FernUniversität in Hagen
Faculty of Mathematics and Computer Science

AIG
Artificial Intelligence Group

Parallel Backtracking for Stable Extensions in Abstract Argumentation

Bachelor's Thesis

in partial fulfilment of the requirements for
the degree of *Bachelor of Science (B.Sc.)*
in Informatik

submitted by
Ricky Berghold

First examiner: Prof. Dr. Matthias Thimm
Artificial Intelligence Group

Advisor: Prof. Dr. Matthias Thimm
Artificial Intelligence Group

Statement

I declare that I have written this bachelor's thesis independently without unauthorized assistance from third parties. I have only used the resources listed in the bibliography, and all passages taken verbatim or paraphrased from these sources are clearly marked as such. This also applies to any figures, sketches, or graphical representations. The work has not been submitted elsewhere in the same or similar form and has not been published. By submitting the electronic version of this thesis, I acknowledge that it will be checked for plagiarism and stored exclusively for examination purposes.

I explicitly agree to have this thesis published on the webpage of the artificial intelligence group and endorse its public availability.

The software created for this work is available as open source; a link to the repository is included in the thesis. The same applies to any research data.

Baiersdorf, 26.03.2026 

(Place, Date) (Signature)

Zusammenfassung

Die meisten Solver in den jüngsten Argumentation-Wettbewerben reduzieren das Problem, eine stabile Extension zu finden, auf SAT oder ASP. In dieser Bachelorarbeit wird WallEParBt vorgestellt, ein Solver, der stabile Extensionen direkt auf dem Argumentationsgraphen mittels parallelem Backtracking lösen kann. Hierbei wird untersucht, wie sich Parallelisierung und Preprocessing auf verschiedene Graphstrukturen auswirken. WallEParBt wird dazu auf dem kompletten ICCMA-2025 Benchmark gegen einige der leistungsstärksten Solver aus dem Main Track des ICCMA-2025-Wettbewerbs evaluiert.

Abstract

Most solvers in recent argumentation competitions reduce the problem of finding a stable extension to SAT or ASP. In this thesis, WallEParBt is presented, a solver that can find stable extensions directly on the argumentation graph using parallel backtracking. We investigate how parallelisation and preprocessing affect different graph structures. WallEParBt is evaluated on the complete ICCMA 2025 benchmark against some of the strongest solvers from the ICCMA 2025 Main Track.

Contents

List of Figures

List of Tables

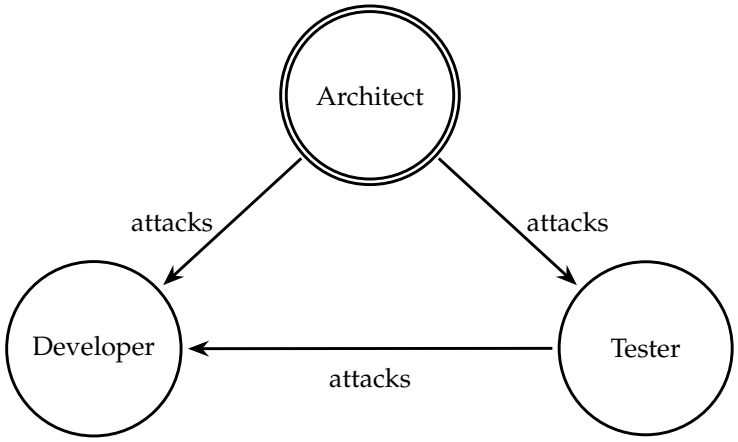


Figure 1: Argumentation framework for the sprint planning example. The only stable extension is {Architect}.

1 Introduction

Everyday project decisions work fine until somebody disagrees. As soon as people defend different and conflicting positions, one has to ask which of them can still be accepted together.

In classical software development, many meetings take place in which team members exchange their opinions and try to reach an agreement. A real-world example is the handling of bugs that customers report shortly before a software release. Assume there is a new performance bug. While working on the fix, the developer realises that it is necessary and reasonable to refactor a software module. Here, the architect has a different opinion and considers this change as too risky shortly before the release date. In contrast, the tester insists on fixing this bug for the upcoming release and therefore classifies the bug as release-relevant. The architect then rejects this opinion and considers the bug to be a low priority one. Finally, during the discussion, the tester also rejects the refactoring because the change could introduce new bugs and the high testing effort would make it impossible to reach the planned release date. Such discussions can become lively, and getting all parties to agree becomes a real challenge.[1]

Abstract argumentation goes back to Dung [Dun95, pp. 321–326], whose idea was to place each position into a node of a directed graph and to draw an edge wherever one position attacks another. When this idea is applied to the sprint discussion, we obtain the digraph shown in Figure 1, with one node for each representative position and one arrow for each attack. One important question is then which positions can be accepted together, i.e., without being attacked by another accepted position. In our example, only the architect is not attacked, while the developer and the tester are both attacked and have nobody who attacks back on their behalf. Since the

[1] For simplicity, Figure 1 collapses the discussion to one representative position per participant.

architect's position attacks the other two and neither of those two attacks back, the architect's position is the only one that survives.

In this formal context, the positions are referred to as *arguments*, and what counts as an acceptable collection of arguments depends on the semantics that is chosen. Each such collection is called an *extension*. Because the sprint example had only three arguments, it was relatively simple to determine what constituted an acceptable extension. However, when dealing with thousands of arguments and complex attack relations, finding acceptable extensions becomes much more difficult. The problem studied in this thesis is called SE-ST [Org25], which stands for **S**ome **E**xtension under **ST**able semantics. It asks whether at least one stable extension exists and, if one does exist, the solver must return it. Dvořák and Dunne [DD18, Table 1] demonstrated that the corresponding decision problem is NP-complete, which means that there likely does not exist a polynomial-time algorithm to solve it. Most solvers out in the field transform the argumentation graph into a SAT or ASP formulation and then pass it to a general-purpose solver [EGW08, TCV23]. In practice, these solvers are often very fast because SAT and ASP technology has been optimised for decades. However, the transformation loses the structure of the graph, which could be used for additional pruning and parallelisation.

By contrast, our solver preserves the argumentation graph as it is. We develop a backtracking-based solver called WallEParBt that assigns labels directly on the argumentation graph based on Caminada labelling [Cam06, pp. 112–113] and resolves all forced labelling choices on the attack structure itself. The solver has been designed for multi-core use through work-stealing [BL99], which distributes the backtracking search among threads. Idle threads will take over tasks from other threads that are still active.

