FLARE: VERIFYING MILP REFORMULATIONS WITH LLM-BASED FORMAL PROOF SYNTHESIS

Henry Robbins
Stanford University
hwr@stanford.edu

Connor Lawless
Stanford University
lawlessc@stanford.edu

Madeleine Udell
Stanford University
udell@stanford.edu

Ellen Vitercik
Stanford University
vitercik@stanford.edu

ABSTRACT

Mixed-Integer Linear Programming (MILP) is a fundamental tool for combinatorial optimization with extensive real-world applications. A central challenge is designing efficient MILP formulations. Large Language Models (LLMs) offer new opportunities to automate the modeling process, from deriving formulations to strengthening them. To ensure correctness, we need robust methods to compare formulations. However, existing approaches evaluate formulations numerically and fail to reason about general problem instances. We resolve this limitation by introducing a constructive notion of MILP *reformulation* that can be formalized in Lean and *machine-checked*. We develop FLARE[1] (Formulation-Level Automated Reformulation Evaluation), a method that uses an LLM-based agent and the Lean proof assistant to verify proposed reformulations against a reference. To evaluate our approach, we introduce *FormulationBench*, a challenging dataset of 20 problems and 116 formulations. FLARE outperforms existing methods with **96.9%** accuracy. For cases where formal guarantees are not necessary, we introduce FLARE-NL, an LLM proxy that achieves **99.7%** accuracy. These methods enable reliable verification in automated optimization modeling.[2]

Keywords Mixed Integer Linear Programming · Automated Formal Proof Synthesis · Large Language Models

1 Introduction

Mixed-Integer Linear Programming (MILP) is a fundamental tool for combinatorial optimization with applications in scheduling [17, 32], planning [48], energy [45, 31], and chip design [53]. A central challenge in MILP is problem formulation: translating a real-world scenario into a concrete mathematical model. Historically, problem formulation has required significant technical expertise. However, recent work has highlighted the ability of LLMs to translate natural-language problem descriptions into MILP formulations [63, 1, 3, 25, 9]. The primary objective is to generate MILP formulations that faithfully represent the underlying optimization problem.

Developing a faithful formulation is just the first stage in the modeling process. The same problem can often be expressed by multiple formulations which vary dramatically in solve times. In practice, experts use techniques like reformulation [56], decomposition [61], and cutting planes [43] to obtain efficient MILP formulations. LLMs offer the potential to automate this process [66, 16]. A related line of research uses LLMs for algorithm design by iteratively evolving efficient algorithms through generation and evaluation [52, 41, 67, 47]. This approach has led to advances in mathematical discovery [22], vehicle routing [24, 64] and system design [23]. However, automated optimization modeling requires verifying that a generated formulation remains faithful to the original problem. This challenge reduces to determining whether one formulation is a *reformulation* of another.

Existing approaches rely heavily on heuristics, most commonly comparing optimal objective values on a single instance [1, 25, 37]. However, such checks are unreliable and can fail under simple transformations such as the addition of cutting planes or rescaling an objective (see [68] for a discussion). Inspired by Karp reductions in complexity theory, recent work by Zhai et al. [68] introduced *EquivaMap*, which uses LLMs to discover how solutions map between formulations.

[1] The FLARE implementation is available at https://github.com/henryrobbins/flare
[2] *FormulationBench* is available at https://huggingface.co/datasets/henryrobbins/formulation-bench

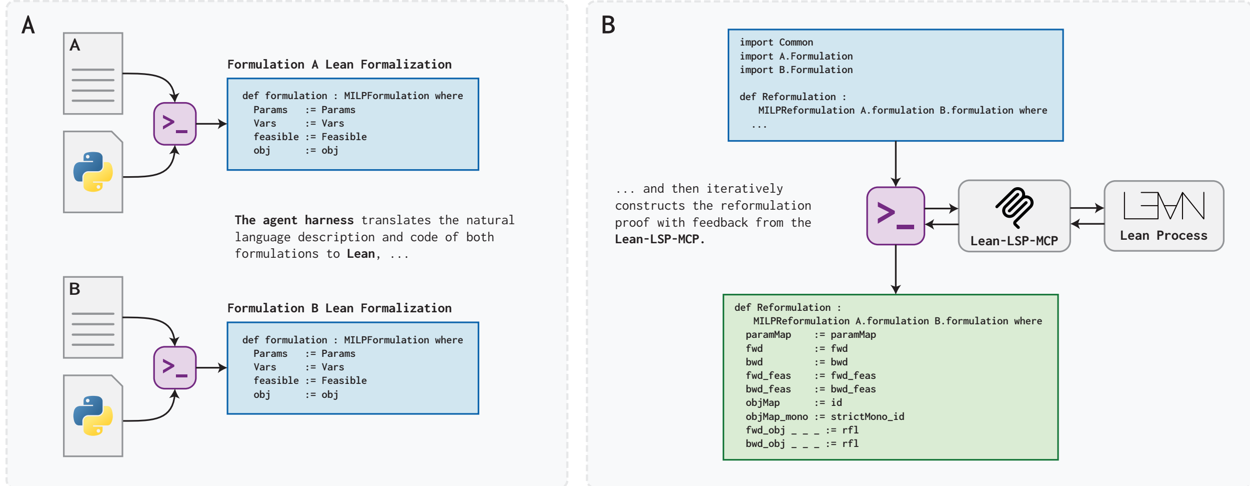

Figure 1: The FLARE workflow. (a) An agent is given natural language and Python representations of two MILP formulations and is instructed to generate Lean formalizations of both. Combined with our formalization of MILP *reformulation*, this yields a formal claim that formulation B is a reformulation of A. (b) In the second phase, the agent attempts to construct a Lean proof of that claim. The Lean-LSP-MCP allows the agent to obtain detailed feedback from the Lean process as it develops the proof.

Key Challenge. *EquivaMap* moves beyond purely heuristic methods, but its guarantees remain *instance-level*. It validates a reformulation for a specific instance (e.g., set of jobs to schedule) but not in general (e.g., any possible set of jobs to schedule). Such instance-level checks can miss formulation inconsistencies that arise on other problem instances. For example, we identify several cutting planes proposed by an LLM-based modeling framework called EvoCut [66] that pass instance-level validation but remove feasible solutions for some instances. To mitigate this risk, we seek *formulation-level* guarantees satisfied by all problem instances. Such guarantees require reasoning over *all* instances, rather than computationally evaluating a small *subset* of instances.

Our Contributions. We introduce FLARE, an automated framework for validating reformulations at the formulation-level. Our key idea is to use formal verification to make universal claims over problem instances *machine-checkable*. We utilize recent advances in automated formal proof synthesis (AFPS) to generate reformulation proofs automatically [42]. Classic notions of reformulation require reasoning about optimal solution sets and are not well-suited to AFPS. To address this challenge, we introduce a constructive definition of reformulation that requires explicit mappings between parameter spaces, feasible regions, and objective values. We formalize this definition in Lean and use an LLM-based agent for AFPS. When successful, a certificate proves a reformulation is valid for all problem instances. A formulation is treated as invalid if FLARE fails to synthesize a certificate. FLARE is the first automated approach for producing verifiable reformulation certificates, enabling trustworthy optimization modeling with LLMs.

Our contributions can be summarized as follows:

- **Constructive definition of MILP reformulation.** We introduce a constructive definition of MILP reformulation that is amenable to formal verification and AFPS, enabling the first automated method for validating reformulations at the formulation-level.

- **AI for formulation verification.** We develop FLARE, a framework that combines LLM-based auto-formalization with AFPS to generate machine-checkable reformulation certificates, and FLARE-NL, an LLM proxy that trades formal guarantees for improved speed and cost.

- **Realistic benchmark for reformulation.** We introduce *FormulationBench*, a benchmark dataset of 20 problems and 116 formulations capturing realistic modeling transformations. Empirically, we show FLARE outperforms existing methods, achieving a **96.9%** accuracy in assessing reformulation validity. FLARE-NL achieves **99.7%** accuracy, showing its value as a cheap proxy for FLARE when a machine-checkable certificate is unnecessary.

- **Demonstrated need for formal proofs.** We verify prior AI-driven MILP reformulations, revealing 5 invalid cutting planes proposed by EvoCut [66] and 3 reformulations proposed in Ferchtandiker [16]: these "equivalent" formulations are wrong or omit necessary assumptions (Appendix F).

2 Related Work

An efficient MILP formulation can decrease the solve time by orders of magnitude [11]. To improve solve time, experts apply transformations such as lifting [6], change of variables [61], and cutting planes [55] that strengthen the formulation while preserving optimal solutions (see [38] for a broader discussion). We study the complementary problem of *verifying* such reformulations, particularly in emerging settings where formulations are generated or modified by LLMs. This perspective connects three lines of research: (i) LLM-based systems that generate and refine MILP formulations, (ii) methods for verifying reformulations, and (iii) auto-formalization and automated proof synthesis.

LLMs for MILP Modeling. A growing body of work has applied LLMs to automate translating natural language into a MILP formulation. Early efforts focused on natural language processing (NLP) pipelines for structured extraction [49], while more recent *optimization copilots* use LLM-based multi-agent systems [1, 2, 46, 63, 37, 3, 15] or fine-tuning [25, 28] to handle complex, multi-constraint problems. These capabilities have been applied across domains including supply chain management [36], diagnosing infeasible models [8, 9], and scheduling [33]. Beyond formulating a *correct* model, recent work has also explored the use of LLMs to generate *stronger* MILP formulations via cutting planes [66], reformulations [16], or better solver configuration [34]. However, these approaches rely on instance-level validation. We demonstrate examples where this limitation leads to invalid cutting planes and reformulations.

Automatic Reformulation Checking Methods. Determining whether one formulation is a *reformulation* of another is central to evaluating LLM-generated MILP formulations. Existing approaches largely operate at the instance level. *Canonical accuracy* checks for a direct mapping between declarations (i.e., constraints and objectives) [49], while *execution accuracy* compares optimal objective values after solving both formulations [1, 25, 37]. More recently, *EquivaMap* [68] used LLMs to infer variable mappings between formulations and validate them by mapping optimal solutions on a specific instance. Structural approaches represent MILPs as bipartite graphs and measure similarity via graph isomorphism or edit distance [65, 54, 59, 60], avoiding the need to solve the optimization problem directly. Unlike existing approaches, FLARE verifies reformulations at the formulation-level by producing machine-checkable certificates that hold across all problem instances.

Auto-Formalization and Automated Formal Proof Synthesis. Auto-formalization translates natural language into formal representations [62, 20, 58, 40, 27], while automated formal proof synthesis (AFPS) generates machine-checkable proofs from formal statements [39, 50, 57, 10, 5, 26, 42, 51]. Recent advances in LLMs have significantly improved both tasks, with agentic frameworks combining (fine-tuned) language models and proof assistants (e.g., Lean) to formalize and solve complex mathematical problems. Most recently, simple agentic harnesses have been shown to be competitive AFPS methods [51, 42]. We adopt an LLM-based agent as the AFPS method in FLARE and introduce a constructive definition of reformulation to make proving reformulations tractable.

3 Formalizing Formulations

We begin by formally defining MILP *formulation*. Here, we make a clear distinction between a *formulation* and the parameter (or data) values that define a particular *instance* of the problem [18]. Take the traveling salesman problem (TSP) as an example. A TSP *formulation* is defined on an abstract set of cities. A TSP *instance* is one such city set. Instantiating the *formulation* with an *instance* yields a concrete MILP to be solved.

Definition 3.1 (Formulation)**.** A MILP *formulation* $\mathcal{M}$ is a tuple $\mathcal{M} = (\mathcal{P}, \mathcal{F}, f_0)$ with parameter space $\mathcal{P}$, feasible region $\mathcal{F}(p) \subseteq \mathbb{R}^{n(p)}$, and objective function f_0. For instance $p \in \mathcal{P}$, the feasible region $\mathcal{F}(p)$ is defined by $m(p)$ linear constraints, $f_i(\cdot; p) : \mathbb{R}^{n(p)} \to \mathbb{R}$ for all $i \in [m(p)]$. The first $k(p) \leq n(p)$ variables are integers. The feasible region is

$$\mathcal{F}(p) = \{x \in \mathbb{Z}^{k(p)} \times \mathbb{R}^{n(p)-k(p)} \mid f_i(x; p) \leq 0 \ \forall i \in [m(p)]\}.$$

When the instance p is clear from context, we use n, m, and k instead of the parameterized forms. The objective is to minimize the linear function $f_0(\cdot; p) : \mathbb{R}^{n(p)} \to \mathbb{R}$. A formulation $\mathcal{M}$ is *instantiated* with an *instance* $p \in \mathcal{P}$. We denote an instantiated formulation as $\mathcal{M}(p) = (\mathcal{F}(p), f_0(p))$.

Running Example. The traveling salesman problem (TSP) aims to find the shortest tour in a graph that visits every node exactly once. A TSP instance is a weighted fully-connected graph $G = (V, E, w)$ on $n = |V| \geq 2$ nodes with edge weights w_{ij}. We write $\mathcal{P}_{\text{TSP}}$ for the set of all such graphs. Two common MILP formulations for the TSP are the Dantzig-Fulkerson-Johnson (DFJ) [12] and Miller-Tucker-Zemlin (MTZ) [44] formulations.

$$\begin{aligned}
\min \quad & \sum_{(i,j)\in E} w_{ij}\, x_{ij} \\
\text{s.t.} \quad & \sum_{j\in V\setminus\{i\}} x_{ij} = 1 && \forall i \in V \\
& \sum_{i\in V\setminus\{j\}} x_{ij} = 1 && \forall j \in V \\
& \sum_{i\in S}\sum_{j\notin S} x_{ij} \leq |S| - 1 && \forall S \subset V,\ 2 \leq |S| \leq n{-}1 \\
& x_{ij} \in \{0,1\} && \forall (i,j) \in E
\end{aligned} \tag{DFJ}$$

$$\begin{aligned}
\min \quad & \sum_{(i,j)\in E} w_{ij}\, x_{ij} \\
\text{s.t.} \quad & \sum_{j\in V\setminus\{i\}} x_{ij} = 1 && \forall i \in V \\
& \sum_{i\in V\setminus\{j\}} x_{ij} = 1 && \forall j \in V \\
& u_i - u_j + n\, x_{ij} \leq n - 1 && \forall i,j \in V \setminus \{1\},\ i \neq j \\
& u_1 = 1 \\
& 2 \leq u_i \leq n && \forall i \in V \setminus \{1\} \\
& x_{ij} \in \{0,1\} && \forall (i,j) \in E
\end{aligned} \tag{MTZ}$$

Both formulations define decision variables $x_{ij} \in \{0, 1\}$ to indicate if the edge (i, j) is used in the tour. To prevent subtours, DFJ uses an exponential family of subtour-elimination constraints, while MTZ uses a polynomial family of constraints with auxiliary position variables $u_i \in \mathbb{R}$ for each node. We denote these formulations as $\mathcal{M}_{\text{DFJ}}$ and $\mathcal{M}_{\text{MTZ}}$, respectively. Both formulations faithfully represent the TSP, but have notable differences that affect solve times. $\mathcal{M}_{\text{DFJ}}$ is a *stronger* formulation than $\mathcal{M}_{\text{MTZ}}$ in that it admits fewer fractional solutions in the linear relaxation. Despite the exponential size of $\mathcal{M}_{\text{DFJ}}$, it can be solved efficiently with constraint generation.

