Performance of Various Negative Cycle Detection Algorithms When Solving Formulas in Quantifier-Free Difference Logic

Sherwin Lai
Stanford University
sherwlai@stanford.edu

Ammar Ratnani
Stanford University
aratnani@stanford.edu

***Abstract*—Difference logic is a somewhat popular fragment of linear arithmetic which has seen applications in temporal reasoning. The problem of solving conjunctive difference logic formulas is equivalent to that of finding negative cycles in a graph. Therefore, this report measures the performance of various negative cycle detection algorithms for this application. Evaluation of our implementation on benchmarks reveals that applying the T, I, and E rules with quadratic blowup for preprocessing along with Tarjan's Algorithm as the theory-solving strategy yields the best performance.**

***Index Terms*—Formal methods, constraint satisfaction.**

I. Introduction

Quantifier-free integer difference logic (`QF_IDL`) is a somewhat popular fragment of quantifier-free linear integer arithmetic (`QF_LIA`) [1, 2]. While the theory is much less expressive than linear arithmetic, it is much more efficient as its conjuctive fragment can be decided in just cubic time, in contrast to the exponential time required for conjunctive `QF_LIA` [1]. Furthermore, despite its limited expressive power, `QF_IDL` has applications in temporal reasoning [3], such as job scheduling [1] and high-level synthesis (HLS) [4].

The problem of solving conjunctive `QF_IDL` can be reduced to that of finding negative cycles in a graph. Even though that problem has a classic solution dating back to the late 1950s [5], it still admits a surprising amount of depth and room for optimization. Therefore, this report measures the performance of various negative cycle detection algorithms in the context of Satisfiability Modulo `QF_IDL`.

The repository for this project is available on GitHub at https://github.com/ammrat13/qf-idl-solver.

II. Background

Most of the background for negative cycle detection comes from [6], a survey and performance analysis of the algorithms known at the time. Even today, it appears to be a foundational work on the topic [7].

A. Integer Difference Logic

In `QF_IDL`, the only places non-Bool terms may appear are in expressions of the form $x - y \leq k$. Here, x and y are free variables of sort Int for which we seek a satisfying assignment, and k is an integral constant. Naturally, $\mathsf{dom}(\mathsf{Int}) = \mathbb{Z}$, and the symbols $-$ and $\leq$ have their standard meanings over the integers. [1]

For now, consider the conjunctive fragment of difference logic, since a decision procedure for that can be extended to one for all of `QF_IDL` via DPLL(T) [1]. So, suppose we wish to decide a conjunction of terms each of which either have the aforementioned form or its negation. We may rewrite

$$\begin{aligned}\neg(x - y \leq k) &\Leftrightarrow (x - y > k) \\ &\Leftrightarrow (y - x < -k) \\ &\Leftrightarrow (y - x \leq -k - 1),\end{aligned}$$

so we only have to consider positive terms without loss of generality.[1] [1]

Given a conjunction of such positive terms ϕ, construct an *inequality graph* $G = (V, E)$, where V consists of all the free variables of ϕ. For each constraint $(x - y \leq k) \in \phi$, create an edge from x to y with weight k.[2] We have that ϕ is satisfiable iff G has no negative cycle, so running an algorithm to find such cycles gives a decision procedure for `QF_IDL`. [1]

B. Label-Correcting Methods

One approach to detecting negative cycles is solving the single-source shortest paths problem, and a popular way to do that is by a *label-correcting method*. An algorithm following this paradigm tags each vertex as one of:

- *unlabeled* if the algorithm has not yet considered it,
- *labeled* if it has been flagged for further processing, or
- *scanned* if no further action is required on it for now.

In addition, the algorithm maintains a *tentative distance* to each vertex $d(v)$. Sometimes d is called the *potential*. Initially, the lone source vertex s is labeled with $d(s) = 0$ and all other vertices v are unlabeled with $d(v) = \infty$. [9]

On each step, an algorithm following the label-correcting method takes a labeled vertex u and *scans* it. To do that, it first tries to *relax* all the outgoing edges uv with weight w_{uv}. A single relaxation consists of computing $d(u) + w_{uv}$ and

[1] Note that the last line requires us to operate over $\mathbb{Z}$. Over $\mathbb{R}$, the last line would read $(y - x \leq -k - \epsilon)$ for sufficiently small $\epsilon > 0$. There exist methods to pick ϵ ahead of time [8], or it can be operated on symbolically.