The central question of this thesis is whether the backtracking search tree offers enough independent work to be split across the available CPU cores. We want to find out if parallelising this search produces measurable improvements on real benchmark instances, or if the overhead of the parallel execution exceeds the gain in speedup. Since various graph structures may react differently to parallelisation and graph processing, we test three hypotheses.

- **Parallelisation (H1)** asks whether work-stealing on the backtracking search tree will result in practical speedups and not simply overhead.
- **Preprocessing (H2)** determines if splitting the graph into its strongly connected components and then testing small root components for the existence of a stable extension, prior to the main search, would speed up the whole solution process or if this additional step costs more than what it saves.
- **Adaptive Configuration (H3)** examines whether it is possible to outperform a fixed configuration by enabling or disabling the preprocessing described in H2 based solely upon the graph structure of each individual problem instance.

We have evaluated on the ICCMA 2025 benchmark suite [Org25], containing 322 instances from a variety of graph families. To allow a direct comparison, we also ran

the three solvers that competed in the SE-ST main track on the same hardware. Section 2 introduces the formal definitions, Section 3 reviews SAT-based, ASP-based, and direct solvers, Section 4 describes WallEParBt itself, and Sections 5–7 present and discuss the experimental results.

2 Preliminaries

In the introduction, the sprint discussion between the architect, the developer, and the tester had only three arguments and three attacks, and it was easy to see that the architect belongs to the stable extension while neither the developer nor the tester did. The ICCMA 2025 benchmark [Org25] contains 322 instances with graphs ranging from 100 to several million arguments. Attack relations can grow so dense that no human could trace which sets are conflict-free, much less which ones are stable. For this reason, we built our solver on top of several existing results from the argumentation literature. The present section walks through each of them on an easy to follow four-node example before moving on to the solver's algorithms.

2.1 Abstract Argumentation Frameworks

Back in 1995, Dung [Dun95, p. 326] came up with a very simple idea. Take a set of arguments, write down which argument attacks which, and draw the result as a directed graph. There is nothing more to it, no probability, no strength, not even what an argument actually states, just nodes and directed edges.

Definition 1 (Abstract Argumentation Framework)**.** An *abstract argumentation framework* (AAF), as introduced by Dung [Dun95, Def. 2, p. 326], is a pair $F = (\mathcal{A}, \mathcal{R})$ with $\mathcal{A}$ being a finite set of *arguments* and $\mathcal{R} \subseteq \mathcal{A} \times \mathcal{A}$ recording which argument *attacks* which. Whenever $(b, a) \in \mathcal{R}$, we say that b attacks a.

As a running example throughout this chapter, consider the framework F_1 with nodes $\{0, 1, 2, 3\}$ and edges $\{(0, 1), (0, 2), (1, 3), (2, 3)\}$. This gives us a diamond shape where argument 0 is at the top with two outgoing edges pointing to 1 and 2, and both 1 and 2 point down to argument 3 at the bottom. Argument 0 has an out-degree of 2 because it attacks two other arguments. No other argument attacks it, so it has an in-degree of 0. Since it is the only argument with an in-degree of 0, it is the only unattacked argument within the framework. See Figure 2 for the resulting graph.

Example 1 (Computational Running Example)**.** Let

$$F_1 = \big(\{0, 1, 2, 3\},\ \{(0, 1), (0, 2), (1, 3), (2, 3)\}\big).$$

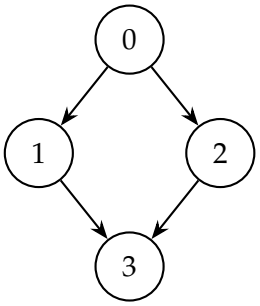


Figure 2: The diamond framework F_1. Argument 0 attacks 1 and 2, which both attack 3. The unique stable extension is $\{0, 3\}$.

2.2 Conflict-Freeness and Defence

Thus far, we have looked at single arguments, but the interesting question is which arguments can be accepted together. On F_1, the set $\{0, 3\}$ does not have any edges connecting its members, and therefore no member of the set attacks another member of the set. Dung [Dun95, p. 326] calls such a set *conflict-free*. However, the set $\{0, 1\}$ contains the edge $(0, 1)$, and therefore fails to meet this criterion, as argument 0 attacks argument 1.

Dung [Dun95, p. 326] also introduced a second property that asks whether a set can defend each of its members against all external attackers. On F_1, the arguments 1 and 2 both attack argument 3 from outside the set $\{0, 3\}$, but argument 0 belongs to the set and attacks back along both edges, namely $(0, 1)$ and $(0, 2)$. That is, every external attacker of $\{0, 3\}$ is counter-attacked by some member of the set. Dung calls a conflict-free set with this property an *admissible set*.

Definition 2 (Conflict-Free Set)**.** A set $S \subseteq \mathcal{A}$ is called *conflict-free* [Dun95, p. 326] when none of its members attacks another member of S. Equivalently,

$$S \times S \cap \mathcal{R} = \emptyset.$$

Definition 3 (Acceptability)**.** We call an argument $a \in \mathcal{A}$ *acceptable with respect to* a set $S \subseteq \mathcal{A}$ [Dun95, p. 326] when every attacker of a is counter-attacked by some member of S. If this holds, we say that S *defends* a. Formally,

$$\forall b \in \mathcal{A} : \ (b, a) \in \mathcal{R} \Rightarrow \exists c \in S : \ (c, b) \in \mathcal{R}.$$

Definition 4 (Admissible Set)**.** A set $S \subseteq \mathcal{A}$ is said to be *admissible* [Dun95, p. 326] if and only if it is conflict-free and defends each of its elements.

Returning to our first example of F_1, the singleton set $\{0\}$ is admissible since argument 0 has an in-degree of 0 on the diamond and there is no attacker available to require a counter-attack. The set $\{3\}$ is not admissible because arguments 1 and 2 attack argument 3 and $\{3\}$ contains no one that will counter-attack either of them. Likewise, the set $\{1, 2\}$ is not admissible because of the attack from argument 0 against both of them.

Theorem 1 (Fundamental Lemma)**.** *If S is admissible, then so is $S \cup \{a\}$ for any acceptable a with respect to S. Similarly, any a' acceptable with respect to S is also acceptable with respect to $S \cup \{a\}$ [Dun95, p. 327].*

Furthermore, any stable extension is also admissible [Dun95, p. 328]. This is yet another motivation for the forcing rules used in the solver. When an argument becomes acceptable with respect to the current partial labelling, adding it preserves admissibility. Of course, the actual forcing rules, which are described in Section 2.5, go beyond this and also include additional constraints on `OUT` labels and coverage, related to the stability of the solution.

