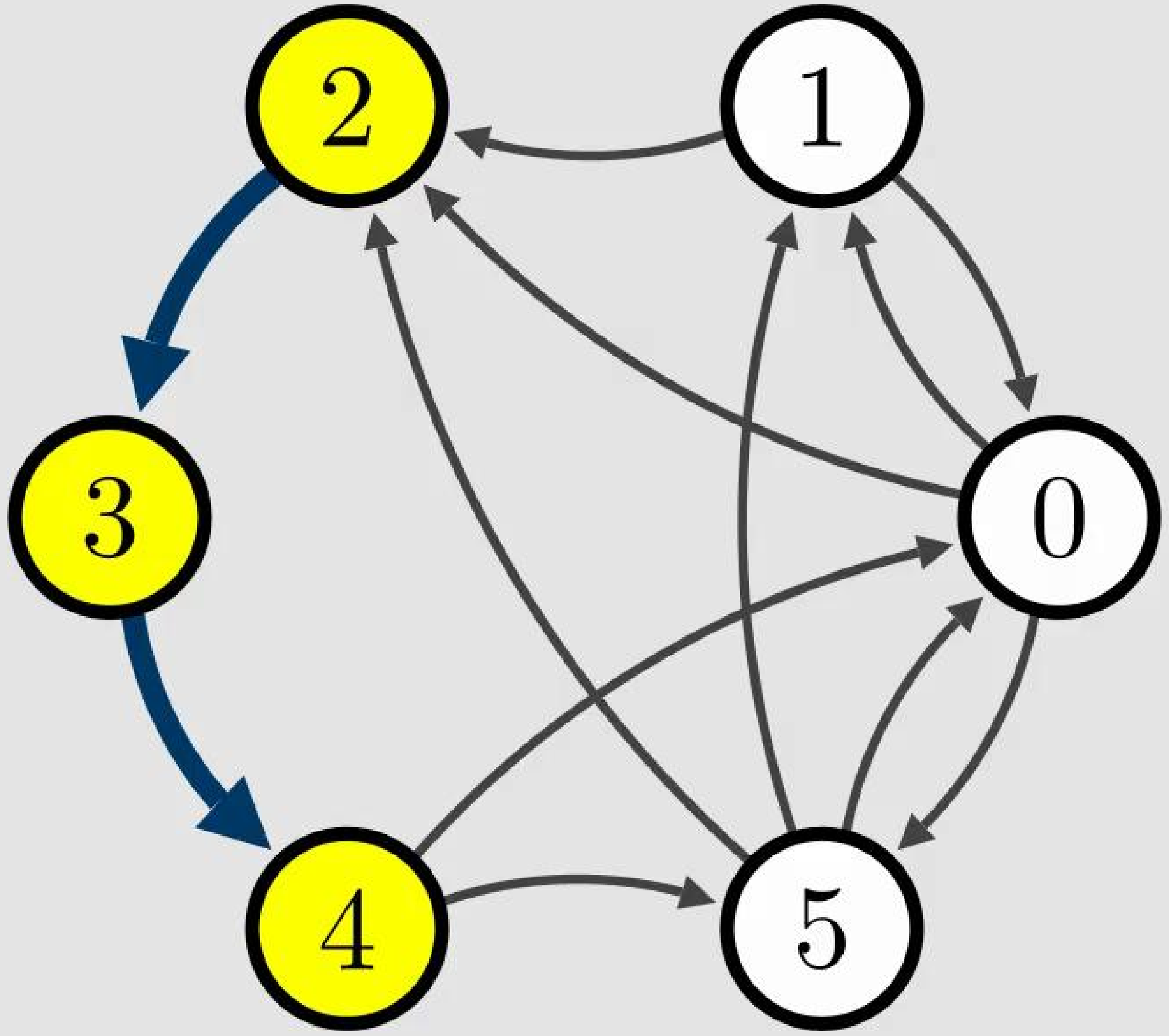
For each $X_i, j \in dom(X_i)\ j \neq 0$:

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

From encoding:

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

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



Circuit PB Encoding

$bits(P_i)$:= Position of vertex i relative to 0

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

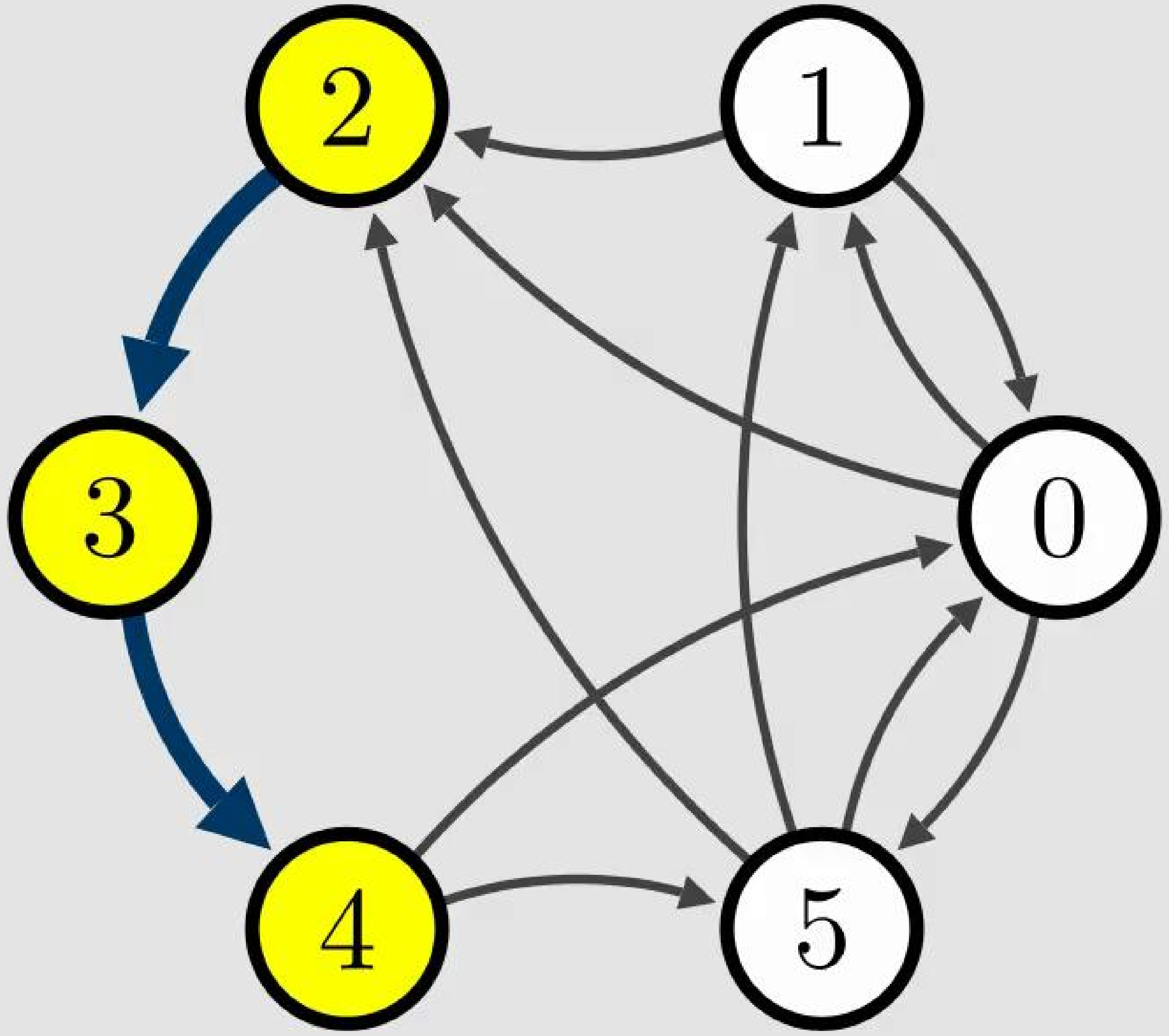
$x_{i=j} \implies bits(P_j) = bits(P_i) + 1$

From encoding:

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

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

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



Circuit PB Encoding

$bits(P_i)$:= Position of vertex i relative to 0

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

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

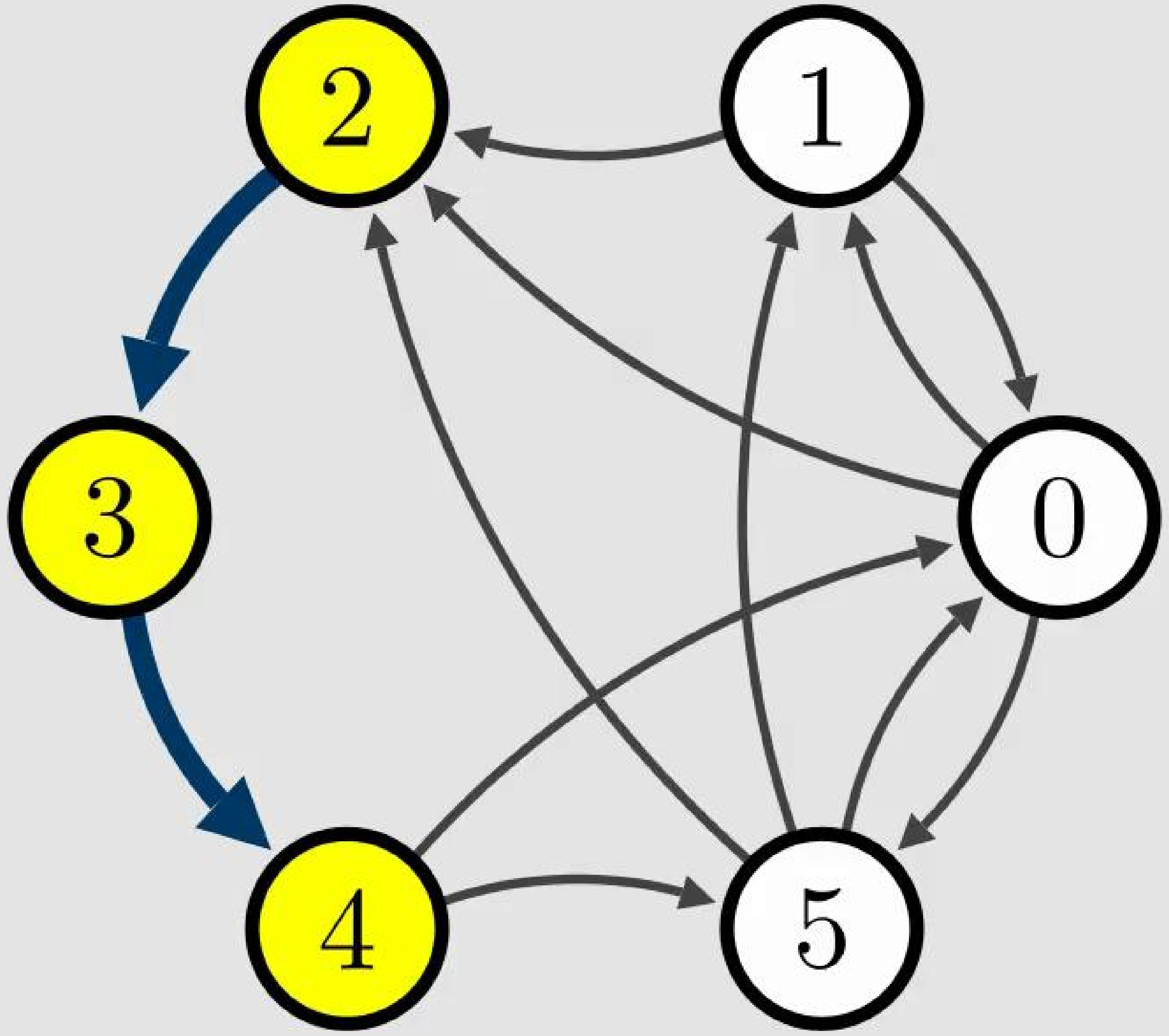
From encoding:

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

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

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

Cutting planes addition:



Circuit PB Encoding

$bits(P_i) :=$ Position of vertex i relative to 0

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

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

From encoding:

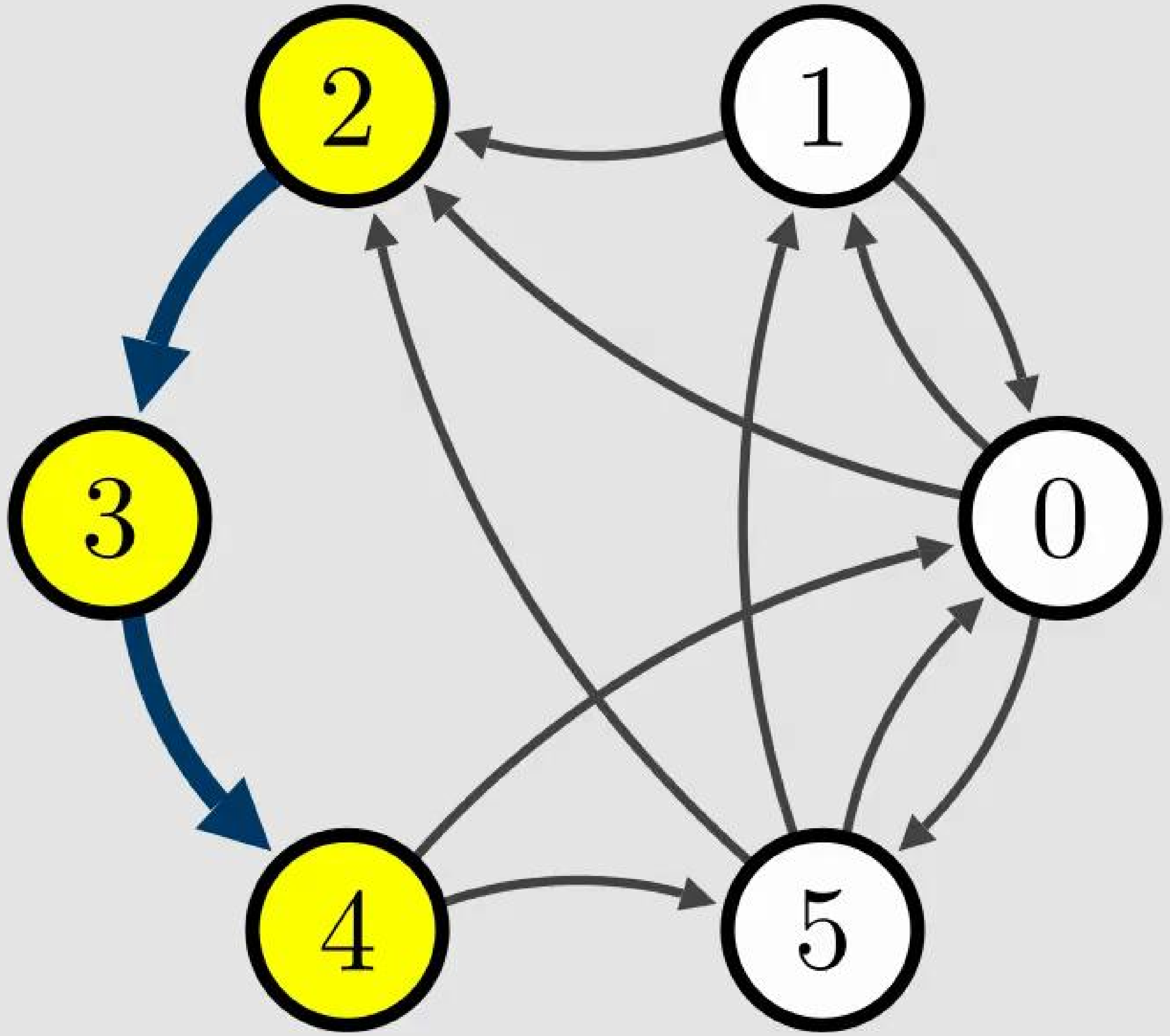
$x_{2=3} \implies bits(P_3) = bits(P_2) + 1$

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

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

Cutting planes addition:

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



Circuit PB Encoding

$bits(P_i) :=$ Position of vertex i relative to 0

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

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

From encoding:

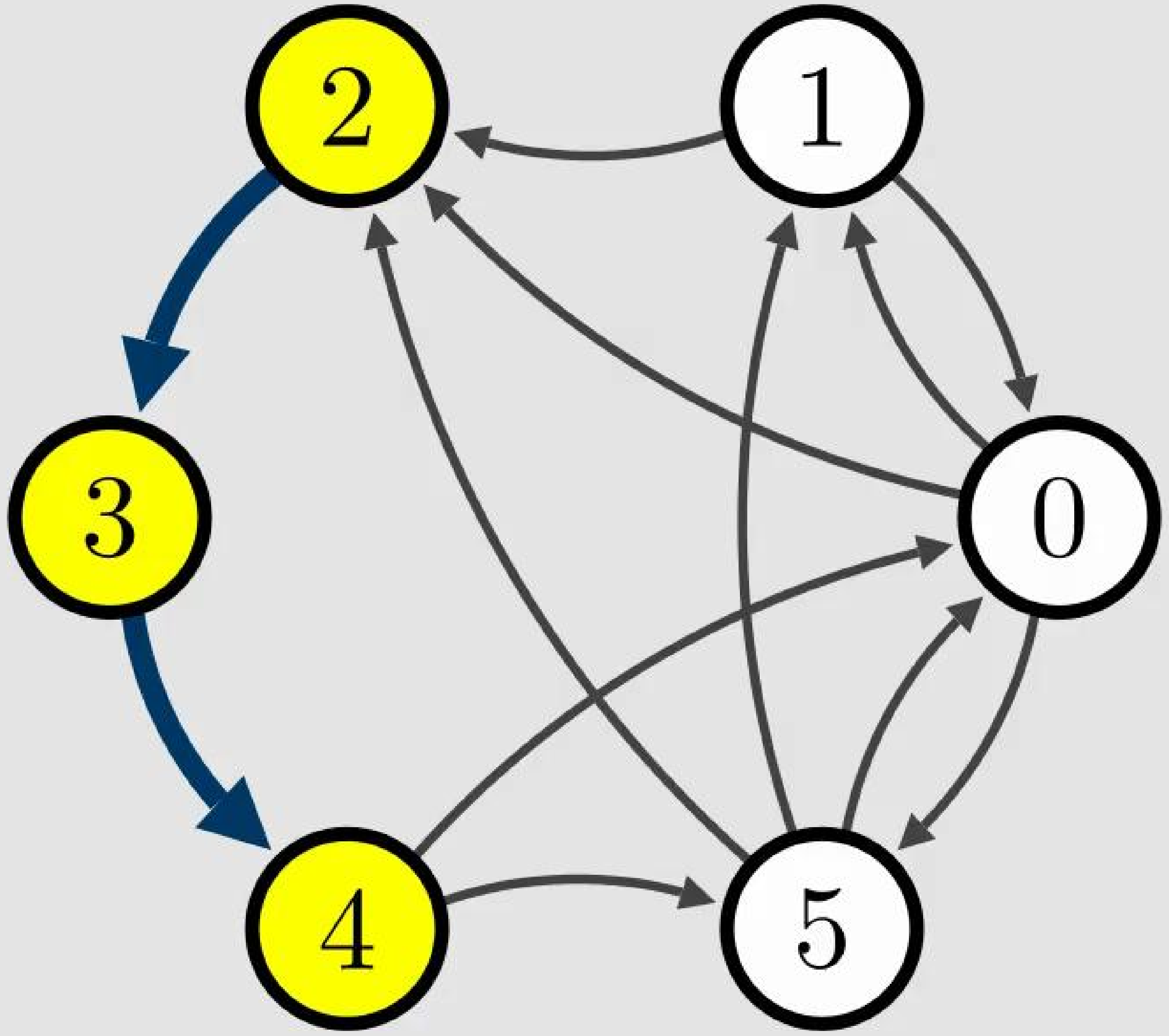
$x_{2=3} \implies bits(P_3) = bits(P_2) + 1$

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

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

Cutting planes addition:

$x_{2=3} \wedge x_{3=4} \wedge x_{4=2} \implies 0 = 3$



Circuit PB Encoding

$bits(P_i) :=$ Position of vertex i relative to 0

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

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

From encoding:

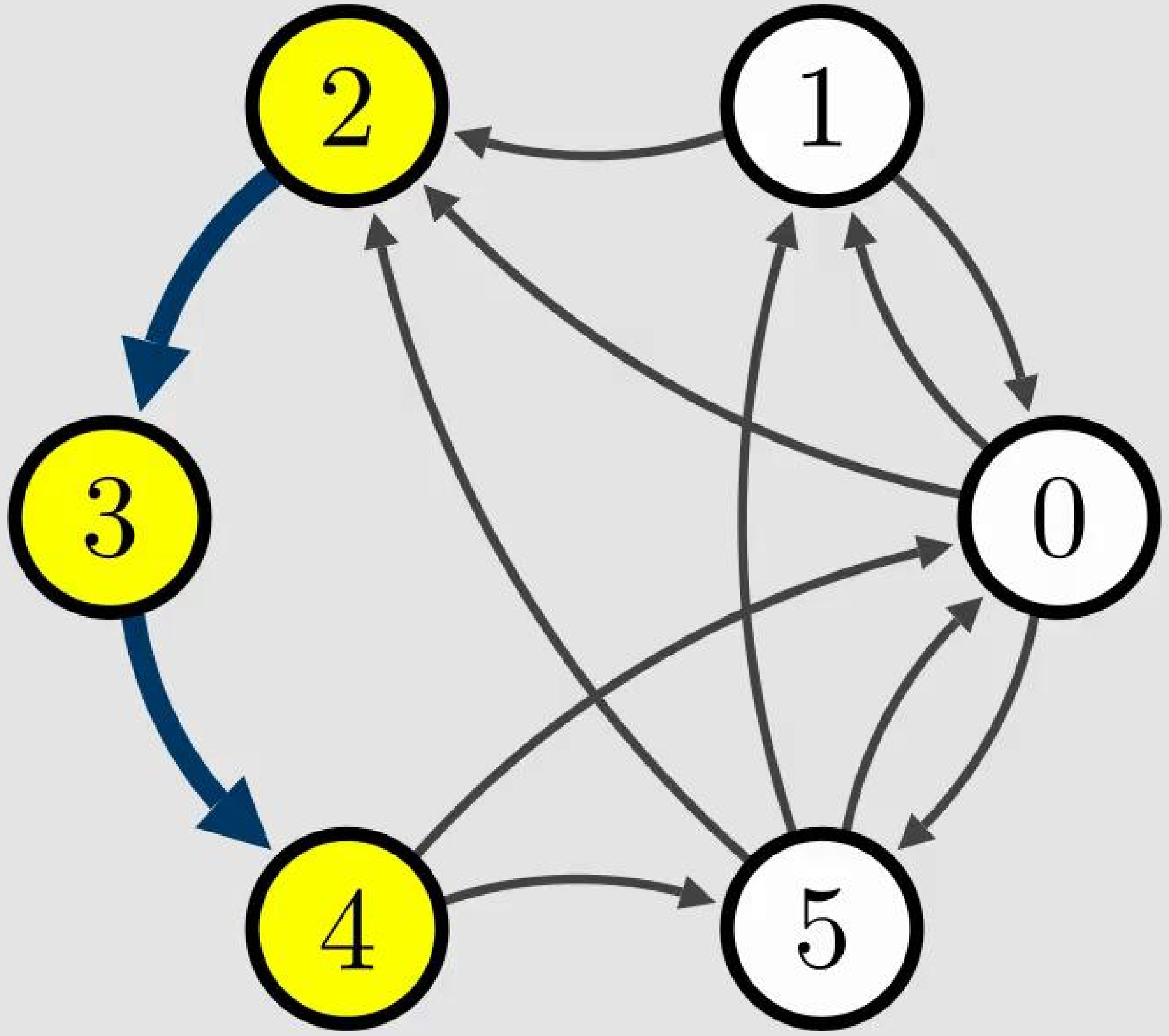
$x_{2=3} \implies bits(P_3) = bits(P_2) + 1$

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

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

Cutting planes addition:

$\overline{x_{2=3}} \vee \overline{x_{3=4}} \vee \overline{x_{4=2}}$



Circuit PB Encoding

$bits(P_i) :=$ Position of vertex i relative to 0

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

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

From encoding:

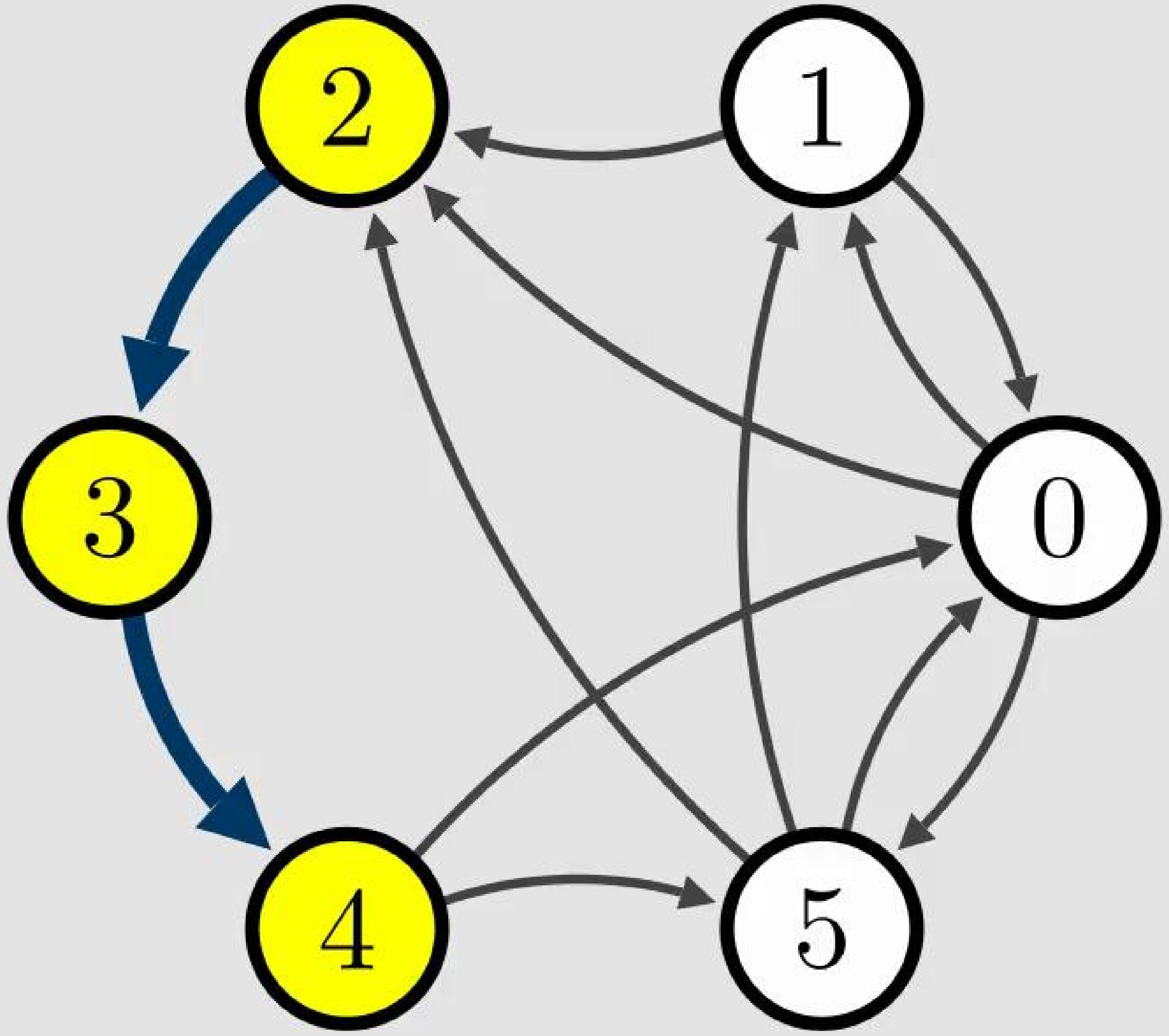
$x_{2=3} \implies bits(P_3) = bits(P_2) + 1$