[2] By mistake, the benchmarks were actually run using the transpose of G instead of G itself. However, this does not compromise correctness — G^R has a negative cycle iff G does.

comparing it to $d(v)$. If going to v via u gives an improvement to v's tentative distance, $d(v)$ is set to $d(u) + w_{uv}$ and v is marked as labeled. After all the outgoing edges have been relaxed, u is marked as scanned. [9]

As an example, Dijkstra's Algorithm (DJ) can be framed as a label-correcting method. For DJ, labeled vertices are precisely those in the priority queue, and at each step the labeled vertex with the shortest tentative distance is selected to be scanned. The scanning order means that DJ never transitions a vertex from scanned to labeled. [9]

C. Derived Graphs

When trying to find a negative cycle in an inequality graph G, several related graphs are often useful. For example, the *augmented inequality graph* G' is formed from G by creating a dummy vertex v^* and adding edges with zero weight from it to every other vertex. Note that G' has a negative cycle iff G does, so we always operate on augmented graphs without loss of generality. Furthermore, v^* provides a natural starting point for a single-source shortest paths algorithm. [3]

The algorithms considered in this report always maintain a path from the source vertex to all the other (known) vertices. This is done by having each vertex record its parent along the shortest known path to it. At any point during the algorithm's execution, the *parent graph* G_p is formed from G by keeping all the edges participating in this parent relation. Nominally G_p is a forest, but it may not be if G has negative cycles. [6]

The algorithms also maintain tentative distances $d(v)$. Call an edge uv with weight w_{uv} *admissible* if it can successfully be relaxed or if it is tight. In other words, uv is admissible if $d(v) \geq d(u) + w_{uv}$. At any point, the *admissible graph* G_d is G with all non-admissible edges removed. [6, 10]

D. Bellman-Ford Algorithm

The Bellman-Ford (BF) algorithm is the classic way to detect negative cycles in a graph. It starts by setting the tentative distance of the source vertex to zero, and the tentative distances of all the other vertices to ∞. The algorithm then runs $|V| - 1$ passes. In each pass, it iterates over all the edges (in some arbitrary order) and attempts to relax each of them. If after all the passes complete there still exists an edge uv that can be relaxed, then the shortest known path to v after that relaxation completes must contain a negative cycle, and the parent graph can be traversed to find it. [5, 9, 11]

The runtime of this algorithm is $\mathcal{O}(|V| \cdot |E|)$. While the other algorithms considered in this report provide speedups in practice, they maintain the same worst-case runtime.

E. Shortest Path Faster Algorithm

BF can be rephrased as a label-correcting scheme. To do so, we specify an iteration order over the edges: we iterate over all the vertices (again in an arbitrary order) and for each vertex iterate over its outgoing edges. With that iteration order, it makes sense to analyze how BF processes individual nodes.

Recall that unlabeled vertices are intuitively those that the algorithm doesn't know about yet. For BF these are vertices u with tentative distance $d(u) = \infty$. When BF processes an unlabeled vertex, it won't make any changes to the potential. Any possible paths through u have infinite length and thus are discarded. Similarly consider the case of a scanned vertex u, which for BF corresponds to a vertex it previously processed which has not has its tentative distance reduced since it was last seen. Note that, again, BF won't be able to relax any outgoing edges from u. This is because, after u was processed previously, $d(v)$ was set to at most $d(u) + w_{uv}$, for every $uv \in E$ of weight w_{uv}. The potentials $d(v)$ only ever decrease, and the fact $d(u)$ didn't change means it can't possibly relax any edge uv this time. Ultimately, this means BF only truly processes labeled vertices.

This insight can be exploited to construct the Shortest Path Faster Algorithm (SPFA). Instead of processing every vertex on each pass, the algorithm only processes those marked as labeled. During each pass, it records which vertices transitioned into the labeled state and uses that as the set of vertices to process on the next pass. The termination criteria is the same as with BF — a negative cycle exists if relaxations are still possible after $|V| - 1$ passes.