2.3 Stable Semantics

To summarise, conflict-freeness means that no member of S is in a position to attack another member of S. Acceptability of an argument a requires that for every attacker of a, some member of S attacks that attacker back. An admissible set must therefore be conflict-free and defend all of its own members. While we know from the running example that $\{0\}$ is admissible, this alone is not enough to determine the stable extension. Under stable semantics, every argument not part of the extension must have at least one attacker within it. On F_1, this leads us directly to $\{0, 3\}$ as the sole stable extension, which we now formalise below.

Definition 5 (Stable Extension)**.** According to Dung [Dun95, p. 328], a set $S \subseteq \mathcal{A}$ is a *stable extension* of $F = (\mathcal{A}, \mathcal{R})$ when two conditions hold at the same time:

1. S is conflict-free (Definition 2).
2. For each argument a not in S, there is an argument $b \in S$ such that $(b, a) \in \mathcal{R}$.

Example 2 (Stable Extensions in Running Example)**.** Arguments 0 and 3 do not attack each other, therefore $\{0, 3\}$ meets the first requirement. Since $(0, 1)$ and $(0, 2)$ show that argument 0 attacks arguments 1 and 2, respectively, every argument outside $\{0, 3\}$ is attacked by an argument within the set. Thus, the second requirement is met as well. There is no other subset of $\{0, 1, 2, 3\}$ that meets both conditions simultaneously. For example, $\{1, 2\}$ does not attack argument 0. On the other hand, any set containing argument 0 together with either 1 or 2 would contain an internal attack. So in total, F_1 has exactly one stable extension. As we will see next, this is not always the case.

To see why stable extensions do not always exist, we informally assign labels `IN` (accepted) and `OUT` (rejected) to arguments and check whether the result is consistent (we will introduce the formal labelling framework in Section 2.4). A small example with an odd cycle is shown in Figure 3. On the **left-hand side**, the triangle consists of three arguments a, b, and c, where a attacks b, b attacks c, and c attacks a. One can see what happens when trying to put labels on it. Say we give the argument a the label `IN`. Because a attacks b, the label of b has to be `OUT`. After that, c has

no attacker left that carries IN or UNDEC, only b which is OUT. At this point, b is the only attacker of c, and b is already OUT, so c would also have to be IN, but c attacks a, and a is already IN, which places an attack between two accepted arguments, and the labelling attempt fails. Picking b or c first makes no difference because three arguments on three edges leave no room for an alternating pattern of IN and OUT, which is why an odd cycle on its own never has a stable extension.

The **right-hand side** of Figure 3 shows a cycle with four arguments where the labelling does not fail. Starting with a on IN, b goes to OUT because a attacks b. Then, c goes to the state IN because its only attacker b is already OUT. Ultimately, d also gets the state OUT as c attacks d. The set $\{a, c\}$ resulting from this has no internal conflicts and attacks both b and d, so it is a stable extension according to Definition 5. What distinguishes the two cycles is their parity. When there is an even number of arguments in the cycle, the IN/OUT labels go back and forth and meet each other at the end of the cycle. When there is an odd number of arguments in the cycle, one IN argument will eventually attack another IN argument.

Figure 3: Odd vs. even cycles (green = IN, red = OUT). The 3-cycle (left) has no stable extension. The 4-cycle (right) has the stable extension $\{a, c\}$.

The symbol a^+ represents the set of all arguments that a attacks,

$$a^+ = \{b \in \mathcal{A} \mid (a, b) \in \mathcal{R}\}.$$

When we want to represent the targets for an entire set S, we take the union

$$S^+ = \bigcup_{a \in S} a^+.$$

On F_1, we get $0^+ = \{1, 2\}$ as the targets of argument 0, from the edges $(0, 1)$ and $(0, 2)$. However, $3^+ = \emptyset$ because there are no other arguments that can be attacked by 3. If we consider the targets of the subset $\{0, 3\}$, and put those into $\{0, 3\}^+ =$

$\{1, 2\}$, the union $\{0, 3\} \cup \{1, 2\} = \{0, 1, 2, 3\}$ gives us the entire set of arguments in the framework, as required by Dung's characterization of a stable extension.

Theorem 2 (Range Characterization)**.** *The combination $S \cup S^+$ is called the* range *of S. Exactly those conflict-free sets $S \subseteq \mathcal{A}$ whose range covers all arguments, i.e.,* $\mathrm{range}(S) = \mathcal{A}$*, are stable extensions of F. In the equivalent complement form, this is $S = \mathcal{A} \setminus S^+$ together with conflict-freeness [Dun95, p. 328].*

2.4 The Stable-Extension Existence Problem

Definition 6 (Labelling)**.** Arguments in an argumentation framework receive a label based on the labelling of Caminada [Cam06, pp. 112–113] as follows: `IN` when it is accepted, `OUT` when it is rejected, and `UNDEC` when the solver has not reached a decision regarding the argument. Caminada's definition of the function used to assign one of the above labels to each argument is

$$\mathrm{Lab} : \mathcal{A} \to \{\texttt{IN}, \texttt{OUT}, \texttt{UNDEC}\}$$

Let $\mathrm{in}(\mathrm{Lab})$ represent the set of all arguments with the label `IN`,

$$\mathrm{in}(\mathrm{Lab}) = \{a \in \mathcal{A} \mid \mathrm{Lab}(a) = \texttt{IN}\}.$$

Definition 7 (Stable Labelling)**.** An assignment of labels is defined to be *stable* [Cam06] when the set of `IN` arguments represents a stable extension in the sense of Definition 5, i.e., no argument receives a label of `UNDEC` and every `OUT` argument has at least one `IN` attacker along some edge in $\mathcal{R}$. In other words, for each $a \in \mathcal{A}$

$$\begin{aligned} \mathrm{Lab}(a) = \texttt{IN} &\iff \forall b \in \mathcal{A} \colon (b, a) \in \mathcal{R} \Rightarrow \mathrm{Lab}(b) = \texttt{OUT}, \\ \mathrm{Lab}(a) = \texttt{OUT} &\iff \exists b \in \mathrm{in}(\mathrm{Lab}) \colon (b, a) \in \mathcal{R}. \end{aligned}$$

Either the solver finds a stable labelling for the given framework, or it determines that none exists. This is the computational problem that we refer to as SE-ST.