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

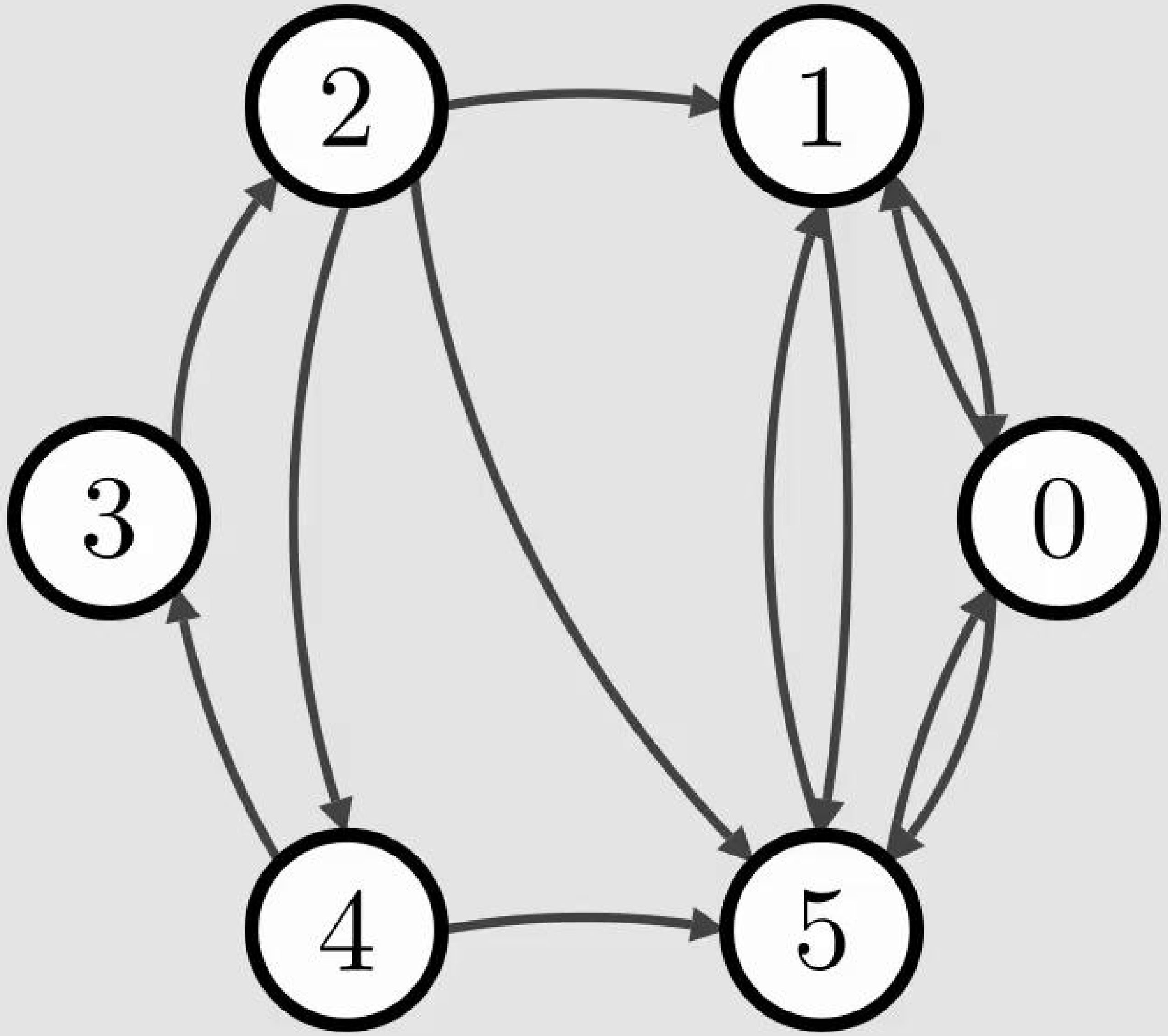
$x_{4=2} \implies bits(P_2) = bits(P_4) + 1$

Cutting planes addition:

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

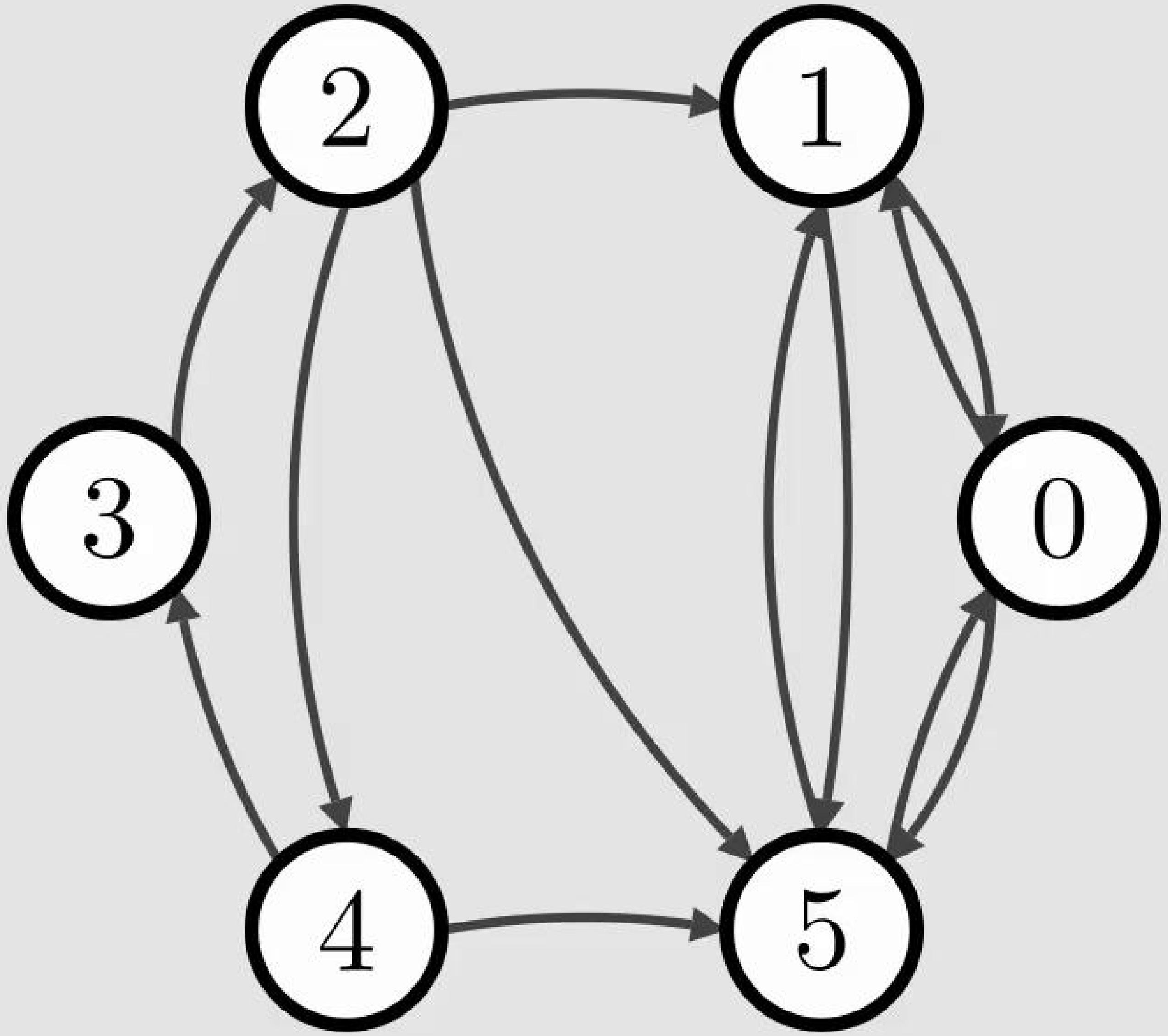


SCC Propagation



SCC Propagation

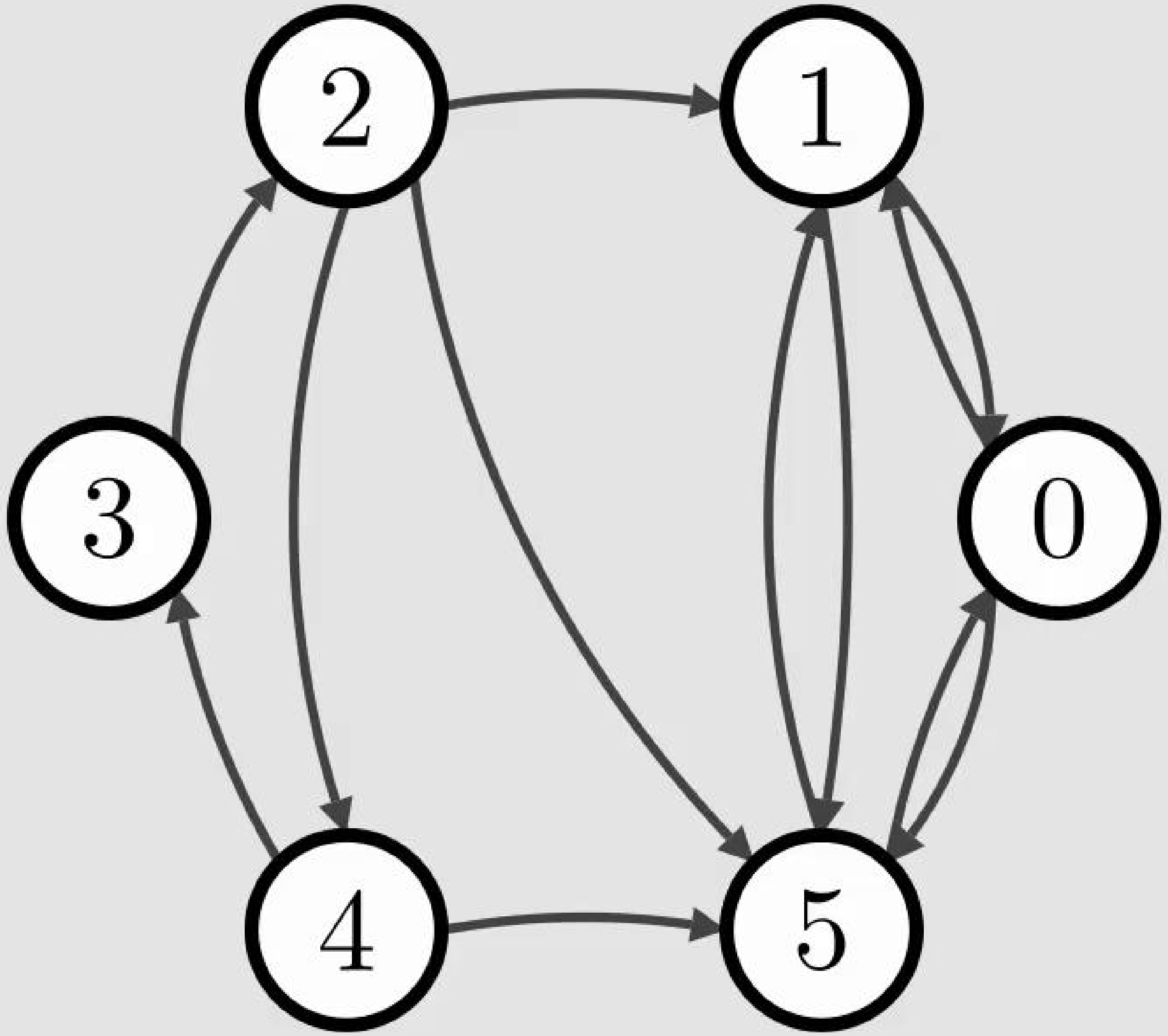
If AllDiff is enforced:



SCC Propagation

If AllDiff is enforced:

No subcycles

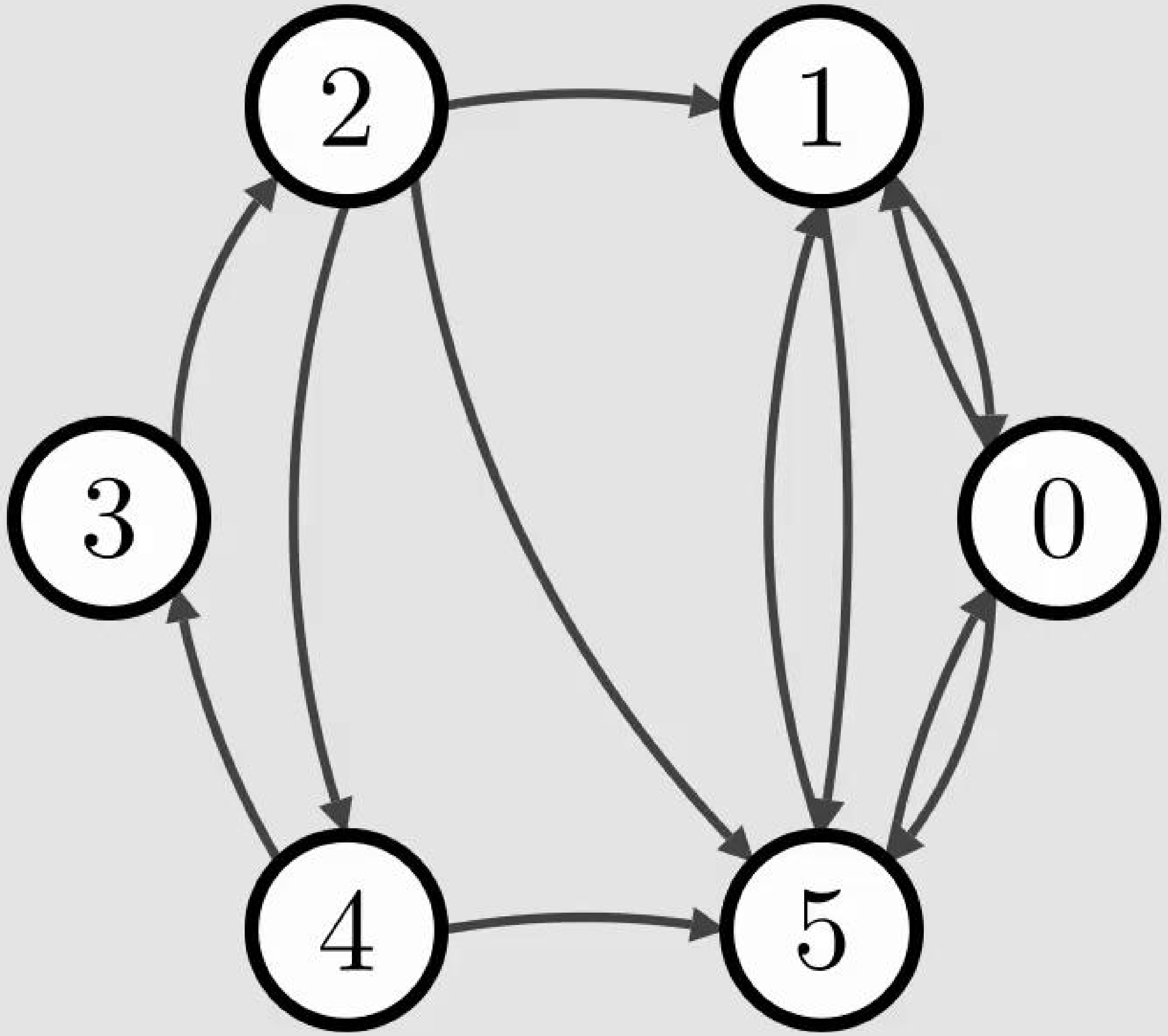


SCC Propagation

If AllDiff is enforced:

No subcycles

$\Longleftrightarrow$



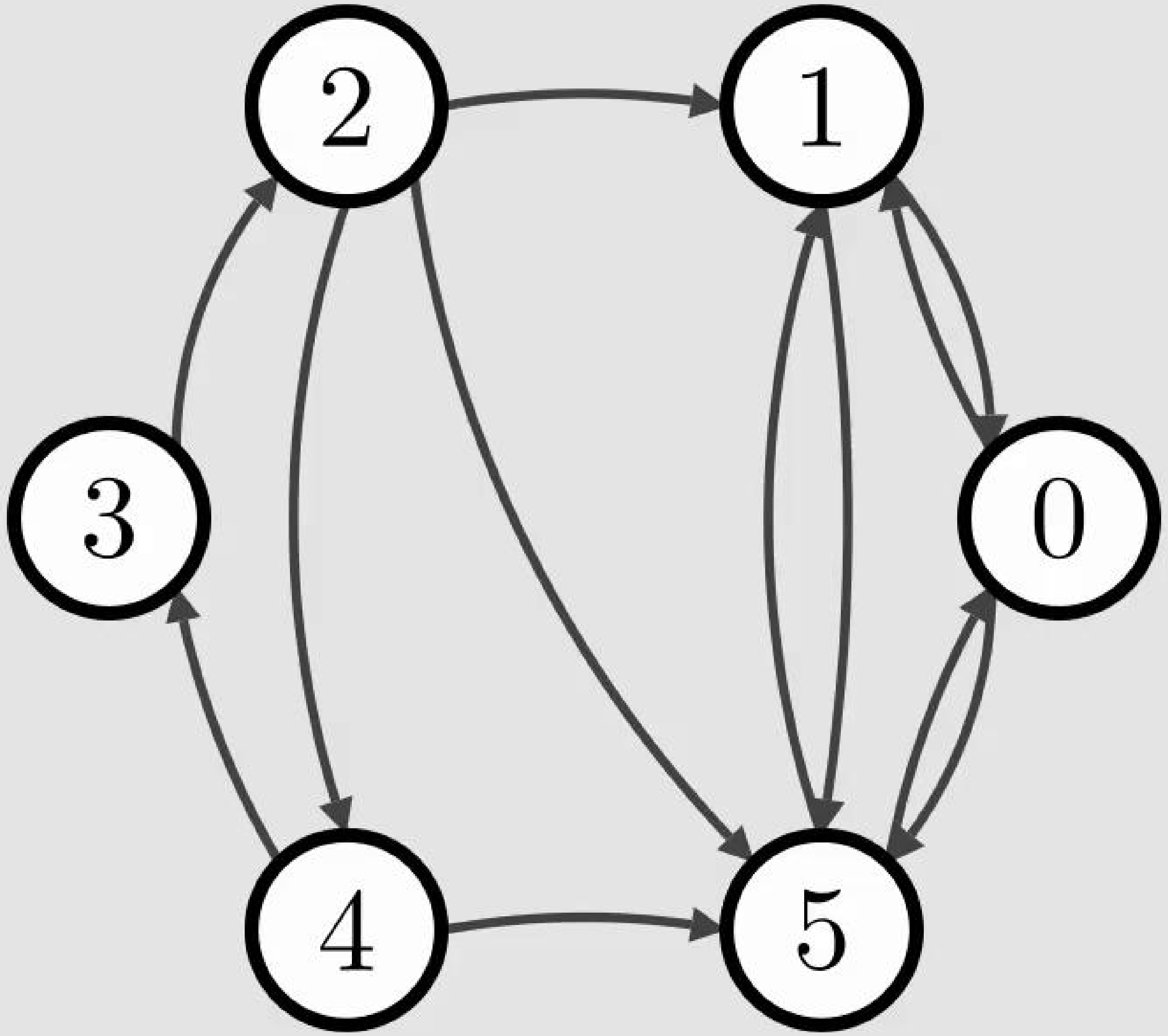
SCC Propagation

If AllDiff is enforced:

No subcycles

$\iff$

All vertices part of one cycle



SCC Propagation

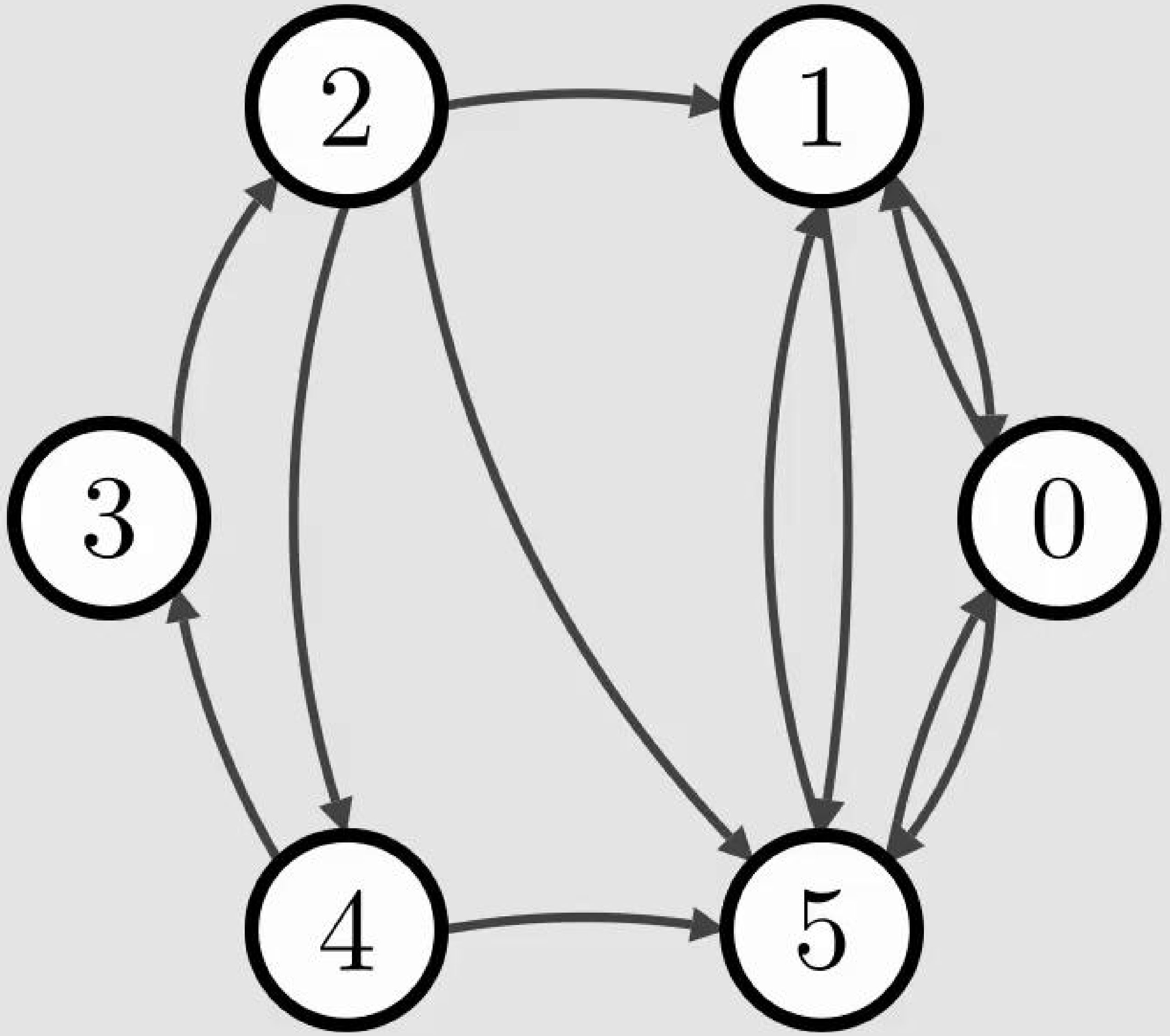
If AllDiff is enforced:

No subcycles

$\iff$

All vertices part of one cycle

$\iff$



SCC Propagation

If AllDiff is enforced:

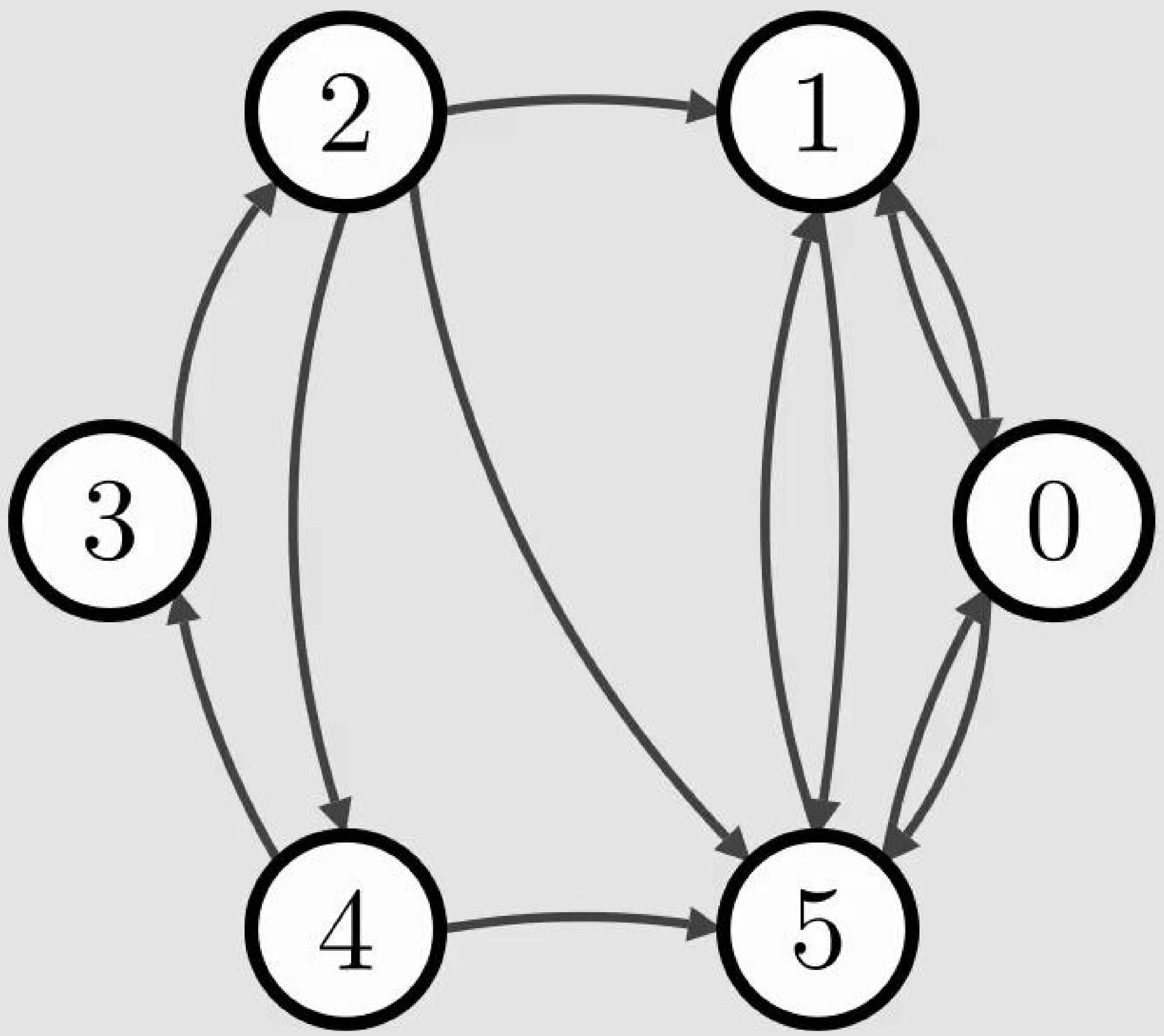
No subcycles

$\iff$

All vertices part of one cycle

$\iff$

Every vertex reachable from every vertex



SCC Propagation

If AllDiff is enforced:

No subcycles

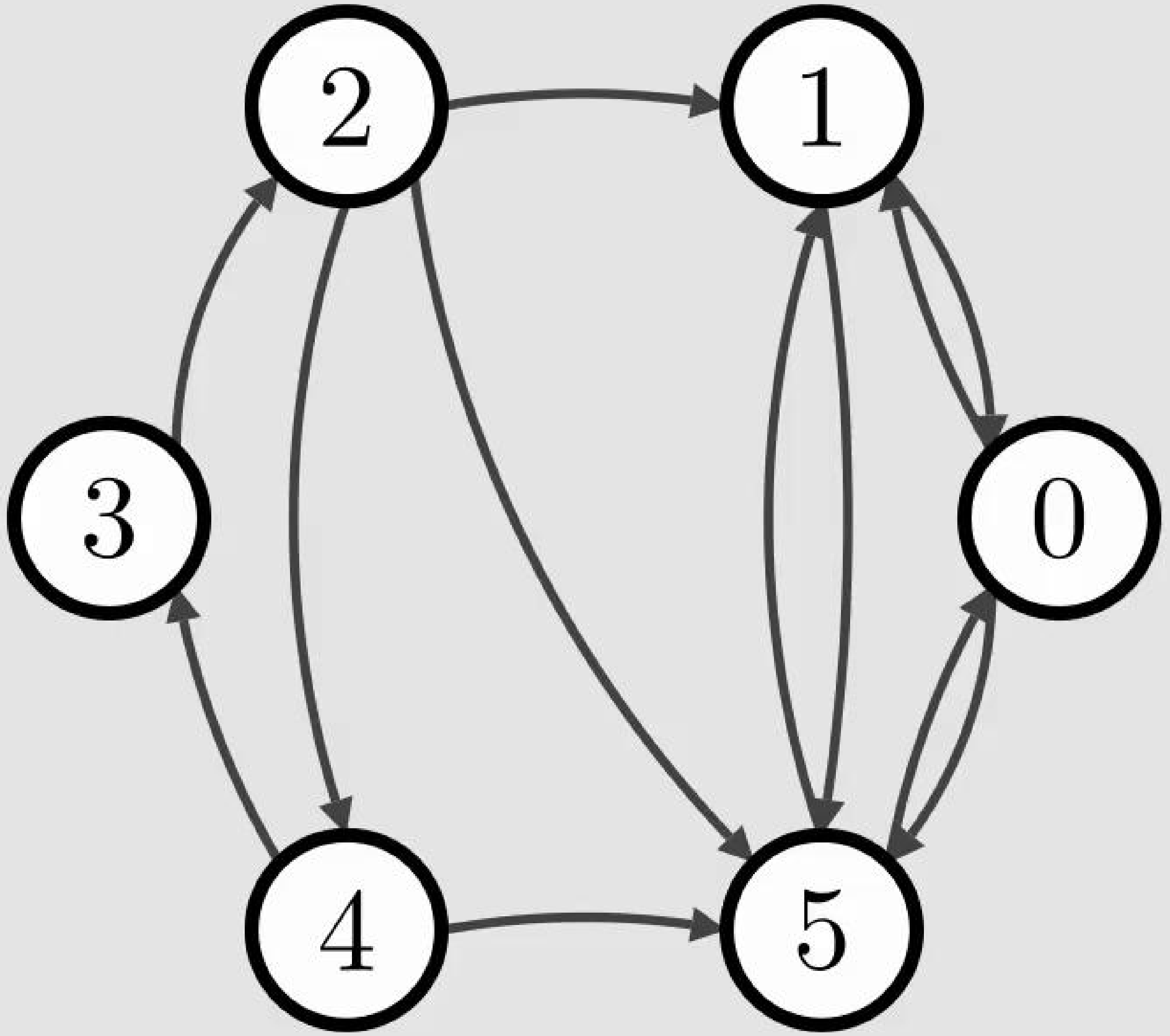
$\iff$

All vertices part of one cycle

$\iff$

Every vertex reachable from every vertex

$\iff$



SCC Propagation

If AllDiff is enforced:

No subcycles

$\iff$

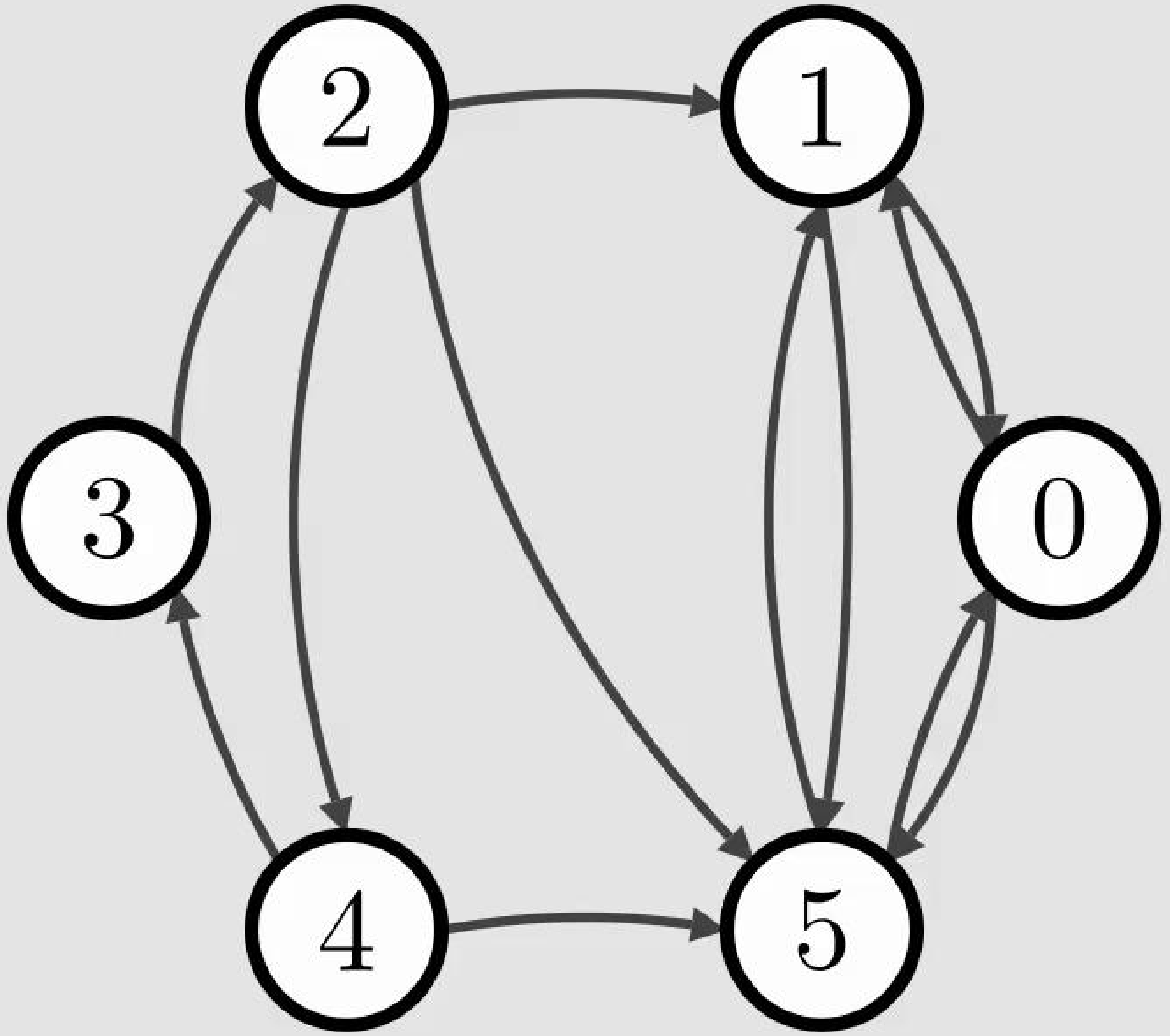
All vertices part of one cycle

$\iff$

Every vertex reachable from every vertex

$\iff$

One one strongly connected component



SCC Propagation

If AllDiff is enforced:

No subcycles

$\iff$

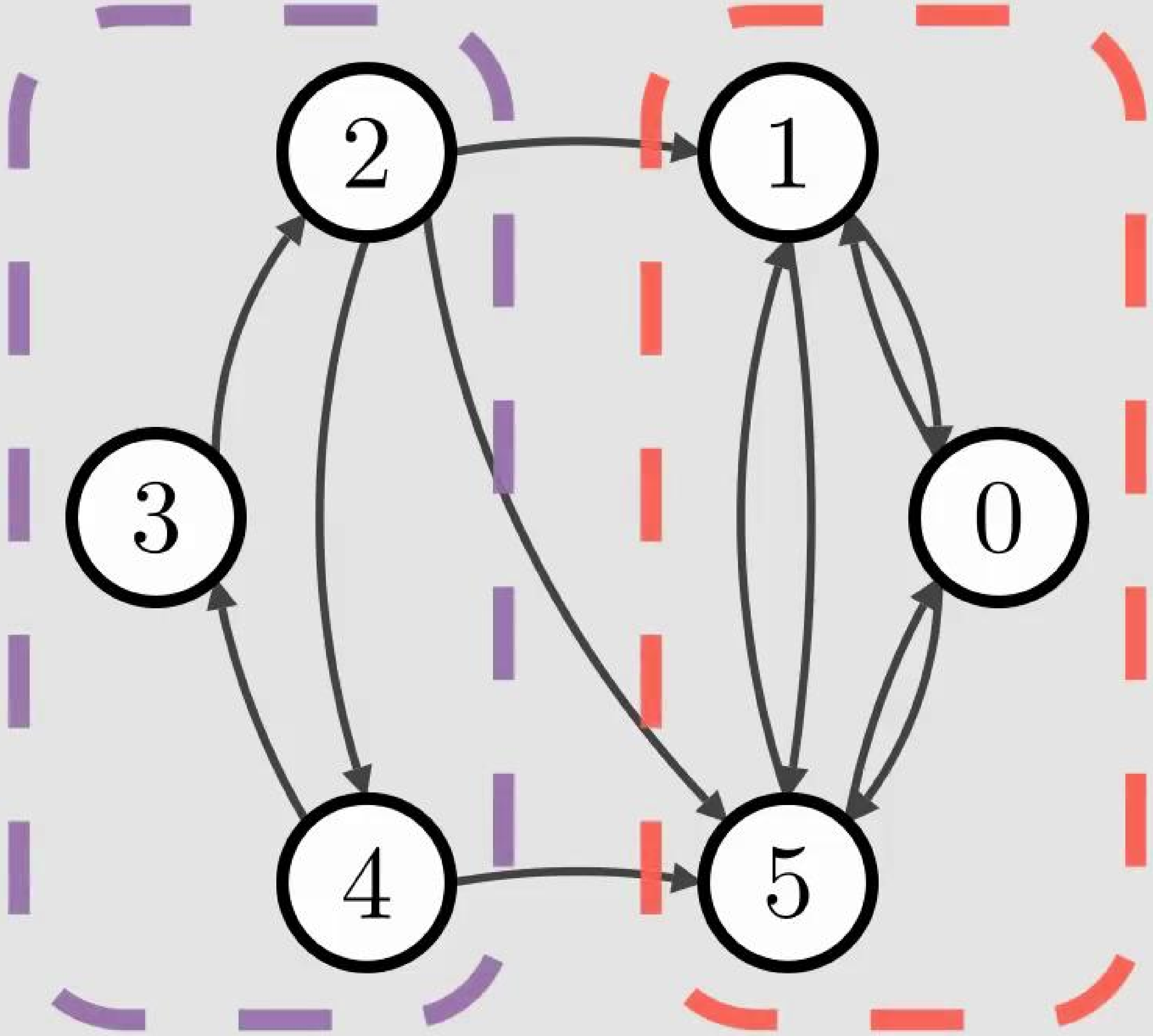
All vertices part of one cycle

$\iff$

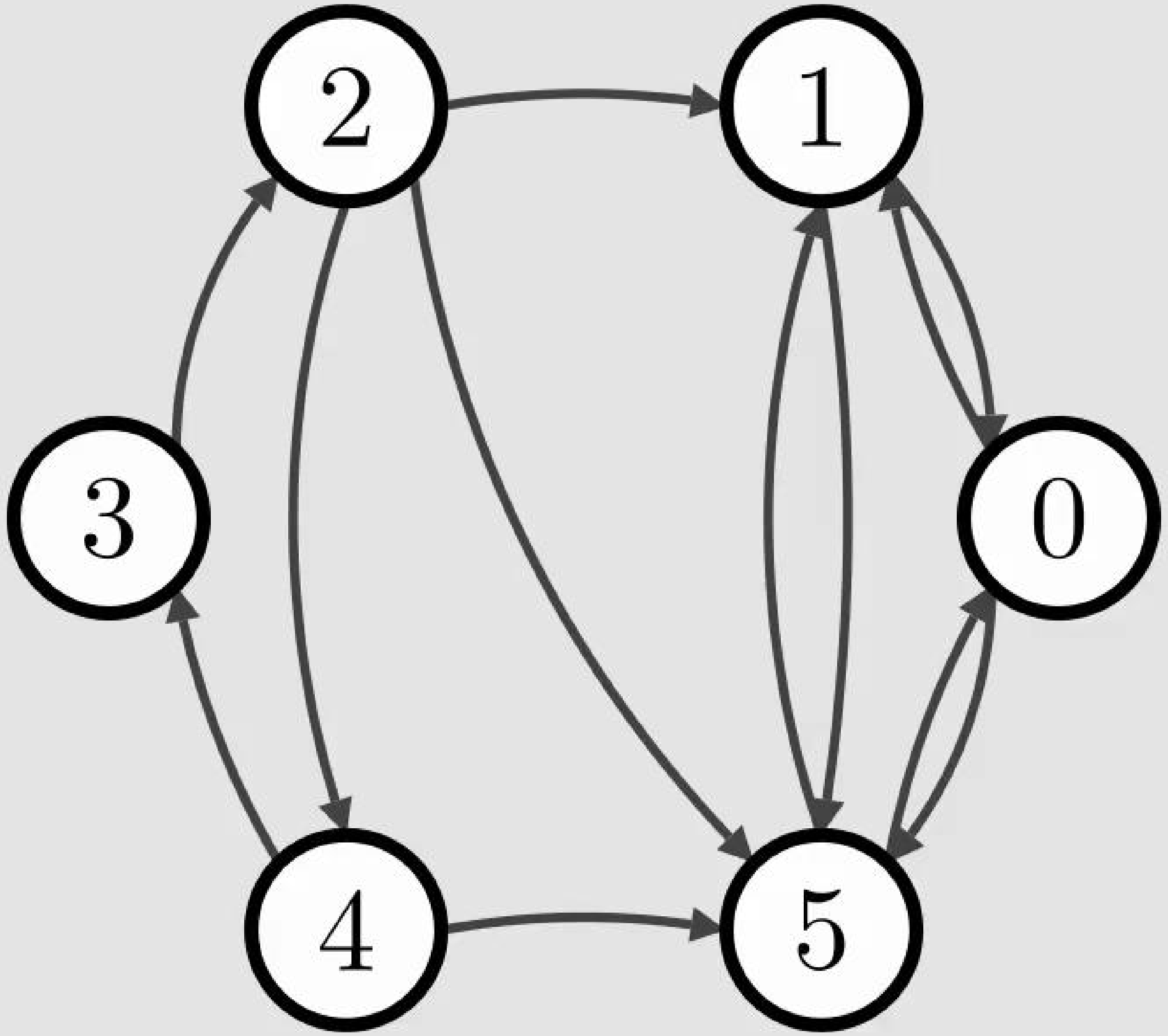
Every vertex reachable from every vertex

$\iff$

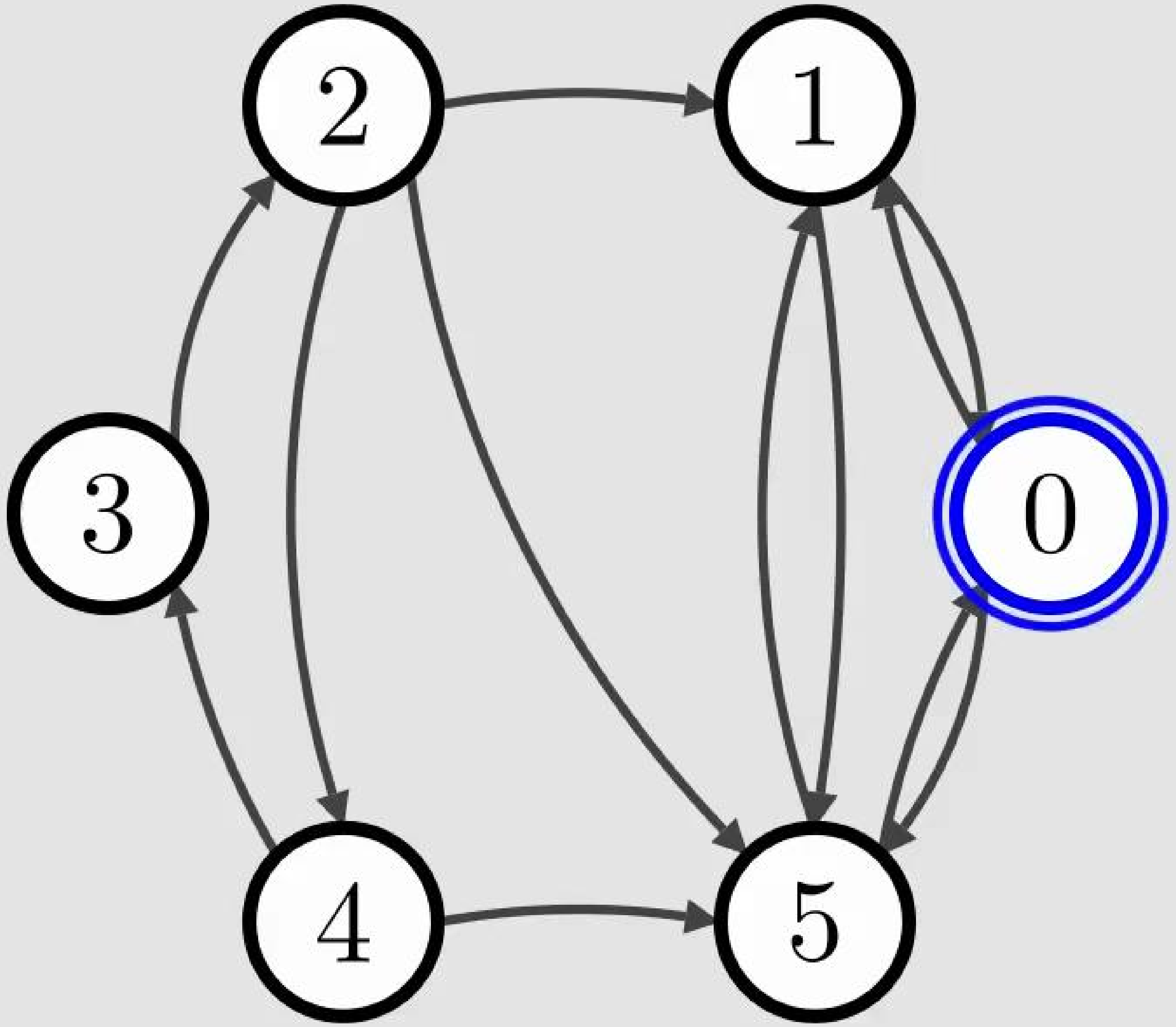
One one strongly connected component



SCC Propagation

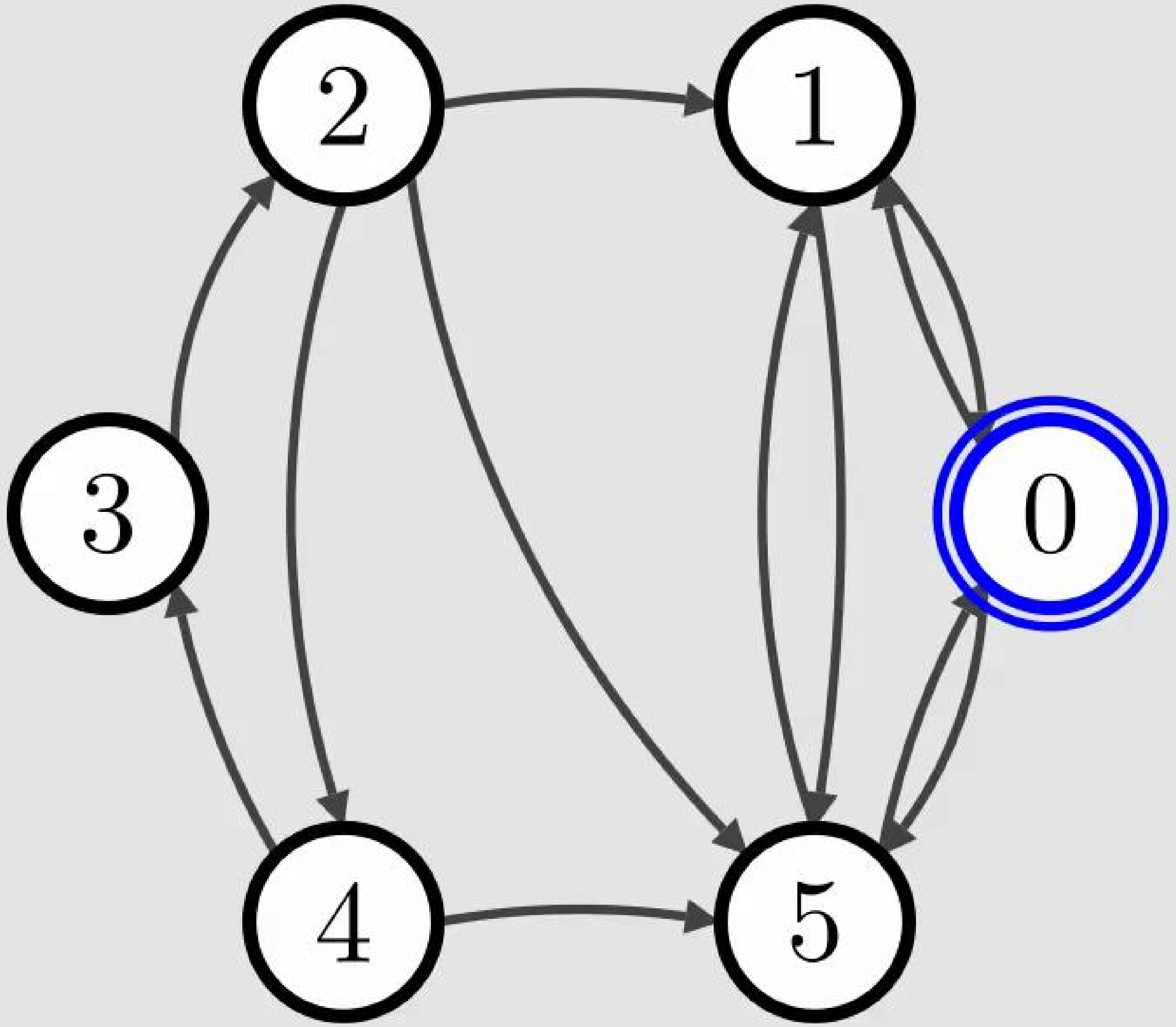


SCC Propagation



SCC Propagation

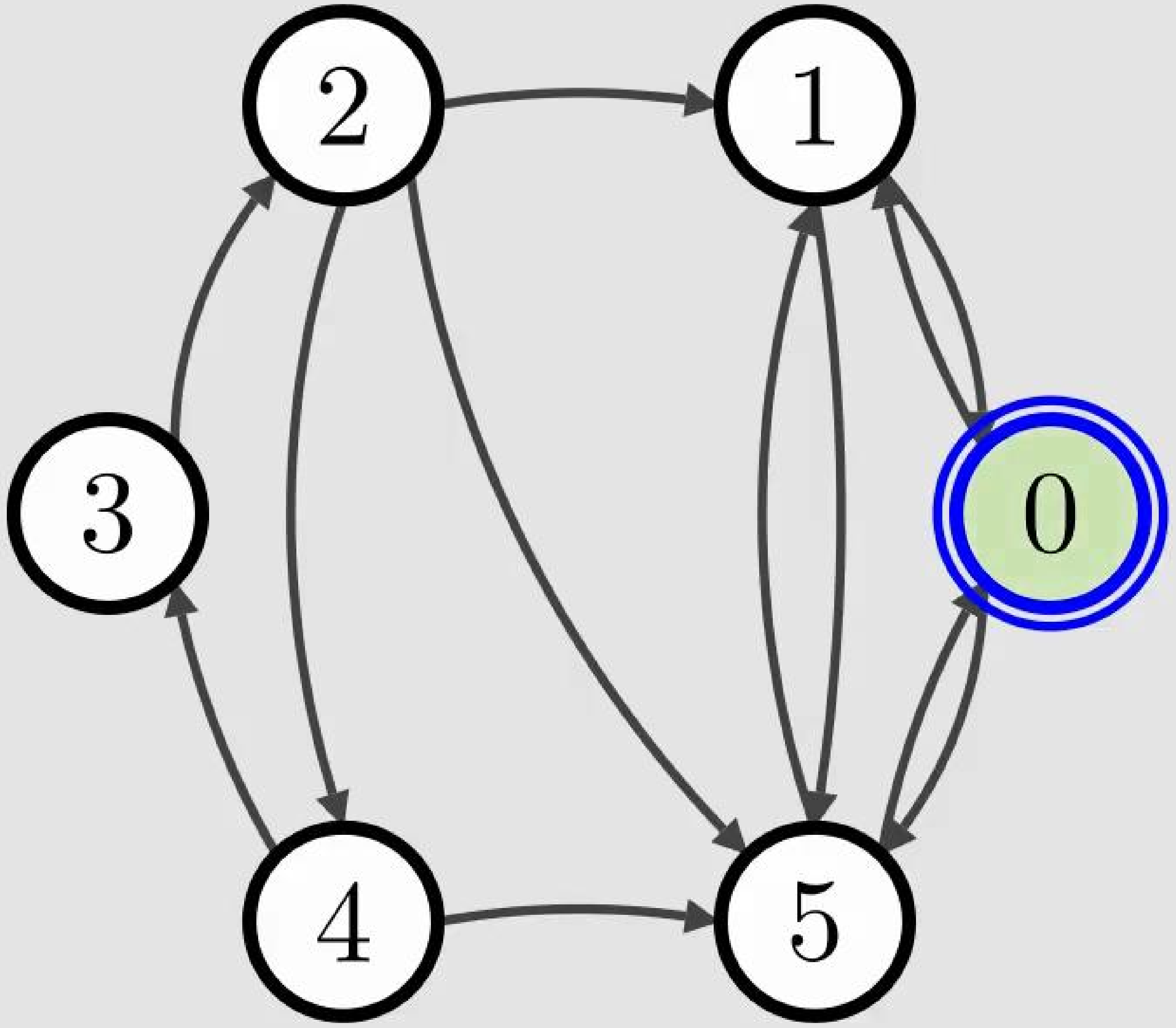
```
ReachTooSmall( 0 )
```



SCC Propagation

ReachTooSmall(0)

$$\{P_0\} = 0$$

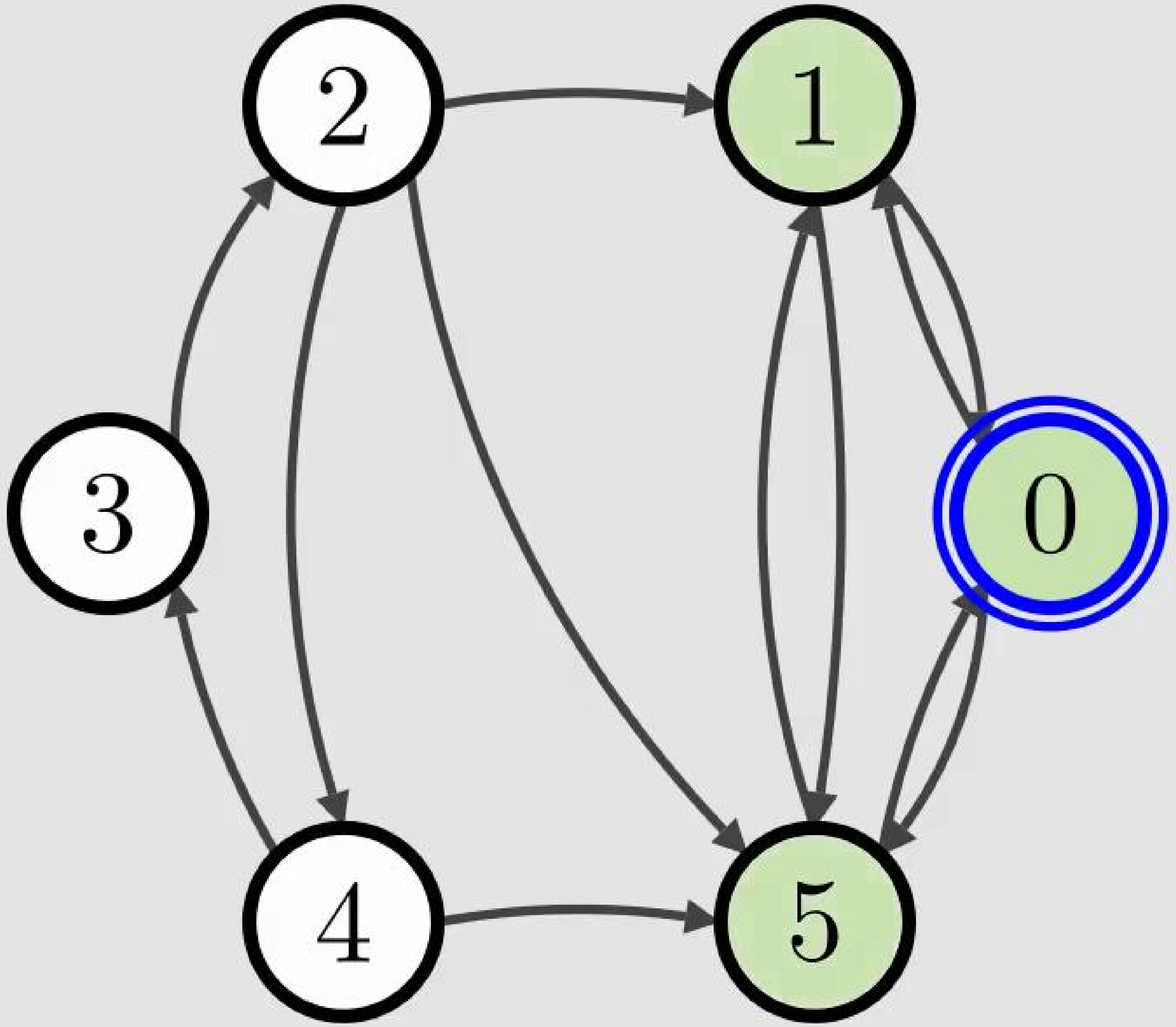


SCC Propagation

```
ReachTooSmall( 0 )
```

$$\{P_0\} = 0$$

$$\{P_1, P_5\} = 1$$



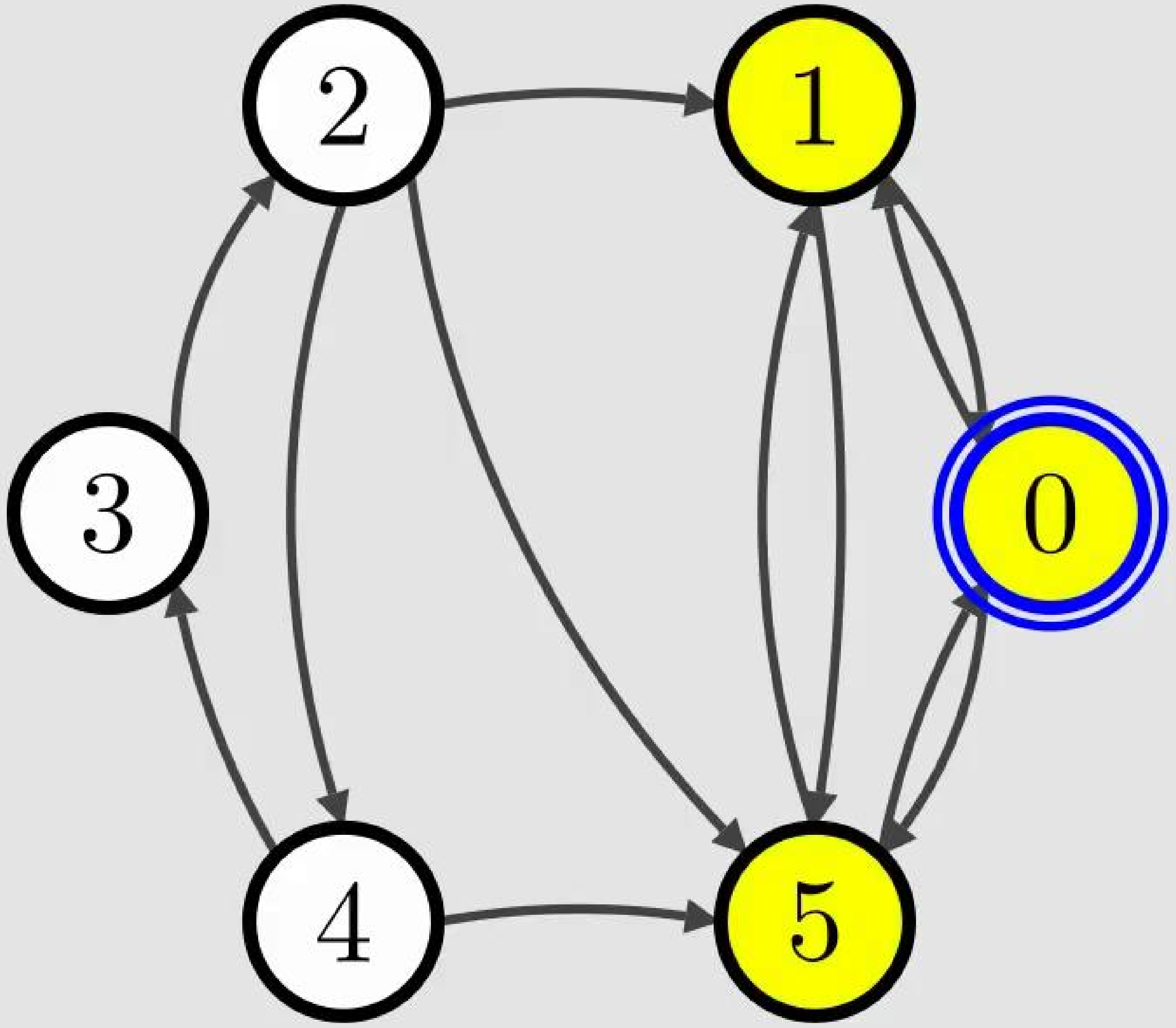
SCC Propagation

ReachTooSmall(0)