Usually, SPFA is implemented in terms of a queue rather than a set of labeled vertices. Additionally, SPFA is often referred to as BF. [9, 12, 13]

F. Amortized Parent Graph Search

While SPFA can stop early if a pass makes no relaxations, it can't do that when G has a negative cycle. This makes its basic form, "uncompetitive," according to [6]. Still, it is possible to augment SPFA with some early termination in that case by consulting the parent graph constructed over the course of the algorithm. Specifically, if G has a negative cycle, then it will eventually be reflected in G_p.

The naïve approach is to check if G_p gained a cycle after every relaxation. After reparenting v onto u, we can follow the parent pointers starting at u and check if we eventually reach v. This check takes $\mathcal{O}(|V|)$ time, increasing the runtime to $\mathcal{O}(|V|^2 \cdot |E|)$ since it has to be done for every relaxation. Alternatively, G_p can be checked for cycles every $r > 1$ steps. Since G_p is unweighted, its cycles can be detected using Depth-First Search (DFS). DFS takes $\mathcal{O}(|V|)$ time, so running it every $r = |V|$ iterations of the inner loop won't change the (asymptotic) worst-case runtime. [6, 12]

This report considers SPFA with Amortized Parent Graph Search (SPFACS). Unfortuntately, this method does not ensure negative cycles are found as soon as they appear. In fact, since cycles may disappear from G_p as BF or SPFA continue running, they can take a long time to be detected. [6]

G. Tarjan's Algorithm

Tarjan's Algorithm (TARJ), also referred to as BFCT, provides a way to acheive immediate cycle detection while maintaining a worst-case $\mathcal{O}(|V| \cdot |E|)$ runtime. The key insight is that, when the potential of a vertex u decreases, the length shortest known path of all of u's children v in G_p decreases as well. Therefore, it does not make sense to try to scan v until

the information from u has had a chance to propagate to it. We may as well consider v unlabeled until it is rediscovered by the relaxations propagated by u.

Implementing this insight into SPFA gives TARJ. Whenever SPFA relaxes $uv \in E$ to mark v as labeled, we first find all of v's children in G_p using DFS. If u is a child of v, then we've found a negative cycle. Otherwise, we mark all of v's children as unlabeled and unparent them. This maintains the worst-case runtime of Bellman-Ford since constantly disassembling subtrees tends to keep them small. Alternatively, TARJ can be seen as traversing all the children without ever unparenting, but amortizing the traversal over multiple relaxations. [6]

There are some important implementation details for TARJ. First, since vertices can now transition from labeled to unlabeled, it is possible for unlabeled vertices to be in the queue for processing. If an unlabeled vertex is popped from the front of the queue, just ignore it. Additionally, don't add a vertex to the queue if it's already in the queue, even if it's unlabeled. Just mark it as labeled and move on.

In [6], TARJ performed the best. Even ten years later in [7], TARJ is still competitive.

III. Methods

A. Design

Algorithm 1 Overview of the implemented solver

```
Require: φ is a formula in QF_IDL
  procedure SOLVE(φ)
    α := e(φ)
    x := TSEITIN(α)
    y := PARTIALEAGER(φ)
    F ← x ∧ y                          ▷ Formula for solver
    loop
      r ← SATSOLVE(F)
      if r is UNSAT then
        return UNSAT
      else
        s ← NEGATIVECYCLE(r)           ▷ Theory solver
        if s is SAT then
          return SAT
        else
          F ← F ∧ s                    ▷ Blocking clause
        end if
      end if
    end loop
  end procedure
```

The high-level structure of the procedure we implemented is given in Algorithm 1. This procedure is a variant of the Algorithm 11.2.1 in [1]. Even though it is less efficient than DPLL(T), this approach lets us to use an off-the-shelf SAT solver. We wrote in Go, so we used Gini [14] for that.