On F_1, the assignment of `IN` on arguments 0 and 3 and `OUT` on 1 and 2 creates a stable labelling, as can be verified by inspection of the graph in Figure 2.

During the search process, the solver may place the `OUT` label on argument 1 of F_1 before placing the `IN` label on argument 0. At that point, we say that argument 1 is *uncovered*, since there is no `IN` attacker that would support the `OUT` label along the edge $(0, 1)$. As a consequence, the propagation rules in Section 2.5 must either find an `IN` attacker that supports the label, or remove it.

Theorem 3 (Complexity)**.** *It is NP-complete to determine whether a given framework contains a stable extension [DD18, Table 1].*

Proof sketch. To show that the problem is in NP, one first provides the candidate set $S \subseteq \mathcal{A}$ as the certificate, and to then validate its correctness in two passes: (1) S is conflict-free, and (2) every argument that does not belong to S is attacked by some

element of S. Both of the above examinations of at most $O(|\mathcal{A}|^2)$ pairs are performed in polynomial time.

NP-hardness is due to the fact that there are well known reduction techniques from the satisfiability (SAT) problem, which have been explained in detail by Dimopoulos and Torres [DT96], and summarized by Dvořák and Dunne [DD18]. More simply, for every Boolean formula ϕ we can build an argumentation framework F_ϕ. The set of all stable extensions of F_ϕ will include exactly those assignments of ϕ that satisfy ϕ. This is achieved in two construction steps:

1. For each variable x_i of the formula ϕ, there is a pair of arguments in the framework, x_i and its negation $\bar{x}_i$, that mutually attack each other.

2. For each clause c_j of the formula ϕ, there is one argument in the framework that has a self-attack, and the argument receives an incoming edge from each literal that appears in c_j.

To understand why the construction works, one first has to look at the mutual attacks between each pair x_i and $\bar{x}_i$. After that, one has to examine the role of the self-attacking clause arguments.

Since S must be conflict-free, it cannot contain both x_i and $\bar{x}_i$ at the same time. On the other hand, if S contained neither x_i nor $\bar{x}_i$, then neither of these two arguments would be attacked by S, so the stability condition would fail. Therefore, a stable extension S must contain exactly one of the two arguments. In other words, picking one of the two corresponds to setting the truth value of that variable.

A similar reasoning applies to clauses. A clause argument attacks itself and therefore cannot belong to a conflict-free set S. Hence, it must be attacked by at least one argument in S. By construction, this happens if and only if at least one literal of the clause is chosen in the extension. This matches the requirement that at least one literal in the clause must evaluate to true.

Therefore, a satisfying assignment for ϕ gives rise to a stable extension of F_ϕ, and vice versa. □

2.5 Graph Structure and Propagation

F_1 has a diamond structure and does not require any branching. Since argument 0 has no attackers, it is labelled as `IN`. Once argument 0 has that label, the labels of the remaining arguments are forced, which gives the stable extension $\{0, 3\}$. Real ICCMA instances are larger than F_1, but they still have graph structure that the solver can use.

Definition 8 (Strongly Connected Component)**.** Given a directed graph, a *strongly connected component* (SCC) is a maximal subset of nodes C in which every pair $a, b \in C$ can reach each other through some directed path.

In other words, two nodes belong to the same SCC if each one can reach the other via a sequence of edges. In F_1, argument 0 can reach argument 3 through the sequence

$0 \to 1 \to 3$, but there is no sequence of edges that allows argument 3 to reach argument 0, and hence they are part of different SCCs. There is a linear-time algorithm due to Tarjan that finds all SCCs of an argumentation framework in $O(n+m)$ [Tar72], and we use the output of this algorithm throughout the search. When we collapse each SCC to a single node, the resulting graph has no cycles. If this were the case, then all of those nodes would also be able to reach each other and thus belong to the same SCC. The resulting cycle-free graph is called the SCC-DAG or condensation. An important point of discussion within Section 4 is that in addition to the SCC-DAG, it would also be advantageous to be able to solve each SCC individually. Choices made in one SCC restrict the search space of all SCCs following it in the SCC-DAG. Baroni and Giacomin [BG07] discuss this SCC-recursive view, where choices made in earlier components affect the components that follow. Accordingly, if a component has no local solution, the entire branch will fail. Thus, a bad decision made on one component may cause our solver to backtrack and attempt another assignment.

In F_1, none of the four edges create a cycle, so each of the arguments forms its own strongly connected component, and the solver need only traverse them in the order $\{0\}$ first, then $\{1\}$ and $\{2\}$, and then $\{3\}$ lastly. Before any branching occurs, five propagation rules (see below) provide all the labels that follow from the current state within each component.

For any given argumentation framework, applying the labelling rules one after another, with no branching at all, leads to a single fixpoint. Those arguments that carry the `IN` label at that fixpoint form the grounded extension. For F_1, the grounded extension is $\{0, 3\}$. In the case of F_1, we have already discussed how, since argument 0 has no attackers, it will have an `IN` label. From there, the remaining labels are assigned one-by-one until all of the arguments in the diamond have been labelled.

Dung [Dun95, pp. 328–330] showed that the grounded extension is always well-defined, regardless of the order in which arguments are processed. For F_1, the fixpoint is the set $\{0, 3\}$ and the remaining arguments in the diamond will have an `OUT` label. All stable extensions of a framework include the labels that were determined from the grounded labelling pass, and in the case of F_1, 0 and 3 are in every stable extension and are therefore not considered again.

Caminada's labelling framework [Cam06], along with the constraint on stability, provides five ways to determine one additional label. Three of these five occur during the grounded labelling pass, as shown in Figure 4 when tracing through each of the diamond's arguments one by one (rules 1, 3, and 2). As the labelling has been completed after the application of only these three rules, the stable extension $\{0, 3\}$ is found from propagation alone, and no branching is required.

1. Let a be an argument that has no attackers. Then, a must be `IN` in every stable extension [Dun95]. Let b be attacked by a. Then, b must be `OUT`.

2. Let v be an argument with all of its attackers being `OUT`. Then v must be

IN [Dun95].

3. Let v be IN. Then all of v's attackers and v's targets must be OUT [Dun95].
4. Let v be OUT with no IN attacker and exactly one UNDEC attacker u. Then, u must be IN [NAD14].
5. Let v be OUT with no IN attacker and no UNDEC attackers. Conflict! [NAD14].

