Dependency calculator for npm

Lucas Pettersson

December 2025

Contents

1 Background **3**
1.1 The npm Ecosystem and Dependency Graphs . . . 3
1.2 Semantic Versioning (SemVer) . . . 3
1.3 Algorithmic Complexity: From SAT to COP . . . 3
1.4 Dependency Classifications and Propagation . . . 4

2 Formal Statement **5**
2.1 Definitions and Notation . . . 5
2.2 Logical Constraints . . . 5
2.3 Optimization Metric . . . 5
2.4 Problem Statement . . . 6

3 Implementation **7**
3.1 Selection . . . 7
3.2 Expansion . . . 7
3.3 Simulation (Rollout) . . . 7
3.4 Backpropagation . . . 8
3.4.1 Reward Calculation . . . 8
3.4.2 Tree Update . . . 8

4 Results **9**
4.1 Experimental Setup . . . 9
4.2 Parameter Sensitivity Analysis . . . 9
4.2.1 Impact of Tree Depth . . . 9
4.2.2 Impact of Computational Budget . . . 10
4.2.3 Impact of Simulation Horizon . . . 11
4.2.4 Impact of Simulation Bias (λ) . . . 11

5 Discussion **13**
5.1 Methodological Efficacy . . . 13
5.2 Computational Complexity . . . 13
5.3 Convergence . . . 13
5.4 Correctness . . . 14

1 Background

To define the dependency resolution problem rigorously, one must first establish the structural properties of the ecosystem in question and the specific rules governing compatibility.

1.1 The npm Ecosystem and Dependency Graphs

The npm registry can be modeled as a directed graph where nodes represent software packages. However, unlike standard static graphs, the npm graph is temporal and versioned. A "package" is not a singular entity but a collection of distinct *versions*.

Let $\mathcal{P}$ denote the set of all package names in the registry. For any package $p \in \mathcal{P}$, there exists a finite, ordered set of versions V_p. A specific instance of a package is defined by the tuple (p, v) where $v \in V_p$.

Dependencies are directional relationships between a specific version of a package and a set of required packages. If package A at version 1.0.0 depends on package B, it does not point to a specific version of B (e.g., B@2.1.0), but rather to a *range* of acceptable versions. This allows for flexibility and automatic patch updates but introduces the nondeterminism that necessitates a complex resolution algorithm.

1.2 Semantic Versioning (SemVer)

The constraints governing the edges of the dependency graph are defined by the Semantic Versioning specification. A SemVer version is a 3-component vector $v = (M, m, p) \in \mathbb{N}^3$, representing the Major, Minor, and Patch numbers, respectively.

The ordering of versions is lexicographical. Given two versions $v_1 = (M_1, m_1, p_1)$ and $v_2 = (M_2, m_2, p_2)$, we define $v_1 < v_2$ if:

1. $M_1 < M_2$, or
2. $M_1 = M_2 \wedge m_1 < m_2$, or
3. $M_1 = M_2 \wedge m_1 = m_2 \wedge p_1 < p_2$.

A *dependency constraint* is a predicate function $f_c : V_p \to \{0, 1\}$ defined by a range string (e.g., `^1.2.0` or `>= 2.0.0`). A version v satisfies the constraint if $f_c(v) = 1$. The core challenge of SemVer in dependency resolution is ensuring that for every dependency relation, the selected version falls within the intersection of all active constraints targeting that package.

1.3 Algorithmic Complexity: From SAT to COP

The problem of dependency resolution is a specific instance of the Boolean Satisfiability Problem (SAT). Specifically, it maps to the "package management problem," which has been proven to be NP-complete.

Consider a system where we must select at most one version for every package p. Let $x_{p,v}$ be a boolean variable representing the installation of package p at version v. The system must satisfy clauses such as:

- **Singleton Constraint:** For a given package p, $\sum_{v \in V_p} x_{p,v} \leq 1$ (if we assume a flat dependency tree).
- **Dependency Implication:** If $x_{p,v}$ is true, and p@v depends on package q with constraint C, then $\bigvee_{u \in V_q, C(u)=1} x_{q,u}$ must be true.

While finding *any* satisfying assignment is an NP-complete decision problem (SAT), the npm ecosystem introduces an additional dimension: optimality. Users generally prefer the most recent release of a package to ensure security and feature parity.