The NegativeCycle subroutine represents a theory solver for conjunctive QF_IDL. It constructs its input's inequality graph and returns SAT if no negative cycle exists in it. Otherwise, it returns a blocking clause for that cycle. Our program lets us choose between:

- BF (`bf-basic`),
- BF with basic early stopping (`bf-full`),
- SPFA (`spfa-basic`),
- SPFACS (`spfa-full`), and
- TARJ (`tarjan`)

for NegativeCycle's implementation. All our implementations operate on augmented inequality graphs, with v^* represented implicitly rather than as an actual node.

$$\frac{C := x - y \le k \quad D := x - y \le \ell \quad k \le \ell}{\neg C \vee D}\ \mathsf{T}$$

$$\frac{C := x - y \le k \quad D := y - x \le \ell \quad k \ge -\ell - 1}{C \vee D}\ \mathsf{I}$$

$$\frac{C := x - y \le k \quad D := y - x \le \ell \quad k \le -\ell - 1}{\neg C \vee \neg D}\ \mathsf{E}$$

Fig. 1. The T, I, and E rules are the the preprocessing rules our program implements. They can alternatively be read as Transitivity, Forced Inclusion, and Forced Exclusion.

Additionally, we preprocess the input formula ϕ in the PartialEager subroutine. We follow the same approach as [8], though we use a slightly different formalism. The preprocessing rules we implement are given in Fig. 1. All these rules consider atomic formulas that share their two variables x and y. The T rule says that if the difference $x - y$ is less than or equal to some number k, then the same also holds for the other numbers $\ell \ge k$. The I and E rules consider what happens when x and y have different "polarity" in the two constraints. In that case, the clauses C and D will describe the set of values $x - y$ can take as rays pointing in opposite directions. If the rays overlap, $x - y$ must be in at least one of the rays — that's I. If they don't, $x - y$ can't be in both rays — that's E. As a special case, if the two rays partition $\mathbb{Z}$, then $x - y$ must be in exactly one of the rays — the rules handle this.

Our program allows us to apply all the preprocessing rules, just apply the T rule, or apply no preprocessing at all. It also allows us to apply variants of these rules that only result in a linear number of extra clauses instead of a quadratic number. To do this, T and I only consider the smallest ℓ that applies, while E only considers the largest. The remaining implications can be derived through Boolean Constraint Propagation.

B. Evaluation

To test the performance of the various algorithms, we employ some QF_IDL benchmarks taken from SMT-COMP'23 (the 2023 International Satisfiability Modulo Theories (SMT) Competition), as made available on StarExec. [2] For each benchmark, we evaluate performance of the preprocessor-solver configurations given in Table 1.

Note that we evaluate only on a subset of all possible preprocessor-solver configurations—specifically, we fix preprocessor on `tie-quad` to test the various solvers, and we fix solver on `tarjan` to test the various preprocessors.

TABLE 1
PREPROCESSOR-SOLVER CONFIGURATIONS EVALUATED

Preprocessor	Solver
nil	tarjan
t-lin	tarjan
tie-lin	tarjan
tie-quad	bf-basic
tie-quad	bf-full
tie-quad	spfa-basic
tie-quad	spfa-full
tie-quad	tarjan

For computational feasibility reasons, we enforce a timeout on the runs of various large problems. If the run's duration exceeds the timeout, the program gives up solving. Timeout excludes the time taken to ingest and preprocess the file. Following termination of the program, we collect and report the following metrics:

- *Ingest duration*: How long it took to parse the file and convert it into CNF.
- *Preprocess duration*: How long preprocessing, if enabled, took.
- *SAT solver duration*: The amount of time spent in the SAT solver.
- *Theory solver duration*: The amount of time spent in the theory solver.
- *Graph overhead duration*: The amount of time taken to construct the constraint graph.
- *Learn overhead duration*: How long it took to learn new clauses from cycles.
- *Solver calls*: The number of times the SAT and theory solvers were invoked.
- *Theory solver loops*: The number of $O(1)$ loops that took place inside the theory solver.[3]

IV. RESULTS

First, let us consider the results, reported in Table 2, of a benchmark whose problem was not so complex so as to require an enforced timeout: `planning/plan-49.cvc.smt2`. (All runs reported fully terminated without premature interruption.) Prepossessing takes a negligible amount of time for all preprocessor-solver combinations. We see that the various preprocessing methods reduce the time spent in the SAT and theory solvers and diminish the number of solver calls and theory solver loops needed, with the `tie-quad` producing the most substantial savings. As for the solver, `spfa-basic` clearly yields the worst performance, but `spfa-full` significantly outperforms `bf-basic` and `bf-full` especially with respect to the theory solver. And `tarjan`'s performance surpasses that of all other solvers.