$$\{P_0\} = 0$$

$$\{P_1, P_5\} = 1$$

$$\{P_0, P_1, P_5\} = 2$$



SCC Propagation

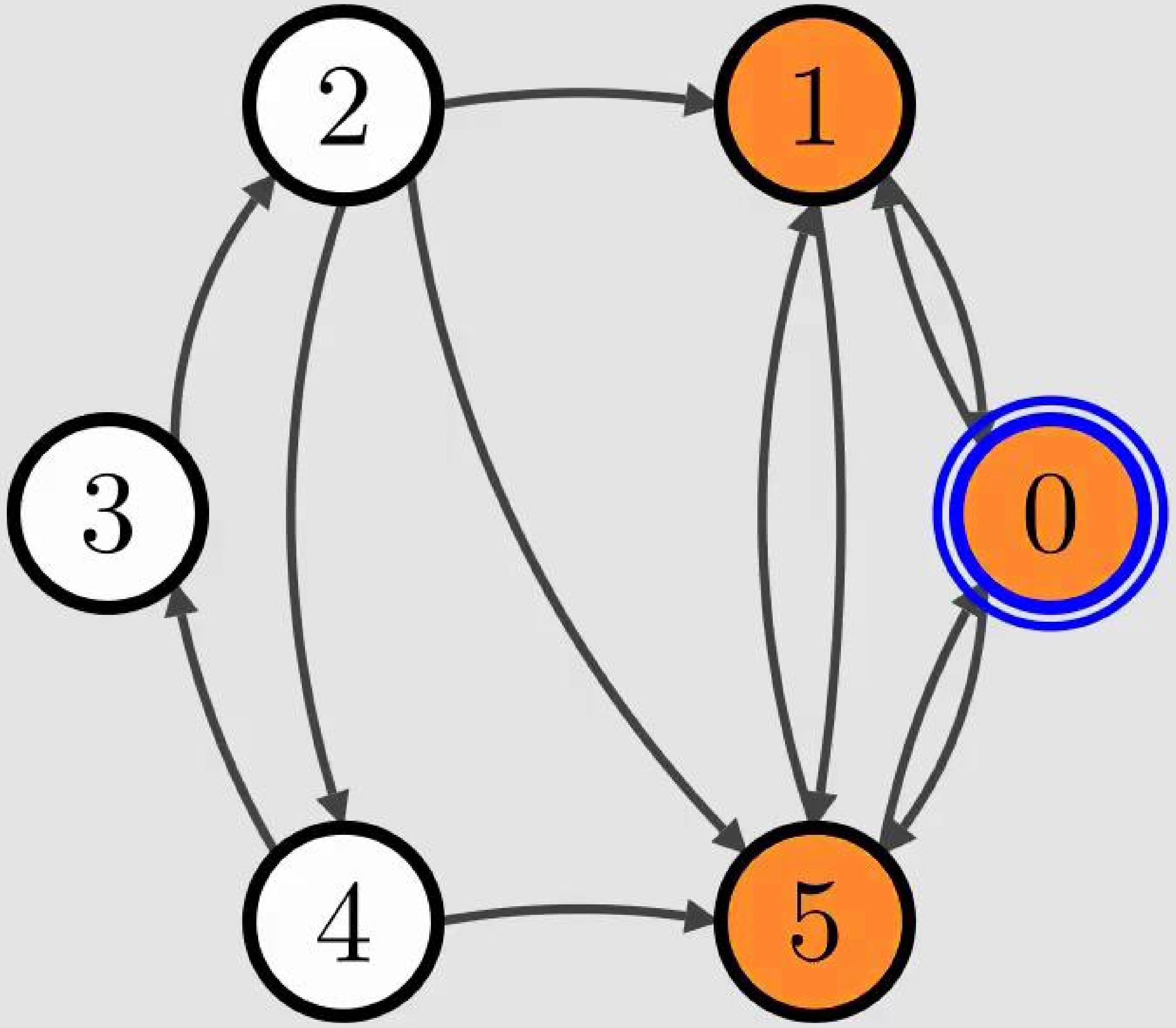
```
ReachTooSmall( 0 )
```

$$\{P_0\} = 0$$

$$\{P_1, P_5\} = 1$$

$$\{P_0, P_1, P_5\} = 2$$

$$\{P_0, P_1, P_5\} = 3$$



SCC Propagation

ReachTooSmall(0)

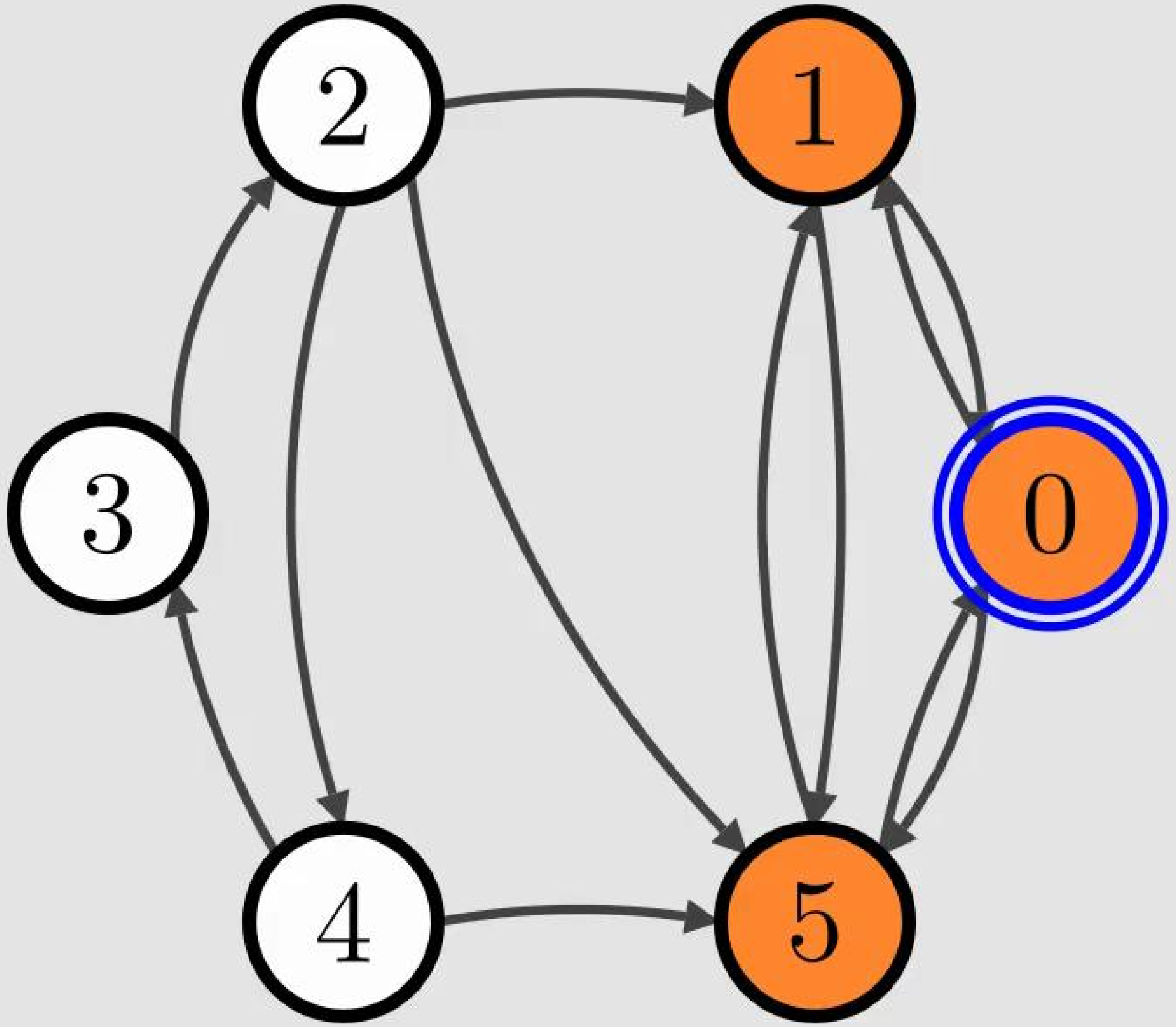
$$\{P_0\} = 0$$

$$\{P_1, P_5\} = 1$$

$$\{P_0, P_1, P_5\} = 2$$

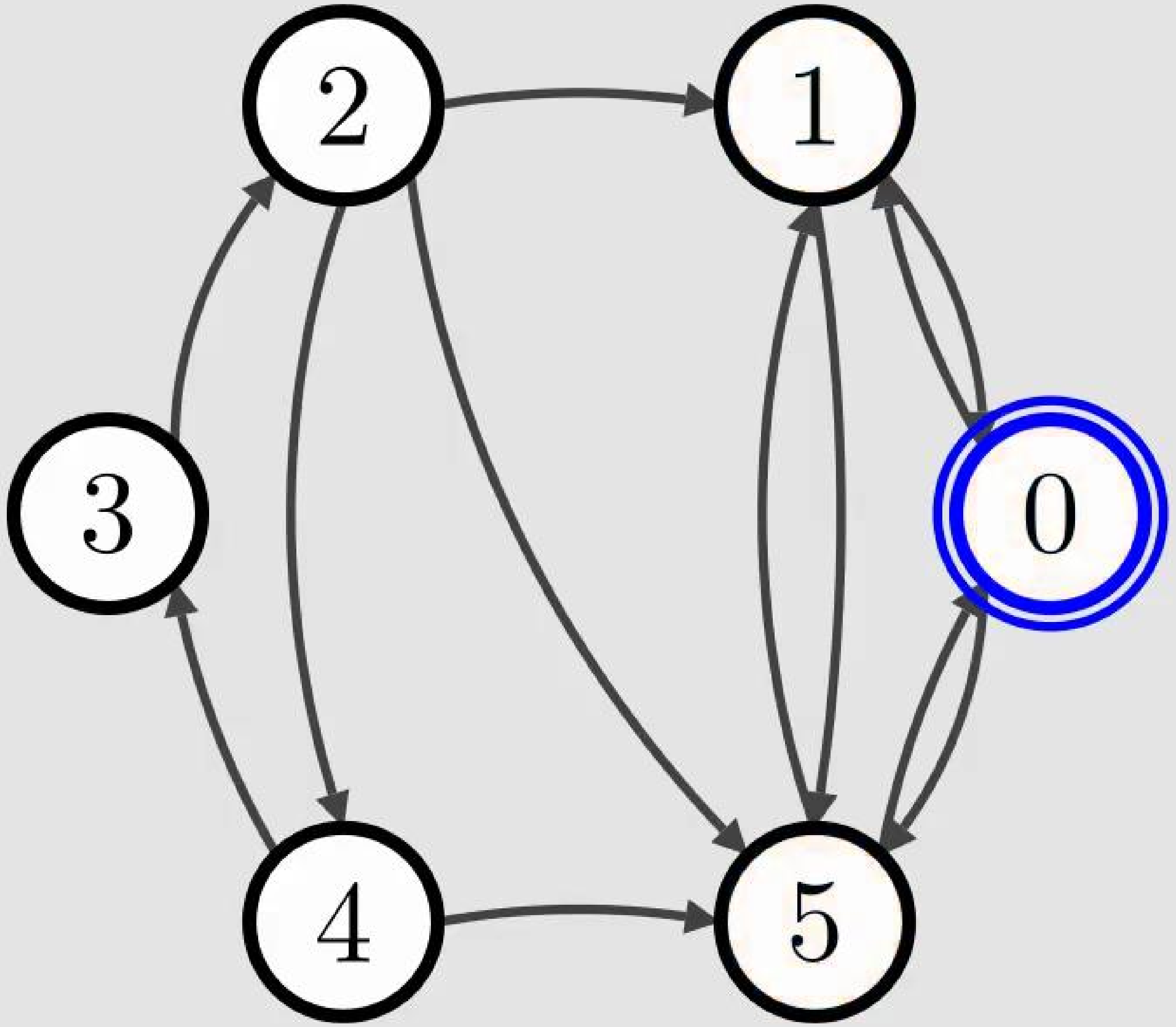
$$\{P_0, P_1, P_5\} = 3$$

$$\mathcal{G} \implies 0 \geq 1$$



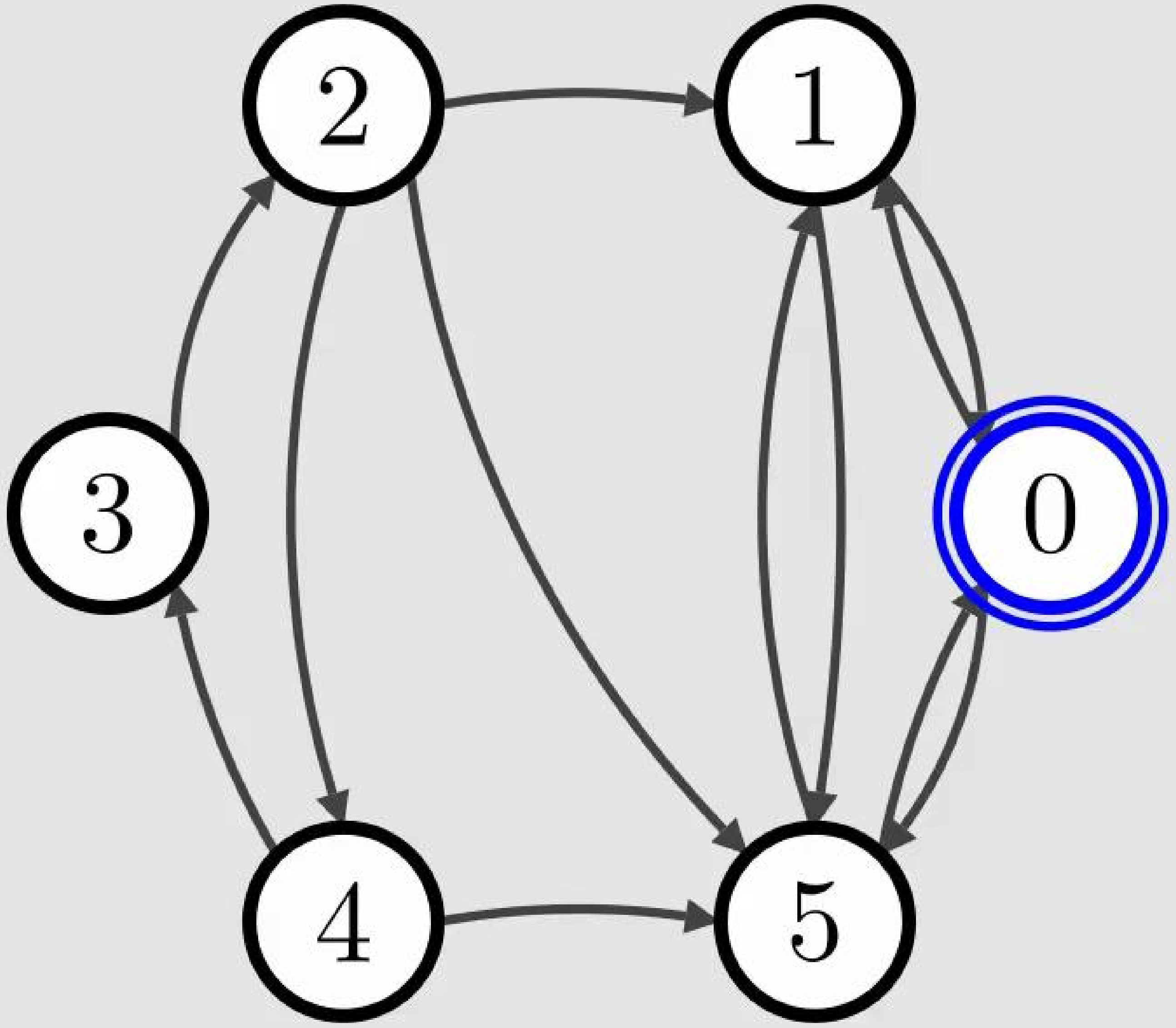
SCC Propagation

`ReachTooSmall( 0 )`



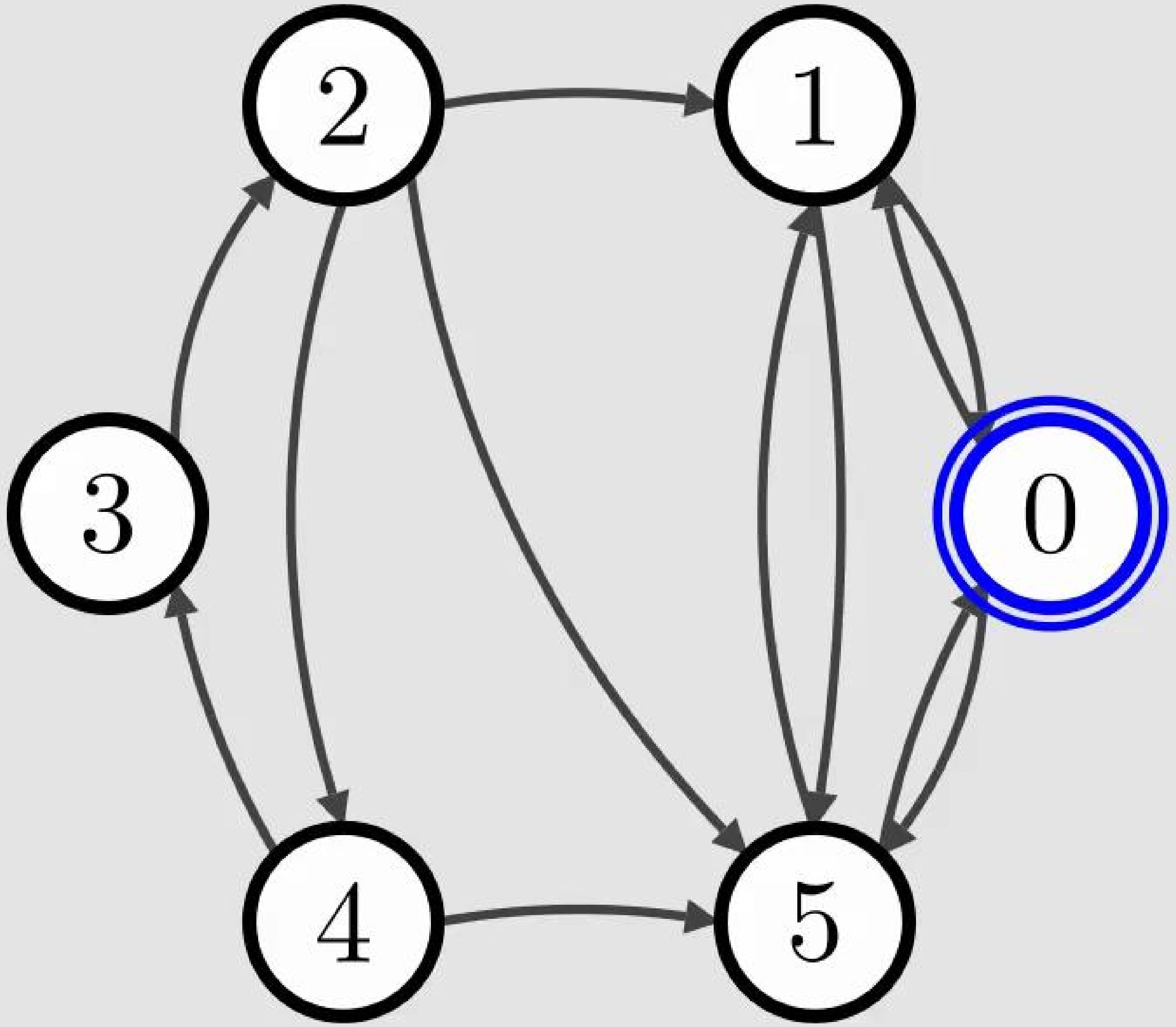
SCC Propagation

`ReachTooSmall( v )`



SCC Propagation

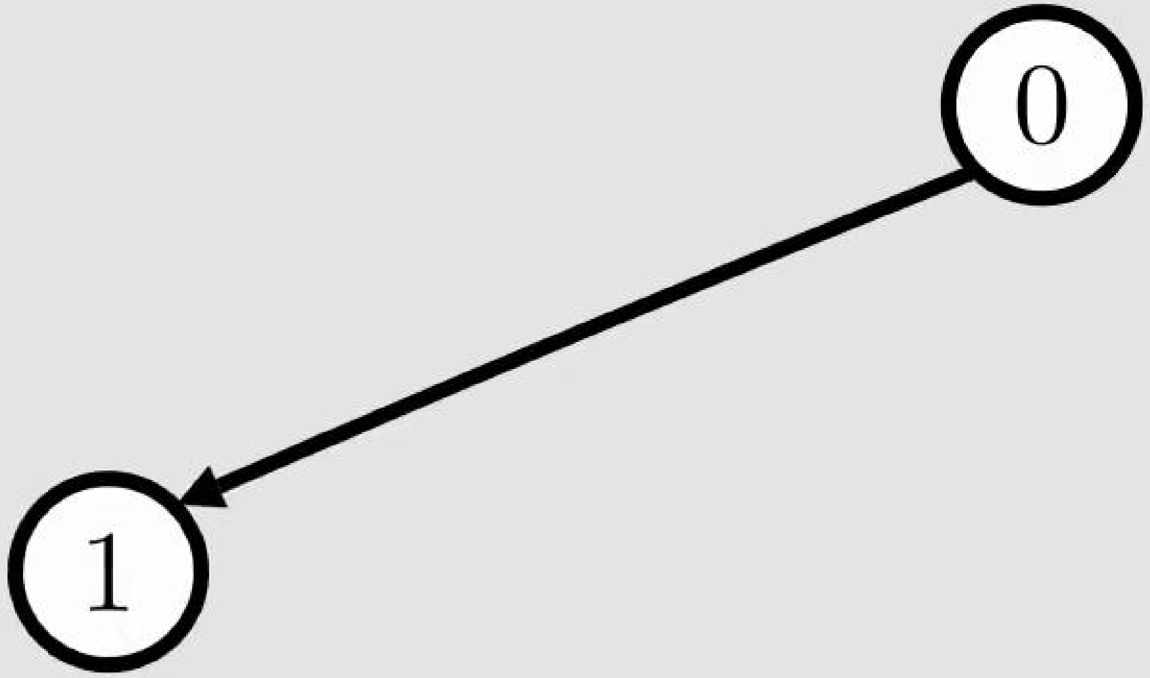
$$c_1 \implies \texttt{ReachTooSmall(v)}$$