This transforms the problem from SAT to a Constraint Optimization Problem (COP). We are not merely searching for a feasible region in the solution space; we are searching for a solution vector $\mathbf{S}$ that maximizes an objective function $R(\mathbf{S})$. This adds significant complexity, as the solver must prune branches of the search tree that are valid but suboptimal, or employ heuristics to traverse the version space in descending order.

1.4 Dependency Classifications and Propagation

The edges of the dependency graph are not uniform; their behavior is determined by the classification of the dependency. We distinguish between three primary types of relationships, each imposing different traversal rules and constraint applications:

1. **Production Dependencies (`dependencies`):** This is the standard directional edge described in graph theory. If package A declares a production dependency on package B, the resolver must ensure B is present in the solution set. These edges are *transitive*: if the root depends on A, and A depends on B, the resolver must traverse from Root to A to B.

2. **Development Dependencies (`devDependencies`):** These dependencies define tooling required for testing or building a package but are not required for its runtime execution. Mathematically, these edges are *non-transitive.* The constraints declared in `devDependencies` are only evaluated for the root node of the dependency tree (the project being developed). For any node at depth $d > 0$ in the graph, `devDependencies` are ignored (pruned) during traversal.

3. **Peer Dependencies (`peerDependencies`):** Unlike production dependencies, which instruct the resolver to *fetch* a package, peer dependencies act as a validation check against the existing hierarchy. If package A lists package B as a peer dependency, it asserts that B must already exist in the host context (typically the parent or root package) that imports A.

   This introduces a *horizontal* or *contextual constraint* rather than a vertical one. The resolver does not necessarily install the peer dependency as a child of A; rather, it verifies that the version of B selected for the project satisfies A's requirement. Failure to satisfy a peer dependency often results in a "missing peer" warning or an invalid dependency tree state.

2 Formal Statement

We define the dependency resolution problem as a search for a truth assignment in a propositional logic system that satisfies a global logical sentence while maximizing a scalar objective function.

2.1 Definitions and Notation

Let $\mathcal{P}$ be the finite set of all available package identifiers in the registry. For each package $p \in \mathcal{P}$, let $\mathcal{V}_p = \{v_1, v_2, \ldots, v_n\}$ be the totally ordered set of available versions, sorted descendingly such that v_1 is the latest version.

We define a configuration state as a partial function $\sigma : \mathcal{P} \rightharpoonup \bigcup_{p\in\mathcal{P}} \mathcal{V}_p$, which maps a package name to a specific concrete version. We define the proposition $x_{p,v}$ as a boolean variable which is true if and only if package p is installed at version v (i.e., $\sigma(p) = v$).

2.2 Logical Constraints

The validity of a resolution set is determined by a set of logical sentences derived from the package manifests.

Let $D(p, v)$ be the set of dependencies for package p at version v. Each element in this set is a tuple (q, r), where $q \in \mathcal{P}$ is the required package and r is the SemVer constraint (a subset of $\mathcal{V}_q$).

For a configuration to be valid, the selection of a specific parent package version necessitates the selection of valid child versions. This can be expressed as a logical implication. For every active package-version pair (p, v), and for every dependency $(q, r) \in D(p, v)$, the following sentence must hold:

$$S_{p,v,q} : x_{p,v} \implies \bigvee_{u\in(\mathcal{V}_q\cap r)} x_{q,u} \tag{1}$$

This sentence $S_{p,v,q}$ states that if p is installed at version v, then q *must* be installed at some version u, where u satisfies the semantic range r. The term $\bigvee$ represents the disjunction (logical OR) of all satisfying versions.

The global validity constraint Φ is the conjunction of all such implications for the root package and all transitively reachable packages:

$$\Phi = \bigwedge_{p\in\mathrm{Dom}(\sigma)} \bigwedge_{(q,r)\in D(p,\sigma(p))} S_{p,\sigma(p),q} \tag{2}$$

Furthermore, to ensure a deterministic state (assuming a flat resolution context for simplicity), we impose the Uniqueness Constraint, enforcing that at most one version of any package is selected:

$$\forall p \in \mathcal{P}, \sum_{v\in\mathcal{V}_p} x_{p,v} \le 1 \tag{3}$$

2.3 Optimization Metric

Since multiple truth assignments for the dependency graph Φ may exist, we require a metric to distinguish the optimal solution. The heuristic goal is to prioritize the "latest" possible versions to ensure the project uses the most recent features and patches.

