FormulaOne: Measuring the Depth of Algorithmic Reasoning Beyond Competitive Programming

Gal Beniamini Yuval Dor Alon Vinnikov Shir Granot Peled Or Weinstein
Or Sharir Noam Wies Tomer Nussbaum Ido Ben Shaul Tomer Zekharya
Yoav Levine Shai Shalev-Shwartz Amnon Shashua

AAI

Abstract

Frontier AI models demonstrate formidable breadth of knowledge. But how close are they to true human — or superhuman — expertise? Genuine experts can tackle the hardest problems and push the boundaries of scientific understanding. To illuminate the limits of frontier model capabilities, we turn away from contrived competitive programming puzzles, and instead focus on real-life research problems.

We construct FormulaOne, a benchmark that lies at the intersection of graph theory, logic, and algorithms, all well within the training distribution of frontier models. Our problems are incredibly demanding, requiring an array of reasoning steps, involving topological and geometric insight, mathematical knowledge, combinatorial considerations, precise implementation, and more. The dataset has three key properties. First, it is of commercial interest and relates to practical large-scale optimisation problems, such as those arising in routing, scheduling, and network design. Second, it is generated from the highly expressive framework of Monadic Second-Order (MSO) logic on graphs, paving the way toward automatic problem generation at scale — ideal for building RL environments. Third, many of our problems are intimately related to the frontier of theoretical computer science, and to central conjectures therein, such as the Strong Exponential Time Hypothesis (SETH). As such, any significant algorithmic progress on our dataset, beyond known results, could carry profound theoretical implications.

Remarkably, state-of-the-art models like OpenAI's o3 fail entirely on FormulaOne, solving less than 1% of the questions, even when given 10 attempts and explanatory fewshot examples — highlighting how far they remain from expert-level understanding in some domains. To support further research, we additionally curate FormulaOne-Warmup, offering a set of simpler tasks, from the same distribution. We release the full corpus along with a comprehensive evaluation framework.

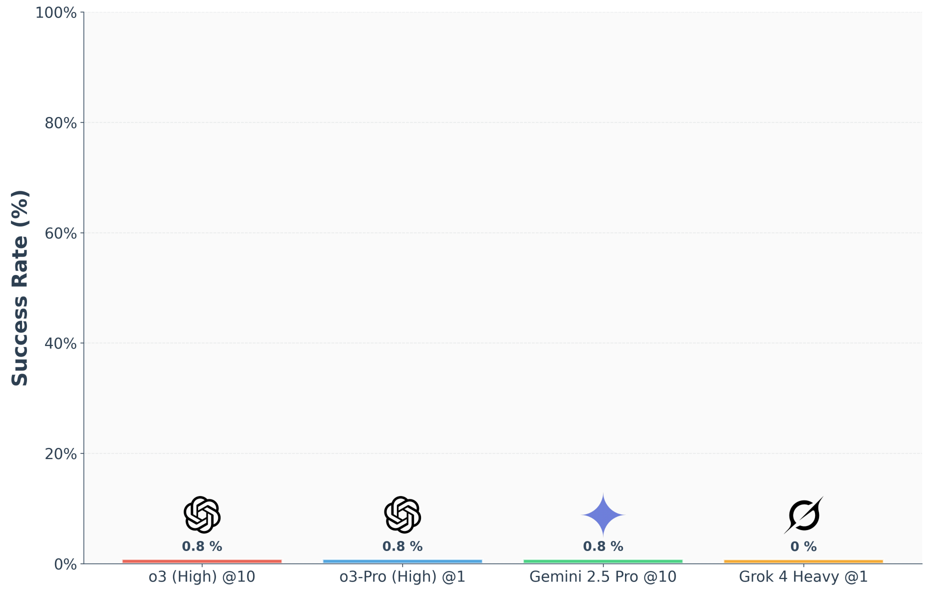


Figure 1: Performance of frontier reasoning models on the FormulaOne dataset.

1 Introduction

Artificial Intelligence (AI) holds the promise of solving the world's hardest scientific, algorithmic, and mathematical challenges—problems so complex they baffle even the brightest human minds. Current benchmarks, however, often do not paint a complete picture of AI's depth of understanding. While recent achievements are remarkable, such as OpenAI-o3 attaining a 2,724 rating on CodeForces or securing a gold medal at the International Olympiad in Informatics [EWS$^+$25], they nevertheless mask a sobering reality: the skills honed for these competitions do not capture the full spectrum of reasoning needed for large-scale, real-world research problems. Tasks such as optimising global supply chains, managing large-scale power grids, and designing resilient network infrastructures are orders of magnitude harder, requiring algorithmic insight that goes far beyond the scope of typical competitive programming.

To this end, we introduce **FormulaOne**, a benchmark centred around dynamic programming over graphs—an algorithmic cornerstone of real-world optimisation. Our framework is constructed in a principled, semi-mechanistic manner based on Monadic Second-Order (MSO) logic, a formal logic on graphs. Its theoretical foundation is an algorithmic meta-theorem due to Courcelle [Cou90], which guarantees that a vast class of problems defined using this formal logic can be solved efficiently for graphs that have a 'tree-like' structure. This allows us to generate a large and conceptually diverse corpus of mathematically deep problems, each guaranteed to have an efficient solution, yet potentially being extraordinarily challenging to discover in practice.

The problems in FormulaOne are designed to serve as a "new ARC" (c.f., [Cho19, CKKL24, CKK$^+$25]) for mathematical reasoning, demanding a synthesis of skills from topological and geometric insight, to knowledge of graph theory, and the need for precise implementation. While ARC is designed to measure fluid intelligence by evaluating performance on tasks that are explicitly out-of-distribution (OOD) relative to the training examples, FormulaOne presents a challenge that is, by design, entirely in-distribution. Every problem, from the simplest to the most complex, is generated from the same family: MSO logic on graphs. Thus our dataset consists entirely of algorithmic coding problems, a task on which frontier reasoning models should, by rights, perform well. Nevertheless, even the best frontier reasoning models, which excel at human-level competitions such as OpenAI's o3, completely fail on our dataset, achieving a stark $<$1% success rate.

One of the primary characteristics of the problems appearing in FormulaOne, is the amount of reasoning required. In Appendix A we exemplify this by proving a solution, in full, to one such problem. Astonishingly, there are no fewer than 15 interdependent, highly complex mathematical reasoning steps, all intertwined in non-obvious ways. We conjecture that this reasoning depth, typical of cutting edge real world research problems, is the main characteristic due to which frontier AI models "flat-line" on FormulaOne. This grim result highlights a pressing need for deeper reasoning environments and better benchmarks, capturing increasing levels of complexity, and perhaps necessitating a more structured approach. To support further research, we will release the full dataset along with a comprehensive evaluation framework. We believe this provides a solid foundation to guide and measure future progress in advanced algorithmic reasoning.

Another key characteristic of FormulaOne is its profound connections to the frontier of theoretical computer science and central conjectures therein. A prime example is the Strong Exponential Time Hypothesis (SETH), a foundational conjecture in fine-grained complexity. Informally, SETH posits that the classic brute-force search algorithm for the Boolean Satisfiability (SAT) problem is essentially optimal, meaning no algorithm can provide a significant exponential speedup. The time complexity of many core graph problems, including several in our dataset, is believed to be optimal under SETH. That is, no algorithm can solve them faster than a particular known lower bound, parameterised by the tree-likeness of the input graph. Therefore, if a powerful AI agent were able to discover a genuinely novel, faster algorithm for one of these hard tasks, it would do more than just solve a puzzle; it would effectively refute a central hypothesis in theoretical computer science.

The semi-mechanistic nature of our problem generation provides the first steps towards an essentially unbounded source of high-depth algorithmic challenges, ideal for building next-generation environments for Reinforcement Learning with Verifiable Rewards (RLVR). Existing RLVR benchmarks are often limited in one of two ways: they either feature problems with low conceptual depth, such as school-level mathematics [CKB$^+$21, HBK$^+$21], or very simple programming tasks [AON$^+$21], or the datasets are more challenging but static and limited in size [IHI$^+$25]. Our framework of problems derived from MSO logic addresses both points: it can provide a virtually infinite stream of problems with profound mathematical depth, for which the solutions are nevertheless automatically verifiable. This combination of unbounded difficulty and guaranteed correctness is critical for training agents to tackle genuinely open-ended scientific discovery.

1.1 Related Work

Recent years have seen a proliferation of benchmarks at the intersection of mathematics, coding, and reasoning. We group our discussion of this prior work into four main areas.

Algorithmic Coding Benchmarks. Several benchmarks have been developed to measure the algorithmic problem-solving abilities of AI systems. ALE-Bench [IHI+25] introduced over 40 hard optimisation tasks from AtCoder contests, focusing on iterative solution refinement. Others, like CodeElo [QYY+25] and LiveCodeBench [JHG+24], provide rigorous evaluation on live contest problems, assigning models human-comparable Elo ratings in authentic execution environments. However, top models are rapidly approaching the performance ceiling on these platforms; OpenAI's o3 model, for instance, has achieved an Elo rating above 2700 on Codeforces [QYY+25], indicating this domain is becoming saturated. In contrast, FormulaOne provides a challenge that is harder to saturate, as its difficulty stems not from eclectic puzzle design but from the profound reasoning depth characteristic of substantive research problems.