With a formal definition of MILP formulation established, we now consider multiple formal notions of *reformulation* and introduce a constructive definition amenable to formalization and AFPS.

4 Formalizing Reformulations

Until now, we have used the term *reformulation* informally. It has an intuitive operational meaning: $\mathcal{M}'$ is a reformulation of $\mathcal{M}$ if one can map an instance $\mathcal{M}(p)$ to an instance $\mathcal{M}'(p')$, solve $\mathcal{M}'(p')$, and efficiently recover an optimal solution to $\mathcal{M}(p)$. Importantly, this is a *formulation-level* claim that holds across *all* problem instances $p \in \mathcal{P}$.

In Section 4.1, we review an existing notion of reformulation capturing this intuition and discuss why this definition is difficult to verify computationally. This limitation has led to proxies for reformulation that are easier to computationally verify but lack formulation-level guarantees (Section 4.2). Our key idea is to utilize tools from formal verification to make formulation-level reformulation claims *machine-checkable*. To this end, we propose a constructive definition of reformulation that is amenable to formalization and tractable for AFPS (Section 4.3).

4.1 Existing Definition

Audet et al. [4] capture our intuitive notion of reformulation with a complexity-theoretic definition inspired by polynomial time Turing reductions [21].

Definition 4.1 (Audet Reformulation [4])**.** Let $\mathcal{M}$ and $\mathcal{M}'$ be two formulations with parameter spaces $\mathcal{P}$ and $\mathcal{P}'$. $\mathcal{M}'$ is an *Audet reformulation* of $\mathcal{M}$ if there exists a mapping $\Phi_{\text{p}} : \mathcal{P} \to \mathcal{P}'$ such that, for any instance $p \in \mathcal{P}$: if $\mathcal{M}(p)$ has an optimal solution, then $\mathcal{M}'(\Phi_{\text{p}}(p))$ also has an optimal solution, and every optimal solution to $\mathcal{M}'(\Phi_{\text{p}}(p))$ can be mapped back to an optimal solution of $\mathcal{M}(p)$ in polynomial time.

TSP Example. Observe that $\mathcal{M}_{\text{MTZ}}$ is an *Audet reformulation* of $\mathcal{M}_{\text{DFJ}}$. The mapping Φ_{p} is the identity as both formulations use the same parameter space $\mathcal{P}_{\text{TSP}}$. Given an optimal solution of $\mathcal{M}_{\text{MTZ}}(\Phi_{\text{p}}(p))$, we can obtain an optimal solution to $\mathcal{M}_{\text{DFJ}}(p)$ by mapping x_{ij} to itself and dropping the position variables u_i. This can be done in polynomial time.

This definition captures a *formulation-level* notion of reformulation as desired. However, it is difficult to verify computationally since it quantifies over *all* instances $p \in \mathcal{P}$. Furthermore, it places a burden on AFPS to reason why an optimal solution to $\mathcal{M}'(\Phi_{\text{p}}(p))$ can recover an optimal solution to $\mathcal{M}(p)$. Our constructive definition of reformulation in Section 4.3 is designed to relieve AFPS of this proof burden.

4.2 Proxy Definitions and Limitations

To address the computational intractability of formulation-level verification, existing proxies consider reformulation at the *instance-level*. For a fixed pair of instances, a proxy verifies if $\mathcal{M}'(p')$ is a reformulation of $\mathcal{M}(p)$. The simplest proxy solves $\mathcal{M}(p)$ and $\mathcal{M}'(p')$ and compares the optimal objective values [1, 25, 37]. Zhai et al. [68] propose a stronger proxy, *Quasi-Karp equivalence*[1], inspired by Karp reductions [30] in complexity theory.

[1]Note that this definition is directional despite the *equivalence* terminology.

Definition 4.2 (Quasi-Karp Equivalence [68])**.** Let $\mathcal{M}(p)$ and $\mathcal{M}'(p')$ be two instantiated formulations. We say $\mathcal{M}'(p')$ is *Quasi-Karp equivalent* to $\mathcal{M}(p)$ if there exists an algorithm $\mathcal{A}(\mathcal{M}(p), \mathcal{M}'(p'))$ that produces a mapping f such that:

- If x^* is an optimal solution to $\mathcal{M}'(p')$, then $f(x^*)$ is an optimal solution to $\mathcal{M}(p)$,
- f can be computed in polynomial time, and
- $\mathcal{A}(\mathcal{M}(p), \mathcal{M}'(p'))$ runs in polynomial time for all $p \in \mathcal{P}$ and $p' \in \mathcal{P}'$.

Quasi-Karp equivalence is an instance-level analogue of *Audet reformulation*. Zhai et al. [68] implement this idea in *EquivaMap*, where an LLM proposes a mapping f and the mapping is checked on the solved instance. This mapping-based view is more expressive than execution accuracy, allowing *EquivaMap* to recognize arbitrary reformulations on a fixed instance, including order-preserving objective transformations. However, it still fails to verify reformulations at the formulation-level.

Pitfalls of Instance-Level Verification. The distinction between instance and formulation-level verification is critical in automated optimization modeling. A generated formulation is meant to be reused on unseen problem data. An instance-level check can validate a reformulation that is invalid at the formulation-level. This failure mode is not hypothetical. EvoCut [66] uses LLMs to propose cutting planes (cuts) for MILP formulations. A valid cut for formulation $\mathcal{M}$ removes feasible points in the linear relaxation of $\mathcal{M}$ while retaining all integer-feasible points. The EvoCut method validates candidate cuts with a simple execution-based proxy. As such, the proposed cuts may be invalid on unseen instances. The following cut proposed for the MTZ TSP formulation illustrates this risk:

$$x_{j1} + x_{ji} + (u_j - u_i - 1) \ \leq \ (n-1)(2 - x_{1i} - x_{ij}) \quad \forall i, j \in V \setminus \{1\}, \ i \neq j.$$

This cut eliminates triangles involving the first node. Let $\mathcal{M}'_{\text{MTZ}}$ be $\mathcal{M}_{\text{MTZ}}$ augmented with this cut family. For any instance $G \in \mathcal{P}_{\text{TSP}}$ with $n > 3$, every feasible tour already avoids triangles with the first node, so the cut is trivially satisfied and $\mathcal{M}'_{\text{MTZ}}(G)$ behaves identically to $\mathcal{M}_{\text{MTZ}}(G)$. However, consider the 3-node instance $G_3 \in \mathcal{P}_{\text{TSP}}$ depicted in Figure 2a with feasible tour $1 \to 2 \to 3 \to 1$. Letting $i = 2$ and $j = 3$, the cut inequality reduces to $1 \leq 0$ and does not hold. Every Hamiltonian tour on 3 nodes violates this cut and $\mathcal{M}'_{\text{MTZ}}(G_3)$ becomes infeasible, proving that this cut is invalid. This example exposes the core problem with using instance-level reformulation proxies: they are unable to validate reformulations across all admissible problem instances.

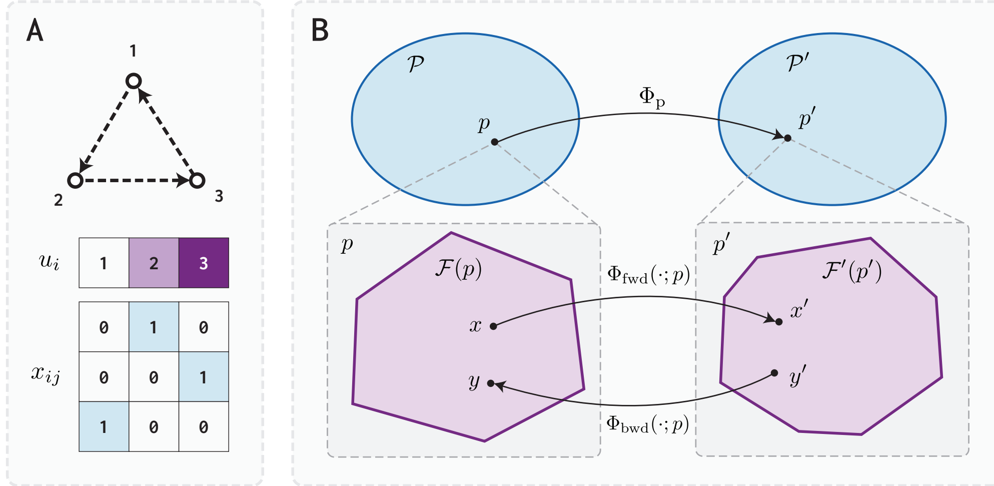

Figure 2: (a) A feasible tour $1 \to 2 \to 3 \to 1$ on a 3-node TSP instance with the MTZ variables x_{ij} and u_i. (b) A graphical depiction of the reformulation construction $\Phi(\mathcal{M}, \mathcal{M}')$. The parameter mapping Φ_{p} yields a pair of feasible regions $\mathcal{F}(p)$ and $\mathcal{F}'(p')$ related by the forward and backward maps such that objective ordering is preserved.

4.3 Constructive Definition

To overcome the pitfalls of instance-level reformulation proxies, we use formal verification to machine-check reformulations at the formulation-level. *Audet reformulation* can be formalized and machine-checked, but it requires AFPS to prove the an optimal solution of $\mathcal{M}'(\Phi_{\text{p}}(p))$ recovers an optimal solution to $\mathcal{M}(p)$. To ease the burden on AFPS, we propose a constructive definition of reformulation. It strengthens *Audet reformulation* so that a reformulation can be verified through explicit forward and backward mappings that preserve objective ordering (see Figure 2b).

Definition 4.3 (Constructive Reformulation)**.** Let $\mathcal{M} = (\mathcal{P}, \mathcal{F}, f_0)$ and $\mathcal{M}' = (\mathcal{P}', \mathcal{F}', f_0')$ be formulations. A *reformulation construction* from $\mathcal{M}$ to $\mathcal{M}'$ is a tuple $\Phi(\mathcal{M}, \mathcal{M}') = (\Phi_{\mathrm{p}},\ \Phi_{\mathrm{fwd}},\ \Phi_{\mathrm{bwd}},\ \Phi_{\mathrm{obj}})$ consisting of:

- a parameter mapping $\Phi_{\mathrm{p}} : \mathcal{P} \to \mathcal{P}'$,
- a forward mapping $\Phi_{\mathrm{fwd}}(\cdot; p) : \mathbb{R}^{n(p)} \to \mathbb{R}^{n'(\Phi_{\mathrm{p}}(p))}$,
- a backward mapping $\Phi_{\mathrm{bwd}}(\cdot; p) : \mathbb{R}^{n'(\Phi_{\mathrm{p}}(p))} \to \mathbb{R}^{n(p)}$ computable in polynomial time, [1] and
- an objective mapping $\Phi_{\mathrm{obj}} : \mathbb{R} \to \mathbb{R}$.

$\mathcal{M}'$ is a *constructive reformulation* of $\mathcal{M}$ if there exists a reformulation construction satisfying the following conditions for every instance $p \in \mathcal{P}$, with $p' = \Phi_{\mathrm{p}}(p)$:

- **Forward feasibility.** For all $x \in \mathcal{F}(p)$: $\Phi_{\mathrm{fwd}}(x; p) \in \mathcal{F}'(p')$.
- **Backward feasibility.** For all $x' \in \mathcal{F}'(p')$: $\Phi_{\mathrm{bwd}}(x'; p) \in \mathcal{F}(p)$.
- **Strictly monotone objective mapping.** Φ_{obj} is strictly monotonically increasing.
- **Internal consistency.** (1) For all $x \in \mathcal{F}(p)$, $f_0'(\Phi_{\mathrm{fwd}}(x; p); p') = \Phi_{\mathrm{obj}}(f_0(x; p))$, and (2) for all $x' \in \mathcal{F}'(p')$, $f_0'(x'; p') = \Phi_{\mathrm{obj}}(f_0(\Phi_{\mathrm{bwd}}(x'; p); p))$.

We say $\mathcal{M}'(p')$ is a *constructive reformulation* of $\mathcal{M}(p)$ if the conditions hold for some instance $p \in \mathcal{P}$.

Like *Audet reformulation*, this definition is only meaningful when computing the optimal solution to both formulations is NP-hard and both formulations admit a feasible solution for at least one parameter. This definition is chosen to ensure our notion of reformulation captures common MILP transformations (e.g., lifting, rescaling, and substitution) and modeling choices (e.g., exponential constraints like the TSP DFJ formulation). Our definition is similar to the definition proposed by Sherali which asserts a much stronger order-preserving bijective mapping between the two feasible regions that excludes common transformations (e.g., lifting) [38].

We prove a *constructive reformulation* is an *Audet reformulation* and, as an immediate corollary, the instantiated claim also implies *Quasi-Karp equivalence*. We defer the proof to Appendix A.

Proposition 4.1. *Let $\mathcal{M}$ and $\mathcal{M}'$ be formulations. If $\mathcal{M}'$ is a constructive reformulation of $\mathcal{M}$, then $\mathcal{M}'$ is an Audet reformulation of $\mathcal{M}$. Equivalently, for any instance $p \in \mathcal{P}$ with $p' = \Phi_{\mathrm{p}}(p)$, if $\mathcal{M}'(p')$ is a constructive reformulation of $\mathcal{M}(p)$, then $\mathcal{M}'(p')$ is Quasi-Karp equivalent to $\mathcal{M}(p)$.*

Our definition is stronger than *Audet* because it maps *all* feasible points, not only optima. This stronger notion of reformulation provides a more structured claim that reduces the proof burden on AFPS while remaining satisfied by common MILP transformations. For the remainder of the paper, reformulation means *constructive reformulation*.

5 Methodology

This section operationalizes our constructive reformulation definition into an automated verification pipeline. We first formalize our definitions in Lean and then introduce `FLARE`, which uses AFPS to generate machine-checkable reformulation certificates. Finally, we introduce `FLARE-NL`, a fast and cheap LLM proxy that reasons about reformulations using only natural language.