We define a *Recency Score* function $\rho(p, v)$ for a chosen version $v \in \mathcal{V}_p$. Let $idx(v)$ be the zero-based index of version v in the descending sorted set $\mathcal{V}_p$ (where index 0 represents the latest version). Let $N_p = |\mathcal{V}_p|$ be the total number of versions available for package p.

The score for an individual package assignment is defined as:

$$\rho(p, v) = 1 - \frac{idx(v)}{N_p} \tag{4}$$

Under this definition, the latest version yields a score of 1.0, while older versions yield scores approaching 0.

The global objective is to maximize the average recency across the entire dependency tree. For a complete assignment σ, we define the Global Reward function $R(\sigma)$:

$$R(\sigma) = \frac{1}{|\mathrm{Dom}(\sigma)|} \sum_{p \in \mathrm{Dom}(\sigma)} \rho(p, \sigma(p)) \tag{5}$$

This function normalizes the score to the interval $(0, 1]$, ensuring that the metric is independent of the total number of packages installed.

2.4 Problem Statement

Given a set of root dependencies $R = \{(p_1, r_1), \ldots, (p_k, r_k)\}$, the goal is to find a complete assignment σ such that:

1. **Root Satisfaction:** $\forall (p, r) \in R, \sigma(p) \in r$.
2. **Global Consistency:** σ satisfies the logical conjunction Φ (2), meaning all transitive dependencies are resolved and valid.
3. **Optimality:** The Global Reward $R(\sigma)$ is maximized.

This formulation defines a search space where nodes represent partial assignments. A crucial distinction in this implementation is that the reward function $R(\sigma)$ is only defined for *complete* solutions. Partial or invalid assignments yield a reward of 0, driving the stochastic search toward terminal states that fully satisfy the graph.

3 Implementation

To solve the constraint optimization problem described in Section 2, we employ the Monte Carlo Tree Search (MCTS) algorithm. MCTS is particularly well-suited for this domain because the search space of dependency graphs is vast and sparse; valid configurations are rare relative to the total number of permutations, and we require a method to effectively balance exploration (finding any valid graph) with exploitation (finding the graph with the latest versions).

We define a search tree where the root node s_0 represents the initial state with only the root project requirements. Each edge represents the assignment of a specific version to a package, effectively "locking in" a truth value for one variable $x_{p,v}$ and resolving one logical sentence.

The algorithm proceeds iteratively through four phases: Selection, Expansion, Simulation, and Back-propagation.

3.1 Selection

In the selection phase, the algorithm traverses the existing tree from the root to a leaf node to identify the most promising path for investigation. At each node s, we select the child node s' that maximizes the Upper Confidence Bound for Trees (UCT). This balances the estimated quality of a version choice against the uncertainty of unexplored paths.

The UCT value for a child node j is calculated as:

$$UCT_j = \bar{X}_j + C_p \sqrt{\frac{\ln n}{n_j}} \tag{6}$$

Where:

- $\bar{X}_j$ is the average reward observed in previous simulations passing through node j. In our context, this reward correlates to the recency of the resulting dependency tree.
- n is the total number of visits to the parent node.
- n_j is the number of visits to child node j.
- C_p is the exploration constant, tuned to control the algorithm's tendency to explore older or less-tested versions versus exploiting known valid configurations.

The traversal continues until a leaf node (a state with unexpanded valid moves) is reached.

3.2 Expansion

Upon reaching a leaf node s_L that represents a partial assignment, we identify the next unsatisfied logical sentence in the dependency queue. Let the next required package be q with constraint range r.

The expansion step generates child nodes for every version $u \in \mathcal{V}_q$ such that $u \in r$. This step is critical as it enforces the satisfiability constraints defined in (2). We prune the search space immediately by only creating nodes for versions that satisfy the intersection of all currently active constraints on q. If no versions satisfy the constraint, the node is marked as a "dead end" (conflict), and the algorithm terminates this branch.

3.3 Simulation (Rollout)

From the newly expanded node, we perform a simulation to estimate the potential value of the current partial assignment. The simulation effectively "fast-forwards" the resolution process to determine if the current path can lead to a complete, valid dependency graph.

Unlike pure random walks used in standard MCTS, we employ a heuristic-guided rollout. The simulation selects versions for remaining dependencies using a biased stochastic policy that favors the latest versions while allowing non-greedy exploration.