What about the results of more complex benchmarks—those that require more computation? Table 3 reports the results of our code on a more "difficult" benchmark: `bench/queens_bench/super_queen/super_queen100-1.smt2`. For computational feasibility we had to limit runtime of our code on this benchmark to 10 minutes per run. The results for `super_queen100-1.smt2` are similar to those for `plan-49.cvc.smt2` in many respects: they reinforce the conclusion that `tie-quad` is the best preprocessor and `tarjan` is the best solver. See Appendix A for results on other benchmarks.

[3]Because SAT solvers use some randomness under the hood [15], even without a timeout the code does not run deterministically, so the precise number of solver calls and theory solver loops may vary with each run.

TABLE 2
BENCHMARK STATS: `PLAN-49.CVC.SMT2`

Preproc	Solver	Ing (s)	Prpr (s)	SAT Slvr (s)	Theory Slvr (s)	Graph OH (s)	Learn OH (s)	Slvr Calls	Theory Slvr Loops
nil	tarjan	1.447	0.000	14.516	2.570	3.221	4.304	5,417	11,630,065
t-lin	tarjan	1.237	0.001	13.199	2.284	2.901	3.683	4,367	10,127,253
t-quad	tarjan	1.026	0.001	10.980	1.955	2.378	3.100	3,722	8,560,882
tie-lin	tarjan	1.350	0.080	7.870	1.617	1.724	2.097	2,726	6,806,141
tie-quad	bf-basic	1.137	0.084	18.659	483.524	3.815	4.430	4,022	2,625,888,988
tie-quad	bf-full	1.141	0.004	18.259	466.166	3.608	4.166	3,911	2,560,383,273
tie-quad	spfa-basic	1.139	0.086	48.318	1051.088	9.563	11.853	9,863	5,488,883,778
tie-quad	spfa-full	1.029	0.004	12.372	2.018	2.572	3.171	3,870	38,498,536
tie-quad	tarjan	1.136	0.003	8.815	1.610	1.881	2.225	2,766	6,867,988

TABLE 3
BENCHMARK STATS: `SUPER_QUEEN100-1.SMT2` (Timeout = 10m)

Preproc	Solver	Ing (s)	Prpr (s)	SAT Slvr (s)	Theory Slvr (s)	Graph OH (s)	Learn OH (s)	Slvr Calls	Theory Slvr Loops
nil	tarjan	1.643	0.000	246.213	38.223	260.727	54.803	36,571	165,678,326
t-lin	tarjan	1.566	0.007	232.348	31.897	285.113	50.592	35,357	129,898,023
t-quad	tarjan	1.562	0.011	233.517	34.759	281.802	49.868	34,367	143,246,445
tie-lin	tarjan	1.556	0.083	244.415	36.266	269.911	49.370	34,247	140,104,389
tie-quad	bf-basic	1.633	0.038	24.877	535.932	36.033	3.149	3,687	2,988,366,121
tie-quad	bf-full	1.659	0.083	25.974	533.988	36.155	3.850	3,726	3,016,065,208
tie-quad	spfa-basic	1.856	0.032	24.526	534.480	37.250	3.650	3,753	2,988,363,670
tie-quad	spfa-full	1.659	0.036	258.388	52.267	242.770	46.542	31,119	1,142,562,019
tie-quad	tarjan	1.669	0.086	257.954	34.315	261.556	46.124	33,464	132,036,170

V. RELATED WORK

This section briefly mentions some other algorithms for negative cycle detection that we were not able to implement within the time constraint.