Out-of-Distribution Reasoning. The Abstraction and Reasoning Corpus (ARC) [Cho19, CKKL24] assesses generalisation to explicitly out-of-distribution (OOD) tasks. It presents visual puzzles that require an agent to infer a transformation from a handful of pixel-grid examples with no linguistic hints. Our work takes a complementary direction, probing deep *interpolation* within a domain — algorithmic programming — the 'bread and butter' of modern reasoning models' training. The failure of these models on our benchmark is therefore particularly notable: it demonstrates that even within a familiar domain, they cannot yet perform the multi-step reasoning that is required in order to successfully tackle such problems.

Frontier Knowledge Benchmarks. Other benchmarks test the limits of broad, specialised knowledge. Humanity's Last Exam (HLE) [PGH+25] assembles thousands of graduate-level questions from dozens of academic disciplines, while FrontierMath [GEB+24] collects challenging research-level problems in mathematics. These benchmarks consist of fixed, static problem sets, whereas our approach allows for semi-automatic generation of problems, in an unbounded domain (MSO). Furthermore, our family of problems is incredibly well-suited to the creation of powerful RLVR environments, with a variety of granular reward signals available, which prior datasets lack.

AI Pushing the Frontier of Theory. Very recently, there has been a push for AI-assisted discovery of new algorithms. Systems like AlphaTensor [FBH+22] and AlphaEvolve [NVE+25] have used deep reinforcement learning to find faster algorithms for specific cases of matrix multiplication, while AlphaDev [MMZ+23] found marginal improvements for sorting routines. These projects are typically structured as closed-loop discovery processes focused on a handful of human-selected challenges. Critically, their discoveries, while impressive, are conceptually distant from improving theoretical bounds, for instance, making progress on the exponent of matrix multiplication, ω. Our work contributes to this area by providing an open-ended suite of problems where discovering a faster-than-previously-known algorithm could have genuine theoretical consequences. Since many of our challenges are related to conjectures like the Strong Exponential Time Hypothesis (SETH), a truly novel solution would be of huge theoretical significance.

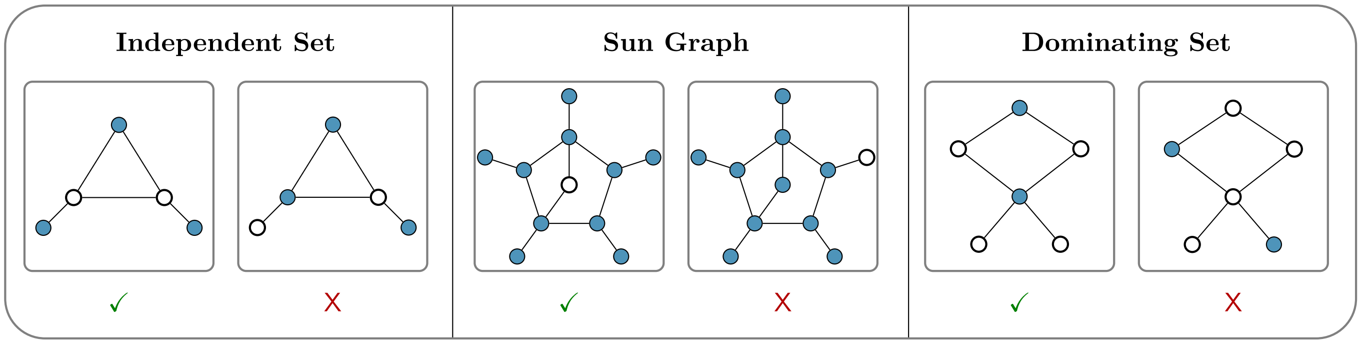


Figure 2: Three problems, expressible by MSO logic, solvable by dynamic programming on tree-like graphs. In each, the left-hand-side showcases a subgraph exhibiting a given property, whereas the right-hand-side showcases one that does not. Filled vertices, marked blue, denote the subset over which the graph is induced.

2 Dataset

We introduce **FormulaOne**, a dataset consisting of a wide range of dynamic programming problems on graphs. At the core of our work is the semi-automatic generation of a large corpus of mathematically deep algorithmic problems, designed to gauge the command of abstract problem-solving, multi-step combinatorial reasoning, and practical implementation. Its theoretical foundation is rooted in an *algorithmic meta-theorem*, due to Courcelle [Cou90], which broadly states:

> "For every *sufficiently tree-like graph*, any problem definable in an expressive formal logic — Monadic Second-Order (MSO) logic — can be solved by a dynamic programming algorithm that operates in time *linear* in the order of the graph."

Our problems all belong to a single underlying distribution: properties expressible through MSO logic. The number of properties definable in Monadic Second Order Logic is, in principle, unlimited. Indeed, this family is incredibly diverse; covering both well-known classical problems, such as 3-colourability or counting maximal independent sets, and entirely novel problems, including local invariants and topological structure (see Figure 2). Concretely, we generate problems of the following form.

`Problem #44`

Input: A tree-like graph $G = (V, E)$, a tree decomposition $\mathcal{T}$ of G, and a weight function $w : V \to \mathbb{N}$.

Objective: Compute the sum of all weights of sets $S \subseteq V$ such that:

The graph $G[S]$, induced[1] over S, does not contain any cycle of length four.

Notation: The weight of a set of vertices S is defined as $w(S) := \sum_{v \in S} w(v)$.

[1]The induced subgraph $G[S]$ consists of all vertices in S and all edges of G with both endpoints in S.

In this context, a tree decomposition $\mathcal{T}$ is an auxiliary structure which enables for an efficient dynamic programming algorithm to operate over the input graph. A detailed exposition is deferred to Section 3.

2.1 Guiding Principles

At the heart of FormulaOne lie three core principles.

1. **A "New ARC" for Mathematical Reasoning.** Our dataset consists of algorithmic coding problems requiring deep mathematical reasoning. Indeed, several of our problems directly relate to research papers at the forefront of complexity theory [KM24, FLR$^+$12]. The domain of algorithmic coding has become the core benchmark for measuring frontier reasoning models' progress. For example, recently OpenAI's o3 reportedly ranked 175th on CodeForces [EWS$^+$25], relative to all human competitors.

However, while a human expert of that calibre should, by rights, be able to score highly on our problems, we find that frontier models do not — achieving a <1% success rate on our hard problems (see Section 5).

2. **An Unbounded, Mathematically Deep Algorithmic Environment for RLVR.** The dynamic programming problems generated through the use of Monadic Second-Order Logic provide an essentially unbounded source of algorithmic problems, of incredible substance (see Section 3). We believe our work provides the first steps towards enabling a truly high-depth environment for Reinforcement Learning with Verifiable Rewards (RLVR), finally moving research beyond existing datasets.

3. **Pushing the Boundaries of Complexity Theory.** The complexity of problems expressible through MSO logic on tree-like graphs is intimately related to the Strong Exponential Time Hypothesis (c.f. [LMS11, CFK+15]), a central conjecture in the field of fine-grained complexity [CIP09]. Concretely, the best-known time complexity of a large portion of our dataset is, in fact, optimal under SETH. Therefore, any *significant* algorithmic progress on our dataset could carry profound theoretical implications.

2.2 The dataset

We introduce two datasets: a core dataset for benchmarking AI performance, and an auxiliary dataset for research and evaluation.

1. **FormulaOne.** We introduce FormulaOne, a dataset of 120 challenging dynamic programming problems that evaluate creativity, sophistication, and expert-level reasoning.

2. **FormulaOne-Warmup.** To facilitate research and evaluation in this demanding setting, we also provide FormulaOne-Warmup, an auxiliary dataset containing 100 simpler problems.

In Section 5, we provide an extensive evaluation of all top frontier reasoning models on our datasets.