On larger ICCMA benchmarks, propagation will most of the time not be able to determine all of the labels by itself. There will typically be some number of arguments that are left in the UNDEC state, and the solver will need to select one of them as a branching point. Each argument that is settled by propagation reduces the potential branching points, which can help to reduce the size of the expensive backtracking search.

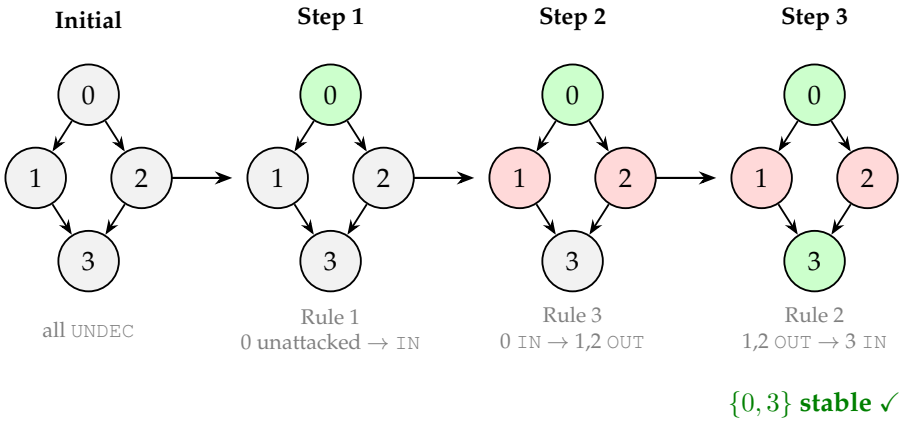


Figure 4: Step-by-step propagation on F_1. Rules 1, 3, and 2 label all four arguments without branching.

All five propagation rules follow from the two properties that characterise a stable extension, namely conflict-freeness and coverage. Consider an argument with no attackers. If it were labelled OUT, that label would need to be justified by some IN attacker to satisfy coverage. Since no such attacker exists, the argument must be labelled IN. All other basic propagation steps follow from the same idea. Once an argument is labelled IN, conflict-freeness forces every attacker and every target of that argument to be labelled OUT. Coverage then implies that any argument whose attackers are all labelled OUT must itself be labelled IN. Figure 4 shows this process step by step on the diamond framework, where these three forced moves already determine every label.

In a larger framework, an OUT-labelled argument may have exactly one attacker u still left in the UNDEC state, while all of its other attackers are already OUT. In that case, the current OUT label still needs an IN attacker to justify it. Since u is the only

remaining argument that could still provide that justification, the solver has to label u as IN [NAD14]. If even u has already left UNDEC without receiving an IN label, then the required justification can no longer be provided at all. At this point, the partial labelling has reached a dead end, and the solver has to backtrack.

To see how Rules 4 and 5 work, assume that argument 1 is OUT while argument 0 is still UNDEC. Argument 1 has exactly one UNDEC attacker, which is argument 0, and no IN attacker at all. Rule 4 therefore forces argument 0 to IN.

Two situations lead to a dead branch. First, if argument 0 had also been OUT already, then argument 1 would have no attacker left on IN or UNDEC, and rule 5 marks that branch as dead. Second, if two arguments that are connected by an edge both carry IN, conflict-freeness is violated. On F_1, labelling both 0 and 1 as IN would place two accepted arguments on the attack $(0, 1)$, so the solver has to undo that branching decision.

Our solver propagates labels using a work-list. Whenever there is a label move from UNDEC to IN or OUT, all neighbours of that argument get added to the work-list. On F_1, we see that the solver performs four label changes and four edge visits during its first pass. As each node and edge are visited a constant number of times within an n-argument, m-attack framework, one pass runs in $O(n + m)$ time and uses $O(n)$ space. We already proved in Theorem 3 that SE-ST is NP-complete, so the search tree can be exponentially large. However, for each search node in the tree, the solver completes a propagation pass in $O(n + m)$ time.

3 Related Work

There exist several ways to solve SE-ST, and these methods generally fall into two distinct groups. In the first category, a solver converts the argumentation framework into a propositional satisfiability instance or an answer set program and then hands it off to a SAT or ASP solver. This can be very effective because SAT and ASP technology has been optimised over decades. The downside is that the solver no longer works directly with the attack graph. In the other case, the original graph is the main data structure used by the solver. It uses backtracking and label propagation directly on the arguments and attacks. As a result, the solver has to implement the main mechanisms itself, such as selecting the next branch and detecting conflicts, instead of relying on a general-purpose search engine.

3.1 Propositional Satisfiability-Based Solvers

A SAT-based argumentation solver converts the argumentation framework into a Boolean formula. It does this by assigning a propositional variable to each argument and then introducing clauses that enforce both conflict-freeness and coverage (Section 2). Once constructed, the formula is expressed in Conjunctive Normal Form (CNF) so that a Conflict-Driven Clause Learning (CDCL) solver can process it.

In Section 5, we compare WallEParBt with three SAT-based argumentation solvers: FUDGE [TCV23], μ-toksia [NJ20], and Scalop [LLM25].

3.2 Answer-Set Programming-Based Solvers

A second common approach is to use Answer Set Programming (ASP) to translate the argumentation framework into a logic program. ASPARTIX [EGW08] is a widely-used ASP-based solver for argumentation frameworks that does not include its own search engine. Instead, ASPARTIX defines a collection of ASP encodings for various argumentation semantics. The argumentation framework is represented as a set of logic-programming facts, and these facts are evaluated together with the encodings using an ASP solver such as Clingo [GKKS14]. In ASP, the primary cost factor is the grounding step, in which the grounder substitutes all logical variables with concrete values. For example, the rule "`out(X) :- att(Y,X), in(Y)`" will be replaced by one ground instance for each attack edge, so that for a graph with 1000 edges, the grounder will produce 1000 rules of the form "`out(6) :- att(2,6), in(2)`". After grounding, the ASP solver searches the resulting program for answer sets using techniques similar to CDCL. While the encoding itself is relatively compact, the grounding operation loses the graph locality of the original problem, so that the solver will see only a flat set of rules, and will no longer be able to ask questions such as "what nodes attack node X?".