The process proceeds as follows:

1. **Identify Pending:** Retrieve the queue of unsatisfied dependencies for the current graph state.
2. **Constraint Intersection:** For the next target package, calculate the set of valid versions $\mathcal{V}_{valid}$ that satisfy all currently active constraints.
3. **Probabilistic Selection:** We select a version $v \in \mathcal{V}_{valid}$ using a Softmax probability distribution. Let $r(v)$ be a scoring function where the latest version has the highest rank. The probability $P(v)$ of selecting version v is given by:

$$P(v) = \frac{\exp(\lambda \cdot r(v))}{\sum_{u \in \mathcal{V}_{valid}} \exp(\lambda \cdot r(u))} \tag{7}$$

   Here, λ is a tunable temperature parameter that controls the "greediness" of the simulation.

   - As $\lambda \to \infty$, the selection becomes purely greedy (always picking the latest).
   - As $\lambda \to 0$, the selection approaches a uniform random distribution.

4. **Recursion:** Apply the selection, update the constraint propagation, and repeat until the dependency queue is empty (success) or a contradiction is found (failure).

3.4 Backpropagation

The final phase of the MCTS iteration updates the tree statistics based on the outcome of the simulation. This process involves calculating the reward for the specific simulation path and propagating that value up the tree to the root.

3.4.1 Reward Calculation

The reward calculation strictly enforces graph completeness and validity. Given the terminal state σ reached by the simulation, the reward scalar R is determined as follows:

First, we verify structural integrity. If the state σ contains any constraint violations (conflicting versions) or if the set of pending dependencies is non-empty (incomplete graph), the reward is strictly zero:

$$R = 0 \tag{8}$$

If the graph is both valid and complete, we compute R as the average recency score of all resolved packages. Using the ranking logic defined in the Optimization Metric, where $idx(v)$ is the zero-based index of the selected version and N_p is the total version count for package p:

$$R = \frac{1}{|\sigma|} \sum_{p \in \sigma} \left(1 - \frac{idx(\sigma(p))}{N_p}\right) \tag{9}$$

This results in a normalized value $R \in (0, 1]$, where a value of 1.0 indicates a solution composed entirely of the latest available versions, and lower values indicate reliance on older package versions to satisfy constraints.

3.4.2 Tree Update

The calculated reward R is propagated recursively from the leaf node s_L back to the root s_0. For every node s_i in the path:

$$N(s_i) \leftarrow N(s_i) + 1 \tag{10}$$

$$Q(s_i) \leftarrow Q(s_i) + R \tag{11}$$

Where N is the visit count and Q is the accumulated reward. This accumulation directs future Selection phases toward branches that consistently yield complete, high-recency dependency graphs.

4 Results

To evaluate the performance characteristics of the MCTS dependency calculator, we conducted a sensitivity analysis on the four primary hyperparameters governing the search: Tree Depth (`MaxDepth`), Computational Budget (`MaxIterations`), Simulation Horizon (`MaxSimulationDepth`), and the Exploration Constant (λ).

4.1 Experimental Setup

The evaluation was performed using a set of 12 open-source packages chosen to represent varying degrees of graph complexity, ranging from small utilities to large enterprise frameworks. The test corpus T consists of:

$$T = \{\text{axios, express, yargs, webpack, eslint, react-scripts, jest, babel-core, react-dom, vue, next, inquirer}\}$$

For each package, the calculator was tasked with resolving a valid dependency tree of the peer dependencies since they have to be unique. Regular dependencies are allowed to differ and was intractable for the algorithm. We measured the Average Execution Time (ms) against the complexity of the resolution, defined by the number of resolved dependencies.

4.2 Parameter Sensitivity Analysis

4.2.1 Impact of Tree Depth

Figure 1 illustrates the performance impact of varying the maximum tree depth (d_{max}). Counter-intuitively, allowing the search tree to grow deeper ($d_{max} = 4$) resulted in significantly lower execution times compared to a shallow tree ($d_{max} = 2$).

This phenomenon highlights the efficiency of the MCTS Selection policy (UCT) compared to the random Simulation policy. When d_{max} is low, the algorithm is forced to switch to the stochastic rollout phase early in the resolution process. Since the rollout is heuristic-based and less "intelligent" than the UCT tree search, it requires more attempts to find a valid intersection of constraints. By increasing d_{max} to 4, we allow the "smart" selection phase to resolve more variables, reducing the burden on the simulation phase and leading to faster convergence.