Problem Formulation. Our problems are defined using MSO formulas. An MSO formula is a statement in a formal logic used to express properties of graphs. In this logic, one can define conditions on graphs, including by quantification over vertices, edges, and sets of vertices and edges ('second-order'). Each problem in our dataset is defined by a *unary open formula* in this logic — a statement with a single free variable that acts as a placeholder for an input set, which is either a set of vertices, or a set of edges. To help make matters concrete, let us consider a problem description in full.

```
## Description

Task Type: Weighted Model Count (CSP-wMC).
Explanation:
    You are given a set of elements, where each element has a weight. Your goal is to compute the sum of weights of all
    the subsets that satisfy the constraints given below, modulo 10^9 + 7.
    If there is no feasible set, the result of this task is -1.
Constraints Definition:
    In this question, the elements are the vertices of a graph, and the constraints are defined in graph-theoretic terms.
    A description of the graph family, the constraints, and the vertex weights is given below.
    The constraints:
        A set of vertices such that the graph induced over these vertices contains no induced square (C4). An induced
        C4 consists of 4 vertices that form a cycle of length 4, with no additional edges between them.
    Input Graph:
        A graph whose tree-width is at most 3, and in whose 'nice tree decomposition' the JOIN nodes have width at most
        2 (i.e., every JOIN node contains at most 3 vertices).
        The following variables are used to represent graphs in this family:
            - `graph`: The graph.
            - `n`: The number of vertices in the graph.
            - `tree_decomposition`: The tree decomposition of the graph.
    Vertex Weights:
        Every vertex is assigned an integer weight. Each weight is between 1 and 10^5, inclusive. The list of weights
        is given in the variable `x_s`.

## Input

The input is composed of the following items, separated by newlines.
```

```
- n:
    An integer written on a single line.

- x_s:
    The first line contains a single integer representing the length of the list.
    The second line contains a list of space-separated integers.

- graph:
    An undirected graph over the vertex set of non-negative integers (starting at zero).
    The first line contains a single integer -- the number of vertices in the graph.
    The second line contains a single integer -- the number of edges in the graph.
    Every edge of the graph is then printed on a single line, as two space-separated integers.

- tree_decomposition:
    A tree decomposition of a graph.
    The first line contains a single integer -- the number of bags in the tree decomposition.
    The second line contains a single integer -- the number of edges in the tree decomposition.
    Then, for every bag in the decomposition, there is a single line of space-separated integers, representing the
    vertices in the bag (the i-th line corresponds to the i-th bag).
    Finally, every edge joining two bags of the tree decomposition is printed on a single line, as two space-separated
    integers, in ascending order. Note that the bags of the tree decomposition are indexed by non-negative integers,
    starting at zero.

## Output

A single integer: the sum of weights of all subsets that satisfy the constraints, modulo 10^9 + 7, or -1 if no such
subset exists.

## Constraints

The following constraints must all be satisfied:

n >= 4
n <= 94

## Time Limit

100.00 seconds per test.
```

As shown above, an input instance to a problem consists of a graph, a tree-decomposition[1] of said graph, and a weight function. If the formula's variable is a vertex set, weights are defined over the vertices; otherwise, they are defined over the edges. The core objective for each problem in the dataset we release is Weighted Model Counting (WMC) — the goal is to compute the total weight of all subsets that satisfy the given formula, where the weight of a set S is the sum of all weights of elements in S, according to the given weight function. Since this number may be large[2], we require only its remainder modulo the safe prime $10^9 + 7$.

2.3 The Tip of the Iceberg – Towards Scalable Problem Generation

The current dataset, while comprehensive, represents just a *fraction* of what is expressible and solvable within the framework of MSO-based dynamic programming over graphs. Let us mention, in passing, why.

- **Logical Expressiveness:** The vast majority of our formulas can be formulated in a subset of MSO logic, known as MSO_1. This leaves out well-known broader classes, such as MSO_2 and myriad other extensions of MSO, for which results similar to Courcelle's [Cou90] hold.
- **Objective Variety:** We are releasing problems with WMC objectives only. However, the same logical framework can be applied to optimisation problems (e.g., finding the minimum or maximum weight of a satisfying set) and other counting variants, which we do not include in this release. Such problems are qualitatively different and can be solved with state designs that are meaningfully different from their WMC counterparts (for instance, they do not require a 'unique witness').
- **Multi-Pass Traversal:** In our evaluation we allow for a single post-order traversal over the tree decomposition. However, it is known (e.g., [FHMW21]) that allowing multiple passes or other traversal strategies may yield substantial performance gains. Such approaches can simplify the required DP state and lead to alternative state representation and improvement to the overall running time.

[1]See section 3.1
[2]Indeed, the truth set may be exponentially large in the order of the graph.

- **Base Graphs and Tree Decompositions:** Currently, we provide both a graph and a corresponding tree decomposition. This can be extended in two ways: by varying the families of input graphs, and by requiring the model to generate the tree decomposition itself.[3] We remark that while there is a linear-time algorithm for finding tree decompositions of tree-like graphs [Bod93], the algorithm is not practical. Instead, for any *particular* family of graphs (such as cactii, or k-trees), one can craft an *efficient and practical* linear-time algorithm. Furthermore, there are non-trivial interactions between the structure of the input graph and the MSO formula. For example, if a problem canonically requires tracking information about odd-length cycles, this part of the DP state becomes redundant when the ambient graph is known to be bipartite. Recognising such properties allows for highly non-obvious state compression, which is *crucial* for efficient solutions.

- **Beyond Treewidth:** This release is predicated on tree decompositions and the *treewidth* parameter. However, the underlying principle of Courcelle's theorem extends to many other graph parameters, such as *pathwidth* [RS83], and *clique-width* [CER93, Wan94], and in certain cases, *tree-depth* [BDJ+98].

3 Complexity and Nuance in MSO-Based Dynamic Programming

To properly contextualise the challenges and subtleties that arise in tackling the problems within our dataset, we begin with a high-level "crash-course" overview of the methodology for dynamic programming on tree decompositions. These topics are presented briefly, intuitively and mostly, visually. We then quickly proceed to give a guided tour of the complexities inherent in our problems.

We remark that, despite their innocuous appearance, our MSO-based problems are complex and multifaceted. They demand combinatorial reasoning, geometric insights arising from the structure of the tree decomposition, exact logic, and a careful implementation. Moreover, often these problems are riddled with *subtle pitfalls*, all of which must be carefully sidestepped when implementing the dynamic programming solution. The intent of this section is to showcase these intricacies.

3.1 Dynamic Programming on Tree-Like Graphs

Treewidth. The methodology for solving these otherwise intractable problems relies heavily on the "tree-likeness" of the given graph, formally known as *treewidth* (c.f. [BB73, Hal76, RS84, Bod88]). Loosely speaking, a graph of low treewidth locally resembles a tree. Any tree-like graph can be decomposed into a collection of small sets of vertices, "bags", which are themselves organised into a tree structure. This construction, known as a *tree decomposition*, provides a roadmap that allows an algorithm to process the entire graph by traversing this tree of bags.

Each bag acts as a small, *local* "window" into the graph. For the traversal process to be sound, every vertex is handled across a contiguous region of the decomposition before it is ultimately processed and removed from view. The tree decomposition is also carefully constructed to ensure that every vertex and edge of the graph is eventually processed. With each step, the view changes only slightly and predictably: a single vertex might be introduced into the window, forgotten from it, or two previously separate parts of the graph are joined (merged) within the view (see Figure 3).

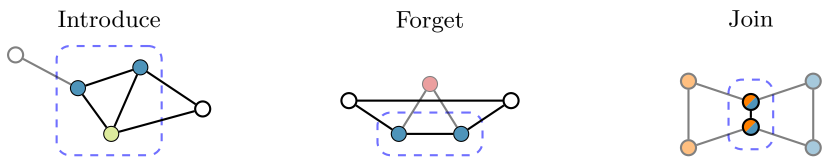


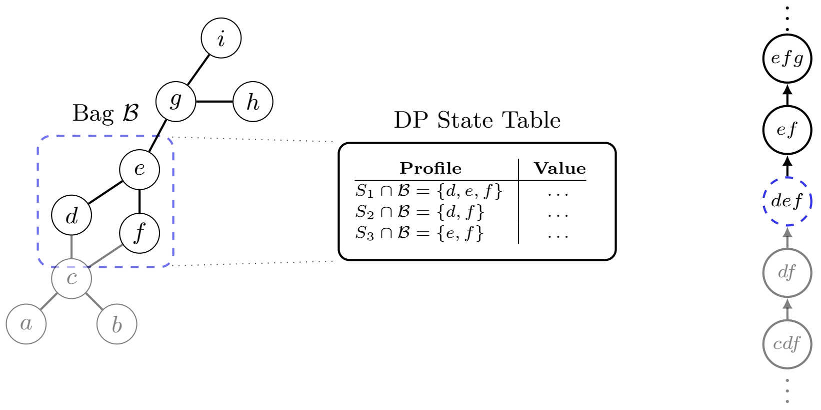
Figure 3: Local modifications to bags (dashed boxes). On the left, a vertex (green) is added ("introduced"), in the centre, a vertex (red) is removed ("forgotten"), and on the right, two previously processed graphs (orange, blue) meet at a bag ("joined"). Already forgotten vertices are semi-transparent. Vertices in the current bag, and those yet to have been processed, are opaque.

[3]In preliminary experiments on a small selection of problems, we found that prompting models to perform both the tree decomposition and the subsequent dynamic programming resulted in complete failure (0% success rate).

The *treewidth* of a graph simply corresponds to the size of the largest window needed for this process. A low treewidth guarantees that our perspective is always limited to a small number of vertices at any given time, making complex problems tractable.