Formalization. To make our reformulation definition machine-checkable, we formalize our definition in the proof assistant Lean [13], chosen for its mature Mathlib library and extensive AFPS tooling (see Appendix C). We provide Lean formalizations of *formulation* (Definition 3.1) and *reformulation* (Definition 4.3) in Appendix B. We assume the input formulations are well-formed MILPs (i.e., have linear constraints and objective, and are defined over real-valued decision variables). Accordingly, our Lean formalization does not explicitly encode these conditions. The only part of Definition 4.3 we omit from the Lean proof obligation is the polynomial-time requirement on Φ_{bwd}. Formalizing computational complexity in Lean would add substantial proof burden, and in our experiments, we never observe a case where the synthesized backward mapping is not computable in polynomial time.

`FLARE`. `FLARE` determines if $\mathcal{M}'$ is a reformulation of $\mathcal{M}$ in the following stages: (1) auto-formalize $\mathcal{M}$ and $\mathcal{M}'$ into Lean, (2) attempt to prove $\mathcal{M}'$ is a reformulation of $\mathcal{M}$ using AFPS, and (3) check if a reformulation certificate was successfully generated by AFPS. In the first two stages, we utilize the Claude Code[2] agent harness for auto-formalization and AFPS. Existing AFPS methods, like the Numina-Lean-Agent [42], demonstrate the use of Claude

[1] Polynomial time is measured under fixed binary encodings of parameters and rational/integer assignments: Φ_{bwd} is polynomial-time computable if a single deterministic algorithm, given p, $\Phi_{\mathrm{p}}(p)$, and x', outputs $\Phi_{\mathrm{bwd}}(x'; p)$ in time polynomial in the total input bit length. This does not impose a polynomial-time requirement on Φ_{p}.

[2] Alternative agent harnesses are considered in Appendix H.

Code as a competitive AFPS method when combined with the Lean-LSP-MCP [7, 42, 29]. We further augment Claude Code with custom agent skills for handling MILP formulations in Lean. In the final stage, we check if a proof was generated and machine-check it with Lean. This workflow is summarized in Figure 1 and additional implementation details can be found in Appendix C.

Limitations. FLARE's guarantee is conditional on faithful formalization. If the auto-formalized formulations are correct, then a Lean-verified certificate proves the reformulation claim under our definition. However, failure to generate a certificate is inconclusive. We highlight three limitations.

- **No certificate of invalidity.** If $\mathcal{M}'$ is not a reformulation of $\mathcal{M}$, FLARE does not prove invalidity. The AFPS component simply fails to produce a reformulation certificate.
- **Dependence on faithful formalization.** If the auto-formalized MILP formulations are unfaithful to $\mathcal{M}$ and $\mathcal{M}'$, FLARE may certify the wrong claim. We only observe this failure mode by GPT 5.5 across 96 formulation pairs in our experiments (Appendix H), and it can be mitigated by deterministically translating from modeling standards such as AMPL [18] or MathOptInterface [35].
- **False negatives from proof synthesis.** If the LLM is unable to construct an appropriate reformulation construction or the proof requires extensive work in Lean to formalize, the agent may exit early, producing a false-negative. Ongoing efforts to formalize computer science foundations in Lean with CSLib may reduce this burden on AFPS.

FLARE-NL. Generating a reformulation certificate using FLARE can take 5-10 minutes. As a fast and cheap proxy, FLARE-NL prompts a frontier reasoning model with natural language descriptions of two formulations, $\mathcal{M}$ and $\mathcal{M}'$, and asks if $\mathcal{M}'$ is a reformulation of $\mathcal{M}$ (see Appendix D for full prompt). FLARE-NL makes three changes to the *naive-LLM* baseline of Zhai et al. [68]: (1) the prompt includes our formal definition of reformulation, (2) the prompt explicitly states all formulation assumptions and instructs not to introduce new ones, and (3) we utilize reasoning and structured output. Unlike FLARE, FLARE-NL lacks any verifiable guarantees on its output. It is useful for estimating whether a certificate likely exists, but it is not a proof method.

6 Experiments

To evaluate if formulation-level verification catches failures missed by instance-level proxies, we introduce *FormulationBench*, an extension of the *EquivaFormulation* dataset [68] saturated by *EquivaMap*. We evaluate FLARE and FLARE-NL against established baselines and conduct an ablation study on FLARE-NL.[1] Additional experimental details are provided in Appendix G.

6.1 FormulationBench

We introduce *FormulationBench*, a benchmark designed to test formulation-level reformulation reasoning. *FormulationBench* extends *EquivaFormulation* [68] with EvoCut cutting plane proposals [66] and a collection of eight MILP formulation pairs provided by Ferchtandiker [16]. These formulation pairs are more challenging than those in *EquivaFormulation*, requiring reasoning about general cutting plane families and meaningfully different modeling techniques. The dataset contains 20 problems, 116 formulations, and 96 formulation pairs with 70 positive and 26 negative examples (Appendix E). *FormulationBench* is available on Hugging Face.[2]

FormulationBench is organized similarly to the NLP4LP dataset [1]; JSON files contain formulation descriptions and problem instance data. However, *FormulationBench* introduces two notable extensions. First, since formal proofs of reformulation validity often rely on assumptions that were implicit in the source dataset, we identify and record these assumptions. We flag these assumptions as implicit to facilitate our ablation study in Section 6.3. Second, we formalize each formulation in Lean (see Appendix B) and include ground-truth reformulation certificates for each valid reformulation pair. In the process of preparing this dataset, we verified prior AI-driven MILP reformulations and found 5 invalid cutting planes proposed by EvoCut [66] and 3 invalid reformulations proposed in Ferchtandiker [16] (see Appendix F).

6.2 Baseline Comparison

We compare FLARE and FLARE-NL against the baselines in Zhai et al. [68]. We exclude graph-related methods due to their poor performance on non-trivial transformations. Notably, we include implicit assumptions in the prompts of all LLM-based methods.

[1] Experimental code is available at https://github.com/henryrobbins/flare

[2] *FormulationBench* is available at https://huggingface.co/datasets/henryrobbins/formulation-bench

- **Execution [1].** Compares the optimal objective value of two formulation on a specific instance.
- **EquivaMap [68].** We re-implement *EquivaMap* to allow for mappings over non-scalar variables. The original mapping prompt is unmodified and we use Opus 4.7 as the mapping LLM.

Performance. Table 1 summarizes the main results. Our methods outperform all existing baselines. FLARE achieves **96.9%** accuracy and is the only method that generates machine-checkable reformulation certificates. Among unverified methods, FLARE-NL outperforms *EquivaMap* with **99.7%** accuracy. We conduct an ablation study on to understand the core components of this improvement (Section 6.3). Despite its stronger instance-level formalism, *EquivaMap* underperforms compared to our methods. *EquivaMap*'s errors reflect limitations of its instance-level mapping framework and failures on non-linear solution maps[•]; see Table 2.

hr deleted here

Table 1: Accuracy of automated reformulation checking methods on the *FormulationBench* dataset. **Bold** indicates the best method on a metric and underlined indicates the second best. Metric cells report mean ± std. dev. across 3 runs. All LLM-based methods use Opus 4.7 with reasoning and high effort level. Due to the high cost, we only do a single run of FLARE. FLARE and FLARE-NL outperform existing methods and FLARE is the only method that generates a machine-checkable certificate.

Method	Model	Certificate	Runs	TP	FP	TN	FN	Precision	Recall	Accuracy	Avg. Time	Avg. Cost
Execution [1]	—	✗	3	195	45	33	15	81.2%±0.0%	92.9%±0.0%	79.2%±0.0%	2.3s	—
EquivaMap [68]	Opus 4.7	✗	3	195	30	48	15	86.7%±0.0%	92.9%±0.0%	84.4%±0.0%	7.7s	$0.042
FLARE	Opus 4.7	✓	1	67	0	26	3	**100.0%**	<u>95.7%</u>	<u>96.9%</u>	561.6s	$2.334
FLARE-NL	Opus 4.7	✗	3	210	1	77	0	<u>99.5%</u>±0.8%	**100.0%**±0.0%	**99.7%**±0.6%	12.9s	$0.043

Perils of Instance-Level Validation. Table 2 summarizes errors by transformation type. Existing instance-level methods fail to catch formulation-level modeling errors, such as transformations (1) and (2), where a transformation can appear valid on the tested instance while failing as a general reformulation. *EquivaMap*'s solution-mapping approach handles transformations (3) and (4), showing its advantage over the *execution* heuristic, but it is unable to certify validity for non-linear reformulations. In contrast, FLARE's formulation-level guarantees eliminate these false positives.

Limits of AFPS. The gap between FLARE and FLARE-NL reflects the current limits of AFPS. FLARE fails to construct certificates for three reformulations from Ferchtandiker [16]. All three reformulation proofs rely on the flow decomposition lemma which states a flow can be decomposed into a collection of paths and cycles. This is non-trivial to formalize in Lean and is not available in standard libraries. However, this is standard computer science result that will be formalized as Lean libraries improve (e.g., CSLib). Once formalized, FLARE can simply invoke the definition.

Table 2: Accuracy of automated reformulation checking methods segmented by challenging transformation types in the *Formulation-Bench* dataset. The *Valid* column indicates if the listed type transformation results in a valid reformulation.

Transformation	Valid	Execution [1]	EquivaMap [68]	FLARE	FLARE-NL
1. Base-10 Representation	✗	0%	0%	**100%**	**100%**
2. Addition of Invalid Cutting Planes	✗	0%	0%	**100%**	<u>93.3%</u>
3. Rescaled Objective	✓	0%	**100%**	**100%**	**100%**
4. Different Formulation (Same Objective)	✗	0%	**100%**	**100%**	**100%**
5. Non-Linear Solution Maps	✓	**100%**	0%	40%	**100%**
Worst Case		0%	0%	<u>40%</u>	**93.3%**

6.3 FLARE-NL Ablation Study

In Table 3, we ablate the FLARE-NL prompt to measure the importance of the reformulation definition and assumption handling across four LLMs. The *Baseline* uses the full prompt; *No Definition* omits the definition of reformulation; and *Allow Implicit* removes implicit assumptions from the prompt and permits the model to introduce reasonable assumptions. See Appendix D for prompt variants and Appendix G for additional details.

Explicit Definitions and Assumptions. The full FLARE-NL prompt performs the best across all model families, indicating that reformulation verification relies on explicit definitions and assumptions. Removing the reformulation definition consistently reduces accuracy, and allowing the model to reason about implicit assumptions causes the largest degradation for most model families. Verifying reformulations requires explicit criteria and assumptions, rather than model-inferred assumptions.

Frontier Model Reasoning. FLARE-NL is most effective when paired with strong frontier reasoning models (also see Appendix H). Under the full prompt, Opus 4.7, GPT-5.5, and DeepSeek V4 Pro all achieve high accuracy, while GPT-4.1 remains recall-limited despite the same instructions.

Table 3: Ablation study of FLARE-NL across LLM model and prompt variations. *No Definition* omits the reformulation definition from the prompt and *Allow Implicit* removes implicit assumptions from the prompt and permits the model to introduce reasonable assumptions. Metric cells report mean ± std. dev. across 3 runs. All models have reasoning enabled, expect for GPT-4.1 which does not support reasoning. The reasoning effort level is shown following the model name.

Model (Effort)	Variant	TP	FP	TN	FN	Precision	Recall	Accuracy	Avg Time	Avg Cost
Opus 4.7 *(high)*	Baseline	210	1	77	0	99.5±0.8%	**100.0±0.0%**	**99.7±0.6%**	12.9s	$0.043
	No Definition	198	6	72	12	97.1±0.0%	94.3±1.4%	93.8±1.0%	11.1s	$0.034
	Allow Implicit	201	8	70	9	96.2±2.2%	95.7±1.4%	94.1±2.6%	14.3s	$0.042
GPT-5.5 *(medium)*	Baseline	204	0	78	6	**100.0±0.0%**	97.1±1.4%	97.9±1.0%	30.7s	$0.058
	No Definition	191	0	78	19	**100.0±0.0%**	91.0±0.8%	93.4±0.6%	20.9s	$0.041
	Allow Implicit	138	8	70	72	94.5±1.0%	65.7±1.4%	72.2±0.6%	46.2s	$0.077
DeepSeek V4 Pro *(high)*	Baseline	203	3	74	7	98.6±2.5%	96.7±0.8%	96.5±2.4%	75.9s	$0.014
	No Definition	198	3	75	11	98.5±1.5%	94.7±2.2%	95.1±2.2%	56.7s	$0.011
	Allow Implicit	179	10	66	30	94.7±0.8%	85.6±2.6%	86.0±1.7%	84.6s	$0.015
GPT-4.1 *(none)*	Baseline	122	4	74	88	97.0±3.4%	58.1±1.6%	68.1±0.6%	7.1s	$0.010
	No Definition	102	4	74	108	96.4±4.1%	48.6±2.9%	61.1±2.2%	<u>5.7s</u>	<u>$0.007</u>
	Allow Implicit	111	5	73	99	95.7±3.9%	52.9±2.5%	63.9±3.3%	7.9s	$0.009

7 Conclusion

LLMs are increasingly used for optimization modeling but lack formal guarantees, necessitating methods to automatically verify LLM-generated formulations. We develop FLARE, the first automated approach for producing machine-checkable reformulation certificates. FLARE combines a constructive definition of reformulation and AFPS to verify reformulation claims at the *formulation-level* while existing methods only make *instance-level* claims. Furthermore, we introduce FLARE-NL, an LLM proxy that trades formal guarantees for improved speed and cost. Both methods outperform existing baselines and achieve high accuracy on the *FormulationBench* dataset.

Limitations and Future Work. FLARE's guarantees depend on faithful formalization. If the Lean formalization of a formulation is unfaithful to its description, the certificate proves the wrong claim. Additionally, failure to generate a certificate is inconclusive, motivating future work on certificates of invalidity, such as generating valid counterexamples. Beyond these limitations, this proof-synthesis perspective may also extend beyond reformulation validity to certifying formulation strength, such as proving that one formulation yields a tighter linear relaxation. Our results also raise questions about how to integrate verification into LLM-driven modeling and automated heuristic design pipelines (e.g., EvoCut [66]). The combination of FLARE's machine-checkable certificates with FLARE-NL's low latency and high accuracy are a potentially powerful combination.

References

[1] Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. OptiMUS: Scalable optimization modeling with (MI)LP solvers and large language models. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[2] Ali AhmadiTeshnizi, Wenzhi Gao, Herman Brunborg, Shayan Talaei, Connor Lawless, and Madeleine Udell. OptiMUS-0.3: Using Large Language Models to Model and Solve Optimization Problems at Scale. *arXiv preprint arXiv:2407.19633*, 2025.

[3] Nicolas Astorg, Tennison Liu, Yuanzhang Xiao, and Mihaela van der Schaar. Autoformulation of mathematical optimization models using LLMs. *Proceedings of Machine Learning Research*, 2025.

[4] C. Audet, P. Hansen, B. Jaumard, and G. Savard. Links Between Linear Bilevel and Mixed 0–1 Programming Problems. *Journal of Optimization Theory and Applications*, 93(2):273–300, May 1997. ISSN 1573-2878. doi: 10.1023/A:1022645805569.