A. Goldberg-Radzik Algorithm

The Goldberg-Radzik Algorithm (GORC) improves on SPFA by changing the scanning order from first-in-first-out. The insight is that, for each pass, it's best to first scan vertices which won't have their potential decreased later in the pass. This way, we propagate information as far as we can, instead of having to wait for the next pass to tell downstream vertices about their new tentative distances. GORC implements this insight by scanning vertices in the order of a topological sort on the admissible graph G_d. [10]

Of course, such a topological sort may not exist. Fortunately, that only happens if G has negative or zero weight cycles [10]. Unfortunately, we allow zero weight cycles. The original authors in [10] suggest resolving such cycles via an expensive contraction operation. The survey in [6] suggests simply ignoring flagged zero weight cycles, but we found that approach to be unsound, and [7] corroborates that. The survey in [7] suggests an approach that gives 20% overhead.

Ultimately, since TARJ outperformed GORC in [6], we decided to just implement that.[4]

[4]Sadly, we only decided to do this after we had implemented GORC. We didn't realize it failed on zero weight cycles until the code was done.

B. D'Esopo-Pape Algorithm

The D'Esopo-Pape Algorithm presents an interesting heuristic improvement over SPFA. It records whether each vertex has been scanned at any point in the past. If a previously scanned vertex has its tentative distance reduced, it is placed at the front of the queue instead of at the end. [16]

While this algorithm usually works well, it has an exponential worst-case runtime [17], so we didn't implement it.

C. Other Algorithms

A more recent survey [7] proposes two more algorithms which are competitive with TARJ. One modifies the first pass to greedily follow vertices as they are labeled so as to find "easy" negative cycles. Another uses a more complicated heuristic to choose the order in which to scan vertices.

The method presented in [18] proposes a simple modification to the scanning order of BF to give a factor of two speedup over it on dense graphs. The method in [19] gives a further speedup on average by using a random scanning order.

D. Incremental Methods

The idea of [4] is to maintain the tentative distance of each vertex between calls to the negative cycle detection algorithm. This way, priority is given to vertices involved in constraint violations. A more aggressive incremental approach is given in [3]. It restricts updates to one edge at a time, so it would not work for this report, but it likely outperforms TARJ since it uses DJ under the hood.

VI. Conclusion

We explored various algorithms and preprocessing techniques for solving formulas in Quantifier-Free Difference Logic (`QF_IDL`), with a focus on their application to negative cycle detection in graphs. We tested various configurations of preprocessors and solvers on a selection of benchmarks from the SMT-COMP'23 competition. Our findings indicate performance differences among these configurations, demonstrating the importance of choosing the right combination for efficient `QF_IDL` solving.

Preprocessing techniques, particularly `tie-quad`, consistently improved solver performance by reducing the time spent in the SAT and theory solvers, as well as the number of solver calls and theory solver loops. Among the solvers, Tarjan's Algorithm (`tarjan`) consistently outperformed others, such as Bellman-Ford (`bf-basic` and `bf-full`) and Shortest Path Faster Algorithm variants (`spfa-basic` and `spfa-full`).

Our results suggest that the combination of `tie-quad` preprocessing and `tarjan` solver offers the most efficient approach for solving `QF_IDL` formulas in the context of negative cycle detection. This finding can guide future development of `QF_IDL` solvers and inform researchers and practitioners in the field of formal methods and constraint satisfaction about the most effective strategies for handling such problems.

References

[1] D. Kroening and O. Strichman, *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.

[2] F. Bobot, M. Bromberger, and J. Hoenicke, "SMT-COMP 2023," https://smt-comp.github.io/2023/, July 2023, [Online; accessed 4-November-2023].

[3] G. Ramalingam, J. Song, L. Joskowicz, and R. E. Miller, "Solving systems of difference constraints incrementally," *Algorithmica*, vol. 23, no. 3, pp. 261–275, Mar 1999. [Online]. Available: https://doi.org/10.1007/PL00009261

[4] N. Chandrachoodan, S. S. Bhattacharyya, and K. R. Liu, "Adaptive negative cycle detection in dynamic graphs," in *ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No.01CH37196)*, vol. 5, 2001, pp. 163–166 vol. 5.

[5] J. Bang-Jensen and G. Z. Gutin, *Digraphs: Theory, Algorithms and Applications*. Springer, 2009.