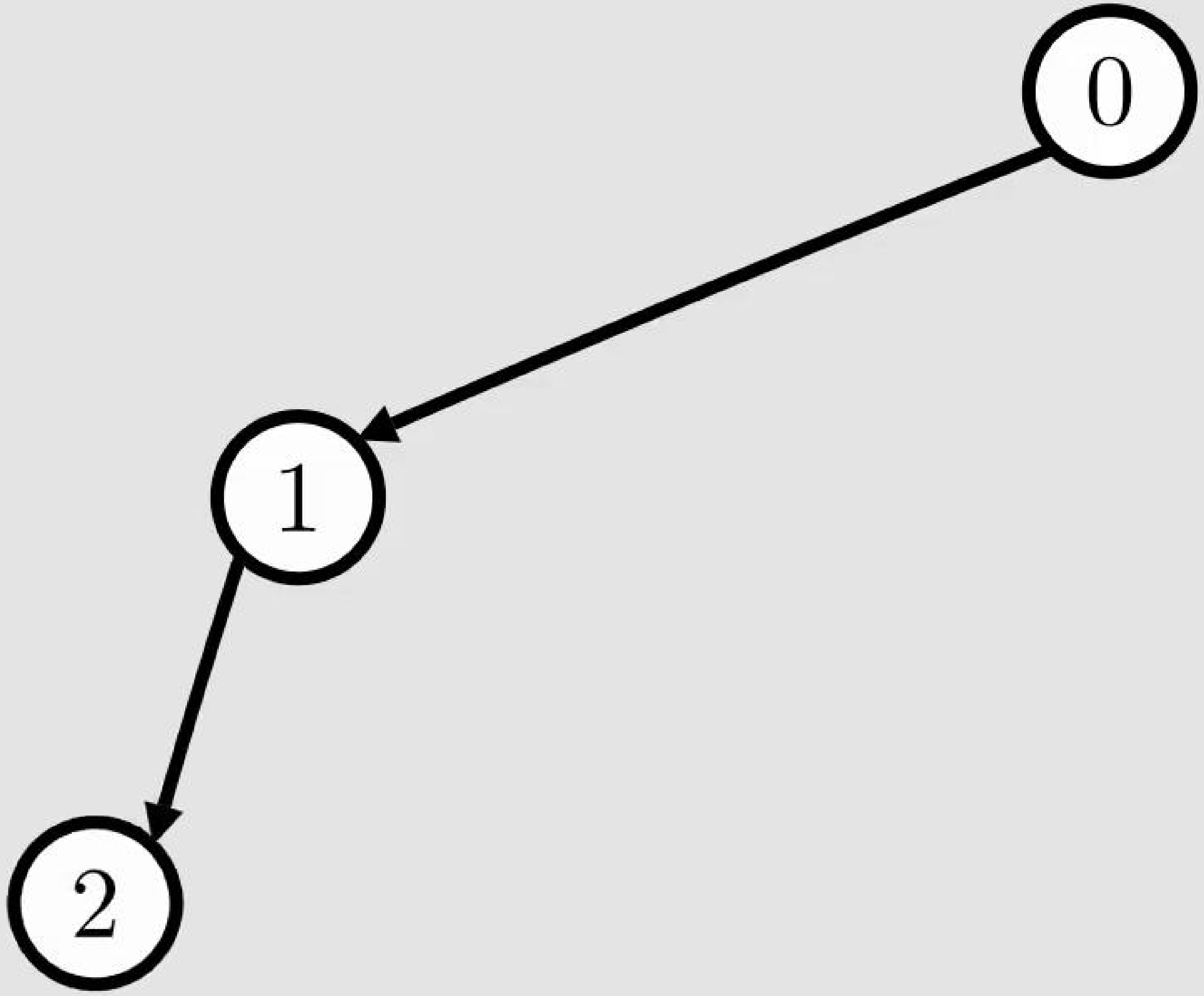
Further Propagation Rules

Further Propagation Rules

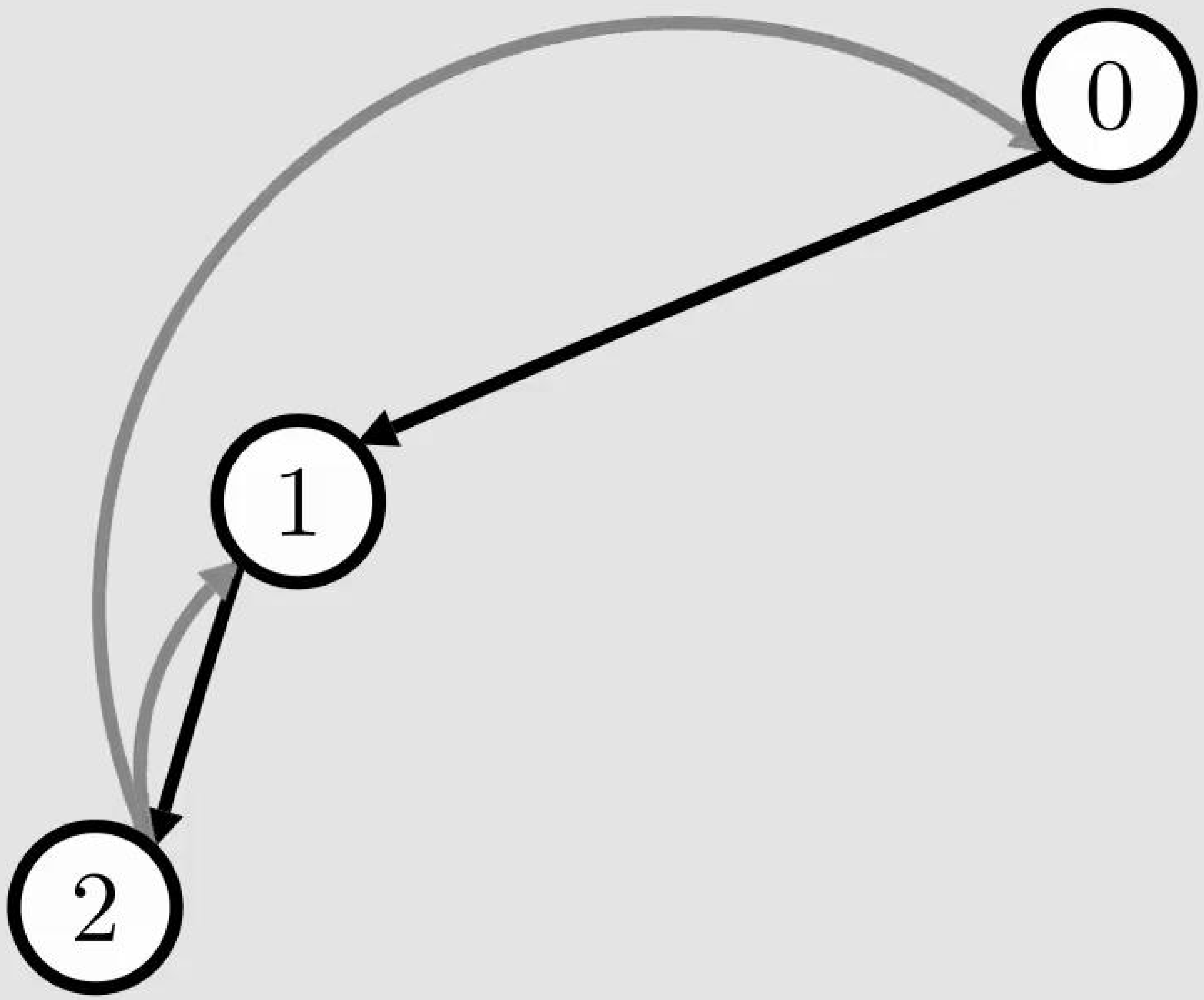
Further Propagation Rules



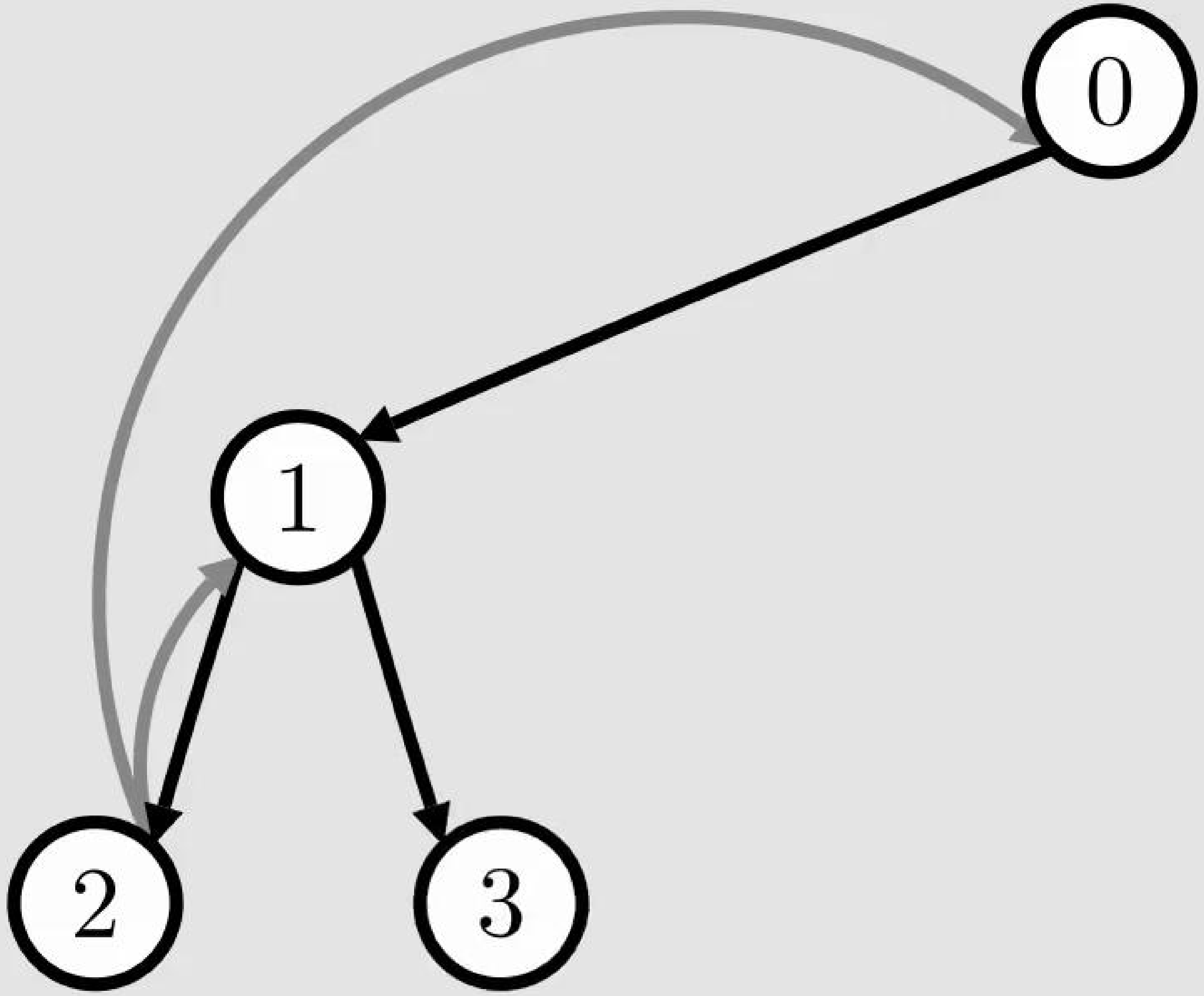
Further Propagation Rules



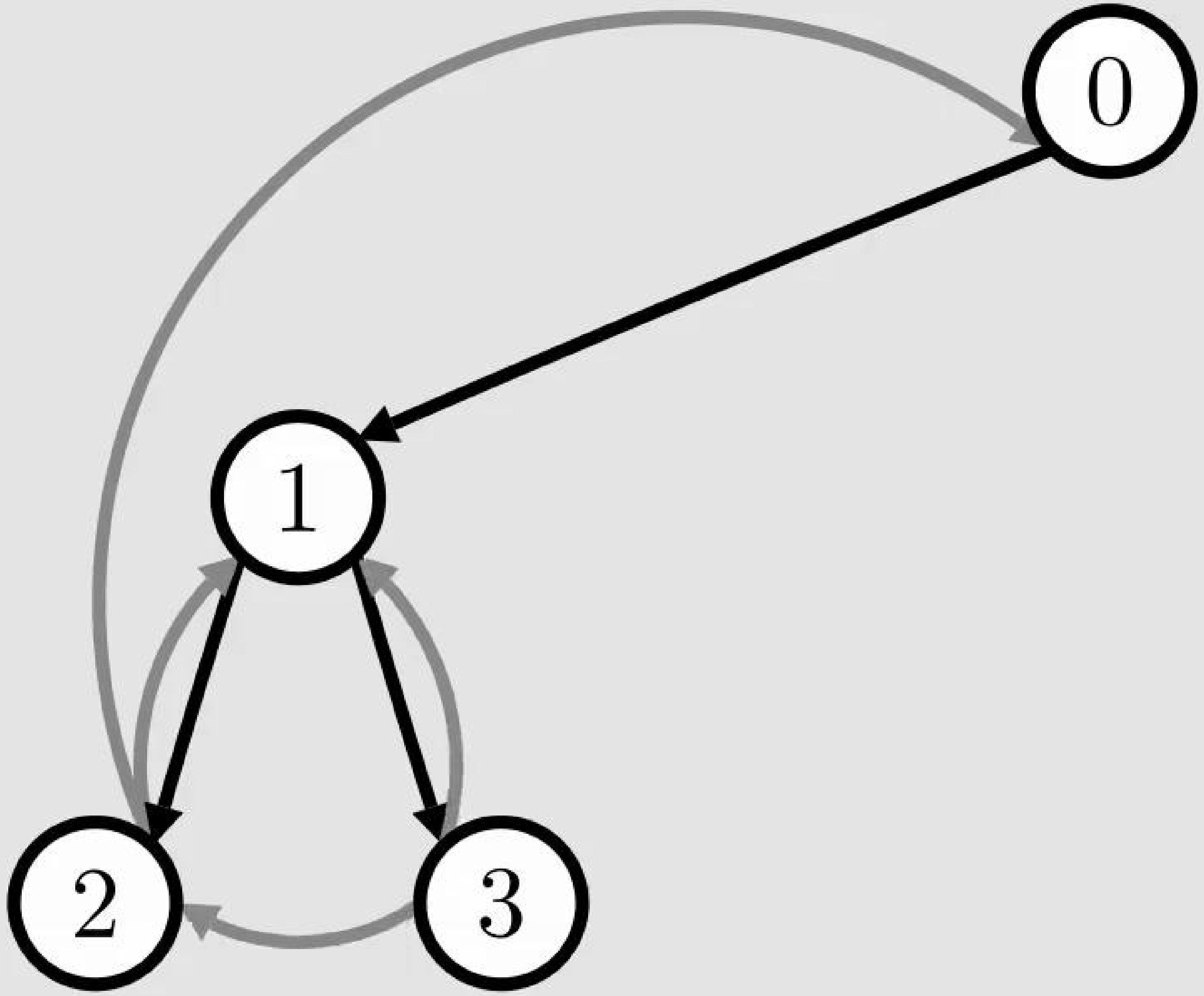
Further Propagation Rules



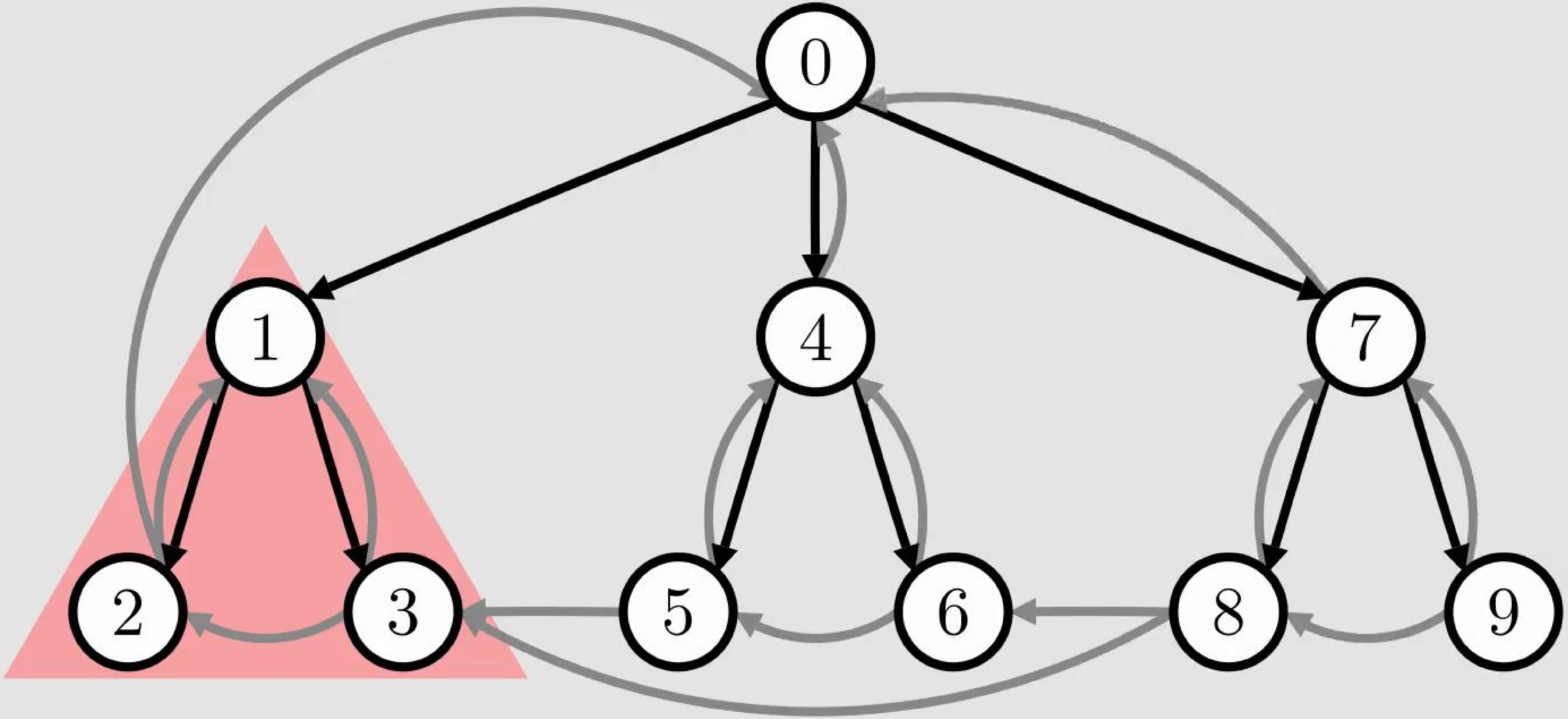
Further Propagation Rules



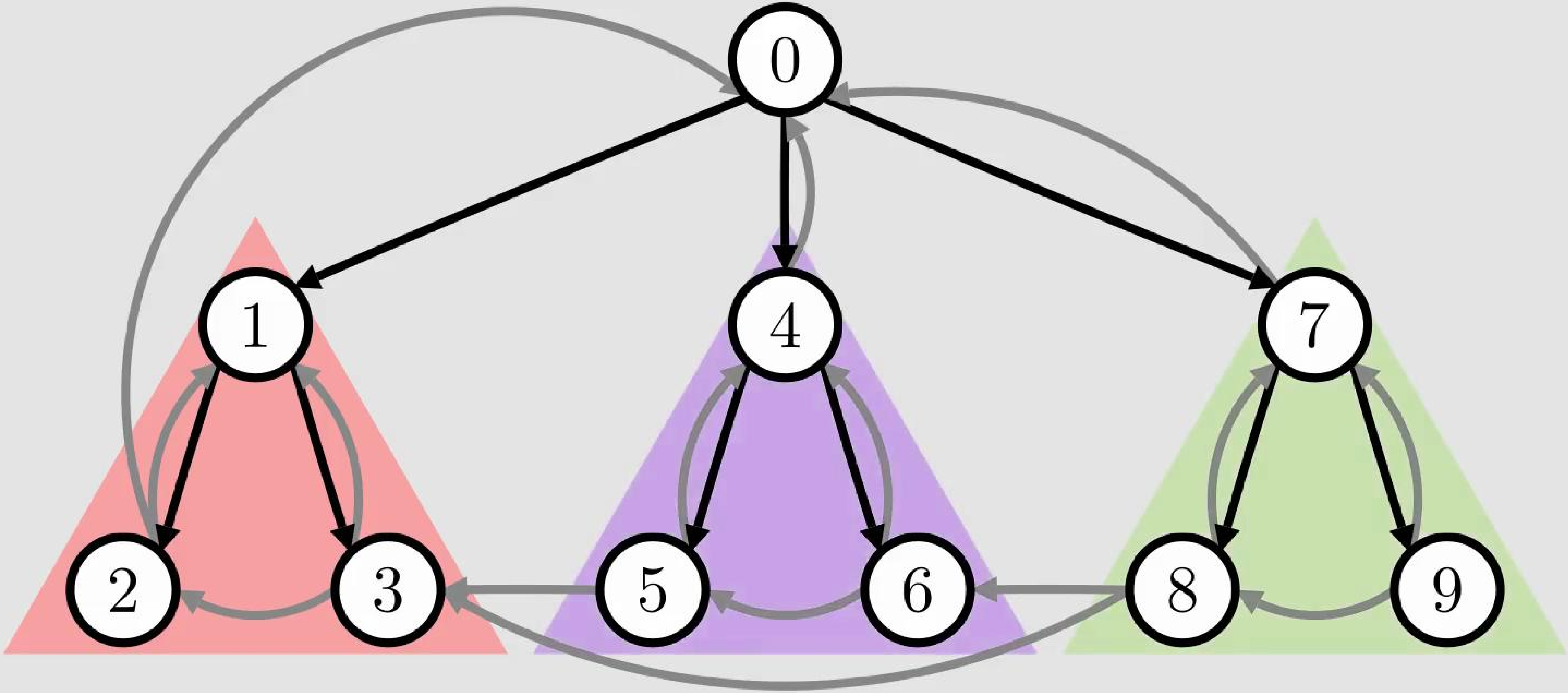
Further Propagation Rules



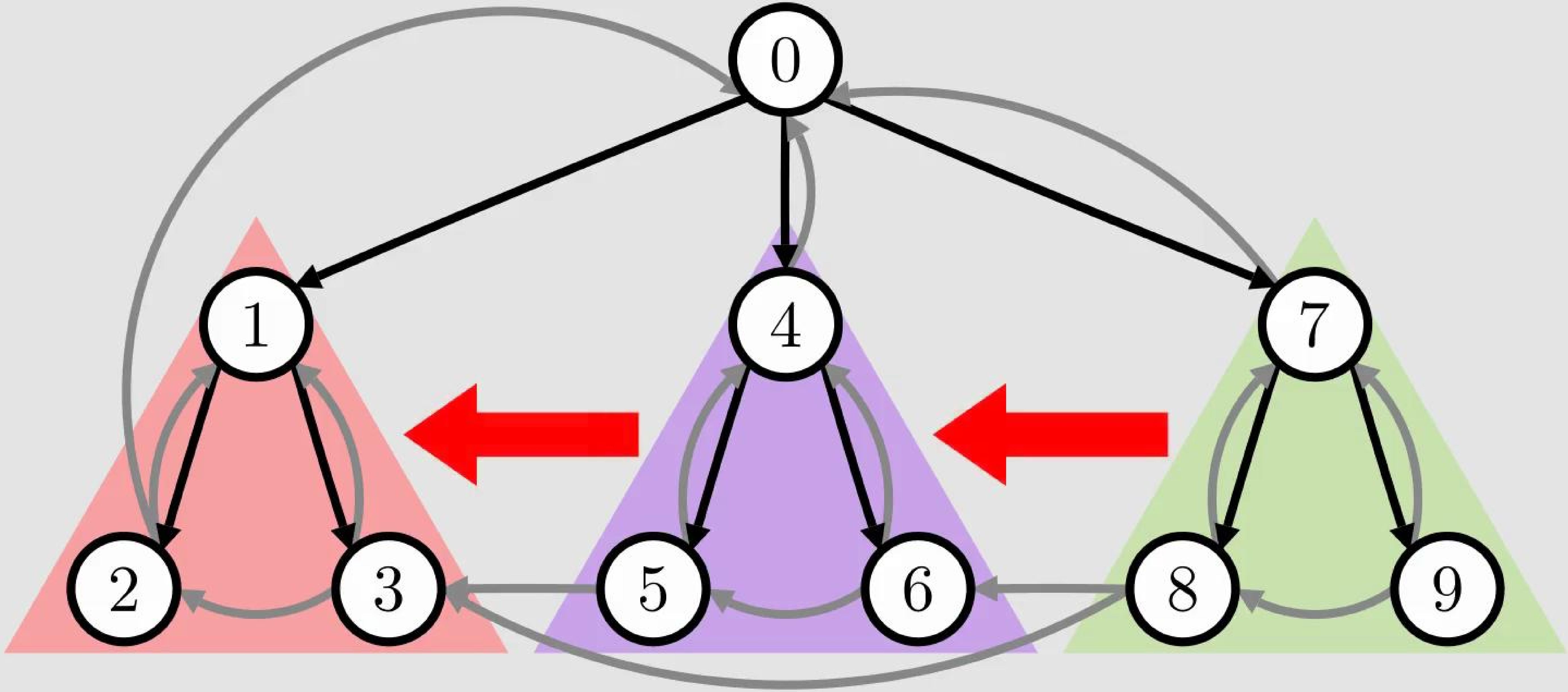
Further Propagation Rules



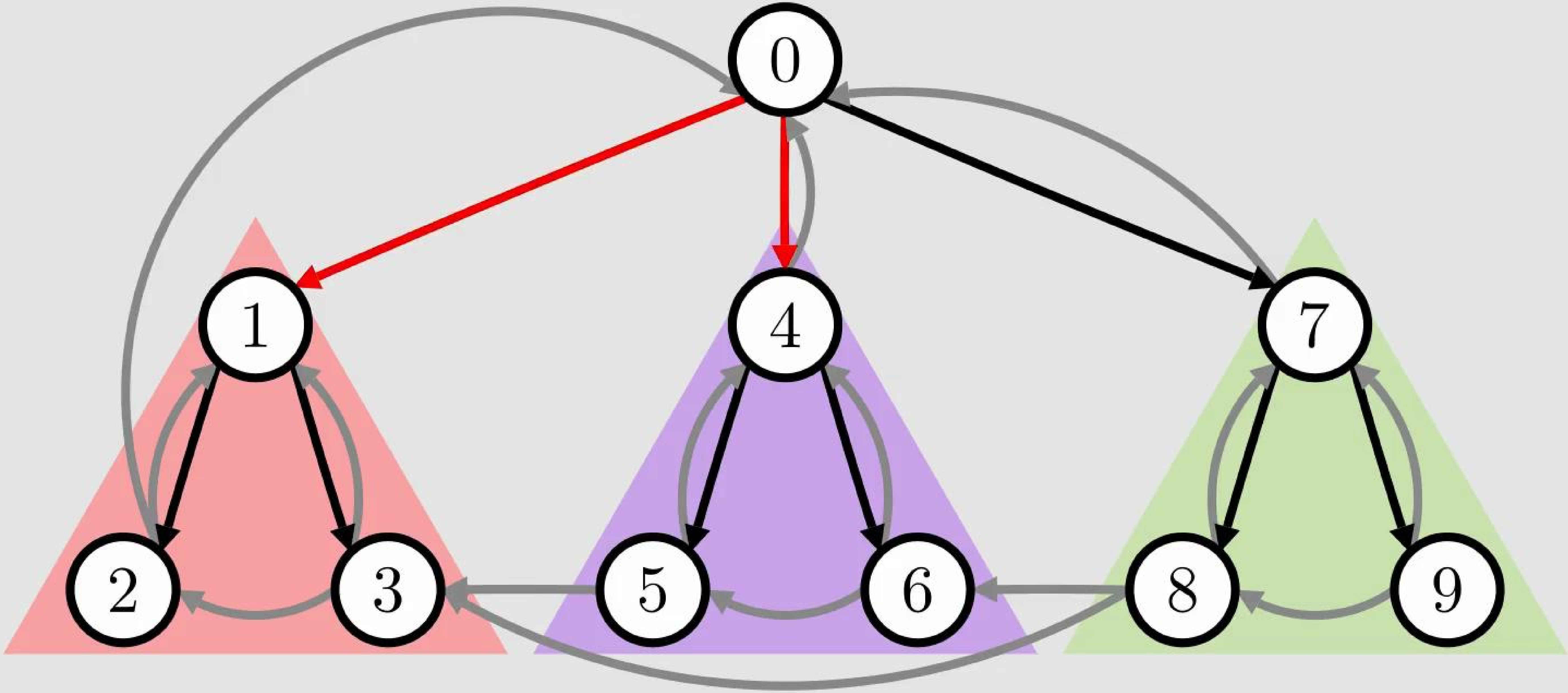
Further Propagation Rules



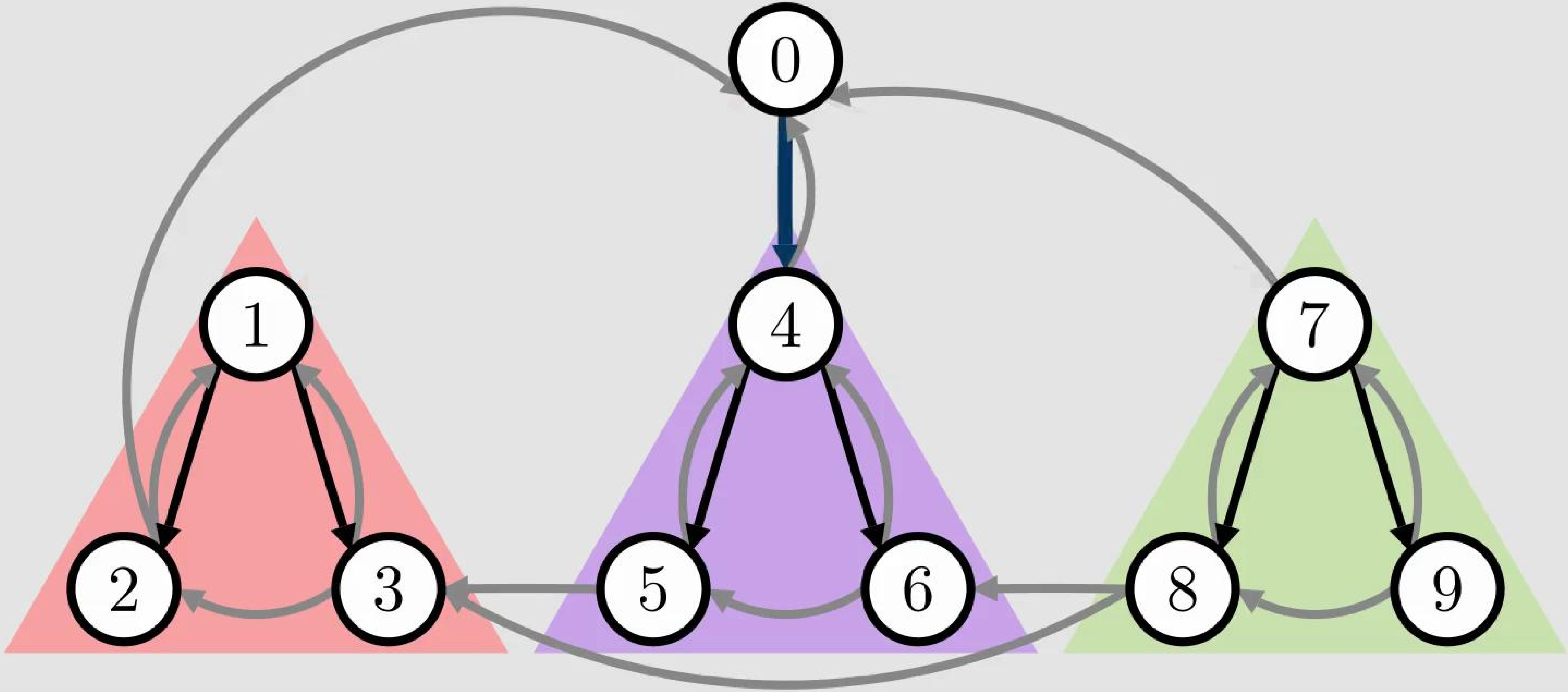
Further Propagation Rules



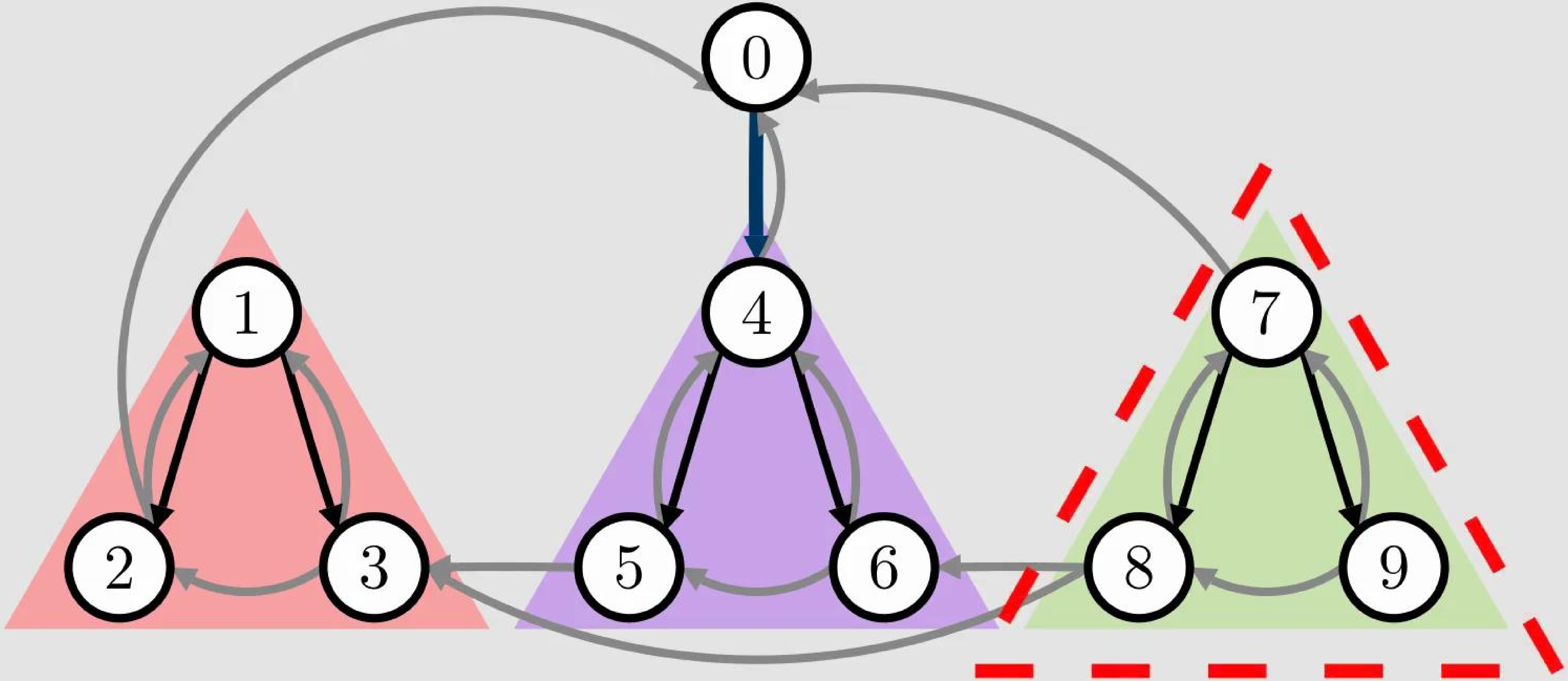
Further Propagation Rules: 'Prune Root'