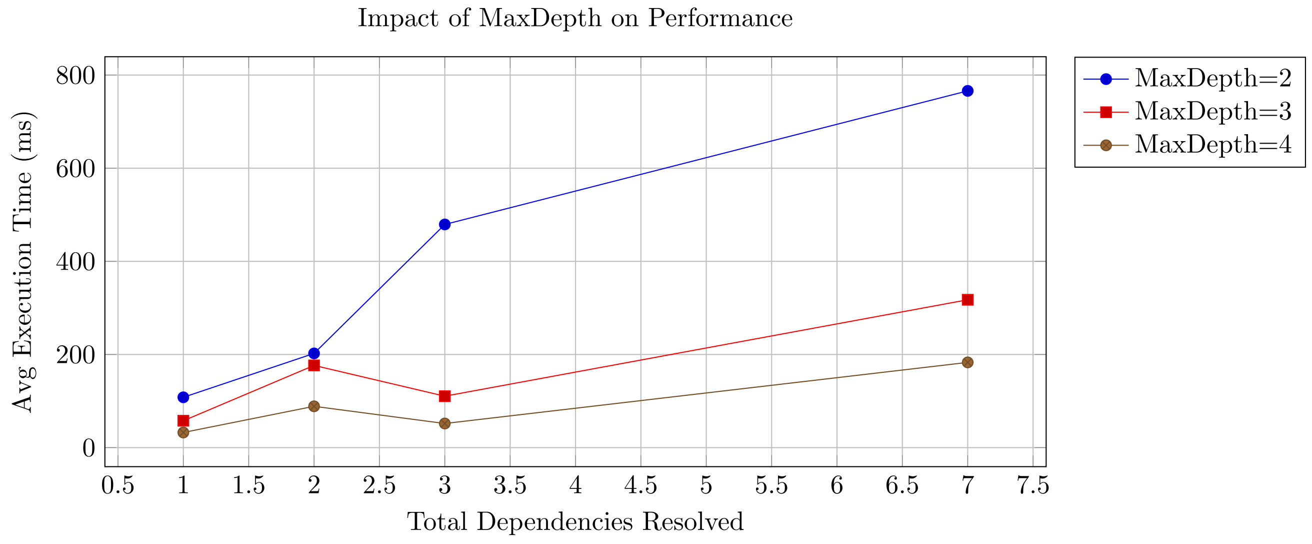

Figure 1: Execution time decreases as Tree Depth increases, indicating that the MCTS tree policy is more efficient than the random rollout.

4.2.2 Impact of Computational Budget

Figure 2 confirms the expected linear relationship between the number of MCTS iterations (k) and execution time.

- At $k = 50$, the solver is extremely fast (< 100ms) but may fail to converge on highly complex graphs like `react-scripts`.
- At $k = 1000$, execution time exceeds 1.5 seconds for complex graphs.

The data suggests that for interactive applications (e.g., CLI tools), a budget of $k = 200$ represents an optimal trade-off, keeping execution time under 400ms while providing sufficient search coverage.