Dynamic Programming. The algorithm that solves problems on graphs of low treewidth is a form of dynamic programming, wherein we traverse the tree decomposition of a given graph in post-order, considering a single bag at a time. In this context, a partial solution refers to a solution to the problem at hand, restricted only to the portion of the graph processed so far, i.e., the subgraph induced by the vertices appearing in the bags that are descendants of the current bag. Often, partial solutions are grouped within a bag according to their *profile*, which acts as a summary for the way in which they interact with the bag (see Figure 4).

The key lies in what information we store for each bag — the "state". The art of state design is in crafting a good profile that summarises the properties of a *partial solution*, as viewed from the vantage point of a bag. Such a state must be rich enough so as to allow it to be updated as we transition from one bag to the next — whether introducing or removing a vertex, or merging two views — yet concise enough to be computationally tractable. To design such a state, one must answer: "What information do we need to know about the past and present, to make valid choices for the future?".



Profile	Value
$S_1 \cap \mathcal{B} = \{d, e, f\}$	...
$S_2 \cap \mathcal{B} = \{d, f\}$	...
$S_3 \cap \mathcal{B} = \{e, f\}$	...

Figure 4: Snapshot of a dynamic programming algorithm, traversing over a tree decomposition. The current bag is encircled in blue. **Left:** The input graph G. Previously processed vertices and edges drawn semi-transparent. The current bag acts as a local window onto the vertices d, e and f. **Centre:** The DP table associated with the current bag lists all possible *profiles* — distinct ways in which *partial solutions* (here, S_1, S_2 and S_3) can interact with the bag. Each row represents a grouping of partial solutions by their profile, and the corresponding value stores an aggregate measure over that group (e.g., maximum weight or total count). **Right:** The tree decomposition of G. The traversal proceeds bottom-up, aggregating information from subtrees until reaching the root, where a solution over the entire graph G is extracted. In this case, the tree decomposition shown is linear (a path), that is, it contains no join bags (where subtrees merge).

3.2 A very easy problem

Before we jump into the deep end, let us dip our toes into the water through a very easy, classical problem.

`Dominating Set`[1]

Objective: Compute the sum of all weights of sets $S \subseteq V$ such that:

Every vertex outside the set is adjacent, in G, to at least one vertex in the set.

[1]Hereafter, we omit the 'Input' specification, as it is identical to the entry listed under Section 2.

To tackle the dominating set problem we consider the different ways in which partial solutions, i.e., dominating sets within the subgraph traversed so far, may interact with the vertices in the bag. The following DP state turns out to be sufficient: for every vertex $v \in \mathcal{B}$ in the current bag $\mathcal{B}$, store one of the following three configurations,

- IN: The vertex is in the emerging dominating set.
- OUT-DOMINATED: The vertex is *not* in the emerging dominating set, but a previously seen vertex that is in the set, dominates it.
- OUT-NEEDS-DOMINATION: The vertex is *not* in the emerging dominating set, and is as of yet not dominated by any vertex chosen to participate in the set.

We stress that not every configuration of an emerging set may lead to a valid dominating set. For instance, if a vertex is forgotten from the bag while in the status of OUT-NEEDS-DOMINATION, then the emerging set corresponding to that state is *not* dominating, and must be invalidated. Notice that the aforementioned state aims to store the *minimum amount of information* necessary in order to identify the way in which the emerging dominating set interacts with the bag, and allows us to maintain the state as we traverse the graph. For instance, we do not store irrelevant details, such as the identity of the vertices' dominators. We provide a full solution to the dominating set problem, written in Python, here.

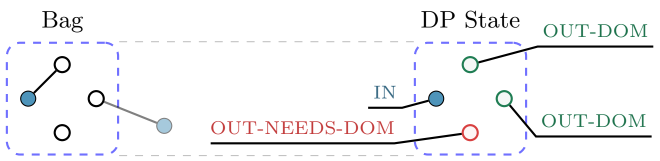


Figure 5: Partial solutions as viewed through a bag, and the corresponding DP state. Vertices in the dominating set S are filled blue. *Semi-transparent* vertices have already been "forgotten". The top vertex is dominated by a neighboring vertex in the set and currently in the bag. The right vertex is dominated by a previously processed vertex in the set and outside the bag. The bottom vertex is not yet dominated.

3.3 A less innocent problem

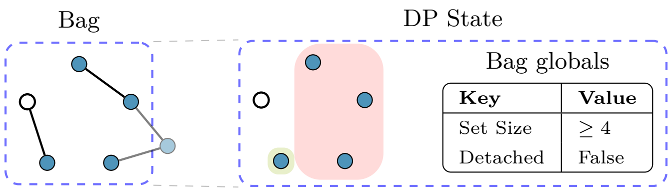
We can now begin to highlight the intricacies through a "simple" representative problem.

<u>`Connected with at least k vertices`</u>

Objective: Compute the sum of all weights of sets $S \subseteq V$ such that:

The graph $G[S]$, induced over S, is connected and contains at least $k = 4$ vertices.

On the face of it, the state one must store within the bag may appear relatively straightforward — a counter for the size of the emerging set, and a running connectivity profile, i.e., a partitioning of the vertices currently within the bag, indicating connectivity with respect to the graph processed thus far.



Key	**Value**
Set Size	≥ 4
Detached	False

Figure 6: A bag and its state. Vertices in the set S are filled blue. *Semi-transparent* vertices have already been "forgotten". The connectivity partition has two classes (red, green) within the bag, in part owing to edges to vertices already forgotten, and the partial solution contains at least four vertices. Furthermore, there is no connected component of selected vertices, that has fully detached from the bag.

So, how may one *get the solution wrong*? Let us count the ways.[4]

Premature invalidation of states. A natural mistake arises when prematurely discarding DP states. In this case, within the current bag, an emerging solution may appear to be disconnected — but one must take into account the possibility that *currently* disjoint components *may merge* into a single connected component *through yet-unseen vertices*, that will be processed at a later stage (i.e., introduced in future bags).

Failure to cap solution cardinality. Maintaining a minimal state profile requires that once the partial solution reaches the minimal size threshold, $k = 4$, the DP state no longer tracks *exact* counts, but rather merely maintains that the threshold has been met. Failing to do so results in a state-space explosion – inflating the running time complexity from linear to cubic[5], due to redundantly tracking detailed counts.

Tracking external connected components. During the traversal of the tree decomposition, there may arise partial solutions that *no longer intersect the current bag*, but nevertheless represent a valid connected subgraph. To handle these cases correctly, a single bit must track whether such an external connected component exists, to be used to invalidate states in two situations: (a) If the intersection of the selected set and the bag is nonempty, we disqualify the state as this bag's vertices are forever disconnected from the external connected component. (b) During a join operation, if the two joined states both have an external connected component, joining them would mean joining two distinct connected components, resulting in a disconnected graph. This is a crucial detail that is easy to overlook.

Avoiding double-counting during joins. When merging states at a join node, one must carefully avoid double-counting vertices contained within the intersection (the bag itself). Naively summing the vertex counts from both states is incorrect; after all, vertices in the intersection must be counted exactly once. Additional subtle errors may occur when handling capped arithmetic, especially when transitioning between capped and uncapped counting schemes.

Careful merging of connectivity partitions. At the join nodes partial solutions from two distinct subtrees are merged. While it is essential that the subset of vertices selected for the emerging set within the intersection bag *matches exactly* between merged states, their connectivity partitions *might differ*. Correctly merging these partitions (by unifying any two sets that share a vertex between the two partitions) should be done efficiently, for instance, by using a DSU data-structure.

3.4 Complex Profiles

Often, even thinking of a valid representation of a state is a challenging task. Here is one such case.

`Cograph`

Objective: Compute the sum of all weights of sets $S \subseteq V$ such that:

The graph $G[S]$, induced over S, contains no induced path of length 3.[1]

[1]The length of a path is the number of edges it contains.

This problem illustrates a more geometric complexity in the state representation. We would like to preserve a state that guarantees that, within the emerging set, there is no induced path of length 3. What sort of state should we keep? To adequately answer, we must first understand the different ways in which a path of length 3 may be *perceived* through a sequence of bags, as we traverse over a tree decomposition.

[4]These listed errors were *all* made by frontier reasoning models, including Google's Gemini 2.5, and OpenAI's o3.

[5]The cubic runtime follows since the number of states per bag may now be linear in the order of the graph. Joining two such states naively requires quadratic time, and may happen $\Theta(n)$ times.

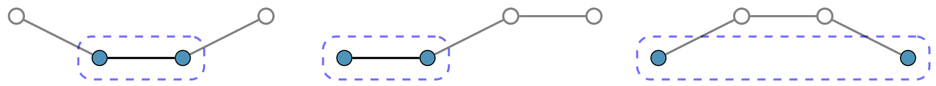


Figure 7: A few different bags of cardinality 2, encircled in blue, that are possible when traversing over P_4, a path with four vertices. Semi-transparent vertices had already been forgotten.