Further Propagation Rules: 'Prune Root'

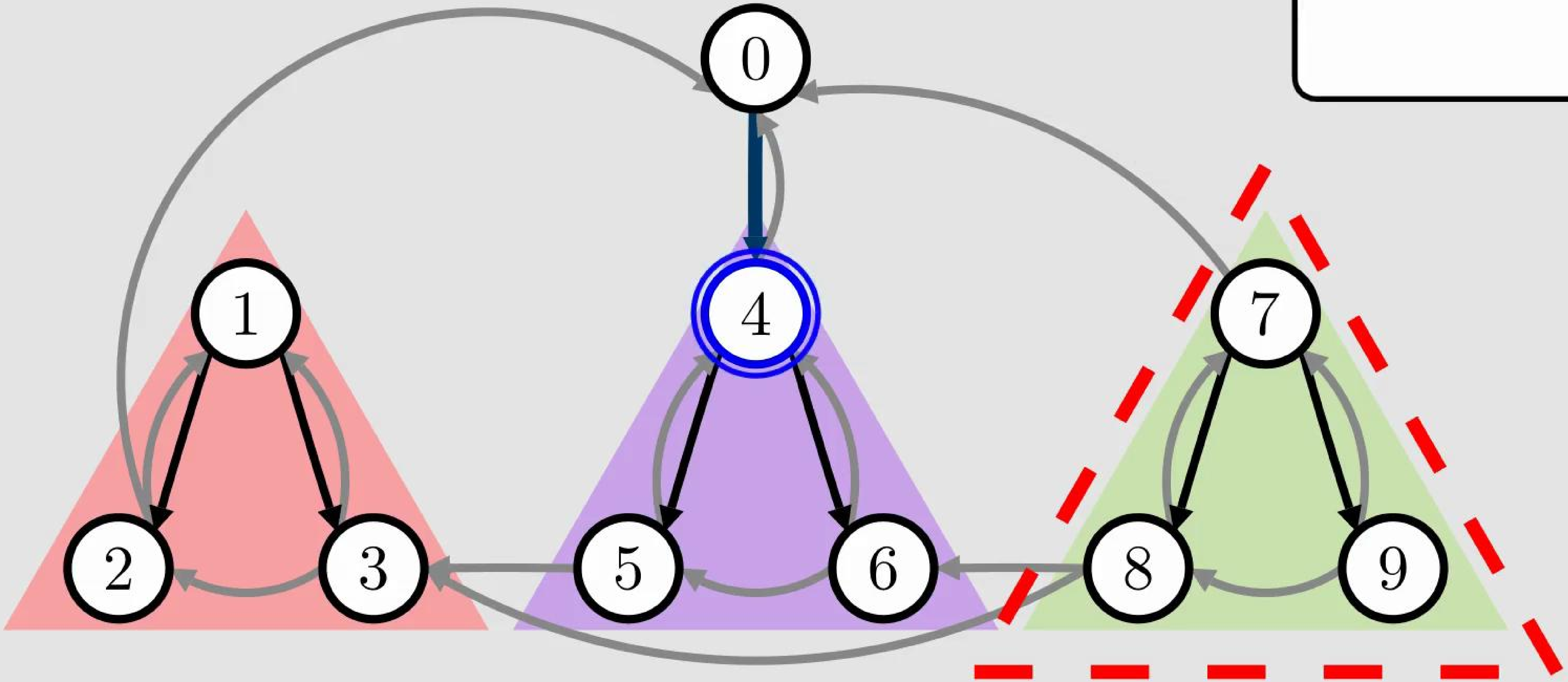


Further Propagation Rules: 'Prune Root'

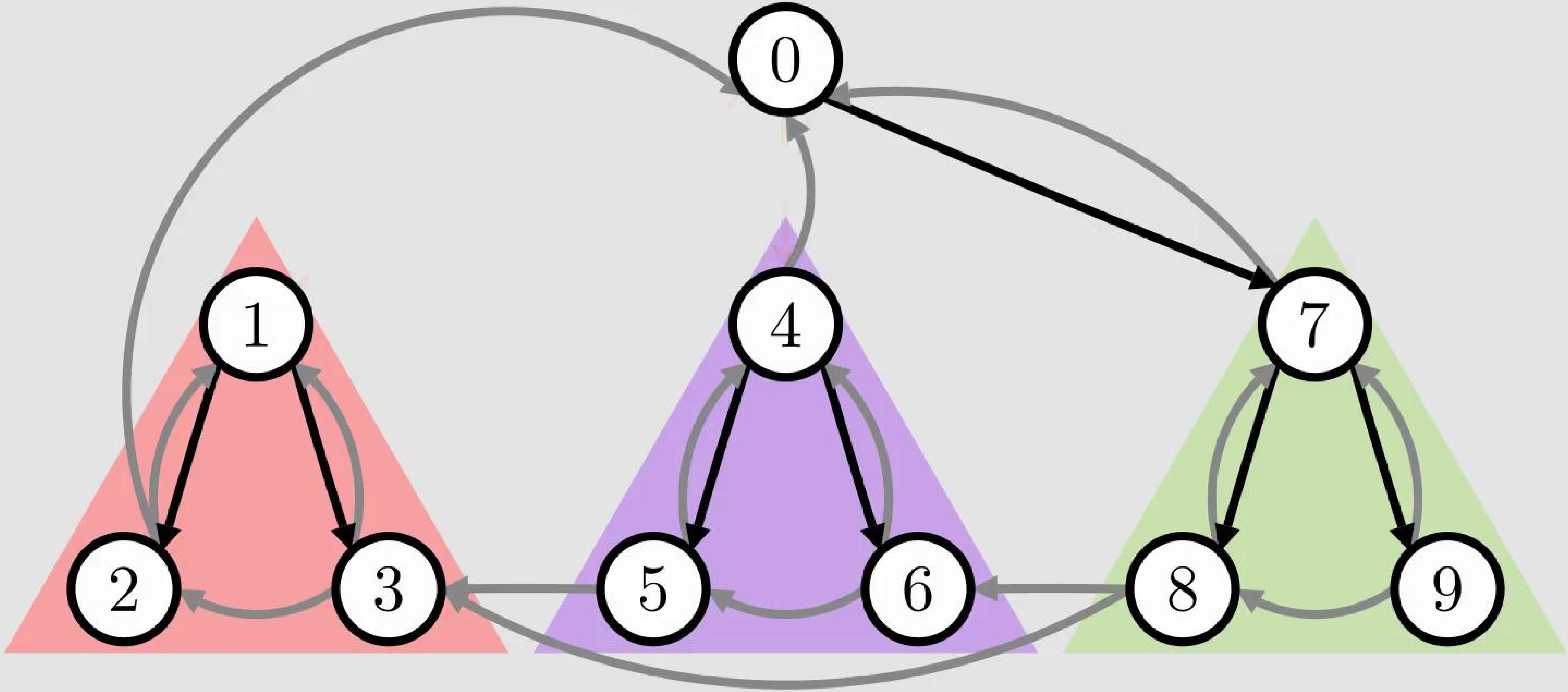


Further Propagation Rules: 'Prune Root'

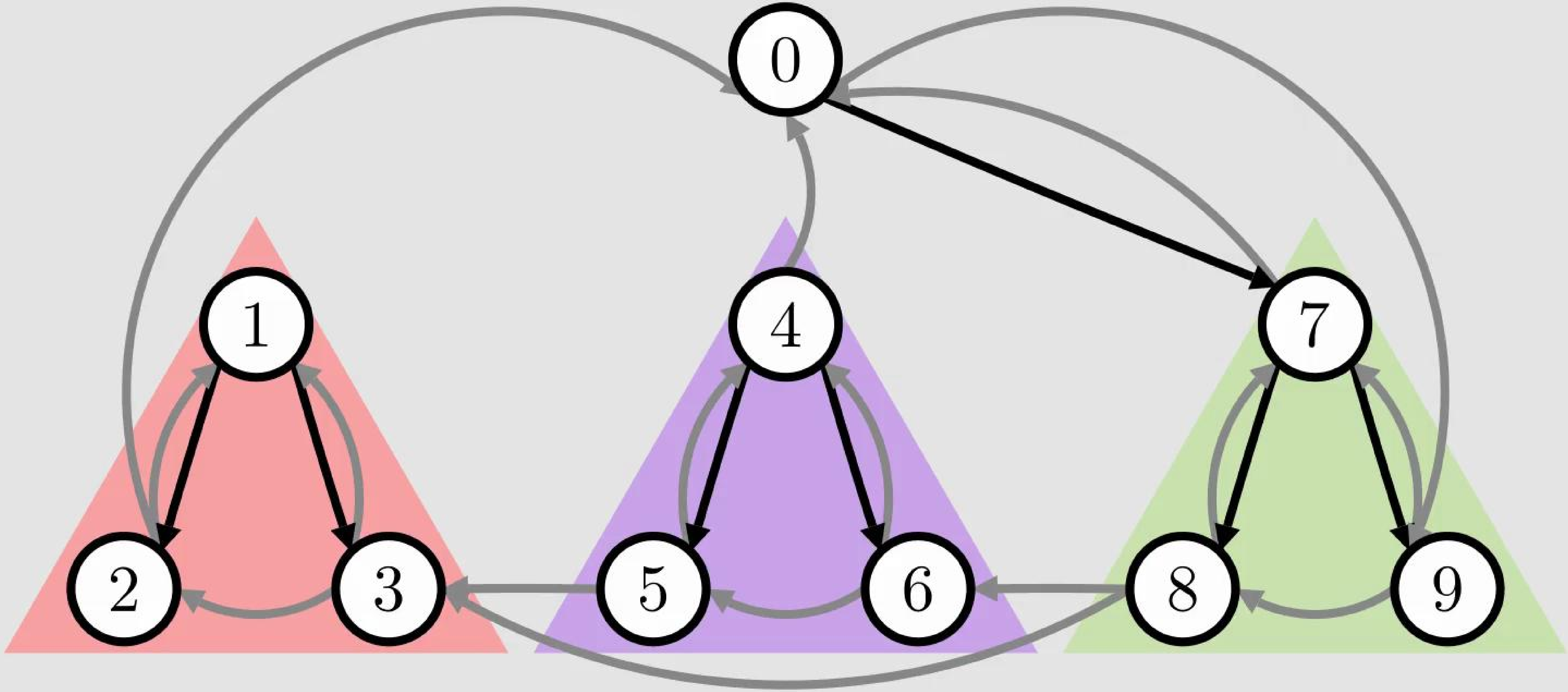
$x_{0=4} \implies$ `ReachTooSmall(4)`



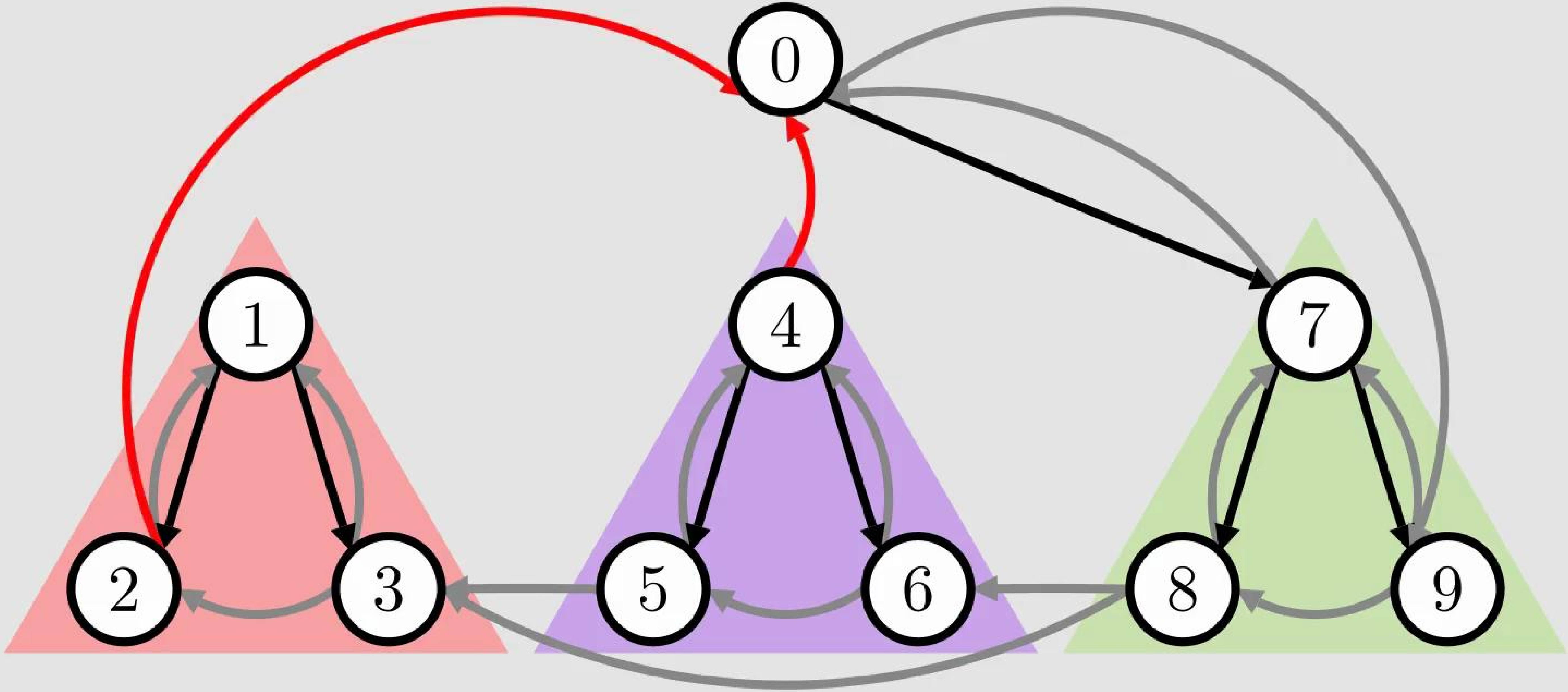
Further Propagation Rules: 'Prune Root'



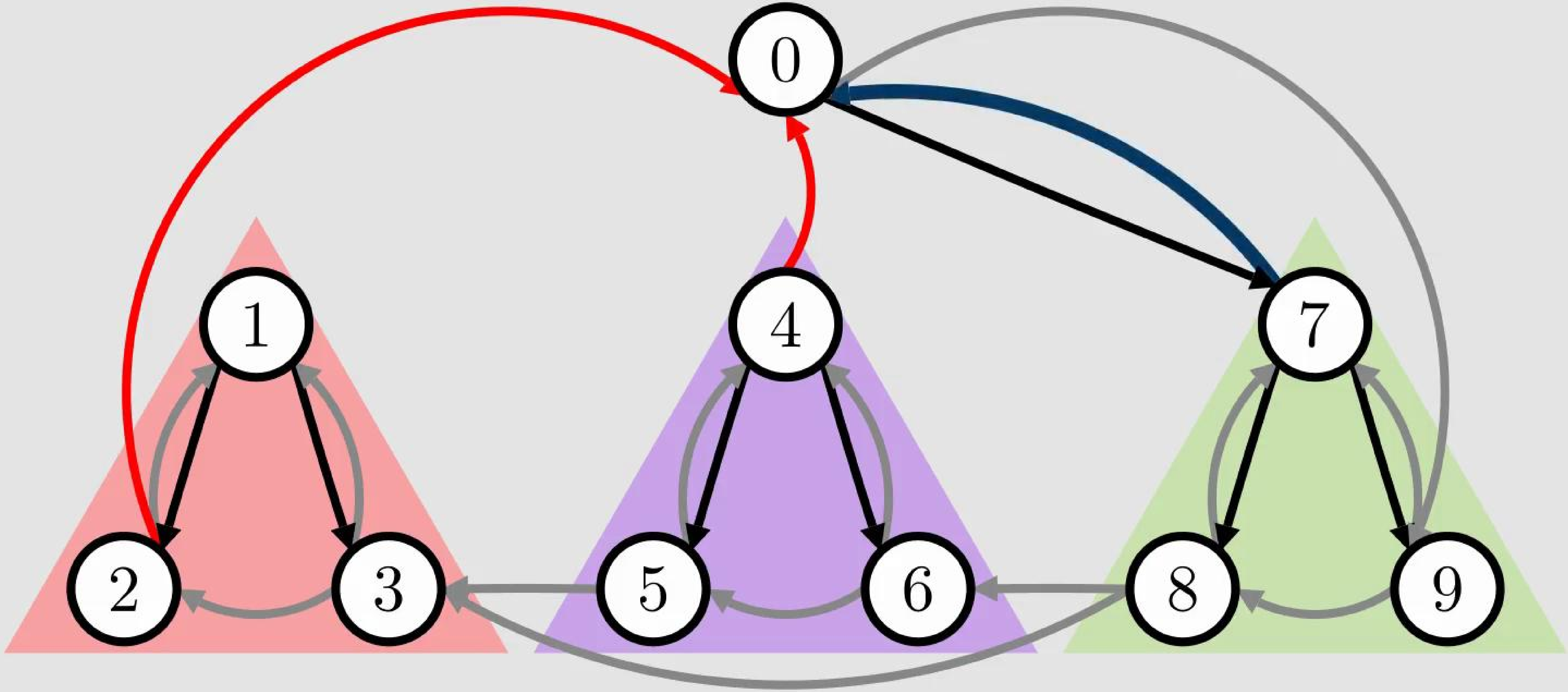
Further Propagation Rules: 'Prune Root'



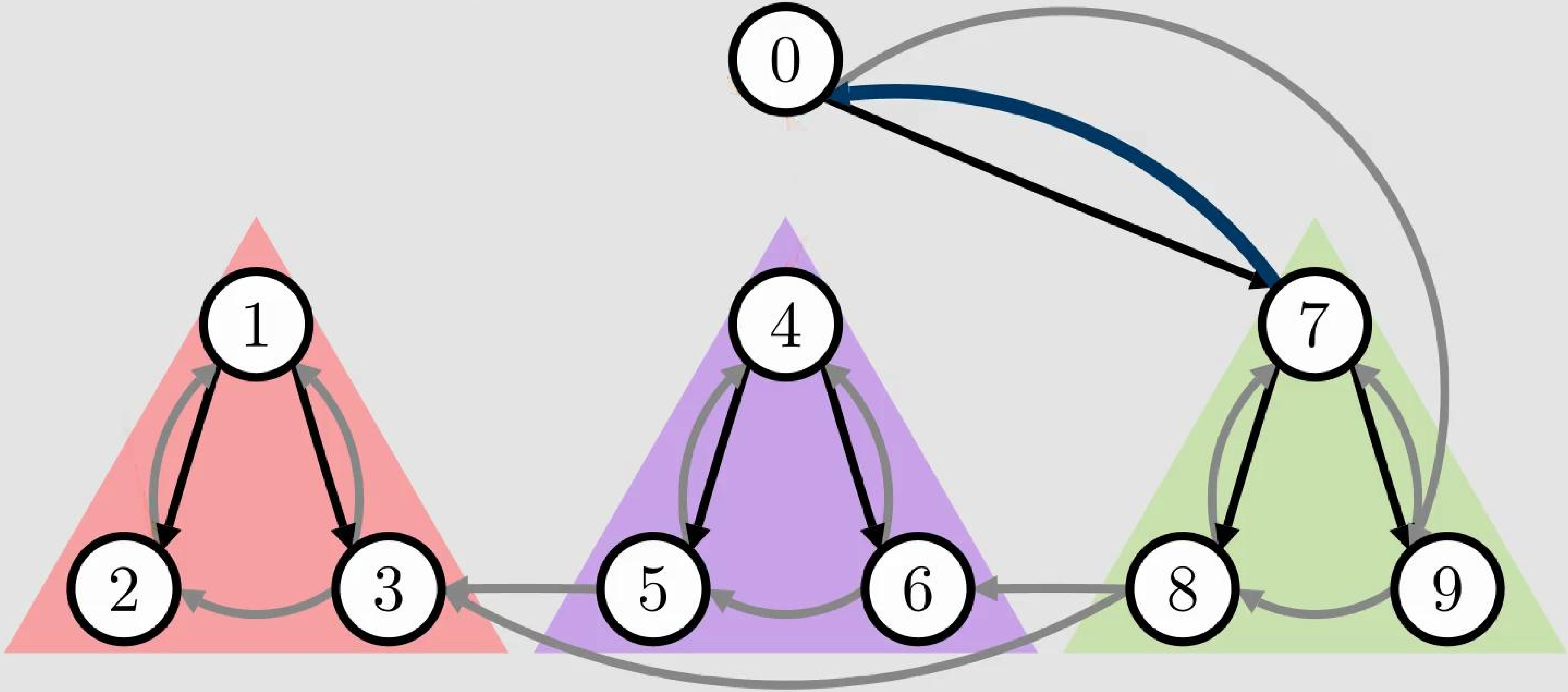
Further Propagation Rules: 'Prune Root'



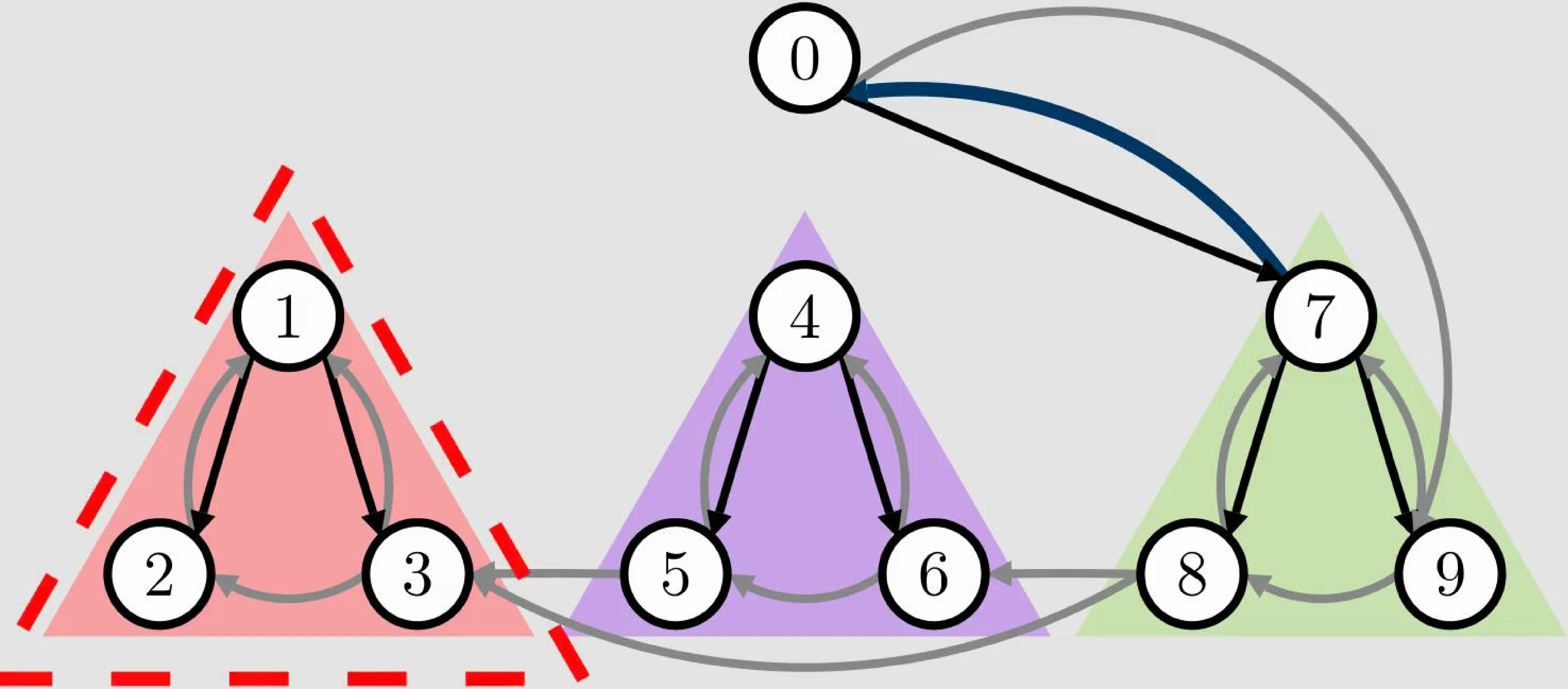
Further Propagation Rules: 'Prune Root'



Further Propagation Rules: 'Prune Root'

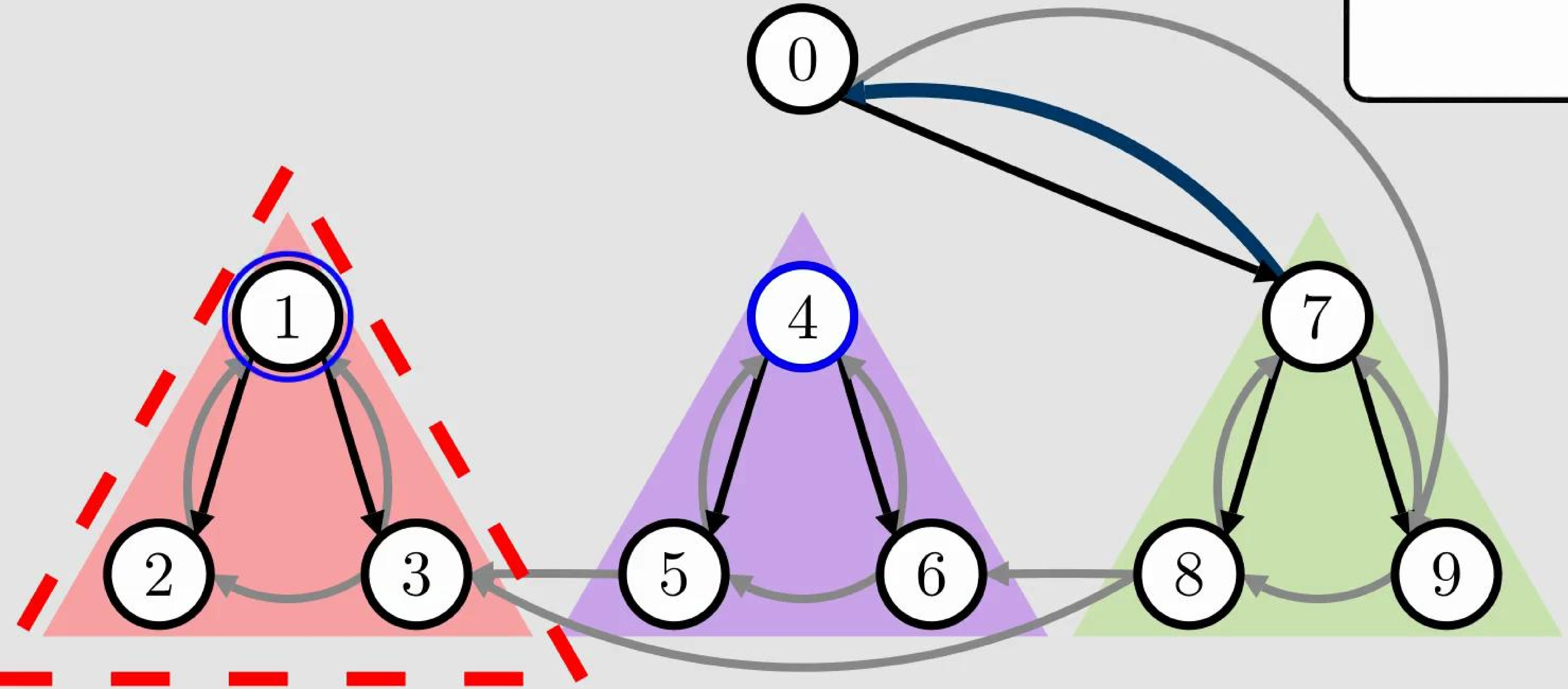


Further Propagation Rules: 'Prune Root'