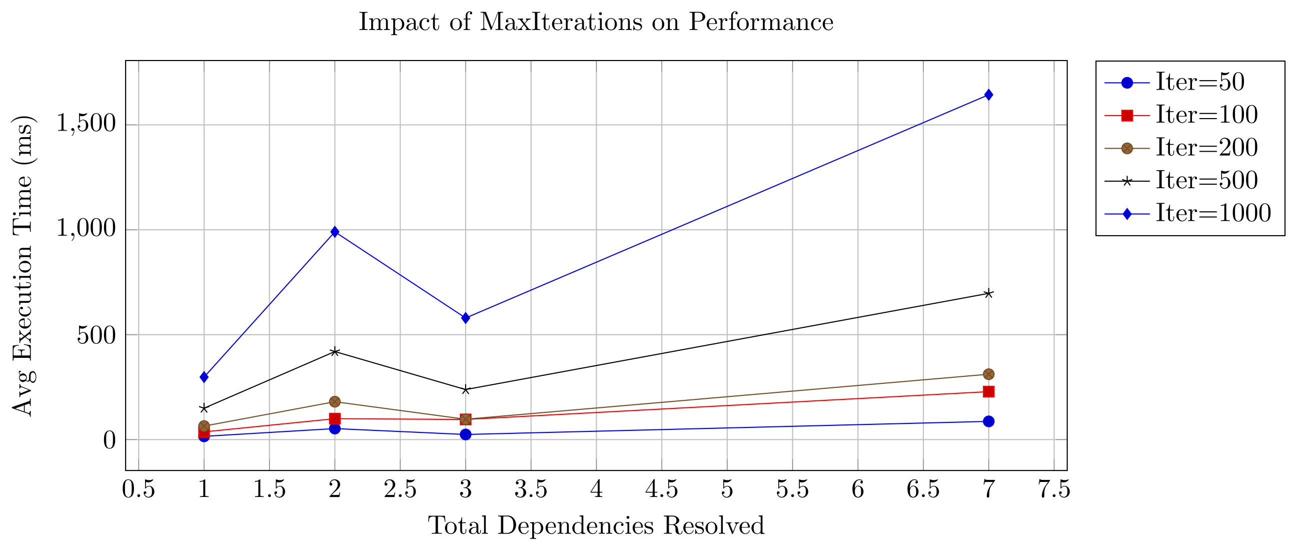

Figure 2: Linear scaling of execution time relative to the iteration budget.

4.2.3 Impact of Simulation Horizon

Figure 3 analyzes the depth of the rollout phase (d_{sim}). We observe a distinct performance cliff. For $d_{sim} \in \{5, 10, 15\}$, performance remains stable and efficient. However, increasing d_{sim} to 50 results in a sharp increase in execution time. This indicates that the critical constraints in the npm ecosystem are typically local (within 10-15 degrees of separation). Extending the simulation beyond this horizon ($d_{sim} = 50$) forces the solver to traverse irrelevant deep dependencies, adding computational overhead without contributing significant information to the Reward function R.

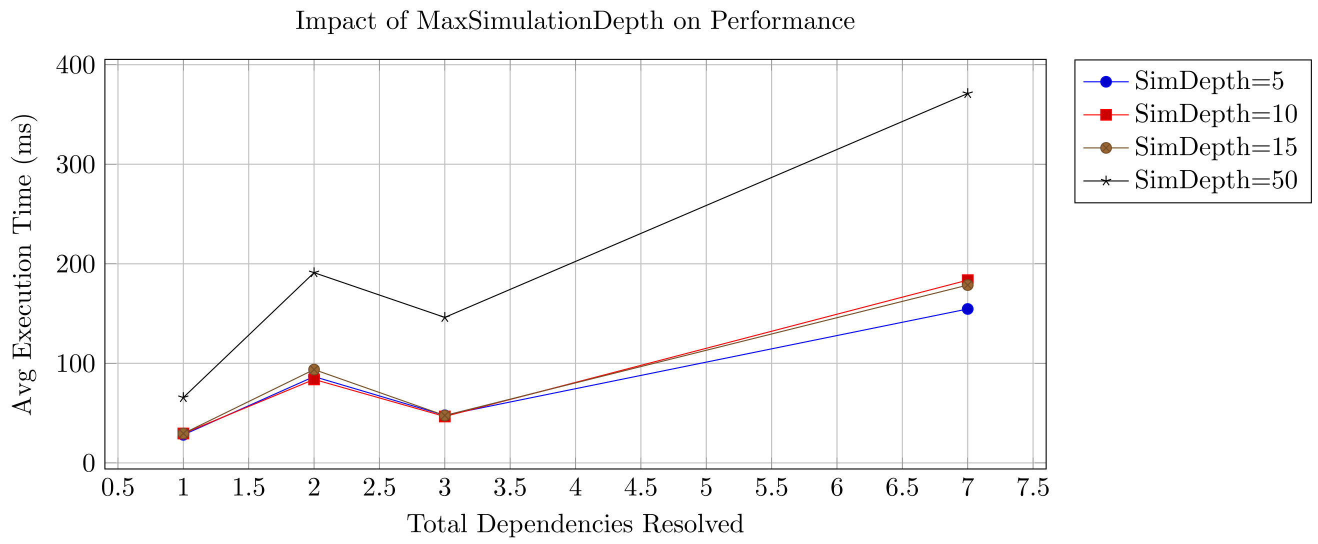

Figure 3: Performance degrades significantly when simulation depth exceeds the effective diameter of the dependency graph.

4.2.4 Impact of Simulation Bias (λ)

Figure 4 analyzes the sensitivity of the solver to the weighting parameter λ used in the probabilistic rollout (7). This parameter controls the "greediness" of the heuristic simulation: a low λ approaches a uniform random walk, while a high λ approaches a deterministic greedy search for the latest version.