As it happens, one can indeed be convinced that no induced path of length 3 exists, by keeping track of the following:

1. OUT: Indicating that the given vertex within the bag does not participate in the emerging set.
2. IN-PAIR-AND-HAVE-COMMON-FORGOTTEN-IN-NEIGH: For every *pair* of vertices within the bag that are selected to participate in the emerging set (IN), whether they share at least one common forgotten selected neighbor.
3. IN-AND-FORGOTTEN-IN-NEIGH: For each selected vertex, record whether or not it has had a forgotten selected neighbor, so far.
4. IN-AND-TWO-FORGOTTEN-IN-NEIGHS: For each selected vertex, record whether or not there is a length-2 path leading up to it, consisting of two forgotten selected vertices.

Preserving this state is no easier. At join nodes, complications emerge, as spurious induced paths of length three can arise from combining subtrees. For instance, such paths can form when merging bags where a vertex from one subtree is an endpoint of a length-3 path, and simultaneously has a forgotten neighbor in the second subtree. Similarly, an induced path can emerge when there is a forgotten vertex connecting two vertices from one subtree, one of whom has an additional forgotten neighbor from the other subtree. And so forth ...

3.5 Mathematical Reasoning

Sometimes, combining two problems can result in a problem exceeding (in elegance) the sum of its parts. Consider the following example.

`Bipartite Cograph`

Objective: Compute the sum of weights of all subsets $S \subseteq V$ such that the following holds:

1. The induced subgraph $G[S]$ contains no induced path of length 3.
2. The induced subgraph $G[S]$ is bipartite; that is, it contains no odd-length cycles.

This example highlights the value of mathematical knowledge. Mechanistically, one might combine separate solutions for tracking cographs[6] and bipartite graphs, resulting in a large state and poor complexity. However, recognising the embedded combinational insight simplifies the solution significantly: a cograph is bipartite if and only if it is triangle-free.[7]

Lemma 3.1. *Let G be a cograph. Then, G is bipartite if and only if G is triangle-free.*

Proof. Recall that a graph is bipartite if and only if it does not contain any odd-length cycle. Therefore, the first implication is immediate. In the opposite direction, let G be a triangle-free cograph. Assume FSOC that G is not bipartite. Therefore, it contains (as a subgraph) an odd length cycle of length greater than three. Let $v_1 \sim v_2 \sim \cdots \sim v_{2k+1} \sim v_1$ be such an odd cycle, of minimum length, where $k > 1$.

[6]A cograph is a graph containing no induced path with four vertices and three edges.

[7]This is illustrative of a more general theme; certain types of graph properties, for instance, those that are 'induced-hereditary' such as the one shown in this problem, are characterised by a set of "forbidden induced subgraphs" (see [Har69]). In this case, the forbidden graphs are P_4, a path with four vertices and three edges, and C_3, a triangle.

Consider the vertices $T = \{v_1, \ldots, v_4\}$. Recall that G is triangle-free, so except for the 'path' joining those vertices, no two vertices may be joined by an edge, except perhaps for v_1 and v_4. If $v_1 \not\sim v_4$, then inducing over T yields a copy of P_4, a contradiction. Otherwise, if $v_1 \sim v_4$, then the original cycle is split into a cycle of length four, and a cycle of length $2k - 1$, a contradiction. □

Another useful fact is that in a tree decomposition, every triangle (in fact, every clique) appears fully within a bag (at some point). Thus, by relying on Lemma 3.1 one may greatly simplify the problem, reducing the *bipartiteness* to simply ensuring no bag introduces a triangle. Indeed, the solution requires minimal additional overhead beyond that of the original `Cograph` problem.

4 Evaluating Solutions

A crucial aspect of our work is the ability to rigorously evaluate candidate solutions, to any of our given algorithmic problems. To this end, we have developed a comprehensive framework, allowing for both systematic generation of dynamic programming problems, and verification of proposed solutions to them.

4.1 Evaluation Environment

To focus the evaluation on the core algorithmic reasoning challenge, and to simplify the task presented to the models to a large extent, we provide a purpose-built Python evaluation environment. This framework handles the input parsing, graph representation, and the traversal of the tree decomposition. Consequently, the model's task is significantly streamlined: it need only implement the *logic* for the dynamic programming updates at each type of node in the tree decomposition. Concretely, only the following five specific *callback* functions must be implemented,

Callback	**Description**
`leaf_callback`	Initialises the DP table at a leaf node of the tree decomposition (Base case).
`introduce_callback`	Handles the introduction of a vertex into a bag.
`forget_callback`	Handles the removal of a vertex from a bag.
`join_callback`	Handles the case in which two previously processed subgraphs meet at a bag.
`extract_solution`	Computes the final answer from the DP table at the root bag.

The environment traverses through the tree decomposition in post-order, invoking the appropriate callback at each node. To enforce that solutions are based purely on the principles of tree decomposition, the graph provided to each callback is restricted to the subgraph induced by the vertices in the current bag.

4.2 Problem Generation and Structure

Our dynamic programming problems are generated by means of a domain-specific language (DSL). For each such problem, the DSL produces:

1. A human-readable description of the task (see Section 2).
2. A *verifier* V, mapping any input $(G, \mathcal{T}, w)$ to the integer answer to the problem at hand.
3. A set of 'tests', where each test consists of an *input* – which is a triplet $(G, \mathcal{T}, w)$ of a graph, a tree-decomposition, and a weight function – and optionally an *output*, which is a single integer x.

To ensure that the generated tests are tractable, our framework generates a complexity estimate for each task. Every formula in our dataset is annotated with a specification for a valid (though not necessarily optimal) dynamic programming state. This specification is constructed from a library of pre-defined, modular building blocks, which represent common patterns arising in MSO-based dynamic programming, like state bits for tracking properties, or profiles for representing incidence (or higher-order) structures.

Our DSL leverages these state specifications to derive a concrete complexity bound for the problem, which in turn informs the test generation. This model can also account for more advanced details, such as rules for optimising computationally expensive operations like the 'join' step (often reducing its complexity from quadratic to near linear). We remark that the time limits for our tests are intentionally very generous; we provide a performance overhead of up to $100\times$ over the expected runtime of the annotated solution, ensuring that the evaluation prioritises algorithmic correctness over minor implementation-level inefficiencies.

4.3 Generation of Tests

To comprehensively test any proposed solution, it is crucial that in the generated tests, both the *graph* and its *tree-decomposition* be rich and interesting. To this end, we employ a stochastic process; we sample the graph and a tree decomposition from a family of Markov chains, designed to explore the space of low-treewidth graphs. This methodology provides us with exact control over the treewidth of the resulting graphs, and other topological properties of the tree decomposition, thereby allowing us to ensure that the tests for the problems remain tractable within an allotted timeframe. Crucially, we remark that our sampling process ensures that (with high probability) all possible "gadgets" (small subgraphs) are *present* in the tested graphs, and moreover, that they are *observed in all possible ways* through the local view of the bags.

4.4 Types of Tests

Our evaluation methodology consists of several key types of test suites, each designed to probe a different aspect of a solution's validity.

Consistency. A fundamental property of the evaluated dynamic programming algorithms is that the final result should be *invariant* to the choice of tree decomposition. That is, for a fixed graph G, and weight function w, the output should not depend on the structure of the tree decomposition $\mathcal{T}$. The consistency tests ensure that this property holds, by fixing a large graph G and an arbitrary weight function w, and enumerating over many different tree decompositions of the same ambient graph, through a set of 'perturbations' that successively modify the structure of an existing tree decomposition.

Correctness. For sufficiently small input graphs, it is feasible to compute the expected output by brute force. This is done by leveraging the aforementioned *verifiers*. These tests serve as a direct verification of the DP algorithm's logic. Our framework generates many small graphs and varied tree decompositions, and the output of the candidate solution is compared against the known correct answer.

Efficiency. These tests are designed to push a solution to its computational limits. The goal is to detect implementations where the DP state, or its handling, is not truly fixed-parameter linear (FPL). By systematically increasing the difficulty of the tests, either by producing larger graphs, or graphs with certain extremal properties, we can identify performance bottlenecks that indicate a flawed or suboptimal implementation.

Sporadic (Exotic) Tests. These includes using small "universal" graphs that are known to contain a wide range of subgraphs and structures, providing a dense set of features for a single test case.

5 Results

In this section we present the performance of frontier reasoning models on our datasets.

5.1 Experimental Setup

For our evaluation, we generate solutions from each model using a detailed prompt that provides the model with all necessary context. Our goal is to help the model to the largest extent possible. The prompt, which can be found in full here, consists of the following components:

- A brief description of the dynamic programming setting, outlining the general approach for designing a state and the transition functions.

- The specific problem description, followed by the request to implement the four transition functions, and the extraction of the final answer from the root table (see Section 4.1).
- A set of basic guidelines, which instruct the model to avoid trying to circumvent the framework, state the expected fixed-parameter linear (FPL) runtime complexity, and encourage it to attempt a solution even for difficult problems.
- A utility class to query the graph induced by the vertices within a bag.
- **Three diverse few-shot example solutions**, included to aid the model, selected from areas where models were observed to struggle.