[5] Axiom. From Seeing Why to Checking Everything, 2026.

[6] Egon Balas. Disjunctive programming. *Annals of discrete mathematics*, 5:3–51, 1979.

[7] Benjamin Breen, Marco Del Tredici, Jacob McCarran, Javier Aspuru Mijares, Weichen Winston Yin, Kfir Sulimany, Jacob M. Taylor, Frank H. L. Koppens, and Dirk Englund. Ax-Prover: A Deep Reasoning Agentic Framework for Theorem Proving in Mathematics and Quantum Physics. *arXiv preprint arXiv:2510.12787*, 2025.

[8] Hao Chen, Gonzalo E Constante-Flores, and Can Li. Diagnosing infeasible optimization problems using large language models. *INFOR: Information Systems and Operational Research*, 62(4):573–587, 2024.

[9] Hao Chen, Gonzalo Esteban Constante-Flores, Krishna Sri Ipsit Mantri, Sai Madhukiran Kompalli, Akshdeep Singh Ahluwalia, and Can Li. OptiChat: Bridging optimization models and practitioners with large language models. *INFORMS Journal on Data Science*, 2025.

[10] Jiangjie Chen, Wenxiang Chen, Jiacheng Du, Jinyi Hu, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Wenlei Shi, Zhihong Wang, Mingxuan Wang, Chenrui Wei, Shufa Wei, Huajian Xin, Fan Yang, Weihao Gao, Zheng Yuan, Tianyang Zhan, Zeyu Zheng, Tianxi Zhou, and Thomas Hanwen Zhu. Seed-Prover 1.5: Mastering Undergraduate-Level Theorem Proving via Learning from Experience. *arXiv preprint arXiv:2512.17260*, 2025.

[11] Michele Conforti, Gérard Cornuéjols, and Giacomo Zambelli. Integer programming models. In *Integer Programming*, pages 45–84. Springer, 2014.

[12] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a Large-Scale Traveling-Salesman Problem. *Journal of the Operations Research Society of America*, 2(4):393–410, 1954. ISSN 0096-3984.

[13] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing. ISBN 978-3-030-79876-5. doi: 10.1007/978-3-030-79876-5_37.

[14] Oliver Dressler. Lean LSP MCP: Tools for agentic interaction with the lean theorem prover, March 2025.

[15] Joshua Drossman, Alexandre Jacquillat, and Sébastien Martin. Let's have a conversation: Designing and evaluating LLM agents for interactive optimization. *arXiv preprint arXiv:2604.02666*, 2026.

[16] Nathan Ferchtandiker. Generating Efficient Optimization Formulations Using Large Language Models. Master's thesis, Universiteit van Amsterdam, July 2025.

[17] Christodoulos A. Floudas and Xiaoxia Lin. Mixed Integer Linear Programming in Process Scheduling: Modeling, Algorithms, and Applications. *Annals of Operations Research*, 139(1):131–162, October 2005. ISSN 1572-9338. doi: 10.1007/s10479-005-3446-x.

[18] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Thomson/Brooks/Cole, Pacific Grove, CA, 2nd ed edition, 2003. ISBN 978-0-534-38809-6.

[19] Cameron Freer. Lean 4 Skills: Theorem proving skill and workflow pack for AI coding agents, October 2025.

[20] Guoxiong Gao, Yutong Wang, Jiedong Jiang, Qi Gao, Zihan Qin, Tianyi Xu, and Bin Dong. Herald: A natural language annotated Lean 4 dataset. In *The Thirteenth International Conference on Learning Representations*, 2025.

[21] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. A Series of Books in the Mathematical Sciences. Freeman, New York [u.a], 27. print edition, 2009. ISBN 978-0-7167-1044-8 978-0-7167-1045-5.

[22] Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. Mathematical exploration and discovery at scale. *arXiv preprint arXiv:2511.02864*, 2025.

[23] Pouya Hamadanian, Pantea Karimi, Arash Nasr-Esfahany, Kimia Noorbakhsh, Joseph Chandler, Ali ParandehGheibi, Mohammad Alizadeh, and Hari Balakrishnan. Glia: A Human-Inspired AI for Automated Systems Design and Optimization. *arXiv preprint arXiv:2510.27176*, 2025.

[24] André Hottung, Federico Berto, Chuanbo Hua, Nayeli Gast Zepeda, Daniel Wetzel, Michael Römer, Haoran Ye, Davide Zago, Michael Poli, Stefano Massaroli, Jinkyoo Park, and Kevin Tierney. VRPAgent: LLM-Driven Discovery of Heuristic Operators for Vehicle Routing Problems. *arXiv preprint arXiv:2510.07073*, 2025.

[25] Chenyu Huang, Zhengyang Tang, Shixi Hu, Ruoqing Jiang, Xin Zheng, Dongdong Ge, Benyou Wang, and Zizhuo Wang. ORLM: A customizable framework in training large models for automated optimization modeling. *Operations Research*, 73(6): 2986–3009, 2025.

[26] Logical Intelligence. Aleph Prover: State-of-the-Art Formal Theorem Prover, January 2026.

[27] Prithwish Jana, Kaan Kale, Ahmet Ege Tanriverdi, Cruise Song, Sriram Vishwanath, and Vijay Ganesh. ProofBridge: Autoformalization of natural language proofs in lean via joint embeddings. In *The Fourteenth International Conference on Learning Representations*, 2026.

[28] Caigao JIANG, Xiang Shu, Hong Qian, Xingyu Lu, JUN ZHOU, Aimin Zhou, and Yang Yu. LLMOPT: Learning to define and solve general optimization problems from scratch. In *The Thirteenth International Conference on Learning Representations*, 2025.

[29] Haocheng Ju, Guoxiong Gao, Jiedong Jiang, Bin Wu, Zeming Sun, Leheng Chen, Yutong Wang, Yuefeng Wang, Zichen Wang, Wanyi He, Peihao Wu, Liang Xiao, Ruochuan Liu, Bryan Dai, and Bin Dong. Automated Conjecture Resolution with Formal Verification. *arXiv preprint arXiv:2604.03789*, 2026.

[30] Richard M. Karp. Reducibility among Combinatorial Problems. In Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger, editors, *Complexity of Computer Computations: Proceedings of a Symposium on the Complexity of Computer Computations, Held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and Sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*, pages 85–103. Springer US, Boston, MA, 1972. ISBN 978-1-4684-2001-2. doi: 10.1007/978-1-4684-2001-2_9.

[31] Bernard Knueven, James Ostrowski, and Jean-Paul Watson. On Mixed-Integer Programming Formulations for the Unit Commitment Problem. *INFORMS Journal on Computing*, 32(4):857–876, October 2020. ISSN 1091-9856. doi: 10.1287/ijoc.2019.0944.

[32] Wen-Yang Ku and J. Christopher Beck. Mixed Integer Programming models for job shop scheduling: A computational analysis. *Computers & Operations Research*, 73:165–173, September 2016. ISSN 0305-0548. doi: 10.1016/j.cor.2016.04.006.

[33] Connor Lawless, Jakob Schoeffer, Lindy Le, Kael Rowan, Shilad Sen, Cristina St. Hill, Jina Suh, and Bahareh Sarrafzadeh. "I Want It That Way": Enabling interactive decision support using large language models and constraint programming. *ACM Transactions on Interactive Intelligent Systems*, 14(3):1–33, 2024.

[34] Connor Lawless, Yingxi Li, Anders Wikum, Madeleine Udell, and Ellen Vitercik. LLMs for cold-start cutting plane separator configuration. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 51–69. Springer, 2025.

[35] Benoît Legat, Oscar Dowson, Joaquim Dias Garcia, and Miles Lubin. MathOptInterface: A data structure for mathematical optimization problems. *INFORMS Journal on Computing*, 2021. doi: 10.1287/ijoc.2021.1067.

[36] Beibin Li, Konstantina Mellou, Bo Zhang, Jeevan Pathuri, and Ishai Menache. Large Language Models for Supply Chain Optimization. *arXiv preprint arXiv:2307.03875*, 2023.

[37] Kuo Liang, Yuhang Lu, Jianming Mao, Shuyi Sun, Chunwei Yang, Congcong Zeng, Xiao Jin, Hanzhang Qin, Ruihao Zhu, and Chung-Piaw Teo. LLM for large-scale optimization model auto-formulation: A lightweight few-shot learning approach. *arXiv preprint arXiv:2601.09635*, 2026.

[38] Leo Liberti. Reformulations in Mathematical Programming: Definitions and Systematics. *RAIRO - Operations Research*, 43 (1):55–85, January 2009. ISSN 0399-0559, 1290-3868. doi: 10.1051/ro/2009005.

[39] Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia LI, Mengzhou Xia, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-Prover: A frontier model for open-source automated theorem proving. In *Second Conference on Language Modeling*, 2025.

[40] Fan Liu, Zhe-Rui Yang, Cancheng Liu, Tianrui SONG, Xiaofeng Gao, and Hao Liu. MM-Agent: LLM as agents for real-world mathematical modeling problem. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

[41] Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of Heuristics: Towards efficient automatic algorithm design using large language model. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[42] Junqi Liu, Zihao Zhou, Zekai Zhu, Marco Dos Santos, Weikun He, Jiawei Liu, Ran Wang, Yunzhou Xie, Junqiao Zhao, Qiufeng Wang, Lihong Zhi, Jia Li, and Wenda Li. Numina-Lean-Agent: An Open and General Agentic Reasoning System for Formal Mathematics. *arXiv preprint arXiv:2601.14027*, 2026.

[43] Hugues Marchand, Alexander Martin, Robert Weismantel, and Laurence Wolsey. Cutting planes in integer and mixed integer programming. *Discrete Applied Mathematics*, 123(1):397–446, November 2002. ISSN 0166-218X. doi: 10.1016/S0166-218X(01)00348-1.

[44] C. E. Miller, A. W. Tucker, and R. A. Zemlin. Integer Programming Formulation of Traveling Salesman Problems. *J. ACM*, 7 (4):326–329, October 1960. ISSN 0004-5411. doi: 10.1145/321043.321046.

[45] Hugo Morais, Péter Kádár, Pedro Faria, Zita A. Vale, and H. M. Khodr. Optimal scheduling of a renewable micro-grid in an isolated load area using mixed-integer linear programming. *Renewable Energy*, 35(1):151–156, January 2010. ISSN 0960-1481. doi: 10.1016/j.renene.2009.02.031.

[46] Mahdi Mostajabdaveh, Timothy T Yu, Rindranirina Ramamonjison, Giuseppe Carenini, Zirui Zhou, and Yong Zhang. Optimization modeling and verification from problem specifications using a multi-agent multi-stage LLM framework. *INFOR: Information Systems and Operational Research*, 62(4):599–617, 2024.

[47] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

[48] Yves Pochet and Laurence A Wolsey. *Production Planning by Mixed Integer Programming*. Springer, 2006.

[49] Rindranirina Ramamonjison, Timothy Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, et al. NL4Opt competition: Formulating optimization problems based on their natural language descriptions. In *NeurIPS 2022 Competition Track*, pages 189–203. PMLR, 2023.

[50] Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. DeepSeek-Prover-V2: Advancing Formal Mathematical Reasoning via Reinforcement Learning for Subgoal Decomposition. *arXiv preprint arXiv:2504.21801*, 2025.

[51] Borja Requena, Austin Letson, Krystian Nowakowski, Izan Beltran Ferreiro, and Leopoldo Sarra. A Minimal Agent for Automated Theorem Proving. *arXiv preprint arXiv:2602.24273*, 2026.

[52] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, January 2024. ISSN 0028-0836, 1476-4687. doi: 10.1038/s41586-023-06924-6.

[53] K. Srinivasan, K.S. Chatha, and G. Konjevod. Linear-programming-based techniques for synthesis of network-on-chip architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(4):407–420, April 2006. ISSN 1557-9999. doi: 10.1109/TVLSI.2006.871762.

[54] Zachary Steever, Kyle Hunt, Mark Karwan, Junsong Yuan, and Chase C Murray. A graph-based approach for relating integer programs. *INFORMS Journal on Computing*, 36(6):1715–1736, 2024.

[55] Tony J Van Roy and Laurence A Wolsey. Solving mixed integer programming problems using automatic reformulation. *Operations Research*, 35(1):45–57, 1987.

[56] François Vanderbeck and Laurence A. Wolsey. Reformulation and Decomposition of Integer Programs. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008*, pages 431–502. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-540-68274-5 978-3-540-68279-0. doi: 10.1007/978-3-540-68279-0_13.

[57] Sumanth Varambally, Thomas Voice, Yanchao Sun, Zhifeng Chen, Rose Yu, and Ke Ye. Hilbert: Recursively building formal proofs with informal reasoning. In *The Fourteenth International Conference on Learning Representations*, 2026.

[58] Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, Jianqiao Lu, Hugues de Saxcé, Bolton Bailey, Chendong Song, Chenjun Xiao, Dehao Zhang, Ebony Zhang, Frederick Pu, Han Zhu, Jiawei Liu, Jonas Bayer, Julien Michel, Longhui Yu, Léo Dreyfus-Schmidt, Lewis Tunstall, Luigi Pagani, Moreira Machado, Pauline Bourigault, Ran Wang, Stanislas Polu, Thibaut Barroyer, Wen-Ding Li, Yazhe Niu, Yann Fleureau, Yangyang Hu, Zhouliang Yu, Zihan Wang, Zhilin Yang, Zhengying Liu, and Jia Li. Kimina-Prover Preview: Towards Large Formal Reasoning Models with Reinforcement Learning. *arXiv preprint arXiv:2504.11354*, 2025.

[59] Zhuohan Wang, Ziwei Zhu, Yizhou Han, Yufeng Lin, Zhihang Lin, Ruoyu Sun, and Tian Ding. OptiBench: Benchmarking large language models in optimization modeling with equivalence-detection evaluation, 2024.

[60] Zhuohan Wang, Ziwei Zhu, Ziniu Li, Congliang Chen, Yizhou Han, Yufeng Lin, Zhihang Lin, Angyang Gu, Xinglin Hu, Ruoyu Sun, et al. ORGEval: Graph-theoretic evaluation of LLMs in optimization modeling. *arXiv preprint arXiv:2510.27610*, 2025.

[61] Laurence A Wolsey. *Integer Programming*. John Wiley & Sons, 2020.

[62] Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

[63] Ziyang Xiao, Dongxiang Zhang, Yangjun Wu, Lilin Xu, Yuan Jessica Wang, Xiongwei Han, Xiaojin Fu, Tao Zhong, Jia Zeng, Mingli Song, and Gang Chen. Chain-of-Experts: When LLMs Meet Complex Operations Research Problems. In *The Twelfth International Conference on Learning Representations*, 2024.