3.3 Direct Backtracking Solvers

Several solvers work directly on the argumentation graph without converting it to some other format. Heureka [GT18] is a direct backtracking solver that dynamically reorders arguments using various heuristics to minimise the number of backtracking steps. Later, Thimm reimplemented Heureka as a C program called Dredd [Thi19], which implements DPLL-style backtracking with domain-independent heuristics and information propagation on the argumentation graph. Dredd participated in ICCMA 2019 and supports grounded, complete, stable, and preferred semantics. In addition, Nofal et al. [NAD14] discussed labelling-based methods for solving decision problems under preferred semantics using propagation. In later work [NAD16], they extended this approach with look-ahead pruning for both extension enumeration and acceptance tasks across several semantics including stable semantics. SMART V1.0 [HKT25] is a more recent solver that combines backtracking with a Graph Convolutional Network (GCN)-guided branching heuristic. SMART supports credulous acceptance under preferred, complete, and grounded semantics, but does not address stable semantics or multi-core parallelisation.

3.4 Parallelism in Argumentation Solvers

None of the argumentation solvers surveyed in this thesis parallelise the backtracking search within a single SCC. One reason may be that the ICCMA 2023 competition [Org23] required submissions in the Main Track to run in a single-core environment, while the No-Limits Track did allow parallel processing. Cerutti et al. [CTV+15] achieved speedups for preferred semantics by solving independent SCCs in parallel, but their approach does not parallelise the backtracking search within a single SCC. By contrast, WallEParBt uses work-stealing to distribute the search tree across threads within each SCC. A large amount of research has investigated parallel SAT solving outside of the context of argumentation. ManySAT [HJS09] uses a portfolio of different DPLL configurations that are run in parallel with clause sharing. Cube-and-conquer is another parallelisation strategy developed by Heule et al. [HKWB12] to partition a formula into thousands of smaller cubes using lookahead strategies, then apply a CDCL solver to each cube independently. Both techniques operate at the SAT level and are therefore only available to solvers that translate the argumentation problem into a Boolean formula.

3.5 Summary

All of the following solvers pass the argumentation framework to a SAT or ASP backend (i.e. the backend is unaware of the original attack graph): FUDGE, μ-toksia, Scalop, and ASPARTIX. On the other hand, Heureka, Dredd, the algorithms of Nofal et al., and SMART all perform searches on the argumentation graph on a single processor, as documented in the references provided in this chapter. Dredd is the most similar predecessor to WallEParBt, since both use DPLL-style backtracking with labelling and propagation on the argumentation graph. However, Dredd only performs its backtracking search sequentially on a single processor and does not include either SCC-based preprocessing or an adaptive configuration. Additionally, there were no direct SE-ST solvers found in the literature reviewed for this thesis that used work-stealing to divide the backtracking search tree across multiple processors with separate thread-local labelling states. WallEParBt extends this direct approach with work-stealing parallelism, a coverage-based branching cascade, and an adaptive preprocessing phase, as discussed in Section 4. Section 5 compares the results against the ICCMA solvers.

4 Methods

This section will present the general architecture of WallEParBt and will describe how it uses the elements previously discussed (stable extensions, SCC decomposition and propagation) to construct a parallel SE-ST solver that operates directly on the argumentation graph.

4.1 Design Overview

Argumentation graphs from the ICCMA benchmarks use a text-based `.af` format. This format provides a header line specifying the number of arguments and each subsequent line specifies an attack between two arguments; the arguments in each line are identified by a unique integer ID. After reading the .af format, the solver stores the graph internally as compressed sparse row arrays (CSR). This allows neighbour ranges of any argument to be obtained in constant time ($O(1)$) and the neighbours to be iterated over in linear time ($O(\deg)$). Due to the heterogeneity of the ICCMA benchmark graphs, they differ widely in SCC structure and density. They can contain either many small strongly connected components (SCCs) or one large SCC, and can be either sparse or nearly fully dense. No single strategy exists to provide efficient performance across all graph structures. Therefore, the solver must react dynamically to the structure of the graph as it is processed at runtime. A three-stage process was implemented to support the varying complexity of the different graph types in solving them as quickly as possible. Each stage reduces the amount of remaining work for the next, to allow the solver to attempt to solve as many of the input instances as possible. Figure 5 illustrates the three stages and how each stage impacts the size of the search tree.

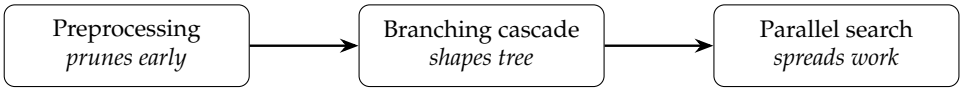


Figure 5: The three stages and their direct effects on the search tree. The relative impact of each stage is quantified in Section 5.4.

Every label that an earlier stage can fix before the search begins may prevent the exploration of a large subtree.

Stage 1 (Preprocessing). The solver begins with Tarjan's algorithm [Tar72] on the input graph (in $O(n + m)$ time, see Section 2.5) to decompose the graph into strongly connected components. The resulting SCC-DAG is then used in two ways. Before the expensive backtracking search begins, eligible source SCCs are checked for solvability. According to Lemma 1, the entire problem instance can be rejected if any such source SCC has no stable extension. During the main search, the SCC order is used as a heuristic preference when the coverage-based stage does not already determine the next branch. Testing small source components separately is significantly less expensive than testing the whole problem. When almost all arguments are located in one large SCC, this preprocessing step cannot reduce the problem size, so this step is skipped and the search proceeds directly (see Section 5.4 for breakdown by family).

Stage 2 (Branching). The method in which open arguments are assigned can have a substantial effect upon the search effort. Therefore, Stage 2 chooses the argu-

ment from which the most propagation is anticipated. If the chosen argument can determine many labels at once, then this can also reduce the size of the tree. As no single strategy will be effective on all possible graph structures, the specific assignment of this choice is based upon a three-rule cascade as presented in Section 4.2.

Stage 3 (Parallelism). Due to the size of the search tree in certain problem instances that can reach millions of nodes, a single thread will most likely reach the timeout. As such, the solver uses work-stealing for distributing the remaining search between multiple threads. Each thread retains its own deque (double-ended queue) of unexplored options. Upon encountering an UNDEC argument, a busy thread can select one label and potentially add the second choice to its deque to enable another thread to choose the second option. Spawn guards (Section 4.4), therefore, determine whether to create a new task or have the thread explore both options itself. If the search tree is wide enough to keep all threads busy, the search time drops by a significant factor. However, due to the nature of graphs having many small strongly connected components, the work within each component will not always be enough to keep all threads busy, which limits the potential speedup (Section 5.4).