To score a model's response, the implemented callback functions are extracted from the completion and are integrated into the evaluation environment described in Section 4.1. They are then run against our test suites to produce a final score for the solution (see Section 4).

5.2 Evaluation Results

We evaluated the four leading reasoning models on our dataset: o3 high (OpenAI), o3-Pro high (OpenAI), Gemini 2.5 Pro (Google DeepMind), and Grok 4 Heavy (xAI). In this evaluation, we provided each model with the maximum number of reasoning tokens, the maximum number of output tokens, and used the highest level of "reasoning" available.[8] The models' only task is to implement the five necessary functions outlined in Section 4. To ensure comparable inference costs, we sampled ten *independent* completions *per problem* (@10) for o3 and Gemini 2.5 Pro, and a single completion (@1) for o3-Pro and Grok 4 Heavy (the latter two models already aggregate multiple samples internally). A model is considered to have succeeded in solving a problem @k, if *any* of its k attempts has been successful.

The success rate on the FormulaOne dataset is remarkably low, with Grok 4 Heavy solving none of the problems, and Gemini, o3 and o3-Pro each solving only one problem out of 120. This poor performance is particularly notable given the substantial assistance provided: models received a diverse few-shot prompt, and the framework handled all input parsing, graph representation, and tree decomposition construction and traversal on their behalf. This indicates that the gap between current model capabilities and the reasoning required for these problems is fundamental and extends beyond prompt engineering. We refer the reader to Section 5.4, where we provide a brief analysis of the failure modes observed in our evaluation of these frontier models.

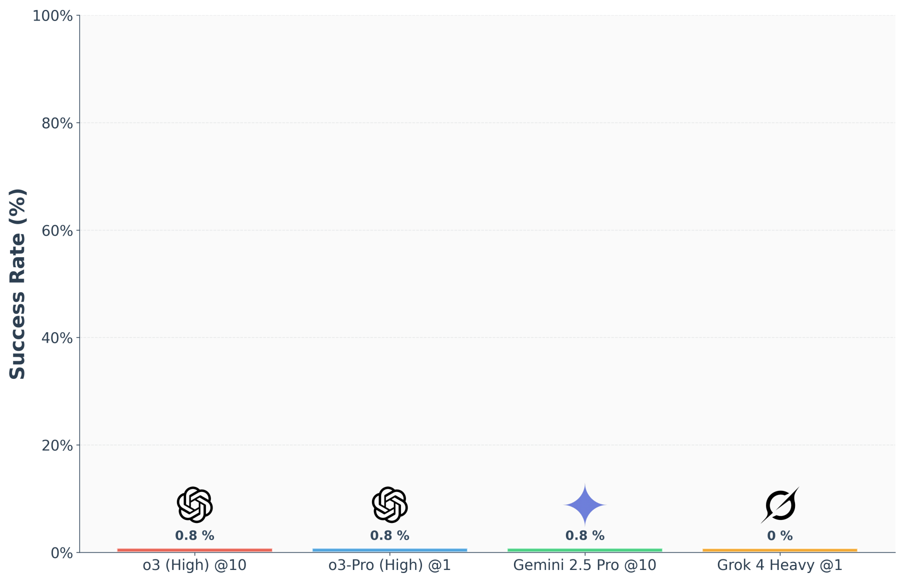


Figure 8: Performance of top frontier models on the FormulaOne dataset.

[8]Grok 4 Heavy was also provided with internet access, whereas other models were not due to technical constraints.

The performance of reasoning models on FormulaOne-Warmup – our dataset of *entry-level* MSO-based problems – was substantially better. We believe these results clearly delineates the complexity spectrum within problems derived from MSO logic on graphs – ranging from accessible entry-level reasoning tasks, to challenges that genuinely test the current upper bounds of algorithmic and logical reasoning capabilities in frontier models. We believe that FormulaOne-Warmup provides a valuable benchmark for the broader research community, including those working with smaller or open-source models, and can serve as a training ground for developing more advanced algorithmic reasoning capabilities.

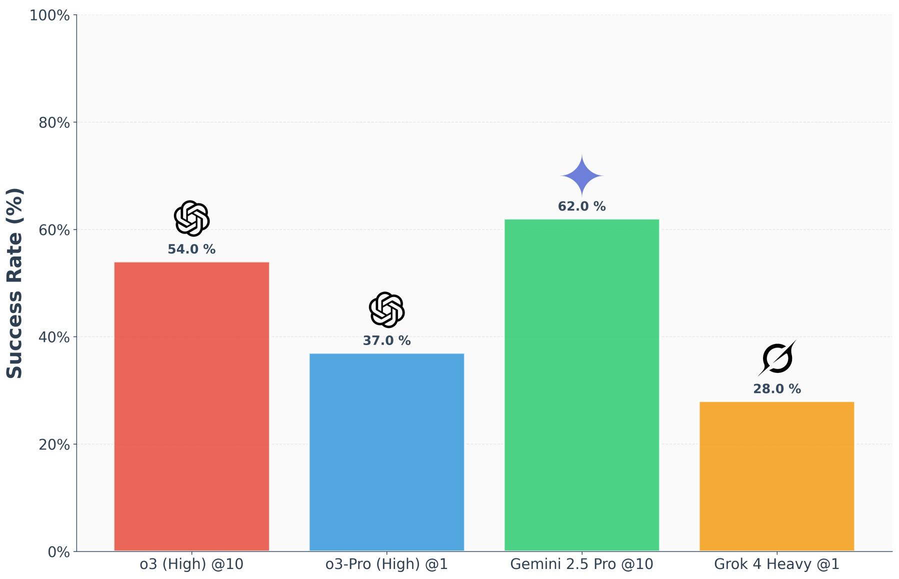


Figure 9: Performance of top frontier models on the FormulaOne-Warmup dataset. There are 100 problems in the dataset. Performance is measured in terms of number of problems solved, where a problem is solved @k if *any* of the k solutions is correct.

5.3 Further Analysis

To better understand model performance and failure modes, each problem in our dataset is annotated according to specific algorithmic skills and state-design techniques required for its solution. Each problem can be assigned multiple labels (and indeed, harder problems often are). This categorisation allows for a fine-grained analysis of the strengths and weaknesses exhibited by the evaluated models.

Label	**Description**
`ADJACENCY`	Requiring neighborhood properties such as degrees, domination, and independence.
`COMPOUND`	Formulas constructed recursively.
`CONNECTIVITY`	Concerning connectivity, including components, cuts, minors, and cycle detection.
`DISTANCE`	Relies on distance information in the ambient graph.
`EPSILON`	"Almost" properties, where a property holds after omitting or adding a single vertex.
`EXTREMAL`	Members of a property, that are also maximal or minimal w.r.t inclusion.
`GADGETS`	Aiming to find fixed subgraphs, such as triangles or cliques, within the selected set.
`GLOBAL`	Pertains to properties that require tracking global information across the entire graph.
`GRAPH-THEORY`	Requires non-trivial theorems of graph-theory.
`INTERFACE`	Tracks how selected vertices inside a bag interact with those outside the bag.
`LOGIC`	Having a complex logical structure, requiring analysis of quantifiers and negations.

LOOKAHEAD	Must utilize states that encode information about future constraints or dependencies.
MODULAR	Compares or analyzes the lattice structure of vertex neighborhoods.
TOPOLOGY	Reasons about global topological structure.

Table 2: Annotations of problems in our dataset. Labels correspond to algorithmic techniques and elements of state-design arising in FormulaOne problems.

Using the categorisation provided in Table 2, we provide a more fine-grained breakdown of the models' performance, for each such category *in isolation*.