[64] Zhuoliang Xie, Fei Liu, Zhenkun Wang, and Qingfu Zhang. Enhancing CVRP Solver through LLM-driven Automatic Heuristic Design. *arXiv preprint arXiv:2602.23092*, 2026.

[65] Linzi Xing, Xinglu Wang, Yuxi Feng, Zhenan Fan, Jing Xiong, Zhijiang Guo, Xiaojin Fu, Rindra Ramamonjison, Mahdi Mostajabdaveh, Xiongwei Han, Zirui Zhou, and Yong Zhang. Towards Human-aligned Evaluation for Linear Programming Word Problems. In Nicoletta Calzolari, Min-Yen Kan, Veronique Hoste, Alessandro Lenci, Sakriani Sakti, and Nianwen Xue, editors, *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 16550–16556, Torino, Italia, May 2024. ELRA and ICCL.

[66] Milad Yazdani, Mahdi Mostajabdaveh, Samin Aref, and Zirui Zhou. EvoCut: Strengthening Integer Programs via Evolution-Guided Language Models. *arXiv preprint arXiv:2508.11850*, 2025.

[67] Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. ReEvo: Large language models as hyper-heuristics with reflective evolution. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[68] Haotian Zhai, Connor Lawless, Ellen Vitercik, and Liu Leqi. EquivaMap: Leveraging LLMs for automatic equivalence checking of optimization formulations. In *Forty-second International Conference on Machine Learning*, 2025.

A Proof of Proposition 4.1

Proposition 4.1. Let $\mathcal{M}$ and $\mathcal{M}'$ be formulations. If $\mathcal{M}'$ is a *constructive reformulation* of $\mathcal{M}$, then $\mathcal{M}'$ is an *Audet reformulation* of $\mathcal{M}$. Equivalently, for any instance $p \in \mathcal{P}$ with $p' = \Phi_{\text{p}}(p)$, if $\mathcal{M}'(p')$ is a *constructive reformulation* of $\mathcal{M}(p)$, then $\mathcal{M}'(p')$ is *Quasi-Karp equivalent* to $\mathcal{M}(p)$.

Proof. Let $\mathcal{M} = (\mathcal{P}, \mathcal{F}, f_0)$ and $\mathcal{M}' = (\mathcal{P}', \mathcal{F}', f_0')$ be formulations. Assume $\mathcal{M}'(p')$ is a *constructive reformulation* of $\mathcal{M}(p)$. By definition, there exists a *reformulation construction* $\Phi(\mathcal{M}, \mathcal{M}') = (\Phi_{\text{p}}, \Phi_{\text{fwd}}, \Phi_{\text{bwd}}, \Phi_{\text{obj}})$ such that the conditions of Definition 4.3 hold. Fix an instance $p \in \mathcal{P}$ and let $p' = \Phi_p(p)$.

We first show that if $\mathcal{M}(p)$ has an optimal solution, then $\mathcal{M}'(p')$ also has an optimal solution. Let $x^* \in \mathcal{F}(p)$ be optimal for $\mathcal{M}(p)$. By forward feasibility, $x' = \Phi_{\text{fwd}}(x^*; p) \in \mathcal{F}'(p')$. We now claim that x' is optimal for $\mathcal{M}'(p')$. Suppose this was not true. There must exist an $\hat{x}' \in \mathcal{F}'(p')$ such that $f_0'(\hat{x}') < f_0'(x')$. By backward feasibility, $\hat{x} = \Phi_{\text{bwd}}(\hat{x}'; p) \in \mathcal{F}(p)$. By internal consistency:

$$f_0'(\hat{x}'; p') = \Phi_{\text{obj}}(f_0(\hat{x}; p)) \quad \text{and} \quad f_0'(x'; p') = \Phi_{\text{obj}}(f_0(x^*; p)).$$

Thus,

$$\Phi_{\text{obj}}(f_0(\hat{x}; p)) < \Phi_{\text{obj}}(f_0(x^*; p)).$$

Since Φ_{obj} is strictly increasing, this implies $f_0(\hat{x}; p) < f_0(x^*; p)$ which contradicts the optimality of x^*. Using an identical argument by contradiction, we can also show that every optimal solution of $\mathcal{M}'(p')$ can be mapped back to an optimal solution of $\mathcal{M}(p)$.

Finally, Definition 4.3 requires $\Phi_{\text{bwd}}(\cdot; p)$ to be computable in polynomial time. Therefore, given any optimal solution of $\mathcal{M}'(\Phi_p(p))$, an optimal solution of $\mathcal{M}(p)$ can be recovered in polynomial time. Hence $\mathcal{M}'$ is an Audet reformulation of $\mathcal{M}$.

For the instantiated claim, fix p and $p' = \Phi_p(p)$. The same argument shows that the mapping

$$f(x') = \Phi_{\text{bwd}}(x'; p)$$

maps every optimal solution of $\mathcal{M}'(p')$ to an optimal solution of $\mathcal{M}(p)$ and is polynomial-time computable. Thus $\mathcal{M}'(p')$ is Quasi-Karp equivalent to $\mathcal{M}(p)$. ☐

B Lean Formalization

We now formalize the *formulation* (Definition 3.1) and *reformulation* (Definition 4.3) definitions in Lean. For both definitions, we utilize a Lean `structure`. A `structure` is a collection of named fields and their types. See Listing 1 for the `MILPFormulation` and `MILPReformulation` structures. In the following sections, we provide details on how each definition is formalized.

B.1 Formulation

A formulation $\mathcal{M} = (\mathcal{P}, \mathcal{F}, f_0)$ is represented by the `MILPFormulation` structure. The `Params` field encodes the parameter space $\mathcal{P}$. Next, the `Vars` and `feasible` fields together encode the feasible region $\mathcal{F}$. Lastly, `obj` encodes the objective function f_0. Notice that both `feasible` and `obj` are functions of `Params` and `Vars`.

Discrepancies. This definition does not restrict `Vars` to $\mathbb{R}^n$ nor does it assert linearity conditions on either `feasible` or `obj`. This additional typing would add modeling complexity to `Vars`, reduce legibility, and place unnecessary burden on AFPS. Our primary motivation is not to prove if a MILP formulation is well-formed. Hence, we make the practical decision to exclude this typing.

B.2 Reformulation

We formalize reformulation with the `MILPReformulation` structure. This defines the reformulation construction $\Phi(\mathcal{M}, \mathcal{M}') = (\Phi_{\text{p}}, \Phi_{\text{fwd}}, \Phi_{\text{bwd}}, \Phi_{\text{obj}})$ along with the four conditions specified in Definition 4.3. The fields `paramMap`, `fwd`, `bwd`, and `obj` encode Φ_{p}, Φ_{fwd}, Φ_{bwd}, and Φ_{obj} respectively. The fields `fwd_feas` and `bwd_feas` encode the forward and backward feasibility conditions. The two objective mapping conditions are encoded by `fwd_obj` and `bwd_obj`. Lastly, `objMap_mono` encodes the strict monotonicity objective condition. A formulation `G` is proven to be a reformulation of `F` by declaring a definition of type `MILPReformulation F G`. This structure encodes both the witnessing reformulation construction and proves it obeys the necessary conditions. Proving `G` is *not* a *reformulation* of `F` requires proving the type `MILPEquiv F G` is uninhabited (i.e., no construction satisfying the conditions exists).

Discrepancies. We omit the restriction that the backward mapping Φ_{bwd} is computable in polynomial time. Practically speaking, we observe no examples of LLMs producing non-polynomial-time backward maps. Hence, adding this property only places undue burden on AFPS as proving computational complexity properties in Lean is non-trivial.

Listing 1: Lean formalizations: `MILPFormulation` and `MILPReformulation`.

```
import Mathlib.Tactic
import Mathlib.Data.Real.Basic
import Mathlib.Order.Basic

structure MILPFormulation where
  Params   : Type                                      -- Parameter space
  Vars     : Params → Type                             -- Variables
  feasible : (p : Params) → Vars p → Prop              -- Feasible region
  obj      : (p : Params) → Vars p → ℝ                 -- Objective function

structure MILPReformulation (F G : MILPFormulation) where
  paramMap    : F.Params → G.Params                    -- Parameter mapping
  fwd         : (p : F.Params) → F.Vars p →            -- Forward mapping
                  G.Vars (paramMap p)
  bwd         : (p : F.Params) → G.Vars (paramMap p) → -- Backward mapping
                  F.Vars p
  fwd_feas    : ∀ p x, F.feasible p x →                -- Forward feasibility condition
                  G.feasible (paramMap p) (fwd p x)
  bwd_feas    : ∀ p x', G.feasible (paramMap p) x' →   -- Backward feasibility condition
                  F.feasible p (bwd p x')
  objMap      : ℝ → ℝ                                  -- Objective mapping
  objMap_mono : StrictMono objMap                      -- Objective monotonicity
  fwd_obj     : ∀ p x, F.feasible p x →                -- Forward objective condition
                  G.obj (paramMap p) (fwd p x) = objMap (F.obj p x)
  bwd_obj     : ∀ p x', G.feasible (paramMap p) x' →   -- Backward objective condition
                  G.obj (paramMap p) x' = objMap (F.obj p (bwd p x'))
```

Listing 2: An example `MILPFormulation` Lean formalization of the TSP MTZ formulation.

```
structure Params where
  n : ℕ                            -- number of cities
  c : Fin n → Fin n → ℝ            -- arc cost
  hn : NeZero n

structure Vars (p : Params) where
  x : Fin p.n → Fin p.n → ℤ        -- arc indicator
  u : Fin p.n → ℝ                  -- position

structure Feasible (p : Params) (v : Vars p) : Prop where
  -- Each city has exactly one outgoing arc
  hout : ∀ i : Fin p.n, ∑ j : Fin p.n, v.x i j = 1
  -- Each city has exactly one incoming arc
  hin : ∀ j : Fin p.n, ∑ i : Fin p.n, v.x i j = 1
  -- MTZ subtour elimination
  hmtz : ∀ (i : Fin p.n) (j : Fin p.n), i.val ≠ 0 → j.val ≠ 0 → i ≠ j →
    v.u i - v.u j + (p.n : ℝ) * (v.x i j : ℝ) ≤ (p.n : ℝ) - 1
  -- Depot position fixed to 1
  hu_depot : haveI := p.hn; v.u 0 = 1
  hx_bin : ∀ (i j : Fin p.n), v.x i j = 0 ∨ v.x i j = 1
  -- u ∈ [2, n] for non-depot cities
  hu_lo : ∀ i : Fin p.n, i.val ≠ 0 → 2 ≤ v.u i
  hu_hi : ∀ i : Fin p.n, v.u i ≤ (p.n : ℝ)
  -- No self-loops
  hx_no_self : ∀ i : Fin p.n, v.x i i = 0

-- Minimize total arc cost
def obj (p : Params) (v : Vars p) : ℝ :=
  ∑ i : Fin p.n, ∑ j : Fin p.n, p.c i j * (v.x i j : ℝ)
```

C FLARE Implementation Details

This section outlines the implementation details for every component of FLARE (Figure 1).[1] We use a general agent harness (Claude Code, Codex, and OpenCode) for auto-formalizing MILP formulations and AFPS for reformulation proofs. This design decision was inspired by recent work achieving competitive AFPS performance with Claude Code [42]. In Section C.1, we describe how we configure the agent harness. To enable the agent to write Lean proofs effectively, we use the Lean-LSP-MCP (Section C.2). Furthermore, we use two custom Agent Skills (Section C.3) to effectively guide the agent to compose Lean reformulation proofs. After the agent exits, FLARE does a final verification step to check if a reformulation certificate was successfully generated (Section C.4).

C.1 Agent Harness

We programmatically initialize the agent harness in a headless mode using the CLI. The agent is given instructions and provided with a working directory with the necessary context. To isolate the agent's working directory and avoid duplicating the Lean environment (the Mathlib library is over 5GB), we run FLARE inside a Docker container.

Agent Prompt. In the agent prompt (Appendix D), we provide the agent with instructions to (1) formalize both natural language formulations as `MILPFormulation` structures `A` and `B` and (2) attempt to declare a definition of type `MILPReformulation A B`.

Working Directory Files. The working directory is initialized with the following context:

- natural language descriptions of both MILP formulations (using the Listing 5 template),
- Gurobi Python implementations for both MILP formulations, and
- a Lean file `Common.lean` with `MILPFormulation` and `MILPReformulation` definitions.

Docker. We pre-build a Docker image called `flare-agent` with every agent CLI (Claude Code, Codex, and OpenCode), the Lean-LSP-MCP, and a Lake-built Lean environment with Mathlib pre-compiled. When FLARE is invoked, it creates a working directory on the host with all the necessary files. It then creates a Docker container from this image and bind mounts the working directory into the container. To make use of the pre-built Lean environment without copying it over to the host, we symlink `.lake` from the pre-built location into the agent's working directory. This configuration prevents duplication of the Lean environment while ensuring FLARE can run in parallel without agents impacting each other's Lean environment. We also attempted to isolate agents with the permissions and sandboxing mechanisms provided by each agent harness. We found the implementations to be immature; Docker was the only reliable way to ensure the agent didn't access files outside its working directory.

C.2 Lean-LSP-MCP

Lean-LSP-MCP [14][2] is a Model Context Protocol (MCP) server providing MCP Tools for Lean theorem proving. It enables the agent to efficiently generate, debug, and compile Lean proofs and has been utilized by numerous Lean AFPS methods [7, 42, 29]. The Lean Language Server Protocol (LSP) traditionally allows editors like VS Code to get rich, interactive feedback from a running Lean process. The Lean-LSP-MCP allows the agent to communicate directly with this LSP and obtain feedback like a human theorem prover. It offers a broad range of tools including:

- `lean_goal`. Get the proof goal at a specific location in the file. This allows the agent to observe precisely what sub-goal must be proved to continue making progress.
- `lean_diagnostic_messages`. Retrieve all the diagnostic messages for a Lean file. This allows the agent to verify if the file is compiling and, if not, where fixes are necessary.
- `lean_verify`. Verify the soundness of a proof. There are two conditions where a proof compiles, but it actually unsound: (1) a new `axiom` was added or (2) the `sorry` tactic is used to complete a goal. This tool allows the agent to verify that neither condition holds.
- `lean_multi_attempt`. Attempt multiple tactics at a proof position and return the new goal state for each. This allows the agent to explore strategies for making progress in parallel.
- `lean_hover_info`. Retrieve documentation for symbols, terms, and expressions. This allows the agent to view underlying definitions and reduces hallucination.

[1] The FLARE implementation is available at https://github.com/henryrobbins/flare

[2] https://github.com/oOo0oOo/lean-lsp-mcp

C.3 Agent Skills

Agent Skills[1] are an "open format for extending AI agent capabilities with specialized knowledge and workflows." They are simply a directory containing context, scripts, or other resources related to a specific task. The directory contains a `SKILL.md` file which tells the agent the Skill's name and provides instructions on when to use it. Skills allow for *progressive disclosure* where the agent loads additional task-specific context as needed.