[6] B. V. Cherkassky and A. V. Goldberg, "Negative-cycle detection algorithms," *Mathematical Programming*, vol. 85, no. 2, pp. 277–311, Jun 1999. [Online]. Available: https://doi.org/10.1007/s101070050058

[7] B. V. Cherkassky, L. Georgiadis, A. V. Goldberg, R. E. Tarjan, and R. F. Werneck, "Shortest-path feasibility algorithms: An experimental evaluation," *ACM J. Exp. Algorithmics*, vol. 14, jan 2010. [Online]. Available: https://doi.org/10.1145/1498698.1537602

[8] A. Armando, C. Castellini, E. Giunchiglia, and M. Maratea, "A SAT-based decision procedure for the boolean combination of difference constraints," in *Theory and Applications of Satisfiability Testing*, H. H. Hoos and D. G. Mitchell, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 16–29.

[9] R. E. Tarjan, *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[10] A. V. Goldberg and T. Radzik, "A heuristic improvement of the bellman-ford algorithm," *Applied Mathematics Letters*, vol. 6, no. 3, pp. 3–6, 1993. [Online]. Available: https://www.sciencedirect.com/science/article/pii/089396599390022F

[11] Wikipedia contributors, "Bellman-ford algorithm — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Bellman%E2%80%93Ford_algorithm&oldid=1188652567, 2023, [Online; accessed 9-December-2023].

[12] T. H. Long, "Using the shortest path faster algorithm to find a negative cycle," https://konaeakira.github.io/posts/using-the-shortest-path-faster-algorithm-to-find-negative-cycles.html, May 2020, [Online; accessed 9-December-2023].

[13] Wikipedia contributors, "Shortest path faster algorithm — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Shortest_path_faster_algorithm&oldid=1187573682, 2023, [Online; accessed 9-December-2023].

[14] S. Cotton, "go-air/gini: Sapeur," Jan. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2553490

[15] N. Eén and N. Sörensson, "An extensible SAT-solver," http://minisat.se/downloads/MiniSat.pdf, 2003, [Online]. Appendix A.

[16] U. Pape, "Implementation and efficiency of Moore-algorithms for the shortest route problem," *Mathematical Programming*, vol. 7, no. 1, pp. 212–222, Dec 1974. [Online]. Available: https://doi.org/10.1007/BF01585517

[17] A. Kershenbaum, "A note on finding shortest path trees," *Networks*, vol. 11, no. 4, pp. 399–400, 1981. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/net.3230110410

[18] J. Y. Yen, "An algorithm for finding shortest routes from all source nodes to a given destination in general networks," *Quarterly of Applied Mathematics*, 1970. [Online]. Available: https://www.ams.org/journals/qam/1970-27-04/S0033-569X-1970-0253822-7/

[19] M. J. Bannister and D. Eppstein, *Randomized Speedup of the Bellman–Ford Algorithm*. ANALCO'12, 2012, pp. 41–47. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611973020.6

APPENDIX A
ADDITIONAL EVALUATION RESULTS

TABLE 4
BENCHMARK STATS: QUEEN100-1.SMT2 (Timeout = 5m)

Preproc	Solver	Ing (s)	Prpr (s)	SAT Slvr (s)	Theory Slvr (s)	Graph OH (s)	Learn OH (s)	Slvr Calls	Theory Slvr Loops
nil	tarjan	1.449	0.000	129.466	14.580	128.916	27.017	18,150	68,042,894
t-lin	tarjan	1.654	0.007	98.797	32.809	147.097	21.276	15,733	142,395,436
t-quad	tarjan	1.639	0.012	105.243	29.518	140.340	24.884	16,115	122,350,857
tie-lin	tarjan	1.435	0.023	106.344	29.826	142.520	21.286	15,972	129,745,423
tie-quad	bf-basic	1.447	0.103	8.522	273.332	16.819	1.326	1,587	1,585,064,527
tie-quad	bf-full	1.560	0.109	8.897	273.257	16.426	1.386	1,562	1,559,875,285
tie-quad	spfa-basic	1.661	0.087	10.206	271.305	16.635	1.700	1,598	1,541,352,304
tie-quad	spfa-full	1.679	0.079	111.307	32.908	133.584	22.185	15,361	690,810,130
tie-quad	tarjan	1.846	0.041	111.492	29.192	137.525	21.780	15,958	131,964,029