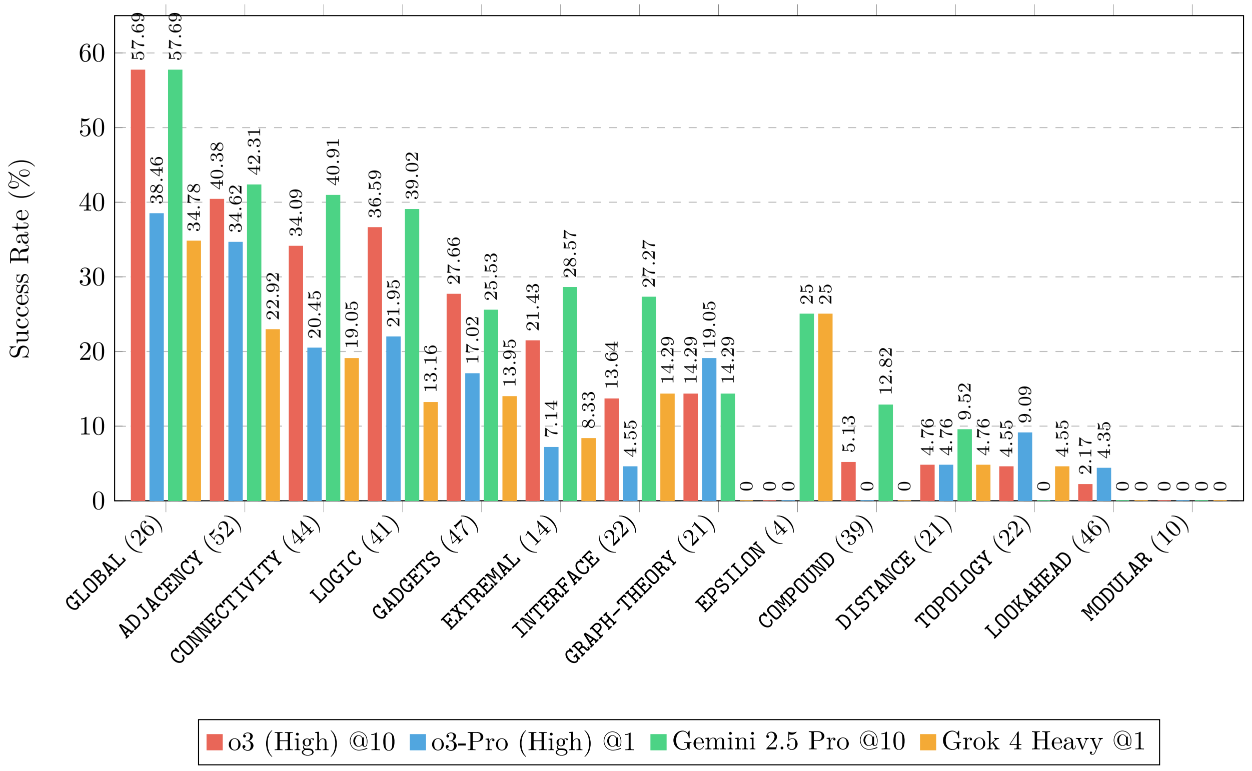


Figure 10: Model Performance by problem label on the *combined* FormulaOne and FormulaOne-Warmup datasets. Problem counts with the given category label are shown next to each label in parentheses. The success rate is the ratio of problems solved, having the given label.

5.4 Observed Central Failure Modes

Below, we provide a brief analysis of several recurrent failure modes observed during our evaluation, highlighting specific areas where frontier reasoning models consistently struggle, despite extensive support.

1. **Premature Finalisation.** Often models make irreversible decisions about a forgotten vertex based on the current, non-final properties of its neighbors that remain in the bag. The failure is one of foresight; as the model does not account for how these neighbors might change critically in the future.

2. **Incomplete Geometric Reasoning.** A particular weak point of models is their inability to account for all the ways in which a small, fixed subgraph "pattern" can be formed by combining vertices across different parts of the tree decomposition. The states, or the transition logic, fails to cover all possible geometric configurations of the pattern's vertices with respect to the bags (see Figure 7 for an illustration of the kind of complications that may arise in such cases.)

3. **Local-to-Global Errors.** This failure happens when the model successfully enforces local rules, but fails to assemble them into the correct *global* structure. In other words, it builds an object that satisfies local constraints but violates the overall definition of the target structure.

4. **Non-Canonical State Representation.** In counting problems, one must make sure to count every set exactly once. Often, the selected dynamic programming state includes auxiliary "witness" information, that distinguishes between states that *should* be considered identical, due to a failure to define a *canonical* representative for each state. When unmitigated (for instance, by inclusion-exclusion), this leads to *overcounting.*

6 Discussion

While frontier AI models achieve high ratings in top human level competitive programming, they fail on more challenging algorithmic challenges, such as the 'hard' problems in our FormulaOne. This serves to highlight that current benchmarks, which often rely on problems solvable by human experts, are insufficient for measuring the deep algorithmic and combinatorial reasoning required for complex, real-world research tasks. Given the depth of reasoning these problems demand, future progress may depend on incorporating more principled approaches, such as systematic search, rather than relying solely on the emergent capabilities of current models.

Another of our contributions is the harnessing of Monadic Second-Order Logic on graphs, in order to make the first steps towards a principled method of creating a virtually unlimited suite of hard yet solvable problems. As such, we believe our dataset, and its extensions, can serve as a high-depth environment for Reinforcement Learning with Verifiable Rewards (RLVR), moving beyond existing static datasets. We remark that the current dataset, while substantial, represents only a fraction of the challenges that can be generated (see Section 2.3).

Lastly, our work has a deep connection to theoretical computer science. The difficulty of many problems in our dataset is linked to the Strong Exponential Time Hypothesis (SETH), a central conjecture in fine-grained complexity. This implies that any significant algorithmic progress on these problems, whether by human or (hopefully) by machine, could have tangible theoretical implications, pushing at the frontiers of complexity theory.

A Algorithm for Maximal Cluster Graph

Some problems are incredibly hard to get right from start to finish. To illustrate why this is the case, let us consider one such problem, in full.

`Maximal Cluster Graph`

Objective: Compute the sum of all weights of sets $S \subseteq V$ such that:

The graph $G[S]$ is a cluster graph[1] that is maximal with respect to inclusion.

[1]A cluster graph is a disjoint union of cliques.

The problem asks us to sum the weights of the sets S for which $G[S]$ is a cluster graph, *and* there exists no superset $T \supseteq S$ such that $G[T]$ is a cluster graph. We start with the following observation: since the cluster graph property is downwards closed (with respect to inclusion), maximality reduces to *local maximality* – that is, for any vertex $v \in V \setminus S$, the induced subgraph $G[S \cup \{v\}]$ is *not* a cluster graph.

A.1 Characterization

A key observation is that a graph is a cluster graph if and only if it does not contain an induced path of length two (i.e., a copy of P_3). By definition a P_3 is a set of three vertices $\{u, v, w\}$ where the only edges are $u \sim v \sim w$. Therefore, the problem is equivalent to finding a maximal vertex set $S \subseteq V$ such that the induced subgraph $G[S]$ is P_3-free. The maximality condition means that for any vertex $u \in V \setminus S$, adding u to S must create at least one induced P_3 in $G[S \cup \{u\}]$.

A.2 Dynamic Programming Formulation

As always, we solve the `Maximal Cluster Graph` problem through dynamic programming on a tree decomposition of the input graph G. For each vertex t in the tree decomposition with bag X_t, we compute a table A_t, which stores information about valid partial solutions in the subgraph induced by the vertices processed so far, V_t. A state in A_t is defined by a *profile* λ for the vertices in the bag X_t. The table entry $A_t(\lambda)$ is a boolean flag, indicating whether this profile can be extended from a valid partial solution on V_t.

The profile λ for X_t consists of the following information for each vertex $v \in X_t$:

- **Status** $\texttt{s}(v) \in \{\texttt{selected}, \texttt{not-selected}\}$: This indicates whether v is part of the solution set S.
- **For each *selected* vertex** v:
  - **Forgotten Neighbour bit** $\texttt{FN}(v) \in \{0, 1\}$, where $\texttt{FN}(v) = 1$ if and only if v is adjacent to a *selected* vertex in $S \cap (V_t \setminus X_t)$.
  - **Obligation bit** $\texttt{O}(v) \in \{0, 1\}$, where $\texttt{O}(v) = 1$ if and only if v is required to become adjacent to a new *selected* vertex in $S \setminus V_t$, to satisfy a maximality condition.
- **For each *non-selected* vertex** u:
  - **Safety bit** $\texttt{SAFE}(u) \in \{0, 1\}$, where $\texttt{SAFE}(u) = 1$ if and only if the maximality condition for u is already satisfied (i.e., adding u to $S \cap V_t$ would create an induced P_3).

A.3 DP Transitions

The DP table for each node t is computed based on the tables of its children. For ease of notation, from here on, let $S_t = \{v \in X_t \mid s(v) = \texttt{selected}\}$ and $U_t = X_t \setminus S_t$.

A.3.1 Leaf Node

For a tree decomposition, a leaf node t has a bag X_t containing a single vertex, say $X_t = \{v\}$. We initialise the DP table for t with two valid base profiles:

- **Profile 1:** v **is selected**. We set $\texttt{s}(v) = \texttt{selected}$, $\texttt{FN}(v) = 0$, and $\texttt{O}(v) = 0$.
- **Profile 2:** v **is not selected**. We set $\texttt{s}(v) = \texttt{not-selected}$ and $\texttt{SAFE}(v) = 0$.

A.3.2 Introduce Node

Let t be an introduce node with child t', such that $X_t = X_{t'} \cup \{z\}$. For each valid profile λ' on $X_{t'}$, we generate profiles λ on X_t by deciding the status of the new vertex z.

Case 1: $s(z) = \texttt{not-selected}$**.** The profile for vertices in $X_{t'}$ remains unchanged. We check if z would *seal* a P_3 (either fully within the bag, or if it is adjacent to a selected vertex whose `FN` bit is on). If so, we fix $\texttt{SAFE}(z) = 1$, and otherwise fix $\texttt{SAFE}(z) = 0$.