We define two custom Agent Skills, `lean-milp-formulation` and `lean-milp-reformulation`, for auto-formalizing MILP formulations and proving reformulations respectively. The agent prompt explicitly instructs the agent to invoke these skills. Both skills contain instructions along with a template Lean file containing detailed comments about Lean modeling choices and file structure. In addition to our custom Skills, we use the general-purpose Lean 4 Skills [19][2].

C.4 Final Verification

After the agent exits, we inspect the working directory. First, we verify the presence of `A/Formulation.lean`, `B/Formulation.lean`, and `Reformulation.lean`. If all of these files are present and compile, we then check that `Reformulation.lean` contains a `MILPReformulation A B` definition. Finally, we inspect the compilation output for `Reformulation.lean` for any use of the `sorry` tactic. In Lean, the `sorry` tactic can be used to complete any proof goal; it indicates the proof has yet to be formally verified. If all files compile with a `sorry`-free `MILPReformulation A B` definition, formulation `B` is deemed a reformulation of `A`.

[1] https://agentskills.io/home

[2] https://github.com/cameronfreer/lean4-skills

D Prompts

Listing 3: FLARE Agent Prompt

```
You are a mixed-integer linear programming (MILP) and Lean 4 expert. You have been tasked with formalizing two MILP
      formulations in Lean 4 and then proving that formulation B is a reformulation of formulation A.

## Working Directory Structure

The working directory will be initialized with the following structure:

‘‘‘
  +-- A
  |   +-- Formulation.lean    # Write the Lean 4 formalization of Formulation A here
  |   +-- formulation.md      # Natural language description of Formulation A
  |   \-- solve.py            # Python script that solves Formulation A
  +-- B
  |   +-- Formulation.lean    # Write the Lean 4 formalization of Formulation B here
  |   +-- formulation.md      # Natural language description of Formulation B
  |   \-- solve.py            # Python script that solves Formulation B
  +-- Common.lean             # Definition of ‘MILPFormulation‘ and ‘MILPEquiv‘
  +-- Reformulation.lean      # Write the reformulation proof here
  +-- lake-manifest.json      # DO NOT EDIT!
  +-- lakefile.toml           # DO NOT EDIT!
  \-- lean-toolchain          # DO NOT EDIT!
‘‘‘

## Workflow
1. Utilize the ‘lean-milp-formulation‘ skill to generate both ‘Formulation.lean‘ files.
2. Validate each ‘Formulation.lean‘ file with ‘mcp__lean-lsp__lean_diagnostic_messages‘. Fix any issues that arise and repeat
      until both files compile cleanly.
3. Run ‘Bash(lake build A.Formulation B.Formulation)‘ to materialize their oleans.
4. Determine whether formulation B is a mathematical reformulation of formulation A.
5. If B is a reformulation of A, use the ‘lean-milp-reformulation‘ skill to generate a compiled ‘MILPReformulation‘ proof in ‘
      Reformulation.lean‘.
6. Validate ‘Reformulation.lean‘ with ‘mcp__lean-lsp__lean_verify‘ to ensure there are no stubs. Fix any issues that arise and
      repeat until both files compile cleanly.

## Rules
- IMPORTANT: Only read/write files that exist in *this* working directory. Do not navigate outside of it for any reason.
- The Lean project root is the current directory. Use ‘import A.Formulation‘ and ‘import B.Formulation‘.
- You MUST use the lean-lsp MCP tools (mcp__lean-lsp__*) to check compilation as you work. Before doing anything else, verify
      the lean-lsp MCP server is available by calling ‘mcp__lean-lsp__lean_diagnostic_messages‘ on ‘Common.lean‘. If that
      initial probe fails because the server itself is unavailable (e.g. "Failed to start Lean language server", tool not
      registered), the environment is unusable: write ‘MCP_UNAVAILABLE: <error>‘ to Reformulation.lean and exit. Do not fall
      back to ‘lake env lean‘ to perform compilation.
- If ‘lake build‘ fails, treat the error message as a real signal -- read it, fix the cause (typically a typo, missing import,
      or stale olean), and retry. Only after the *same* failure persists across two clean retries should you write ‘
      LAKE_BUILD_FAILED: <error>‘ to Reformulation.lean and exit. Persistent toolchain-level failures (e.g. elan/toolchain
      missing) are the exit case; ordinary compile errors are not.
- Generate both Formulation.lean files before attempting the reformulation proof.
- Confirm the final reformulation proof with ‘mcp__lean-lsp__lean_verify‘ on the ‘MILPReformulation‘ definition. The returned
      axioms must NOT contain ‘sorryAx‘ -- if it does, the proof has a stub and you must finish it.
- You are expected to iterate on the proof until it compiles and ‘lean_verify‘ reports no ‘sorryAx‘. A non-trivial
      reformulation proof typically runs 100+ lines with many manual ‘refine‘/‘rcases‘/‘simp‘ steps and multiple rounds of
      compile-error fixing. You should only exit before this point if there is concrete evidence that B is *not* a reformulation
       of A under the given assumptions.

## Common Mistakes
- Interpret ‘lean_diagnostic_messages‘ carefully: ‘success:true, items:[]‘ means the file compiles cleanly. ‘success:false,
      items:[]‘ typically means imports aren’t built yet -- build them, don’t assume the file is broken. Real errors come back
      as ‘items‘ with severity/message fields.
- If there issues with the Lean environment or MCP tools, it is imperative to handle them as described in the rules above.
      Report and exit.
- If you are stuck in proving the reformulation due missing problem data assumptions in either ‘Formulation.lean‘, verify that
      the assumptions specified in ‘formulation.md‘ are all present in the corresponding ‘Formulation.lean‘. If they are not, DO
       NOT ADD ASSUMPTIONS YOURSELF. Instead, report the missing assumptions as an issue preventing the reformulation proof in ‘
      Reformulation.lean‘ and exit.

## Available Tools
**Filesystem:** ‘Bash(ls ./*)‘, ‘Bash(find ./*)‘,
**Read:** ‘Read(./*)‘, ‘Bash(cat ./*)‘, ‘Bash(head ./*)‘, ‘Bash(tail ./*)‘, ‘Bash(less ./*)‘, ‘Bash(more ./*)‘, ‘Bash(bat ./*)‘
**Edit:** ‘Edit(./*)‘
**Write:** ‘Write(./*)‘
**Skills:** ‘lean4:lean4‘, ‘lean-milp-formulation‘, ‘lean-milp-reformulation‘
**MCP Tools:** ‘mcp__lean-lsp__*‘
**Lake:** ‘Bash(lake env lean:*)‘, ‘Bash(lake build:*)‘
```

Listing 4: FLARE-NL Prompt. The *No Definition* variant omits L9-27. The *Allow Implicit* variant omits L31 as well as implicit assumptions and constraints from the formulation descriptions.

```
1  You are given the following two Mixed-Integer Linear Programming (MILP) formulations. You are
       tasked with deciding if formulation B is a reformulation of formulation A.
2
3  ## Formulations
4
5  {{ problem_a }}
6
7  {{ problem_b }}
8
9  ## Definitions
10
11 Use the following definition of formulation and reformulation to guide your reasoning.
12
13 **Formulation.** A MILP *formulation* $A$ is a tuple ${A = (P, F, f_0)}$ with parameter space $P$
       , feasible region $F(p) \subseteq R^{n(p)}$, and objective function $f_0$. For instance $p \
       in P$, the feasible region $F(p)$ is defined by $m(p)$ linear constraints, $f_i(\cdot ; p) :
        R^{n(p)} \to R$ for all $i \in [m(p)]$. The first $k(p) \leq n(p)$ variables are integers.
       The feasible region is
14 $$F(p) = \{x \in \mathbb{Z}^{k(p)} \times R^{n(p)-k(p)}~|~f_i(x;p) \leq 0 ~\forall i\in[m(p)]\}.
       $$
15 The objective is to minimize the linear function $f_0(\cdot ; p) : R^{n(p)} \to R$. A formulation
        $A$ is *instantiated* with an *instance* $p \in P$. We denote an instantiated formulation
       as ${A(p) = (F(p), f_0(p))}$.
16
17 **Reformulation.** Let $A = (P, F, f_0)$ and $B = (P', F', f'_0)$ be formulations. A *
       reformulation construction* from $A$ to $B$ is a tuple $\Phi(A, B) = (\Phi_{p},\, \Phi_{fwd
       },\, \Phi_{bwd},\, \Phi_{\text{obj}})$ consisting of:
18 - a parameter mapping $\Phi_{p} : P \to P'$,
19 - a forward mapping $\Phi_{fwd}(\cdot;p) : R^{n(p)} \to R^{n'(\Phi_{p}(p))}$,
20 - a backward mapping $\Phi_{bwd}(\cdot;p) : R^{n'(\Phi_{p}(p))} \to R^{n(p)}$ computable in
       polynomial time, and
21 - an objective mapping $\Phi_{\text{obj}} : R \to R$.
22
23 $B$ is a *reformulation* of $A$ if there exists a reformulation construction satisfying the
       following conditions for every instance $p \in P$, with $p' = \Phi_{p}(p)$:
24 - **Forward feasibility.** For all feasible points $x \in F(p)$, the forward mapping maps to a
       feasible point $x' = \Phi_{fwd}(x;p) \in F'(\Phi_{p}(p))$.
25 - **Backward feasibility.** For all feasible points $x' \in F'(p')$, the backward mapping maps to
        a feasible point $x = \Phi_{bwd}(x';p) \in F(p)$.
26 - **Objective mapping.** The forward and backward mappings induce an objective mapping. The
       following two conditions must hold. (1) For all feasible points ${x \in F(p)}$, the forward
       mapped point $x' = \Phi_{fwd}(x;p)$ has objective value $f_0'(x';p') = \Phi_{\text{obj}}(f_0
       (x;p))$. (2) For all feasible points ${x' \in F'(p')}$, the backward mapped point is $x = \
       Phi_{bwd}(x';p)$ and $f_0'(x';p') = \Phi_{\text{obj}}(f_0(x;p))$.
27 - **Strictly monotone.** The objective mapping $\Phi_{\text{obj}}$ is strictly monotonically
       increasing.
28
29 ## Instructions
30
31 - Do NOT make any assumptions about the formulation or parameter space that are not explicitly
       stated in the formulation descriptions.
32 - When uncertain, state that formulation B is *not* a reformulation of A.
33 - Provide a short summary of your conclusion (at most 2,500 characters) and a final determination
        of whether B is a reformulation of A (true or false).
```

Listing 5: Formulation Prompt Template

```
# {{ problem_name }}

## Problem Description

{{ problem_description }}

## Formulation

### Parameters

{% for name, p in parameters.items() %}
- **{{ name }}** (type: {{ p.type.value }}, shape: `{{ p.shape }}`): {{ p.description }}
{% endfor %}

{% if assumptions %}

### Assumptions

{% for a in assumptions %}
- {{ a.description }}
$${{ a.formulation }}$$
{% endfor %}
{% endif %}

{% if definitions %}

### Definitions

{% for name, d in definitions.items() %}
- **{{ name }}**: {{ d.description }}
$${{ d.formulation }}$$
{% endfor %}
{% endif %}

### Variables

{% for name, v in variables.items() %}
- **{{ name }}** (type: {{ v.type.value }}, shape: `{{ v.shape }}`): {{ v.description }}
{% endfor %}

### Constraints

{% for c in constraints %}
- {{ c.description }}
$${{ c.formulation }}$$
{% endfor %}

### Objective

{{ objective.description }}
$${{ objective.formulation }}$$
```

E FormulationBench Details

The *FormulationBench* dataset is a collection of 20 optimization problems, 116 MILP formulations, and 96 reformulation pairs (Table 4). It is comprised of three sources:

- **EquivaFormulation [68].** The five *EquivaFormulation* problems were sampled at random. These are simple optimization problems with a few scalar variables and constraints. Each one contains an original formulation and 10 transformations. These are enumerated in Table 1 of [68]. We make the following modifications to support our use case.
  - The *Add Valid Inequalities* transformation is instance-specific. Because we are interested in formulation-level reformulation, we omit this transformation type.
  - We change the label on the *Replace by Base-10 Representation*. This formulation replaces each integer variable with a base-10 decimal expansion. It relies on using enough digit variables to support the size of the test instance data. Hence, this transformation is not valid under our formulation-level notion of reformulation.
- **EvoCut [66].** They define 7 optimization problems for which their method EvoCut proposes numerous cutting planes. We construct reformulation pairs by pairing the original MILP formulation with one in which the cut is added. A valid cutting plane implies a valid reformulation.
- **Ferchtandiker [16].** They define 8 real-world optimization problems, each admitting an efficient and inefficient formulation. We add each as a pair to *FormulationBench*.

Table 4: The 20 optimization problems in the *FormulationBench* dataset with their source and number of formulation pairs.

Problem	Name	Source	Valid	Invalid
`p1`	EquivaFormulation Instance 47	EquivaFormulation [68]	5	4
`p2`	EquivaFormulation Instance 74	EquivaFormulation [68]	5	4
`p3`	EquivaFormulation Instance 92	EquivaFormulation [68]	5	4
`p4`	EquivaFormulation Instance 183	EquivaFormulation [68]	5	4
`p5`	EquivaFormulation Instance 217	EquivaFormulation [68]	5	4
`p6`	Capacitated Warehouse Location (CWLP)	EvoCut [66]	9	0
`p7`	Rectangular Tiling (IMO6)	EvoCut [66]	6	2
`p8`	Job Shop Scheduling (JSSP)	EvoCut [66]	3	0
`p9`	Multi-Commodity Network Design (MCND)	EvoCut [66]	3	0
`p10`	Pickup and Delivery with Time Windows (PDPTW)	EvoCut [66]	4	0
`p11`	Sub-Hour Unit Commitment (SHUC)	EvoCut [66]	8	0
`p12`	Traveling Salesman Problem (TSP)	EvoCut [66]	5	3
`p13`	Air Traffic Flow Management	Ferchtandiker [16]	1	0
`p14`	Blood Bank Netherlands	Ferchtandiker [16]	1	0
`p15`	Dutch Housing Problem	Ferchtandiker [16]	1	0
`p16`	Park and Bike Hub Location (Mobian)	Ferchtandiker [16]	1	0
`p17`	Open-Pit Mine Production Scheduling	Ferchtandiker [16]	0	1
`p18`	Timor-Leste Hospital Location	Ferchtandiker [16]	1	0
`p19`	UN Humanitarian Disaster Response Hub (UNHDR)	Ferchtandiker [16]	1	0
`p20`	World Food Program Food Distribution	Ferchtandiker [16]	1	0

Dataset Structure. Each problem contains (1) a Markdown problem description file, (2) a JSON information file, and (3) a JSON file with instance data and the corresponding optimal solution. Each problem contains at least two formulations. Each formulation contains (1) a JSON information file, (2) a Python script to transform raw problem data into the formulation's parameter space, and (3) a ground-truth `MILPFormulation` Lean formalization. The JSON information file format is an extension of the format introduce by the NLP4LP dataset [1]. In addition to the `parameters`, `variables`, `constraints`, and `objective` fields, we add `assumptions` (see 6.1 for a discussion) and `definitions`. Lastly, each formulation is flagged if it is valid, i.e., it is a faithful representation of the underlying optimization problem.