The results reveal a non-linear relationship with a distinct performance "sweet spot" at $\lambda = 1.5$:

- **Low Bias ($\lambda = 0.5$ - Random):** The solver exhibits poor performance (Avg: 317ms). With a flatter probability distribution, the simulation phase frequently selects older versions during the rollout. This "stale" guidance results in low recency rewards (R), providing a weak gradient for the MCTS backpropagation to identify the optimal path.
- **High Bias ($\lambda \geq 3.0$ - Greedy):** Performance recovers and becomes efficient ($\approx$ 145ms). For the majority of the test corpus (standard healthy packages), the latest versions are valid. A high λ forces the simulation to immediately verify the "latest" path. If valid, it returns a high reward ($R \approx 1.0$) quickly. However, this setting is risky for complex conflict scenarios, as the lack of randomness prevents the simulation from stumbling upon valid older versions.
- **Optimal Balance ($\lambda = 1.5$):** The fastest execution times ($\approx$ 140ms) are observed here. This setting provides strong heuristic guidance (favoring new versions) while retaining sufficient entropy to explore alternative version combinations when the latest versions induce a conflict.
- **Transition Instability ($\lambda = 2.0$):** A notable performance degradation (spike to 392ms) occurs at $\lambda = 2.0$. We hypothesize this represents a region of probabilistic interference where the distribution is neither random enough to bypass conflicts easily nor greedy enough to force a rapid (but potentially failed) conclusion, causing the solver to thrash between suboptimal branches.

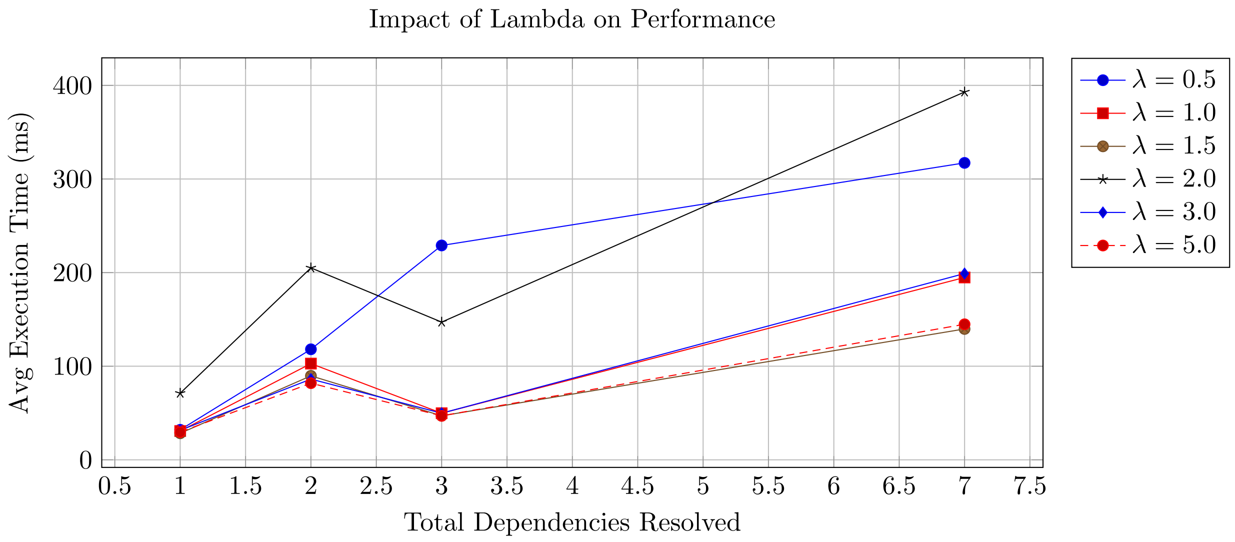

Figure 4: Exploration parameter tuning. $\lambda = 1.5$ provides the optimal balance, while deviations lead to increased search times.

5 Discussion

The results presented in Section 4 demonstrate that Monte Carlo Tree Search is a viable and effective strategy for solving the npm dependency resolution problem. In this section, we analyze the theoretical implications of this approach, specifically regarding how it addresses the dual challenges of satisfiability and optimization, its algorithmic complexity, and its convergence properties.

5.1 Methodological Efficacy

The core contribution of this work is the reformulation of dependency resolution from a standard Constraint Satisfaction Problem (CSP) to a stochastic optimization problem. Traditional approaches typically fall into two distinct categories:

1. **Greedy Algorithms:** These select the latest version at every step. While computationally efficient, they fail catastrophically in "Diamond Dependency" scenarios where a transitive choice necessitates an older version of a shared package.