TABLE 5
BENCHMARK STATS: TOROIDAL_QUEEN100-1.SMT2 (Timeout = 10m)

Preproc	Solver	Ing (s)	Prpr (s)	SAT Slvr (s)	Theory Slvr (s)	Graph OH (s)	Learn OH (s)	Slvr Calls	Theory Slvr Loops
nil	tarjan	2.071	0.000	319.411	38.408	171.140	71.015	21,300	159,824,587
t-lin	tarjan	2.012	0.017	275.423	21.393	238.257	64.894	20,638	87,553,756
t-quad	tarjan	1.957	0.026	282.574	28.558	226.191	62.630	18,948	115,381,428
tie-lin	tarjan	2.018	0.049	281.024	18.622	239.644	60.676	20,638	75,473,975
tie-quad	bf-basic	1.941	0.085	34.420	522.808	37.466	5.249	3,116	3,146,417,159
tie-quad	bf-full	2.024	0.088	35.870	521.243	37.186	5.649	3,107	3,137,423,877
tie-quad	spfa-basic	2.038	0.087	33.971	525.370	35.440	5.196	3,113	3,160,422,309
tie-quad	spfa-full	1.984	0.095	293.364	37.379	213.872	55.357	18,534	801,610,666
tie-quad	tarjan	1.939	0.089	302.925	19.525	220.636	56.878	20,186	87,553,462

TABLE 6
BENCHMARK STATS: DIAMONDS.18.10.I.A.U.SMT2 (Timeout = 10m)

Preproc	Solver	Ing (s)	Prpr (s)	SAT Slvr (s)	Theory Slvr (s)	Graph OH (s)	Learn OH (s)	Slvr Calls	Theory Slvr Loops
nil	tarjan	0.016	0.000	418.490	114.632	47.622	3.762	262,145	423,756,426
t-lin	tarjan	0.012	0.000	420.419	112.858	47.440	3.830	262,145	422,417,173
t-quad	tarjan	0.015	0.000	386.029	111.673	46.464	3.716	262,145	423,780,495
tie-lin	tarjan	0.013	0.000	420.630	112.552	46.724	3.778	262,145	422,617,029
tie-quad	bf-basic	0.013	0.000	4.145	590.304	5.077	0.452	23,763	3,579,442,156
tie-quad	bf-full	0.015	0.000	4.306	590.072	5.175	0.433	24,153	3,638,232,404
tie-quad	spfa-basic	0.012	0.000	4.230	589.961	5.367	0.409	28,034	3,166,118,091
tie-quad	spfa-full	0.014	0.000	415.269	63.803	43.939	3.415	262,145	959,796,850
tie-quad	tarjan	0.015	0.000	413.052	114.389	47.201	3.884	262,145	425,219,949

TABLE 7
BENCHMARK STATS: JOBSHOP60-2-30-30-4-4-12.SMT2 (Timeout = 10m)

Preproc	Solver	Ing (s)	Prpr (s)	SAT Slvr (s)	Theory Slvr (s)	Graph OH (s)	Learn OH (s)	Slvr Calls	Theory Slvr Loops
nil	tarjan	0.517	0.000	160.191	90.508	308.902	40.367	43,222	456,456,625
t-lin	tarjan	0.521	0.000	154.582	95.651	309.479	40.236	45,849	506,108,098
t-quad	tarjan	0.720	0.000	157.273	96.052	306.190	40.430	44,550	494,490,165
tie-lin	tarjan	0.823	0.097	163.218	92.510	306.154	38.065	44,218	482,100,485
tie-quad	bf-basic	0.613	0.008	4.358	583.757	10.479	1.057	1,283	3,380,463,490
tie-quad	bf-full	0.526	0.094	5.023	583.063	10.479	1.135	1,281	3,365,834,647
tie-quad	spfa-basic	0.434	0.086	6.056	582.085	10.765	1.039	1,316	3,332,857,606
tie-quad	spfa-full	0.519	0.007	149.881	126.829	286.802	36.456	46,207	2,578,741,641
tie-quad	tarjan	0.524	0.006	151.123	94.097	315.461	39.290	50,418	508,192,863