Constructing Reformulation Test Set. Formulation pairs are constructed by taking the first formulation $\mathcal{M}$ of each problem and comparing it to each of the remaining formulations $\mathcal{M}'$. The first formulation is always valid. If the other formulation $\mathcal{M}'$ is also valid, $\mathcal{M}'$ is considered a reformulation of $\mathcal{M}$. Otherwise, it is not a reformulation. For every valid reformulation, we provide a ground-truth `MILPReformulation` proof. Note that additional formulation pairs can be generated by considering *all* possible combinations of formulations within each problem.

F Invalid Reformulations

While preparing the *FormulationBench* dataset, we verified prior LLM-generated MILP reformulations: cutting planes proposed by EvoCut [66] and reformulation pairs proposed by Ferchtandiker [16]. In the process, FLARE failed to produce reformulation certificates for 5 cutting planes and 3 formulations that were confirmed invalid upon manual inspection. In this section, we provide proofs of invalidity.

F.1 EvoCut Cutting Planes

Yazdani et al. [66] recently proposed the EvoCut framework to automatically generate *acceleration cuts* for MILP formulations using an LLM-based evolutionary search. Acceleration cuts are constraints added to a MILP formulation with the aim of reducing solve time. Importantly, such cuts are *not* guaranteed to be valid cutting planes or even optimality preserving. This pragmatic choice to consider acceleration cuts enables automation by reducing the computational burden required to verify candidate cuts. A cut is considered an acceleration cut if it does not change the optimal objective value on a small verification set of problem instances.

In the *FormulationBench* dataset, we construct a reformulation from a cutting plane by adding the cut to the base formulation. A formulation constructed from a valid cutting plane trivially satisfies our *constructive reformulation*. Using FLARE, we identify that 5 of the 43 acceleration cuts proposed by EvoCut are invalid cutting planes. These cuts span the TSP and a rectangle tiling problem. In the case of TSP, all three invalid cutting plane families are only invalid on TSP instances of size $n \leq 3$. While these counter-examples are not practically meaningful, they illustrate the importance of explicitly stating all assumptions of the problem data. In the case of rectangle tiling, the cuts meaningfully change the set of optimal solutions (see Figure 3). This motivates the importance of formally verifying reformulations, especially for non-standard optimization problems where an LLM is more likely to have an error in reasoning.

F.1.1 Traveling Salesman Problem (TSP)

For the TSP, EvoCut generates acceleration cuts for the Miller–Tucker–Zemlin (MTZ) formulation defined in Section 3. Between v1 and v2 of the arXiv preprint, there are 8 acceleration cuts proposed. FLARE identifies the following three as invalid cutting planes. Both cuts v1-EC3 and v2-EC2 eliminate triangles including the depot node and cut v2-EC1 eliminates all two-city subtours.

$$x_{j1} + x_{ji} + (u_j - u_i - 1) \leq (n-1)(2 - x_{1i} - x_{ij}) \qquad \forall i,j \in V \setminus \{1\},\ i \neq j \qquad \text{(v1-EC3)}$$
$$x_{ij} + x_{ji} \leq 1 \qquad \forall i,j \in V,\ i < j \qquad \text{(v2-EC1)}$$
$$x_{1i} + x_{i1} + x_{1j} + x_{j1} + x_{ij} + x_{ji} \leq 2 \qquad \forall i,j \in V \setminus \{1\},\ i < j \qquad \text{(v2-EC2)}$$

Proposition F.1. The cut v1-EC3 is *not* a valid cutting plane for the MTZ TSP formulation.

Proof. Consider the 3-node TSP instance depicted in Figure 2a with feasible tour $1 \to 2 \to 3 \to 1$. Letting $i = 2$ and $j = 3$, the cut v1-EC3 reduces to $1 \leq 0$, which does not hold. Hence, there exists an feasible integer point that is not satisfied by the cut. Therefore, the cut is invalid. □

Proposition F.2. The cut v2-EC1 is *not* a valid cutting plane for the MTZ TSP formulation.

Proof. Consider the 2-node TSP instance with the unique feasible tour $1 \to 2 \to 1$. The MTZ formulation permits $x_{12} = x_{21} = 1$, corresponding to traversing both arcs of this tour. Setting $i = 1$ and $j = 2$, the cut v2-EC1 reduces to $2 \leq 1$, which does not hold. Hence, there exists a feasible integer point that is not satisfied by the cut. Therefore, the cut is invalid. □

Proposition F.3. The cut v2-EC2 is *not* a valid cutting plane for the MTZ TSP formulation.

Proof. Consider the 3-node TSP instance depicted in Figure 2a with feasible tour $1 \to 2 \to 3 \to 1$. Setting $i = 2$ and $j = 3$, the cut v2-EC2 reduces to $3 \leq 2$, which does not hold. Hence, there exists a feasible integer point that is not satisfied by the cut. Therefore, the cut is invalid. □

F.1.2 Rectangular Tiling with One Hole per Row and Column (IMO6)

This problem is inspired by IMO 2025 Problem 6. Given an $N \times N$ grid of unit squares, one must place rectangular tiles (of various sizes) such that each row and column of the grid contain exactly one uncovered square (a *hole*). The objective is to minimize the number of tiles used. Yazdani et al. [66] introduce the following notation and formulation.

Notation.

- $R = \{1, \ldots, N\}$ rows and $C = \{1, \ldots, N\}$ columns
- $I = \{(a, b) \in C^2 \mid a \leq b\}$ contiguous column-intervals
- $h_{ij} \in \{0, 1\}$ hole indicator
- $x_i^{ab} \in \{0, 1\}$ indicates if columns a through b on row i are covered by the same tile
- $s_i^{ab}, t_i^{ab} \in \{0, 1\}$ indicate if a tile spanning columns a to b begins on row i or ends on row i, respectively

Formulation.

$$
\begin{aligned}
\min \ & \sum_{i \in R} \sum_{(a,b) \in I} s_i^{ab} \\
\text{s.t.} \ & \sum_{j \in C} h_{ij} = 1 && \forall i \in R \\
& \sum_{i \in R} h_{ij} = 1 && \forall j \in C \\
& \sum_{\substack{(a,b) \in I \\ a \leq j \leq b}} x_i^{ab} + h_{ij} = 1 && \forall i \in R, \ \forall j \in C \\
& x_1^{ab} - s_1^{ab} = 0 && \forall (a,b) \in I \\
& x_i^{ab} - x_{i-1}^{ab} - s_i^{ab} + t_{i-1}^{ab} = 0 && \forall i = 2, \ldots, N, \ \forall (a,b) \in I \\
& x_N^{ab} - t_N^{ab} = 0 && \forall (a,b) \in I \\
& h_{ij} \in \{0, 1\} && \forall i \in R, \ \forall j \in C \\
& x_i^{ab}, s_i^{ab}, t_i^{ab} \in \{0, 1\} && \forall i \in R, \ \forall (a,b) \in I
\end{aligned}
\tag{IMO6}
$$

Between `v1` and `v2` of the arXiv preprint, there are 8 acceleration cuts proposed. `FLARE` identifies the following two as invalid cutting planes. The cut v1-EC1 enforces that a tile end (bottom-right corner) must be immediately to the left of any hole, and the cut v1-EC2 enforces that a tile start (top-left corner) must be immediately to the right of any hole.

$$
h_{ij} \leq \sum_{\substack{(a,b) \in I \\ b = j-1}} t_i^{ab} \qquad \forall i \in R, \ \forall j \in \{2, \ldots, N\} \tag{v1-EC1}
$$

$$
h_{ij} \leq \sum_{\substack{(a,b) \in I \\ a = j+1}} s_i^{ab} \qquad \forall i \in R, \ \forall j \in \{1, \ldots, N-1\} \tag{v1-EC2}
$$

Proposition F.4. The cut v1-EC1 is *not* a valid cutting plane for the IMO6 formulation.

Proof. Consider the $N = 3$ counterexample depicted in Figure 3. Let $i = j = 2$. The inequality reduces to $h_{22} \leq t_2^{11}$. Substituting yields $1 \leq 0$, which does not hold. Hence, there exists a feasible integer point that is not satisfied by the cut. Therefore, the cut is invalid. □

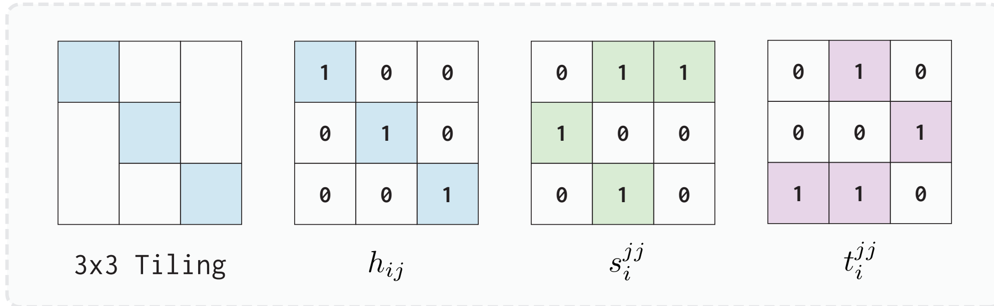

Figure 3: A feasible rectangular tiling for the $N = 3$ instance of IMO 2025 Problem 6. Decision variable values for the hole indicator h_{ij}, start flag s_i^{jj} and end flag t_i^{jj} are depicted in the grids. Note the figure does not specify s_i^{ab} and t_i^{ab} for $a < b$. However, since all tiles have unit width, these variables all have value 0.

Proposition F.5. The cut v1-EC2 is *not* a valid cutting plane for the IMO6 formulation.

Proof. Consider the $N = 3$ counterexample depicted in Figure 3. Let $i = 2$ and $j = 2$. The inequality reduces to $h_{22} \leq s_2^{11} + s_2^{12} + s_2^{13}$. Substituting yields $1 \leq 0$, which does not hold. Hence, there exists a feasible integer point that is not satisfied by the cut. Therefore, the cut is invalid. □

F.2 Ferchtandiker Formulation Pairs

Ferchtandiker [16] introduces a dataset[1] containing an efficient and inefficient MILP formulation for 8 optimization problems. The dataset includes both LaTeX and GurobiPy code for each formulation. In the process of incorporating these reformulations in to the *FormulationBench* dataset, FLARE identified four invalid reformulations. Note that FLARE verifies our stronger *constructive reformulation* definition, not *Audet reformulation*. The Open Pit Mining reformulation is intentionally a *relaxation*. Hence, it is expected to violate our definition of reformulation and we omit any further discussion. Also notable is the World Food Program reformulation. While this formulation is an *Audet reformulation*, it is not a *constructive reformulation*.

[cl: worth making it clear that we are checking our definition of reformulation not audet (I think the last example I would consider a reformulation but not under our definition. Actually a good discussion point for the difference and why our stricter notion can be a bad fit)]

F.2.1 Air Traffic Management (ATM)

This problem consists of assigning each plane in an airline's fleet to a location (airport or airspace sector) over a sequence of time periods. The assignment must respect each location's capacity over time and the network's adjacency structure. Planes can only travel to locations that are a single time unit away. The objective is to maximize the total reward accrued for visiting locations. Ferchtandiker [16] introduces the following formulations of this problem.

Notation.

- P: set of vehicles (planes).
- A: set of locations (airports and airspace sectors).
- T: set of time periods.
- $\text{cap}_{a,t}$: capacity of location $a \in A$ at time $t \in T$.
- $\tau_{a,a'}$: travel time from location a to location a'.
- $r_{a,t}$: reward for being at location $a \in A$ at time $t \in T$.

Formulations. Both formulations share the binary departure variable $x_{p,a,a',t} \in \{0,1\}$, which indicates whether plane p departs from a to a' at time t.[2] The Efficient formulation is purely event-based. The Inefficient formulation additionally tracks plane locations via $y_{p,a,t} \in \{0,1\}$, which indicates whether plane p is at location a at time t.

$$
\begin{aligned}
\max \quad & \sum_{p\in P}\sum_{a\in A}\sum_{t\in T} r_{a,t}\, y_{p,a,t} \\
\text{s.t.} \quad & \sum_{a\in A} y_{p,a,t} = 1 && \forall p \in P,\ t \in T \\
& \sum_{p\in P} y_{p,a,t} \leq \text{cap}_{a,t} && \forall a \in A,\ t \in T \\
& y_{p,a,t} = y_{p,a,t-1} + \sum_{a'\in A}\ \sum_{t':t'+\tau_{a',a}=t} x_{p,a',a,t'} - \sum_{a'\in A} x_{p,a,a',t} && \forall p \in P,\ a \in A,\ t > 0 \\
& y_{p,a,t} \in \{0,1\} && \forall p \in P,\ a \in A,\ t \in T \\
& x_{p,a,a',t} \in \{0,1\} && \forall p \in P,\ a, a' \in A,\ t \in T
\end{aligned}
\qquad \text{(Inefficient)}
$$

$$
\begin{aligned}
\max \quad & \sum_{p\in P}\sum_{a\in A}\sum_{t\in T} r_{a,t} \left(\sum_{a'\in A}\ \sum_{t':t'+\tau_{a',a}=t} x_{p,a',a,t'} \right) \\
\text{s.t.} \quad & \sum_{a\in A}\sum_{a'\in A}\sum_{t\in T} x_{p,a,a',t} \geq 1 && \forall p \in P \\
& \sum_{p\in P}\sum_{a'\in A} x_{p,a,a',t} \leq \text{cap}_{a,t} && \forall a \in A,\ t \in T \\
& x_{p,a,a',t} \in \{0,1\} && \forall p \in P,\ a, a' \in A,\ t \in T
\end{aligned}
\qquad \text{(Efficient)}
$$

Proposition F.6. The inefficient formulation of ATM is *not* a *constructive reformulation* of the efficient formulation.