Running example We use an easy example to illustrate each mechanism. Figure 6 shows an argumentation framework with six arguments (a–f) and the attacks between them.

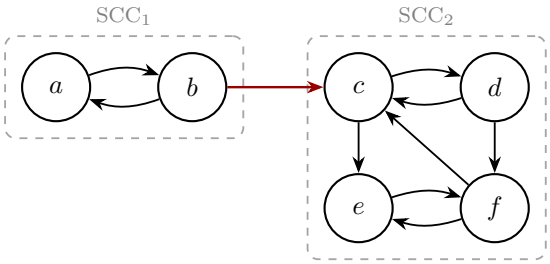


Figure 6: Running example. $\mathrm{SCC}_1 = \{a, b\}$ is a mutual attack. $\mathrm{SCC}_2 = \{c, d, e, f\}$ contains two mutual attacks ($c \leftrightarrow d$, $e \leftrightarrow f$) connected by $c \to e$, $d \to f$, and $f \to c$. The cross-component attack $b \to c$ (red) links the two SCCs.

Tarjan's algorithm partitions this graph into two strongly connected components, where $\mathrm{SCC}_1 = \{a, b\}$ is a source in the condensation DAG, and $\mathrm{SCC}_2 = \{c, d, e, f\}$ is its sole successor. There are two stable extensions of the framework, namely, $\{a, d, e\}$ and $\{b, d, e\}$, however, the solver will terminate after finding the first stable extension since SE-ST only requires a single witness.

4.2 Algorithm

This section describes the data structures, the backtracking search with its propagation rules, and the heuristics that decide which argument to branch on. Because the algorithms work on data structures, we start with this topic first.

Data structures Internally, a single byte is used to record the status of each argument. An argument can be in one of three states: `IN` means that we have chosen to include it in the extension, `OUT` means that it has been excluded, and `UNDEC` means that no decision has been made yet. In addition, the solver keeps track of three additional counters for each argument. These counters include the number of attackers of the argument that have not yet been labelled `OUT`, the number of attackers of the argument that have not yet been assigned a label, and whether there exists an attacker of the argument that is currently `IN`. When any of these counters falls to zero, we know that a propagation rule applies. Because the counter value alone tells us which rule to apply, there is no need to examine the status of all neighbours.

No duplicate copy of the entire label array is created with every decision. Instead, it records the label changes on a trail (a log of label changes), and can restore the original labels when it backs up to a previous decision point by replaying the trail in reverse. Recording label changes on a trail, followed by restoration of the original labels upon backing up, is effective for sequential search. When the solver wishes to send a subtree to another thread of execution, however, the solver requires a full copy of the labels. A full copy of the labels is necessary because the two threads can backtrack independently (Section 4.4), and therefore cannot share the same copy of the labels. Each thread also uses a deque of unexplored subtrees to enable work-stealing, as described in Section 4.4.

Backtracking and propagation The loop itself follows the DPLL algorithm (Algorithm 1) as it goes through each argument for a given problem, chooses an undecided argument from the list, and adds a label to it. After adding the label to the chosen undecided argument, the loop performs the propagation required for the newly added label. Based on the results of these propagations, the loop decides how to proceed. There are three possible outcomes. The first is to backtrack to the previous choice point when a contradiction is identified along the current path of the search. The second is to stop searching once all undecided arguments have been labelled. This indicates that a potential candidate solution has been reached; however, before declaring this candidate solution as complete, the solver must test whether or not this candidate solution meets the criteria of the stability condition defined in Section 2. The third is to use the cascading strategy defined below to select the next argument for branching and the order in which to attempt the two labels for the selected argument. If both labels at a search node fail, the loop returns to the previous node in the search tree.

Algorithm 1 defines how an argument is selected, how a label is assigned, and

Algorithm 1 Backtracking Search for Stable Extensions (DPLL-style [DLL62])

Require: Argumentation framework $F = (\mathcal{A}, \mathcal{R})$
Ensure: YES if a stable extension exists, NO otherwise

$L \leftarrow$ labelling with all arguments UNDEC
return SEARCH(L)

function SEARCH(L)
    $ok \leftarrow$ PROPAGATE(L)
    if $ok =$ CONFLICT **then**
        return NO
    if L has no UNDEC arguments **then**
        if ISSTABLE(L) **then**
            return YES ▷ Found stable extension, $\{a \mid L(a) = \texttt{IN}\}$
        else
            return NO
    $v \leftarrow$ SELECTVARIABLE(L) ▷ Cascade: coverage → SCC → degree
    for $\ell \in \{\texttt{IN}, \texttt{OUT}\}$ **do** ▷ Order chosen by CHOOSELABEL
        $L' \leftarrow$ COPY(L) ▷ Conceptual; DFS uses trail/undo
        $L'(v) \leftarrow \ell$
        if SEARCH(L') = YES **then**
            return YES
    return NO ▷ Both branches failed

how we propagate the consequences. Based on the outcome, the solver determines whether to continue down the current path or to backtrack. The solver's ability to succeed depends on how quickly the propagation algorithm can detect an inconsistency and thereby prevent the solver from continuing to explore the dead-end branch of the search tree.

After an argument has had a label assigned to it, the propagation stage applies the inference rules from Section 2 to propagate all consequences of this assignment through a queue-based fixed-point loop (Algorithm 2). This loop keeps running until one of two things happens: either a contradiction is found and the search goes back to the last choice point, or no further label changes can be made and control returns to the main loop.