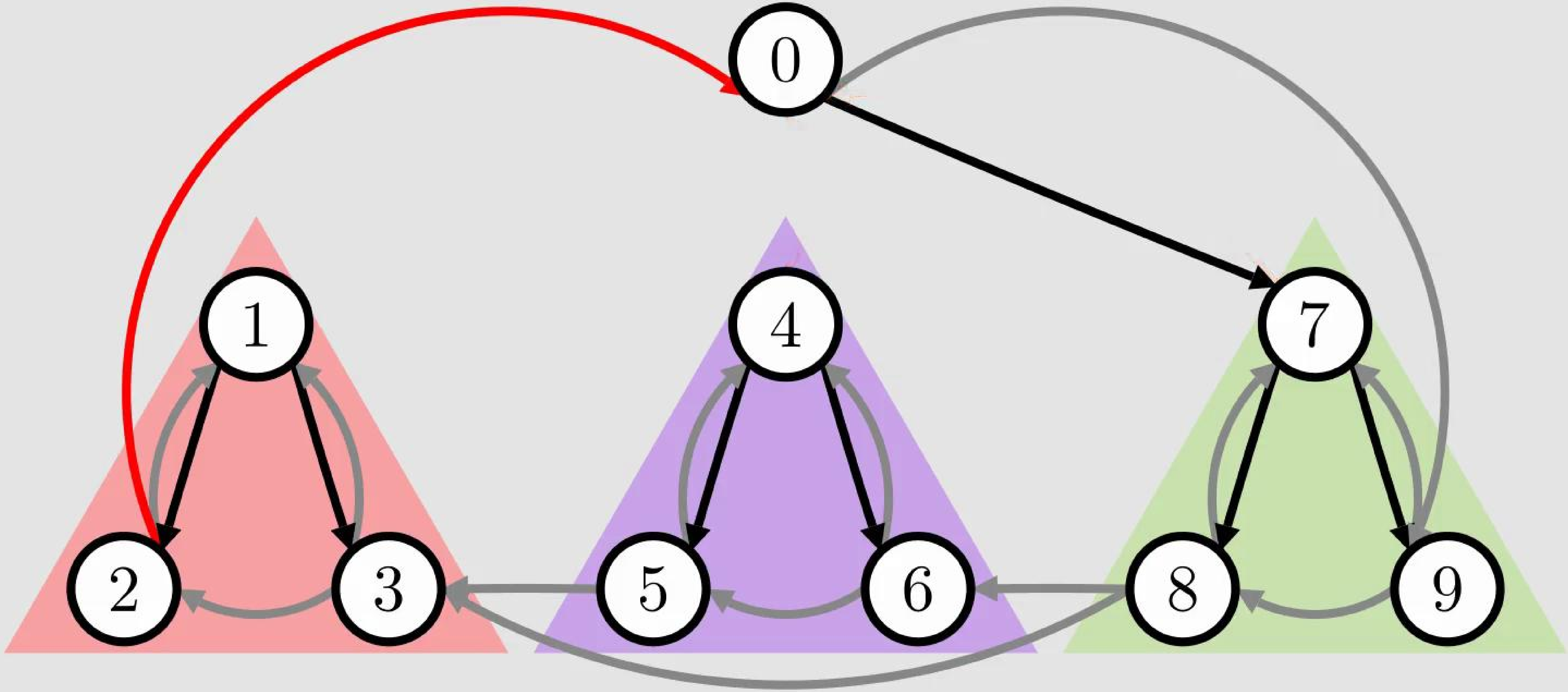


Further Propagation Rules: 'Prune Root'

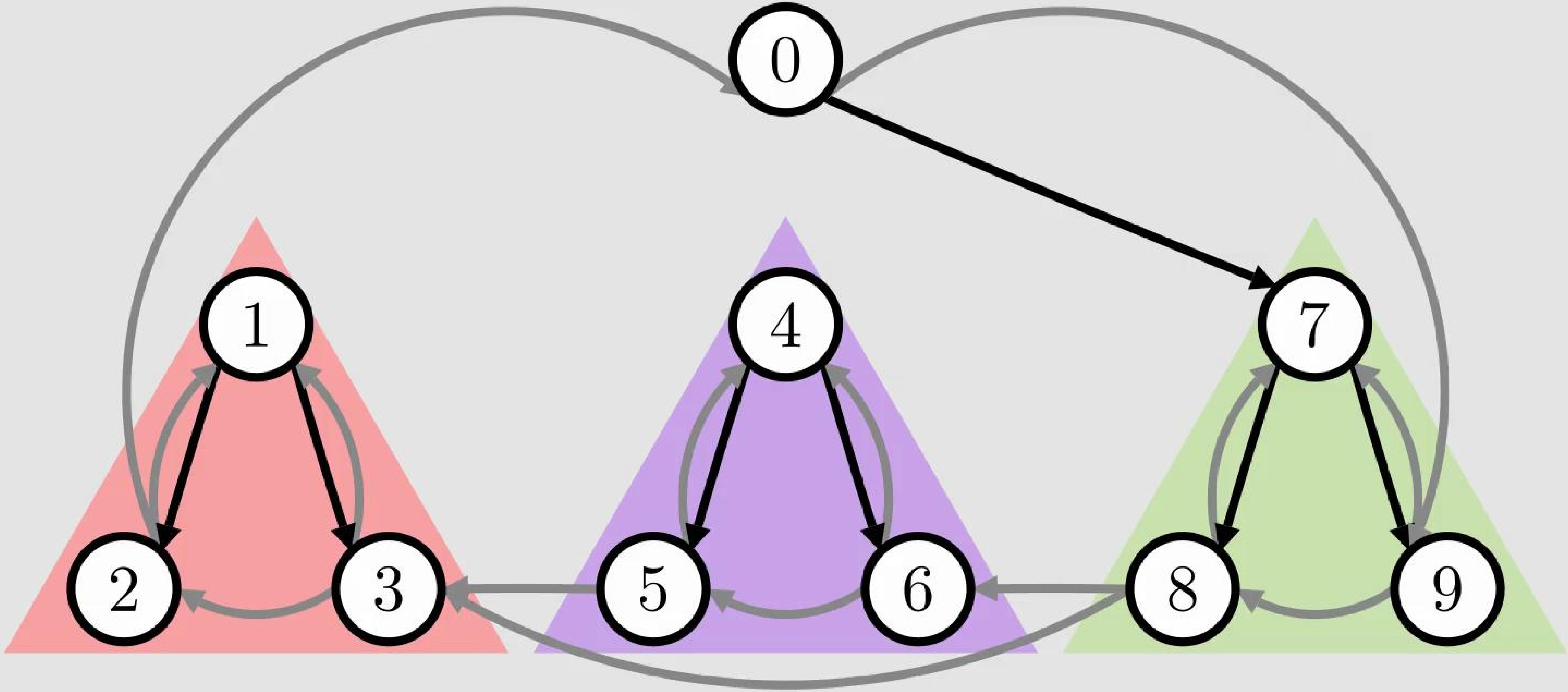
$x_{7=0} \implies$ `ReachTooSmall(1)`



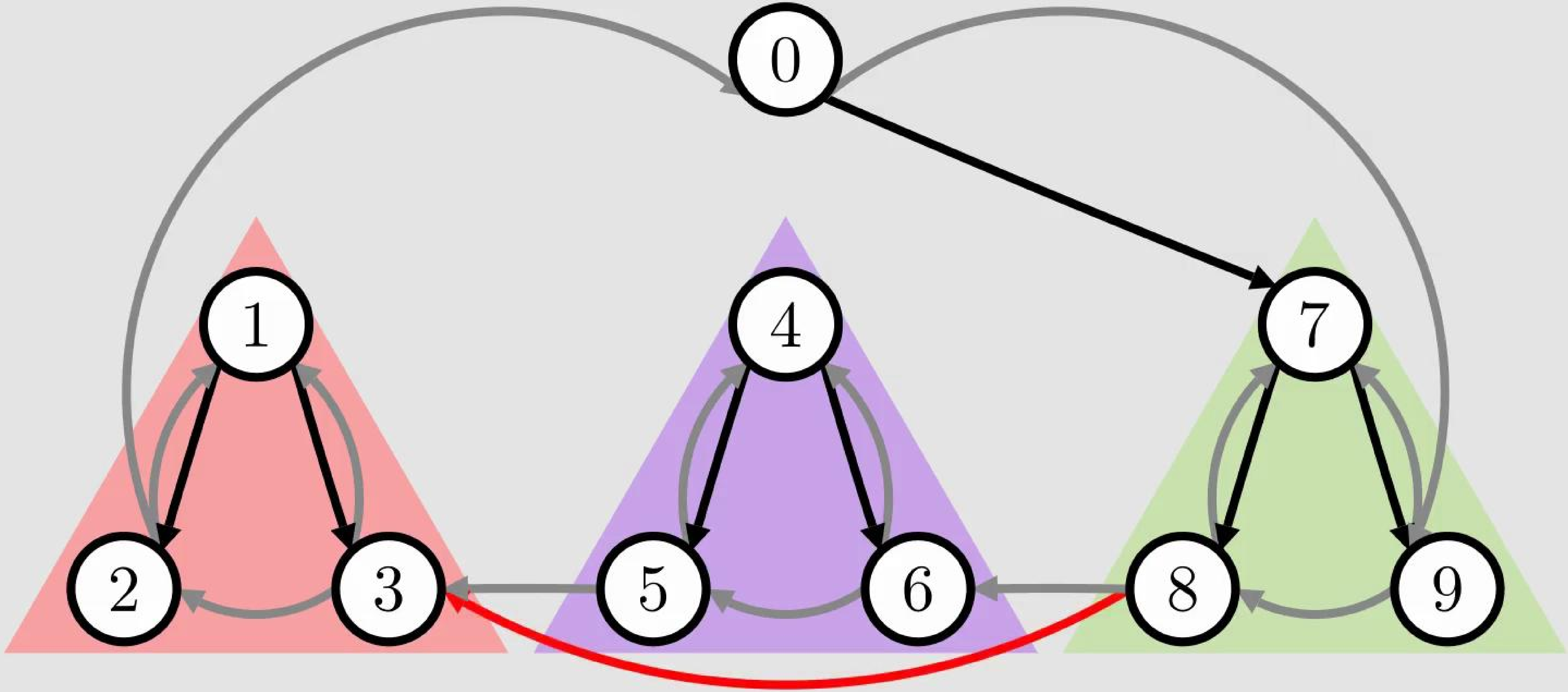
Further Propagation Rules: 'Prune Skip'



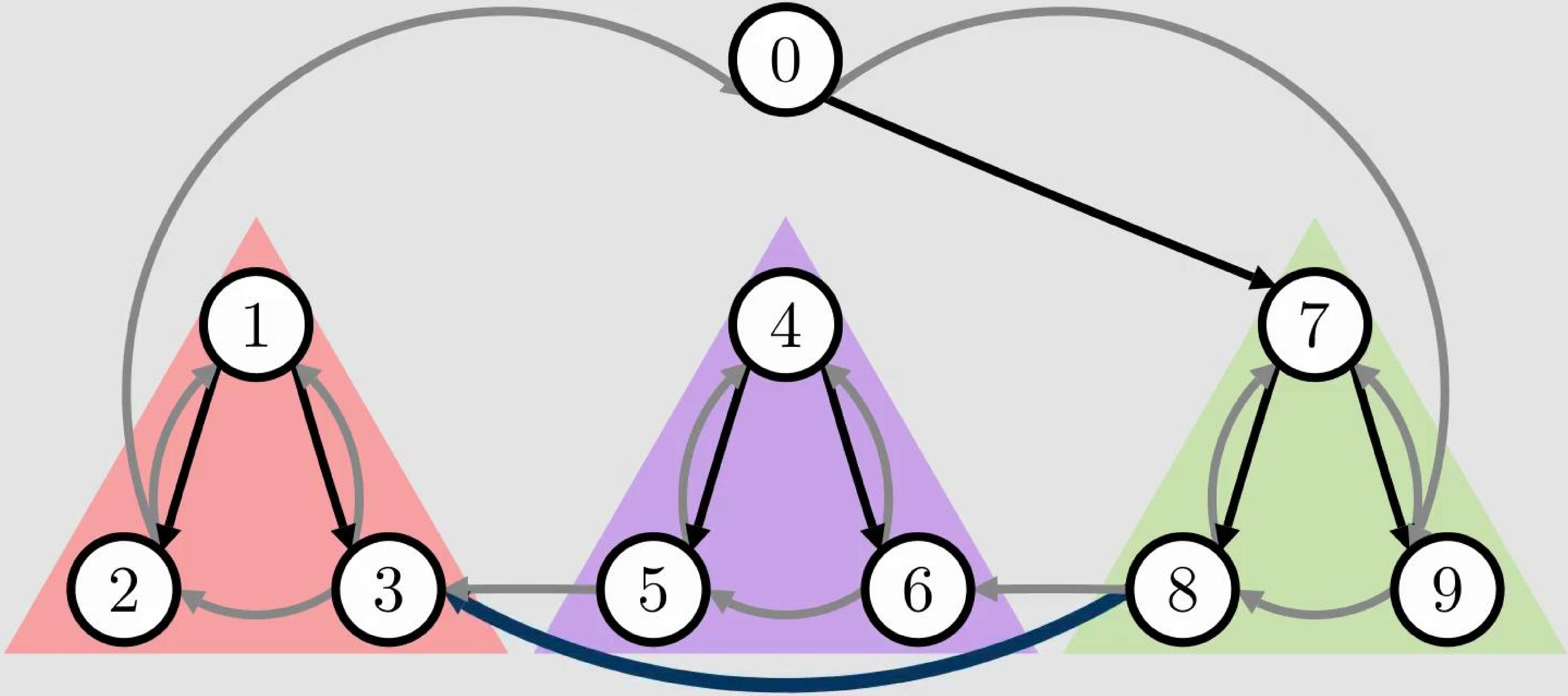
Further Propagation Rules: 'Prune Skip'



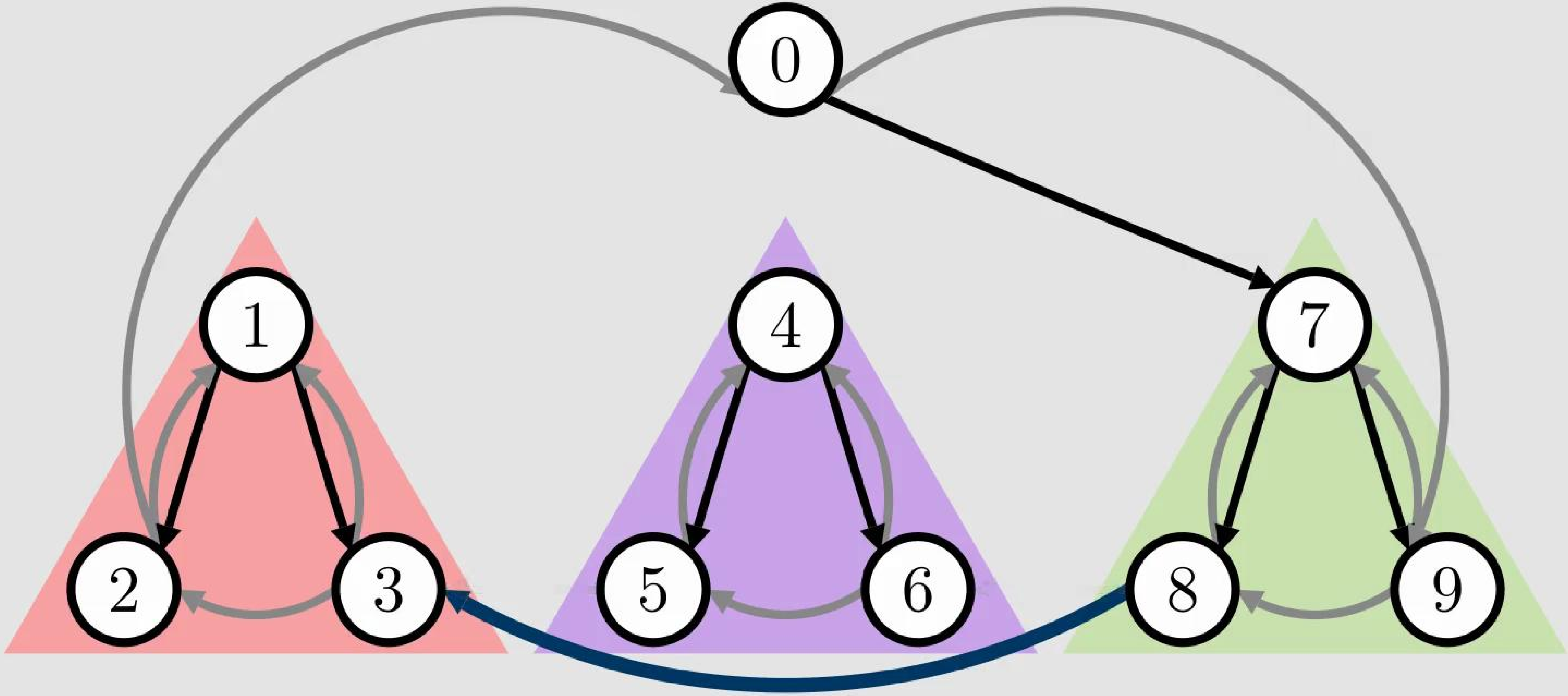
Further Propagation Rules: 'Prune Skip'



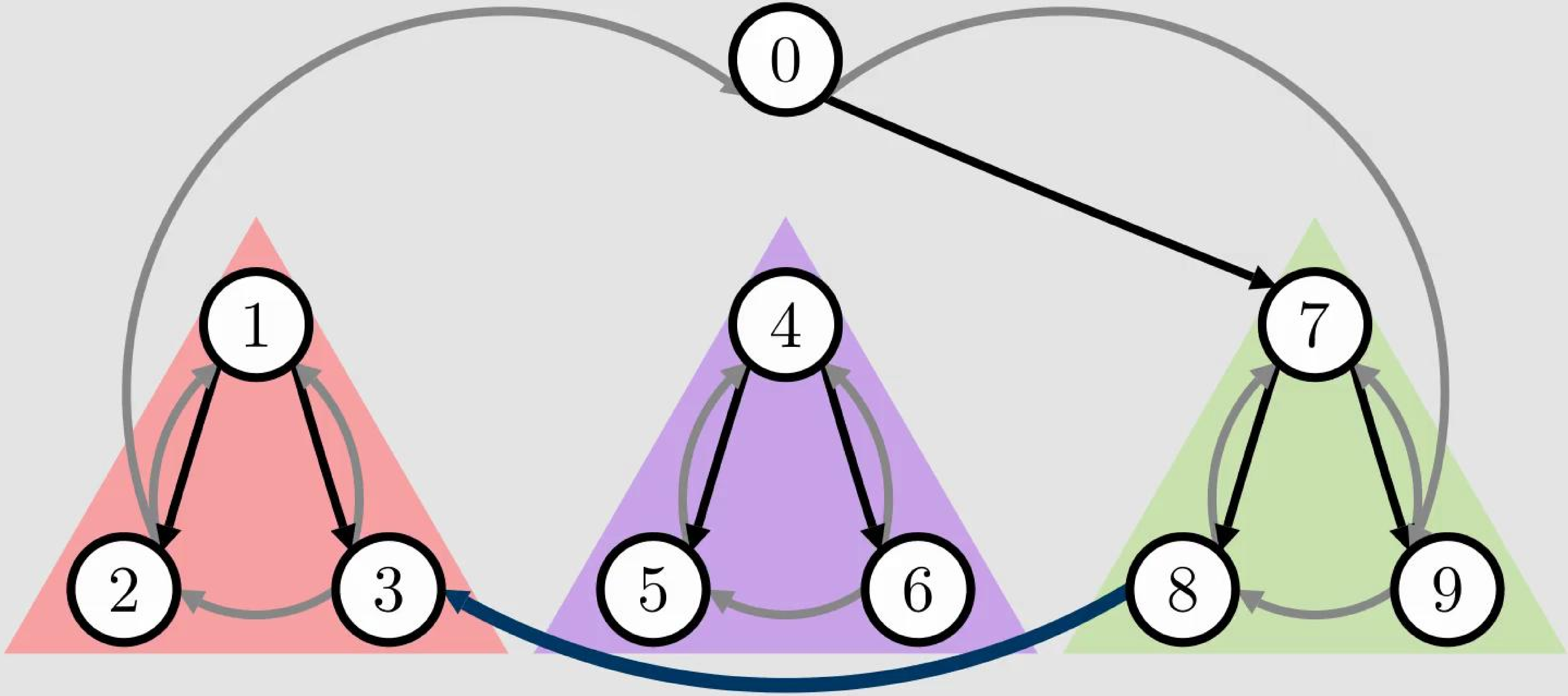
Further Propagation Rules: 'Prune Skip'



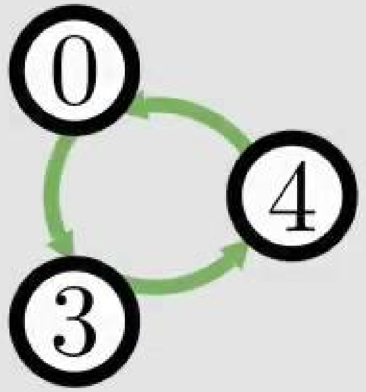
Further Propagation Rules: 'Prune Skip'



Further Propagation Rules: 'Prune Skip'

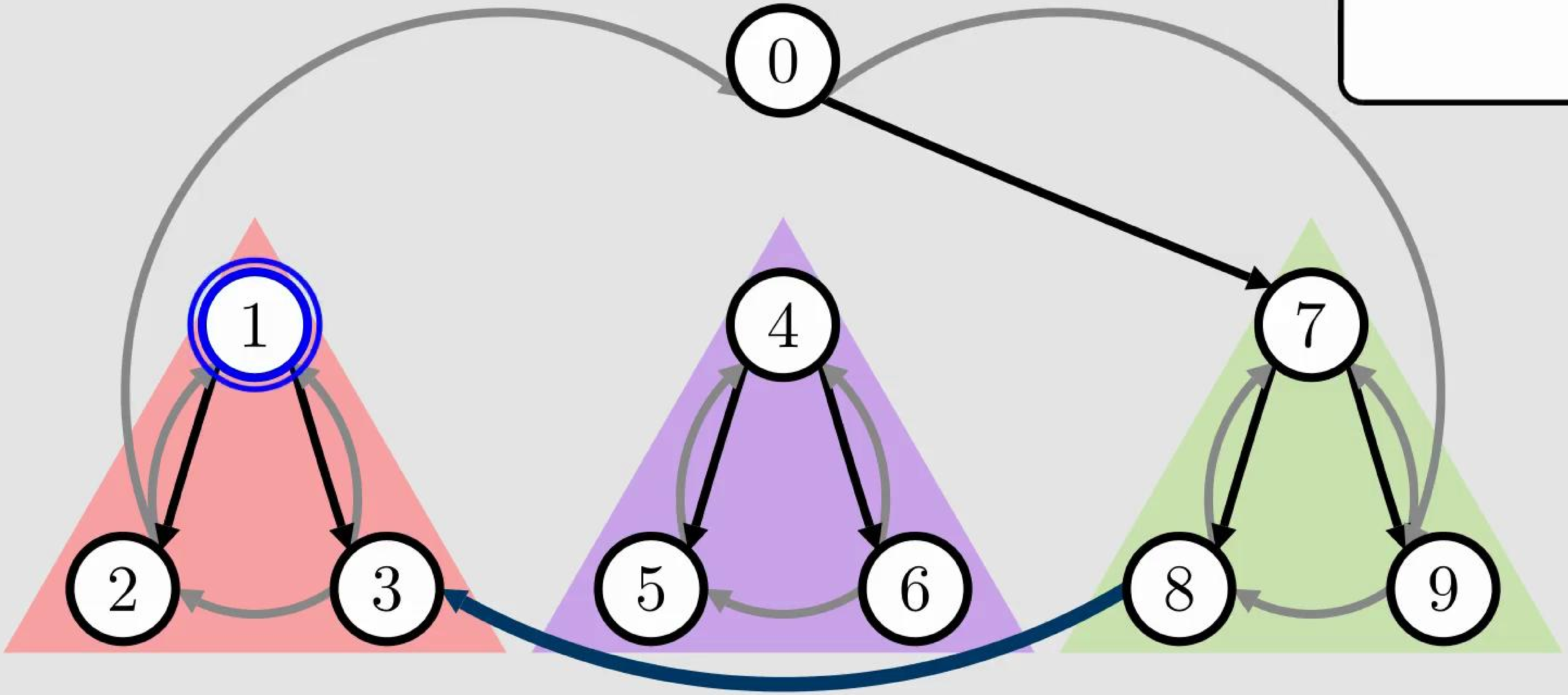


$a_1 \iff$

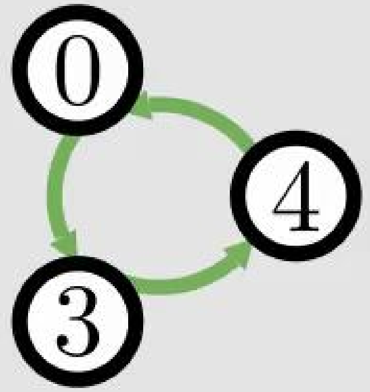


Further Propagation Rules: 'Prune Skip'

$x_{8=3} \wedge a_1 \implies \texttt{ReachTooSmall(1)}$

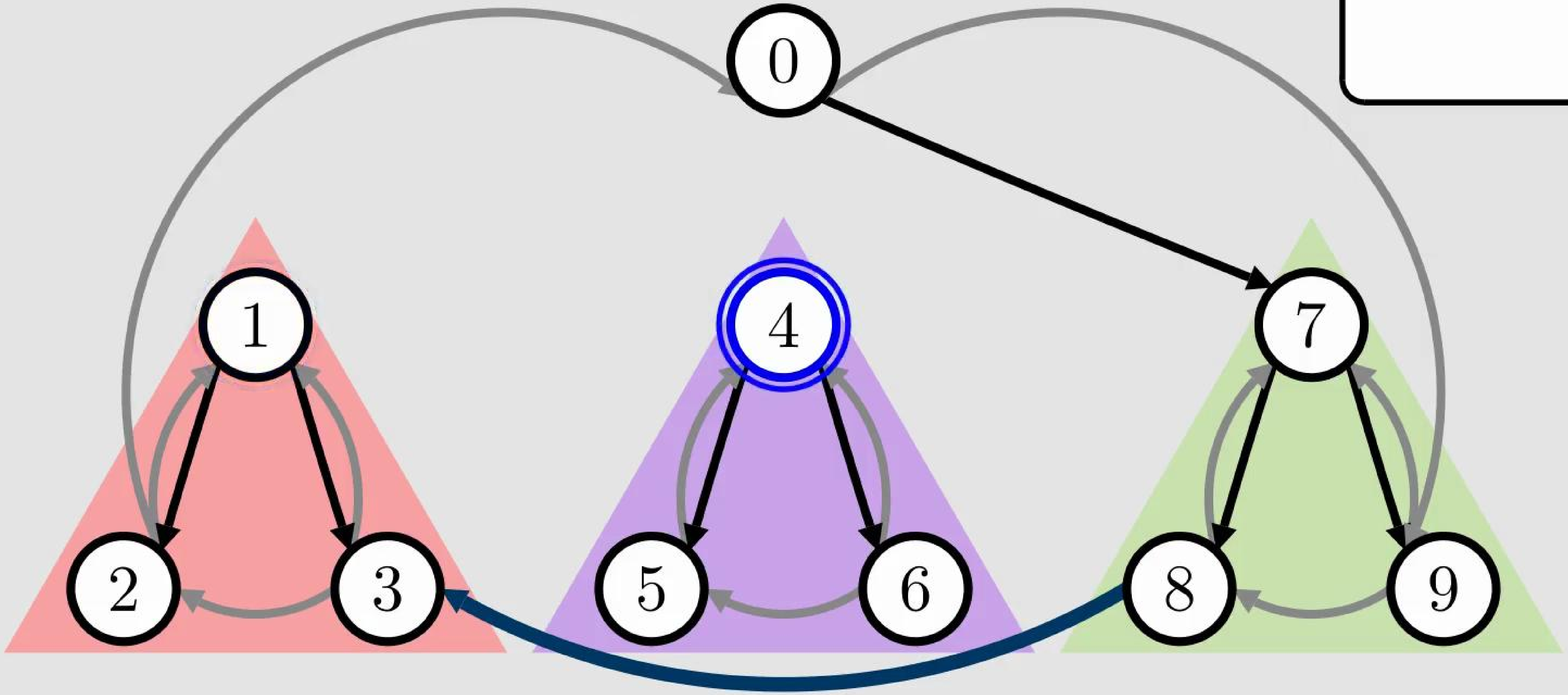


$a_1 \iff$

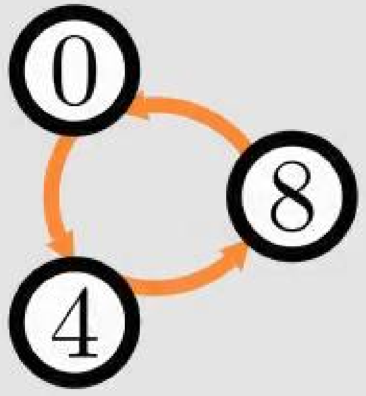


Further Propagation Rules: 'Prune Skip'