Proof. The two formulations have misaligned objectives and capacity semantics. In the inefficient formulation, the reward $\sum_{p,a,t} r_{a,t} y_{p,a,t}$ accrues at every time period a plane is present at a location, so a plane that remains at a for

[1] The dataset is available at https://github.com/nathan-ferchtandiker/LLMs-For-Optimization-Reformulations

[2] In the dataset, this variable is denoted $z_{p,a,a',t}$ in the inefficient formulation. We use $x_{p,a,a',t}$ for both formulations.

k consecutive time periods contributes $\sum_t r_{a,t}$ over those k periods. In the efficient formulation, the reward only accrues on arrivals, so the same solution contributes only the single reward $r_{a,t}$ at the arrival time. Furthermore, the inefficient capacity constraint $\sum_p y_{p,a,t} \leq \text{cap}_{a,t}$ bounds the number of planes *present* at a location, while the efficient capacity constraint $\sum_{p,a'} x_{p,a,a',t} \leq \text{cap}_{a,t}$ bounds only the number of *departures*, so a feasible departure schedule in the efficient formulation may correspond to an infeasible occupancy in the inefficient one. Since the objectives and feasible sets disagree on corresponding points under the natural variable mapping, the inefficient formulation is not a reformulation of the efficient one. □

F.2.2 UN Humanitarian Disaster Response Hub (UNHDR)

This problem consists of selecting a fixed number of response hubs to service a set of disaster regions. The selection must satisfy demand and response time constraints. The objective is to minimize transportation cost. Ferchtandiker [16] introduces the following formulations of this problem.

Notation.

- H: set of candidate hubs.
- $H^f \subseteq H$: set of hubs that must be open (fixed hubs).
- C: set of disaster regions.
- a_c: number of people affected in region $c \in C$.
- C_{hc}: cost per person from hub h to region c.
- t_{hc}: travel time from hub h to region c.
- T: maximum allowed (weighted) travel time per region.
- n: maximum number of hubs that can be opened.
- M: sufficiently large constant (used in big-M constraints).

Formulations. The Inefficient formulation introduces a binary indicator variable y_h to indicate if hub $h \in H$ is opened and binary indicator z_{hc} to indicate if hub $h \in H$ is assigned to serve region $c \in C$. Lastly, x_{hc} is the fraction of region c's demand that is served by hub h.[1] The Efficient formulation drops the hub indicator z_{hc} variables.

$$
\begin{aligned}
\min \quad & \sum_{h\in H}\sum_{c\in C} a_c\, C_{hc}\, x_{hc} \\
\text{s.t.} \quad & x_{hc} \leq M z_{hc} && \forall h \in H,\ c \in C \\
& \sum_{h\in H} z_{hc} = 1 && \forall c \in C \\
& \sum_{c\in C} z_{hc} \leq |C| y_h && \forall h \in H \\
& \sum_{h\in H} x_{hc} = 1 && \forall c \in C \\
& \sum_{h\in H} y_h \leq n && \\
& y_h = 1 && \forall h \in H^f \\
& \sum_{h\in H} t_{hc} x_{hc} \leq T && \forall c \in C \\
& x_{hc} \geq 0 && \forall h \in H,\ c \in C \\
& z_{hc} \in \{0,1\} && \forall h \in H,\ c \in C \\
& y_h \in \{0,1\} && \forall h \in H
\end{aligned}
\tag{Inefficient}
$$

$$
\begin{aligned}
\min \quad & \sum_{h\in H}\sum_{c\in C} a_c C_{hc} x_{hc} \\
\text{s.t.} \quad & \sum_{c\in C} x_{hc} \leq |C| y_h && \forall h \in H \\
& \sum_{h\in H} x_{hc} = 1 && \forall c \in C \\
& \sum_{h\in H} y_h \leq n && \\
& y_h = 1 && \forall h \in H^f \\
& \sum_{h\in H} t_{hc} x_{hc} \leq T && \forall c \in C \\
& x_{hc} \geq 0 && \forall h \in H,\ c \in C \\
& y_h \in \{0,1\} && \forall h \in H
\end{aligned}
\tag{Efficient}
$$

Proposition F.7. The inefficient formulation of UNHDR is *not* a *constructive reformulation* of the efficient formulation.

Proof. The two formulations are identical with the exception of the constraints involving z_{hc}. The inefficient formulation adds an additional constraint $x_{hc} \leq M z_{hc}$ to ensure $x_{hc} = 0$ if $z_{hc} = 0$. Furthermore, it adds $\sum_{h\in H} z_{hc} = 1$. This constraint enforces that every disaster region $c \in C$ is assigned to *exactly one* response hub $h \in H$. The efficient formulation has no such restriction; a disaster region can be supplied by multiple hubs simultaneously. Therefore, the inefficient formulation is not a reformulation of the efficient one. □

F.2.3 World Food Program (WFP)

This problem consists of designing a food distribution plan for the World Food Program (WFP) to deliver commodities from suppliers, through transshipment points, to beneficiary camps in crisis-affected regions. The plan must satisfy each

[1] In the dataset, this variable is denoted q_{hc}. We use x_{hc} for consistency with the efficient formulation.

camp's ration demand and meet per-person nutritional requirements across all nutrients. The objective is to minimize the total procurement and transportation cost. Ferchtandiker [16] introduces the following formulations of this problem.

Notation.

- K: set of commodities.
- L: set of nutrients.
- pc_k is the procurement cost per kg of commodity k.[1]
- $\mathrm{nutval}_{k\ell}$: nutrient-ℓ content (per kg) of commodity $k \in K$.
- nutreq_ℓ: per-person requirement for nutrient $\ell \in L$.
- dem_j: number of beneficiaries at camp j.
- $R_k \geq 0$: ration size (kg per person) of commodity $k \in K$.

Formulations. The Efficient formulation is node-based: N is the set of nodes, partitioned into suppliers N_S, transshipment points N_T, and beneficiary camps N_B. $E_{ij} \in \{0, 1\}$ indicates whether an edge from i to j exists and we have a transportation cost tc_{ijk} per kg of commodity k on edge (i, j). The decision variable $F_{ijk} \geq 0$ encodes the amount of commodity k shipped from i to j. The Inefficient formulation is path-based: P is the set of all simple paths from a supplier to a beneficiary camp. The indicator $e_{jp} \in \{0, 1\}$ indicates whether path p ends at camp j and c_{pk} is the cost of shipping one kg of commodity k along path p. The decision variable $x_{pk} \geq 0$ is the amount of commodity k shipped along path p.

$$
\begin{aligned}
\min\quad & \sum_{k\in K} pc_k \left(\sum_{p\in P} x_{pk}\right) + \sum_{p\in P}\sum_{k\in K} c_{pk}\, x_{pk} \\
\text{s.t.}\quad & \sum_{p\in P} e_{jp}\, x_{pk} \geq \mathrm{dem}_j\, R_k && \forall j \in N_B,\ k \in K \\
& \sum_{k\in K} \mathrm{nutval}_{k\ell}\, R_k \geq \mathrm{nutreq}_\ell && \forall \ell \in L \\
& x_{pk} \geq 0 && \forall p \in P,\ k \in K \\
& R_k \geq 0 && \forall k \in K
\end{aligned}
\tag{Inefficient}
$$

$$
\begin{aligned}
\min\quad & \sum_{k\in K} \mathrm{pc}_k \left(\sum_{j\in N_B} \mathrm{dem}_j R_k\right) + \sum_{i,j\in N}\sum_{k\in K} \mathrm{tc}_{ijk}\, F_{ijk} \\
\text{s.t.}\quad & \sum_{i\in N} E_{ij} F_{ijk} = \sum_{i\in N} E_{ji} F_{jik} && \forall j \in N,\ k \in K \\
& \sum_{i\in N} E_{ij} F_{ijk} \geq \mathrm{dem}_j R_k && \forall j \in N_B,\ k \in K \\
& \sum_{k\in K} \mathrm{nutval}_{k\ell} R_k \geq \mathrm{nutreq}_\ell && \forall \ell \in L \\
& F_{ijk} \geq 0 && \forall i, j \in N,\ k \in K \\
& R_k \geq 0 && \forall k \in K
\end{aligned}
\tag{Efficient}
$$

Proposition F.8. The inefficient formulation of WFP is *not* a *constructive reformulation* of the efficient formulation.

Proof. The two formulations have misaligned objectives. In the efficient formulation, the procurement cost term $\sum_k \mathrm{pc}_k \sum_{j\in N_B} \mathrm{dem}_j R_k$ depends only on the ration size R_k, while the demand constraint $\sum_i E_{ij} F_{ijk} \geq \mathrm{dem}_j R_k$ permits shipping in excess of $\mathrm{dem}_j R_k$ without additional procurement penalty. In the inefficient formulation, the procurement cost $\sum_k q_k \sum_p x_{pk}$ instead depends on the total amount shipped, so any slack in the demand constraint is charged. Since the objectives disagree on corresponding feasible points under the natural variable mapping, the inefficient formulation is not a reformulation of the efficient one. □

Notably, this objective misalignment does not hold at optimal solutions where no excess demand is shipped. Thus, the inefficient formulation of WFP is still an *Audet reformulation* of the efficient formulation.

G Experiment Details

All experiments were conducted on a MacBook Pro (Apple M3 Pro, 12-core CPU, 18 GB unified memory) running macOS 15.7.1. For FLARE, we use Lean 4.28.0 and the following agent harnesses.

- **Claude Code.** Version 2.1.140 with Claude Code Max subscription ($200/month)
- **Codex.** Version 0.130.0 with ChatGPT Pro subscription ($100/month)
- **OpenCode.** Version 1.14.45 (open-source)

We used Claude Code and ChatGPT subscriptions in order to avoid higher API costs. During development, we consumed 100% of the weekly quota. Furthermore, we benchmarked FLARE in batches to avoid hitting the session limits.

[1] In the dataset, the variable q_k is used for procurement cost in the inefficient formulation. We use pc_k for both formulations.

All the results in Section 6 and Appendix H specify the LLM model and reasoning effort used. With the exception of Table 5, reasoning is enabled for all models in every experiment. We specify a max token budget of 8192 when reasoning is disabled and 16384 when reasoning is enabled, regardless of effort level. We use high effort for Anthropic models (Opus 4.7 and Sonnet 4.6), medium effort for OpenAI models (GPT-5.4 and GPT 5.5), and high effort for DeepSeek models (DeepSeek V4 Pro and DeepSeek V4 Flash). We found these settings produce comparable reasoning efforts across model families. DeepSeek models reason for longer, but *high* is the lowest reasoning effort available.

H Additional Results

Table 5: Evaluation of FLARE-NL across LLM models and reasoning effort levels. Results show FLARE-NL evaluated on the *FormulationBench* dataset TSP problem. Metric cells report mean ± std. dev. across 3 runs. Broadly, frontier models with reasoning enabled dominate. To reduce computational costs, we only evaluate these models in the ablation study (Table 3).

Model	Effort	TP	FP	TN	FN	Precision	Recall	Accuracy	Avg Time	Avg Cost
Reasoning Disabled										
Opus 4.7		15	1	8	0	94.4±9.6%	**100.0±0.0%**	95.8±7.2%	10.3s	$0.031
Sonnet 4.6		7	1	8	8	88.9±19.2%	46.7±11.5%	62.5±12.5%	19.8s	$0.024
GPT-5.4		3	1	8	12	50.0±50.0%	20.0±20.0%	45.8±14.4%	9.2s	$0.013
GPT-5.5		7	0	9	8	**100.0±0.0%**	46.7±11.5%	66.7±7.2%	11.8s	$0.024
GPT-4.1		2	4	5	13	22.2±38.5%	13.3±23.1%	29.2±19.1%	6.5s	$0.007
DeepSeek V4 Pro		4	1	8	11	88.9±19.2%	26.7±11.5%	50.0±0.0%	8.6s	$0.005
DeepSeek V4 Flash		3	2	7	12	55.6±50.9%	20.0±20.0%	41.7±14.4%	3.1s	$0.0003
Reasoning Enabled										
Opus 4.7	high	15	0	9	0	**100.0±0.0%**	**100.0±0.0%**	**100.0±0.0%**	15.5s	$0.052
Sonnet 4.6	high	15	0	9	0	**100.0±0.0%**	**100.0±0.0%**	**100.0±0.0%**	97.7s	$0.121
GPT-5.5	medium	15	0	9	0	**100.0±0.0%**	**100.0±0.0%**	**100.0±0.0%**	44.6s	$0.082
GPT-5.4	medium	15	2	7	0	88.9±9.6%	**100.0±0.0%**	91.7±7.2%	49.3s	$0.048
DeepSeek V4 Pro	high	15	4	5	0	79.4±6.9%	**100.0±0.0%**	83.3±7.2%	141.5s	$0.020
DeepSeek V4 Flash	high	13	2	7	2	88.9±9.6%	86.7±23.1%	83.3±7.2%	47.5s	$0.001

Table 6: Evaluation of FLARE across agent harnesses and LLM models. Accuracy results on the *FormulationBench* dataset are provided in aggregate and segmented by source dataset. Metric cells report results from a single run and **Bold** indicates the best method on a metric. The Claude Code harness with Opus 4.7 achieves the best accuracy and is cheaper than Codex with GPT 5.5. The open-source OpenCode harness with DeepSeek V4 Pro has competitive accuracy and cheaper cost, but takes substantially longer.

Harness	Model	Effort	TP	FP	TN	FN	Precision	Recall	Accuracy	Avg. Time	Avg. Cost
Overall (p1–20)											
Claude Code	Opus 4.7	high	67	0	26	3	**100.0%**	**95.7%**	**96.9%**	561.6s	$2.334
Codex	GPT-5.5	medium	66	0	26	4	**100.0%**	94.3%	95.8%	439.3s	$3.646
OpenCode	DeepSeek V4 Pro	high	60	0	26	10	**100.0%**	85.7%	89.6%	2331.8s	$0.909
EquivaFormulation [68] (p1–5)											
Claude Code	Opus 4.7	high	25	0	20	0	**100.0%**	**100.0%**	**100.0%**	328.6s	$1.180
Codex	GPT-5.5	medium	25	0	20	0	**100.0%**	**100.0%**	**100.0%**	285.8s	$2.116
OpenCode	DeepSeek V4 Pro	high	25	0	20	0	**100.0%**	**100.0%**	**100.0%**	572.5s	$0.160
EvoCut [66] (p6–12)											
Claude Code	Opus 4.7	high	38	0	5	0	**100.0%**	**100.0%**	**100.0%**	796.5s	$3.495
Codex	GPT-5.5	medium	35	0	5	3	**100.0%**	92.1%	93.0%	619.0s	$5.043
OpenCode	DeepSeek V4 Pro	high	32	0	5	6	**100.0%**	84.2%	86.0%	3913.6s	$1.542
Ferchtandiker [16] (p13–20)											
Claude Code	Opus 4.7	high	4	0	1	3	**100.0%**	57.1%	62.5%	609.4s	$2.581
Codex	GPT-5.5	medium	6	0	1	1	**100.0%**	**85.7%**	**87.5%**	336.6s	$4.741
OpenCode	DeepSeek V4 Pro	high	3	0	1	4	**100.0%**	42.9%	50.0%	3725.7s	$1.725