Case 2: $s(z) = \texttt{selected}$**.**

1. P_3**-freeness check**: The new profile is invalid if z forms an induced P_3 with vertices already in the partial solution. This occurs whenever:
   - z is part of a P_3 contained entirely in the induced subgraph (either $z \sim u \sim v$ or $v \sim z \sim u$).
   - z is adjacent to a vertex $v \in S_{t'}$ that has a forgotten selected neighbour ($\texttt{FN}(v) = 1$). This would induce a P_3 where the non-edge $z \not\sim x$ is guaranteed by x being forgotten.
2. **State update**: If all the checks pass, we create a new valid state λ. We initialise $\texttt{FN}(z) = 0$ and $\texttt{O}(z) = 0$. For any $v \in S_{t'}$ adjacent to z, its obligation $\texttt{O}(v)$ is now fulfilled and is set to 0. For any $u \in U_{t'}$, we update its safety bit $\texttt{SAFE}(u)$ to 1 if z and some other vertices in S_t form an induced P_3 with u.

A.3.3 Forget Node

Let t be a forget node with child t', such that $X_t = X_{t'} \setminus \{z\}$. We project the profiles from $X_{t'}$ down to X_t.

Case 1: $z \in S_{t'}$**.**

1. **Obligation check**: The state is invalid if $\texttt{O}(z) = 1$. A vertex with an outstanding obligation cannot be forgotten.
2. **State update**: If the check passes, we form the new profile on X_t. For each neighbour $v \in S_t$ of the forgotten vertex z, we update its forgotten neighbour bit: $\texttt{FN}(v) \leftarrow \texttt{FN}(v) \vee 1$.

Case 2: $z \in U_{t'}$**.**

1. If $\texttt{SAFE}(z) = 0$, the omission of z is as of yet unjustified. We must ensure that z would seal a P_3 in the future.
   - The profile is invalid if the vertex z has no selected neighbours in the bag ($N(z) \cap S_{t'} = \emptyset$). Without a selected neighbour, z can never be part of an induced P_3.
   - The set of selected neighbours of z in the bag forms a clique (otherwise it would seal a P_3 and the safety bit would be on).
   - **State update**: We place an obligation on all of z's selected neighbours in the bag: for each $v \in N(z) \cap S_{t'}$, we set $\texttt{O}(v) \leftarrow \texttt{O}(v) \vee 1$.
2. If $\texttt{SAFE}(z) = 1$ then no action is needed; the profile is projected to parent node, omitting z, and all the other vertices have the same status.

A.3.4 Join Node

Let t be a join node with children t_1 and t_2, where $X_t = X_{t_1} = X_{t_2}$. A profile λ for X_t is valid if it can be formed by merging compatible valid profiles λ_1 from t_1 and λ_2 from t_2.

1. P_3**-freeness check**: The merge is invalid if for any vertex $v \in S_t$, it has a forgotten neighbour from both branches ($\texttt{FN}_1(v) = 1$ and $\texttt{FN}_2(v) = 1$). This would create an induced P_3 with v as the center and endpoints in $V_{t_1} \setminus X_t$ and $V_{t_2} \setminus X_t$.
2. **State merge**: If the check passes, the new profile λ is created by combining λ_1 and λ_2:
   - For $u \in U_t$, the safety bit is merged with a logical OR, $\texttt{SAFE}(u) = \texttt{SAFE}_1(u) \vee \texttt{SAFE}_2(u)$.
   - For $v \in S_t$, the forgotten neighbour bit is also merged with a logical OR, $\texttt{FN}(v) = \texttt{FN}_1(v) \vee \texttt{FN}_2(v)$.
   - For $v \in S_t$, the obligation bit $O(v)$ is handled carefully. An obligation from one branch is discharged if the vertex acquires a forgotten neighbour from the other branch. The new obligation is set according to the rule: $\texttt{O}(v) = (\texttt{O}_1(v) \wedge \neg\texttt{FN}_2(v)) \vee (\texttt{O}_2(v) \wedge \neg\texttt{FN}_1(v))$.

A.4 Final Answer

At the root node a profile is valid if for every outside vertex the safety bit is on, and for every selected vertex the obligation bit is off. At this point we can extract the final answer.

References

[AON+21] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[BB73] Umberto Bertele and Francesco Brioschi. On non-serial dynamic programming. *J. Comb. Theory, Ser. A*, 14(2):137–148, 1973.

[BDJ+98] Hans L Bodlaender, Jitender S Deogun, Klaus Jansen, Ton Kloks, Dieter Kratsch, Haiko Müller, and Zsolt Tuza. Rankings of graphs. *SIAM Journal on Discrete Mathematics*, 11(1):168–181, 1998.

[Bod88] Hans L Bodlaender. Dynamic programming on graphs with bounded treewidth. In *Automata, Languages and Programming: 15th International Colloquium Tampere, Finland, July 11–15, 1988 Proceedings 15*, pages 105–118. Springer, 1988.

[Bod93] Hans L Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 226–234, 1993.

[CER93] Bruno Courcelle, Joost Engelfriet, and Grzegorz Rozenberg. Handle-rewriting hypergraph grammars. *Journal of computer and system sciences*, 46(2):218–270, 1993.

[CFK+15] Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, Saket Saurabh, Marek Cygan, Fedor V Fomin, et al. Lower bounds based on the exponential-time hypothesis. *Parameterized Algorithms*, pages 467–521, 2015.

[Cho19] François Chollet. On the measure of intelligence. 2019.

[CIP09] Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. The complexity of satisfiability of small depth circuits. In *International Workshop on Parameterized and Exact Computation*, pages 75–85. Springer, 2009.

[CKB+21] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. 2021.

[CKK+25] Francois Chollet, Mike Knoop, Gregory Kamradt, Bryan Landers, and Henry Pinkard. Arc-agi-2: A new challenge for frontier ai reasoning systems. *arXiv preprint arXiv:2505.11831*, 2025.

[CKKL24] Francois Chollet, Mike Knoop, Gregory Kamradt, and Bryan Landers. Arc prize 2024: Technical report. *arXiv preprint arXiv:2412.04604*, 2024.

[Cou90] Bruno Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Information and computation*, 85(1):12–75, 1990.

[EWS+25] Ahmed ElKishky, Alexander Wei, Andre Saraiva, Borys Minaiev, Daniel Selsam, David Dohan, Francis Song, Hunter Lightman, Ignasi Clavera, Jakub Pachocki, et al. Competitive programming with large reasoning models. 2025.

[FBH+22] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J.R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610:47–53, 2022.

[FHMW21] Johannes K Fichte, Markus Hecher, Michael Morak, and Stefan Woltran. Dynasp2.5: Dynamic programming on tree decompositions in action. *Algorithms*, 14(3):81, 2021.

[FLR+12] Fedor V Fomin, Daniel Lokshtanov, Venkatesh Raman, Saket Saurabh, and BV Raghavendra Rao. Faster algorithms for finding and counting subgraphs. *Journal of Computer and System Sciences*, 78(3):698–706, 2012.

[GEB+24] Elliot Glazer, Ege Erdil, Tamay Besiroglu, Diego Chicharro, Evan Chen, Alex Gunning, Caroline Falkman Olsson, Jean-Stanislas Denain, Anson Ho, Emily de Oliveira Santos, et al. Frontiermath: A benchmark for evaluating advanced mathematical reasoning in ai. *arXiv preprint arXiv:2411.04872*, 2024.

[Hal76] Rudolf Halin. S-functions for graphs. *Journal of geometry*, 8:171–186, 1976.

[Har69] F. Harary. *Graph Theory.* Addison-Wesley, Reading, MA, 1969.

[HBK$^+$21] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. 2021.

[IHI$^+$25] Yuki Imajuku, Kohki Horie, Yoichi Iwata, Kensho Aoki, Naohiro Takahashi, and Takuya Akiba. Ale-bench: A benchmark for long-horizon objective-driven algorithm engineering. 2025.

[JHG$^+$24] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. 2024.

[KM24] Batya Kenig and Dan Shlomo Mizrahi. Enumeration of minimal hitting sets parameterized by treewidth. 2024.

[LMS11] Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Known algorithms on graphs of bounded treewidth are probably optimal. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*, pages 777–789. SIAM, 2011.

[MMZ$^+$23] Daniel J. Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618:257–263, 2023.

[NVE$^+$25] 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. 2025.

[PGH$^+$25] Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, Hugh Zhang, Dan Hendrycks, Alexandr Wang, et al. Humanity's last exam. 2025.

[QYY$^+$25] Shanghaoran Quan, Jiaxi Yang, Bowen Yu, Bo Zheng, Dayiheng Liu, An Yang, Xuancheng Ren, Bofei Gao, Yibo Miao, Yunlong Feng, et al. Codeelo: Benchmarking competition-level code generation of llms with human-comparable elo ratings. 2025.

[RS83] Neil Robertson and Paul D Seymour. Graph minors. i. excluding a forest. *Journal of Combinatorial Theory, Series B*, 35(1):39–61, 1983.

[RS84] Neil Robertson and Paul D Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

[Wan94] Egon Wanke. k-nlc graphs and polynomial algorithms. *Discrete Applied Mathematics*, 54(2-3):251–266, 1994.