2. **SAT Solvers:** These guarantee a valid solution if one exists but lack an inherent mechanism to prioritize recency. A pure SAT solver is as likely to return version 1.0.0 as it is to return 5.0.0, provided both satisfy constraints.

Our MCTS implementation effectively bridges this gap. By encoding the recency metric into the Reward function (9), we convert the qualitative preference for new software into a quantitative gradient that guides the search.

The sensitivity analysis of the simulation bias (λ) provides a crucial insight into the topology of the npm ecosystem. The distinct performance degradation observed at high λ values (Greedy) confirms that the dependency graph is not monotonic; "newer" does not always imply "compatible." Conversely, the poor performance at low λ values (Random) confirms that the search space is too vast for unguided exploration. The experimentally derived optimum of $\lambda \approx 1.5$ suggests that dependency resolution requires a hybrid strategy: a strong heuristic bias to drive optimization, coupled with sufficient entropic exploration to bypass local constraint optima (breaking changes).

5.2 Computational Complexity

The search space for dependency resolution is combinatorial. Let N be the number of packages in the dependency closure, and let b be the average number of available versions per package. The total state space size is $O(b^N)$. Since checking satisfiability is NP-complete, an exhaustive search is intractable.

The computational complexity of the MCTS approach is nominally determined by the iteration budget k and the rollout cost. However, our experimental results regarding 'MaxDepth' reveal a counter-intuitive efficiency characteristic. We observed that increasing the tree depth limit actually *decreased* execution time.

This implies that the UCT selection policy (6) is significantly more computationally efficient than the stochastic rollout. By allowing the tree to grow deeper, the algorithm shifts the burden of resolution from the "blind" simulation phase to the "informed" selection phase. The UCT policy effectively prunes the search space by locking in variables that have high statistical promise, whereas a shallow tree forces the solver to rely on random sampling to resolve complex, deep constraints.

Consequently, while the worst-case complexity remains exponential, the *effective* complexity for practical npm trees is managed by the "Effective Horizon." The sharp performance cliff observed when 'MaxSimulationDepth' > 15 indicates that constraint propagation in this domain exhibits strong locality of reference; conflicts are typically resolved within a shallow radius of the root, allowing the solver to prune vast sections of the theoretical search space.

5.3 Convergence

Convergence in this context refers to two distinct properties: converging to a *valid* solution and converging to the *optimal* (most recent) solution.

The Upper Confidence Bound (UCT) formula guarantees that as $k \to \infty$, the probability of exploring the optimal branch approaches 1. The exploration term ($C_p\sqrt{\ln n/n_j}$) prevents the solver from getting stuck in local optima (e.g., a valid but old set of versions).

In our implementation, convergence is accelerated by the heuristic rollout. By weighting the random simulation toward newer versions, we inject domain knowledge that steers the solver toward high-reward states faster than a uniform random walk. However, this creates a tension in "Adversarial" scenarios (e.g., legacy projects). The solver must "unlearn" its bias toward new versions through repeated constraint violations (zero rewards) before the UCT exploration term forces it to consider the older, valid versions. This unlearning curve explains the linear scaling of execution time with graph complexity: larger graphs increase the probability of encountering a "breaking change" deep in the tree, requiring more iterations to overcome the heuristic bias.

5.4 Correctness

We distinguish between *soundness* (if a solution is returned, is it valid?) and *completeness* (if a solution exists, will it be found?).

Soundness: The implementation guarantees soundness through the expansion and reward mechanisms. A node is only expanded if the version satisfies the immediate parent's constraints (1), and a positive reward is only generated if the simulation reaches a state with zero pending dependencies and zero conflicts. Therefore, any solution returned with $R > 0$ is mathematically guaranteed to be a valid configuration of the dependency graph.

Completeness: As a stochastic method, MCTS is probabilistically complete but not deterministically complete within a bounded timeframe. Given infinite iterations, it will traverse every valid state. However, under a finite iteration cap, it is possible for the solver to fail to find a valid configuration for highly constrained graphs that a deterministic CDCL (Conflict-Driven Clause Learning) solver might find. This is a deliberate trade-off: we sacrifice the guarantee of finding *obscure* solutions in exchange for the ability to reliably find and optimize *standard* solutions.