$x_{8=3} \wedge a_1 \implies$ `ReachTooSmall(1)`



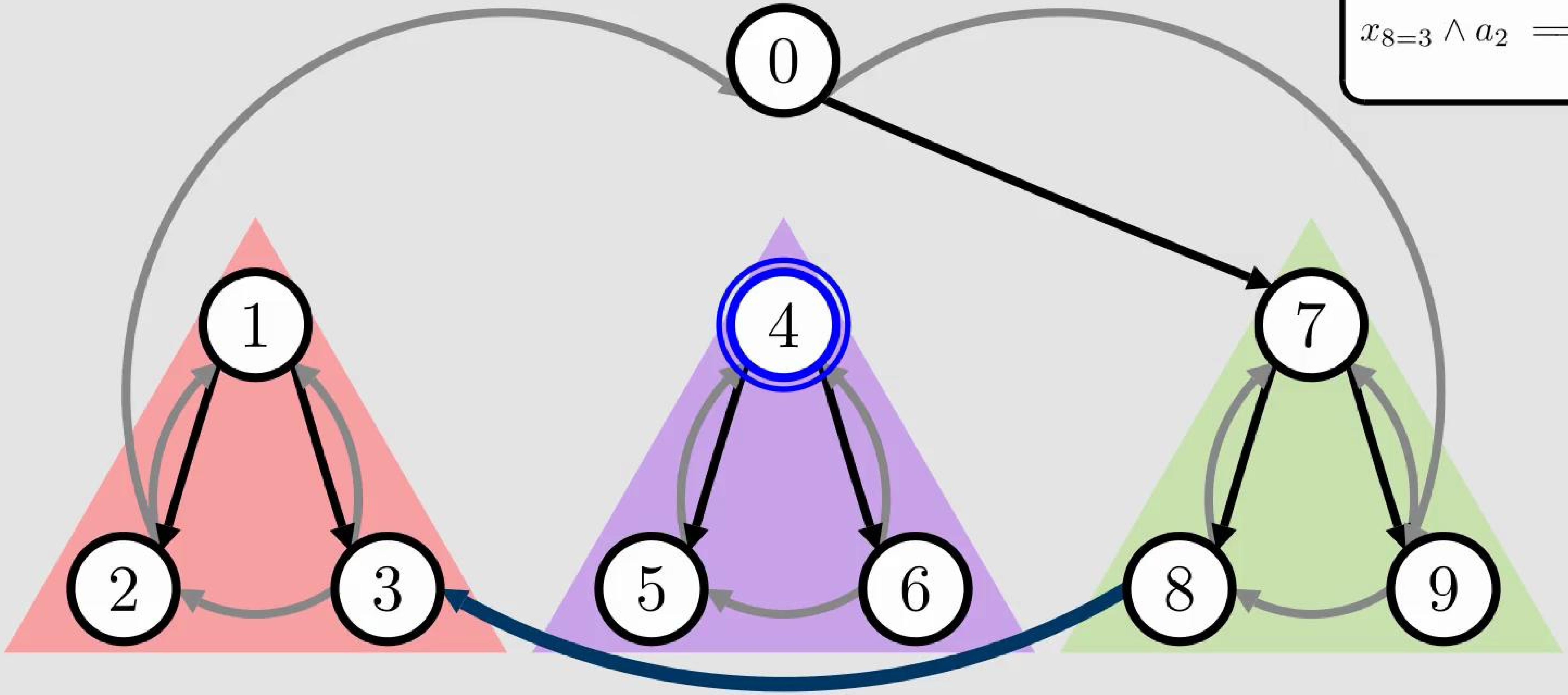
$a_2 \iff$



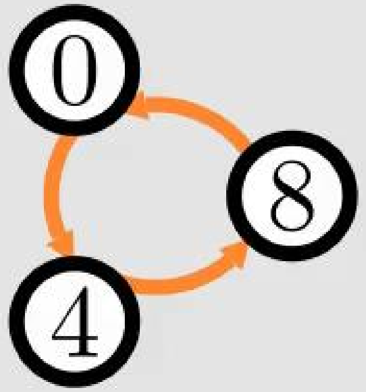
Further Propagation Rules: 'Prune Skip'

$x_{8=3} \wedge a_1 \implies$ `ReachTooSmall(1)`

$x_{8=3} \wedge a_2 \implies$ `ReachTooSmall(4)`



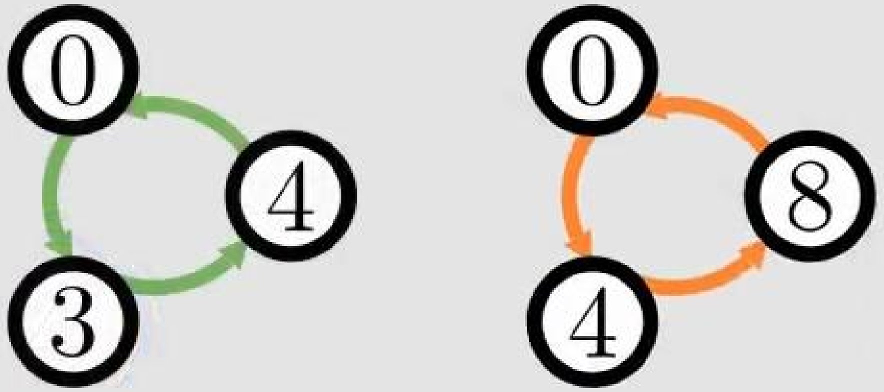
$a_2 \iff$



Further Propagation Rules: 'Prune Skip'

$$x_{8=3} \land a_1 \implies \texttt{ReachTooSmall(1)}$$

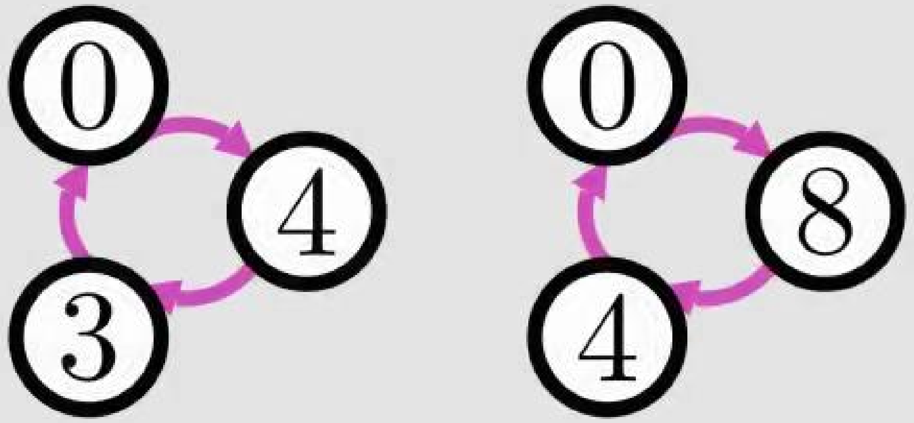
$$x_{8=3} \land a_2 \implies \texttt{ReachTooSmall(4)}$$



Further Propagation Rules: 'Prune Skip'

$x_{8=3} \land a_1 \implies$ `ReachTooSmall(1)`

$x_{8=3} \land a_2 \implies$ `ReachTooSmall(4)`



So Far

So Far

- All Different
- Equals/Not equals
- Array MinMax
- Element
- (Reified) Linear (In)equalities
- Logical (and/or)
- Table
- NValue
- Count
- Among

So Far

- All Different
- Equals/Not equals
- Array MinMax
- Element
- (Reified) Linear (In)equalities
- Logical (and/or)
- Table
- NValue
- Count
- Among

And lately:

- Circuit*
- Multiplication*(somewhat awkard but doable)
- Any constraint with an efficient 'Smart Table' representation* (e.g. Lex, Diffn, Notallequal)
- Any constraint with an efficient MDD representation* (e.g. Knapsack, Regular)
- (Lately) Any constraint with a Network Flow Propagator or Totally Unimodular ILP relaxation (e.g. GCC, Inverse, Sequence)

So Far

- All Different
- Equals/Not equals
- Array MinMax
- Element
- (Reified) Linear (In)equalities
- Logical (and/or)
- Table
- NValue
- Count
- Among

And lately:

- Circuit*
- Multiplication*(somewhat awkard but doable)
- Any constraint with an efficient 'Smart Table' representation*
  (e.g. Lex, Diffn, Notallequal)
- Any constraint with an efficient MDD representation*
  (e.g. Knapsack, Regular)
- (Lately) Any constraint with a Network Flow Propagator
  or Totally Unimodular ILP relaxation
  (e.g. GCC, Inverse, Sequence)

*Citations available on request :-)

So Far

- All Different
- Equals/Not equals
- Array MinMax
- Element
- (Reified) Linear (In)equalities
- Logical (and/or)
- Table
- NValue
- Count
- Among

And lately:

- Circuit*
- Multiplication (somewhat awkward but doable)
- Any constraint with an efficient 'Smart Table' representation* (e.g. Lex, Diffn, Notallequal)
- Any constraint with an efficient MDD representation* (e.g. Knapsack, Regular)
- (Lately) Any constraint with a Network Flow Propagator or Totally Unimodular ILP relaxation (e.g. GCC, Inverse, Sequence)

*Citations available on request :-)

`vec_eq_tuple`

`visible`

`weighted_partial_alldiff`

`xor`

`zero_or_not_zero`

`zero_or_not_zero_vectors`

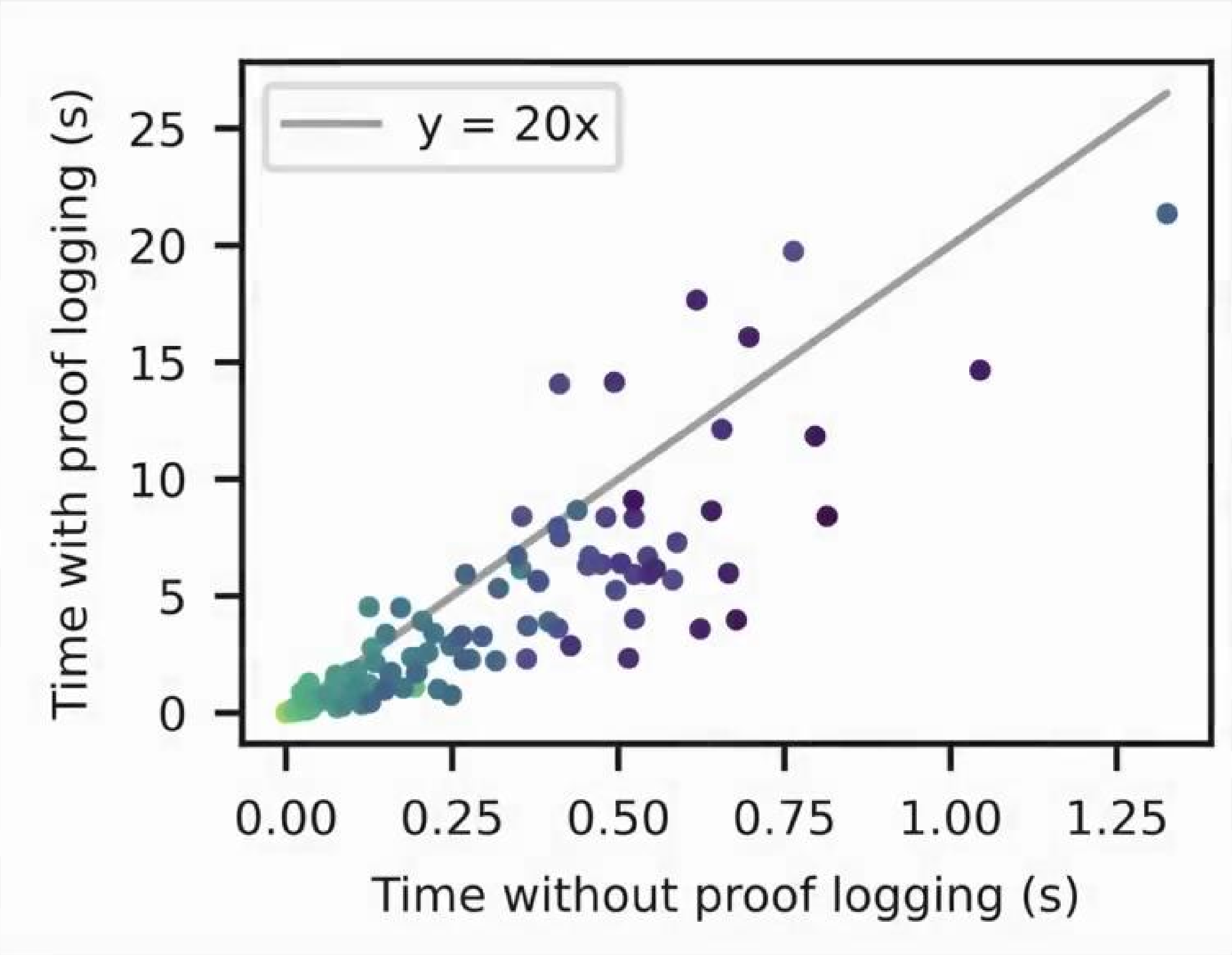
Further Challenges

Further Challenges

- Painful overheads on top of solving

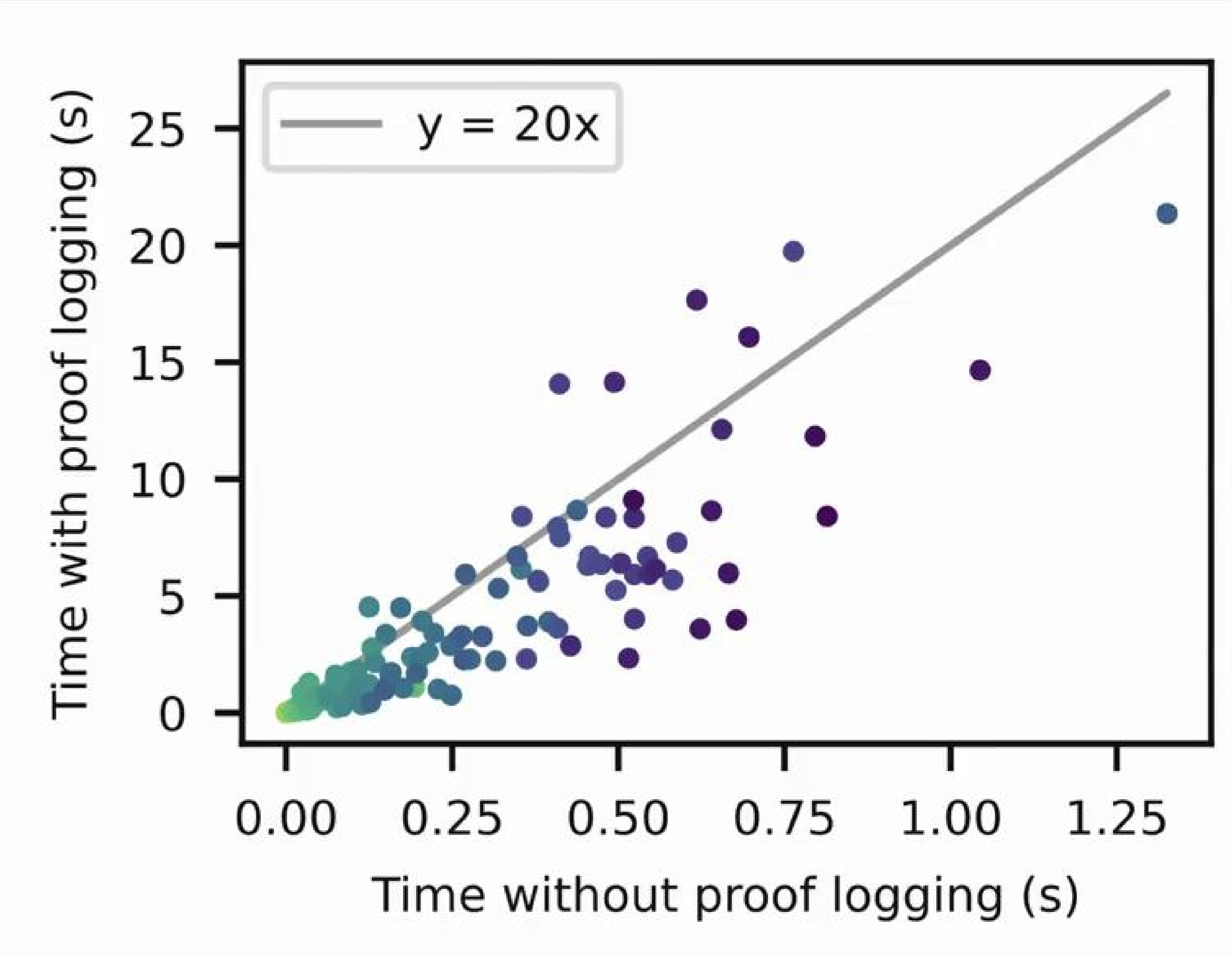
Further Challenges

- Painful overheads on top of solving



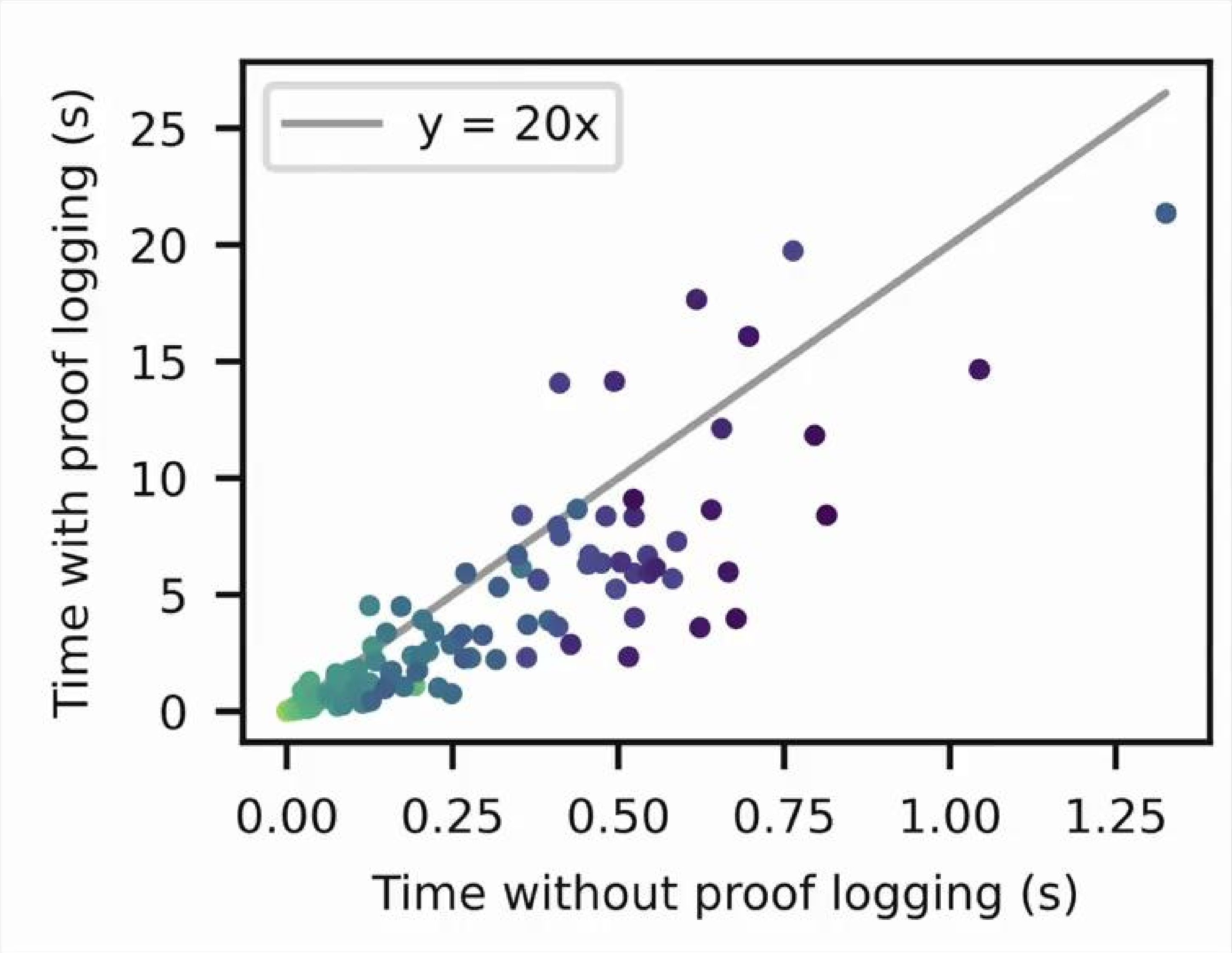
Further Challenges

- Painful overheads on top of solving
- (Can be) difficult to implement



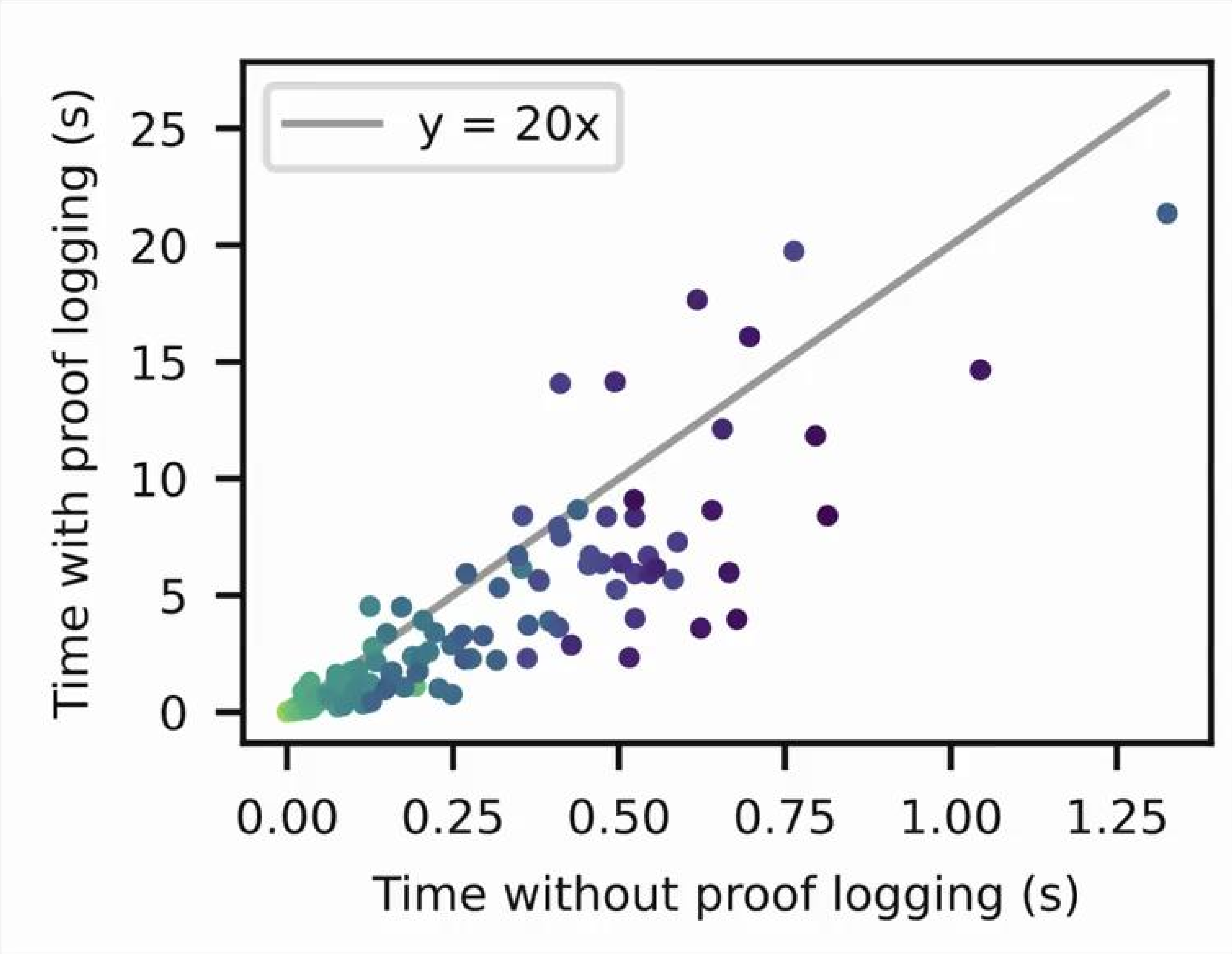
Further Challenges

- Painful overheads on top of solving
- (Can be) difficult to implement
- Verification overhead



Further Challenges

- Painful overheads on top of solving
- (Can be) difficult to implement
- Verification overhead
- Trusting the PB Encoding (or the verifiers's input more broadly)



Multi-Stage Proof Logging, 2024

A Multi-Stage Proof Logging Framework to Certify the Correctness of CP Solvers

Maarten Flippo
Delft University of Technology, The Netherlands
Konstantin Sidorov
Delft University of Technology, The Netherlands
Imko Marijnissen
Delft University of Technology, The Netherlands
Jeff Smits
Delft University of Technology, The Netherlands
Emir Demirović
Delft University of Technology, The Netherlands

Abstract

Proof logging is used to increase trust in the optimality and unsatisfiability claims of solvers. However, to this date, no constraint programming solver can practically produce proofs without significantly impacting performance, which hinders mainstream adoption. We address this issue by introducing a novel proof generation *framework*, together with a CP proof format and proof checker. Our approach is to divide the proof generation into three steps. At runtime, we require the CP solver to only produce a proof sketch, which we call a scaffold. After the solving is done, our proof processor trims and expands the scaffold into a full CP proof, which is subsequently verified. Our framework is agnostic to the solver and the verification approach. Through MiniZinc benchmarks, we demonstrate that with our framework, the overhead of logging during solving is often less than 10%, significantly lower than other approaches, and that our proof processing step can reduce the overall size of the proof by orders of magnitude and by extension the proof checking time. Our results demonstrate that proof logging has the potential to become an integral part of the CP community.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization; Theory of computation → Logic and verification

Keywords and phrases proof logging, formal verification, constraint programming

Multi-Stage Proof Logging, 2024

A Multi-Stage Proof Logging Framework to Certify the Correctness of CP Solvers

Maarten Flippo
Delft University of Technology, The Netherlands

Konstantin Sidorov
Delft University of Technology, The Netherlands

Imko Marijnissen
Delft University of Technology, The Netherlands

Jeff Smits
Delft University of Technology, The Netherlands

Emir Demirović
Delft University of Technology, The Netherlands

Abstract

Proof logging is used to increase trust in the optimality and unsatisfiability claims of solvers. However, to this date, no constraint programming solver can practically produce proofs without significantly impacting performance, which hinders mainstream adoption. We address this issue by introducing a novel proof generation *framework*, together with a CP proof format and proof checker. Our approach is to divide the proof generation into three steps. At runtime, we require the CP solver to only produce a proof sketch, which we call a scaffold. After the solving is done, our proof processor trims and expands the scaffold into a full CP proof, which is subsequently verified. Our framework is agnostic to the solver and the verification approach. Through MiniZinc benchmarks, we demonstrate that with our framework, the overhead of logging during solving is often less than 10%, significantly lower than other approaches, and that our proof processing step can reduce the overall size of the proof by orders of magnitude and by extension the proof checking time. Our results demonstrate that proof logging has the potential to become an integral part of the CP community.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization; Theory of computation → Logic and verification

Keywords and phrases proof logging, formal verification, constraint programming

First output a 'scaffold';

then find which justifications are needed;

then then fill in the derivations.

If nothing else

- Proof logging is worth doing, generally speaking.

If nothing else

- Proof logging is worth doing, generally speaking.
- Constraint Programming Solvers have a huge potential to be turned into certifying algorithms.

If nothing else

- Proof logging is worth doing, generally speaking.
- Constraint Programming Solvers have a huge potential to be turned into certifying algorithms.
- Pseudo-Boolean proof logging seems to be very effective for a wide range of constraint propagation algorithms.

If nothing else

- Proof logging is worth doing, generally speaking.
- Constraint Programming Solvers have a huge potential to be turned into certifying algorithms.
- Pseudo-Boolean proof logging seems to be very effective for a wide range of constraint propagation algorithms.
- In particular, high-level constraint reasoning can be reduced to simple steps in a (relatively) simple proof system.

Open Questions

- Are there going to be CP constraints fundamentally difficult for PB justifications?

Open Questions

- Are there going to be CP constraints fundamentally difficult for PB justifications?
- Can we integrate low-level proofs with external trusted justifiers?

Open Questions

- Are there going to be CP constraints fundamentally difficult for PB justifications?
- Can we integrate low-level proofs with external trusted justifiers?
- How else can we encourage uptake in the CP community?

Open Questions

- Are there going to be CP constraints fundamentally difficult for PB justifications?
- Can we integrate low-level proofs with external trusted justifiers?
- How else can we encourage uptake in the CP community?
- How can we get faster logging, proof trimming, faster checking?