Algorithm 2 Label Propagation (BCP-style fixed point [DLL62])

Require: Labelling L with at least one freshly assigned argument, framework $F = (\mathcal{A}, \mathcal{R})$
Ensure: OK if L stays consistent, CONFLICT if a contradiction is detected

function PROPAGATE(L)
  $Q \leftarrow$ queue of arguments whose label was just set
  for each a with $(a, a) \in \mathcal{R}$ **do** ▷ Self-attack (conflict-freeness)
    $L(a) \leftarrow$ `OUT`; enqueue a in Q
  for each a with $L(a) =$ `UNDEC` and *attackers*$(a) = \emptyset$ **do** ▷ Unattacked (Rule 1)
    $L(a) \leftarrow$ `IN`; enqueue a in Q
  while $Q \neq \emptyset$ **do**
    $v \leftarrow$ dequeue from Q
    if $L(v) =$ `IN` **then** ▷ Conflict-freeness (Rule 3)
      for each $b \in$ *attackers*$(v) \cup$ *targets*(v) **do**
        if $L(b) =$ `IN` **then return** CONFLICT
        else if $L(b) =$ `UNDEC` **then**
          $L(b) \leftarrow$ `OUT`; enqueue b in Q
    else if $L(v) =$ `OUT` **then**
      for each $b \in \{v\} \cup$ *targets*(v) with $L(b) =$ `OUT` **do** ▷ v itself may now be uncovered (R. 4–5)
        if $|att_{\texttt{IN}}(b)| = 0$ and $|att_{\texttt{UNDEC}}(b)| = 0$ **then**
          return CONFLICT ▷ Coverage conflict
        else if $|att_{\texttt{IN}}(b)| = 0$ and $|att_{\texttt{UNDEC}}(b)| = 1$ with attacker u **then**
          $L(u) \leftarrow$ `IN`; enqueue u in Q ▷ Unit propagation
      for each $b \in$ *targets*(v) with $L(b) =$ `UNDEC` **do** ▷ All attackers OUT
        if every attacker of b is `OUT` **then**
          $L(b) \leftarrow$ `IN`; enqueue b in Q
  return OK

Algorithm 2 applies the inference rules from Section 2 to compute the consequences of each labelling choice. Five examples below demonstrate these conditions and map each line of the pseudocode onto the corresponding inference rule.

All five inference rules applied during propagation are:

1. **Unattacked argument.** No attackers $\Rightarrow v$ is `IN` (lines 5–6).
2. **All attackers rejected.** Every attacker `OUT` $\Rightarrow v$ is `IN` (lines 20–22).
3. **Conflict-freeness.** v is `IN` $\Rightarrow$ all neighbours `OUT` (lines 9–13).
4. **Unit propagation.** v is `OUT`, uncovered, one `UNDEC` attacker u left $\Rightarrow u$ is `IN` (lines 18–19).
5. **Coverage conflict.** v is `OUT`, no `IN` attacker, no `UNDEC` attackers $\Rightarrow$ dead end (lines 16–17).

Running example (continued). Figure 7 continues the running example with a branch on a. Because of the cross-SCC edge $b \rightarrow c$, the full cascade would have ranked b ahead of a. Nevertheless, the example branches on a, because assigning $b =$ `IN` would determine the whole framework by propagation and would leave no backtracking step to illustrate. Once a is assigned the label `IN`, conflict-freeness forces b to `OUT`. Once b is `OUT`, the first SCC is settled, because b is covered by the `IN` attacker a.

Next, the solver proceeds to SCC_2. According to the heuristic cascade presented in Section 4.2, the solver should select c and attempt to assign $c =$ `IN`. Assigning $c =$ `IN` causes d, e, and f to be assigned the label `OUT`. This is due to conflict-freeness, since d and e are targets of c, and f is an attacker of c. Now, f is `OUT`, but f has no `IN` attacker. Both attackers of f (i.e., d and e) are also `OUT`. Hence, f is uncovered. At this point, a coverage conflict (Rule 5) has occurred and the branch is dead, as shown in Figure 8(b).

Since $c =$ `IN` was a dead branch, the solver backtracks and attempts $c =$ `OUT`. When the solver attempts $c =$ `OUT`, the solver propagates all remaining labels until the stable extension $\{a, d, e\}$ is reached, as shown in Figure 8(c).

Decision heuristics Unlike μ-toksia, which is a CDCL-based SAT solver and therefore only knows clauses and variables, WallEParBt operates at the graph level of the argumentation framework and chooses the next branch based upon graph structure. There is no perfect metric that will always be optimal for every type of graph. Therefore, a cascaded three-stage approach is used (Algorithm 3). First, it chooses the argument that covers the most open attack targets. If the first step cannot make a choice, the second stage narrows down the choices to the first open SCC in topological order. Finally, if a choice still remains to be made, the third stage uses a degree-based scoring system in addition to a bonus for each critical uncovered target.

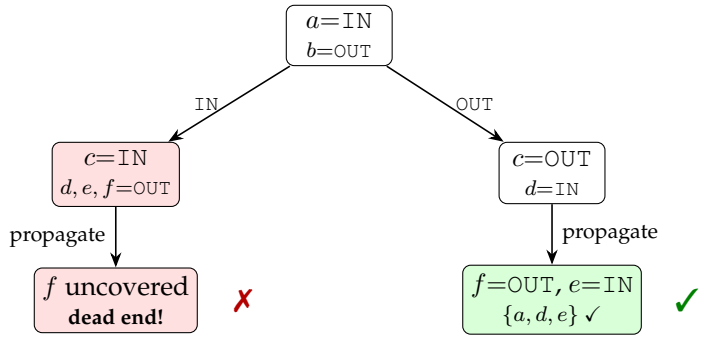


Figure 7: Search tree for the running example. On the left, the solver tries $c = \texttt{IN}$; propagation forces d, e, and f to OUT (conflict-freeness), but f has no IN attacker (coverage conflict), so the branch fails. The solver backtracks and tries $c = \texttt{OUT}$ on the right, where propagation produces the stable extension $\{a, d, e\}$.

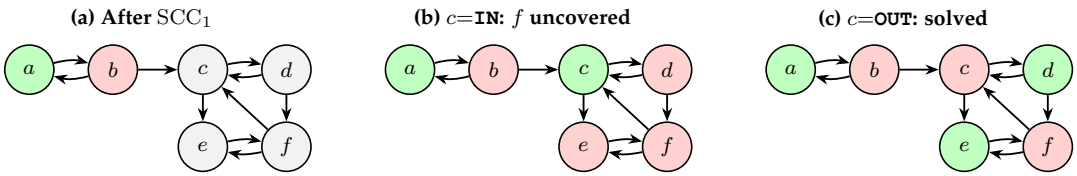


Figure 8: How the labels change during the running example. Green = IN, red = OUT, gray = UNDEC. (a) After SCC_1 is resolved, only a and b have labels. (b) Trying $c = \texttt{IN}$ forces d, e, and f to OUT (conflict-freeness); f has no IN attacker—coverage conflict. (c) Trying $c = \texttt{OUT}$ leads to the stable extension $\{a, d, e\}$.

Algorithm 3 Variable and Label Selection (Cascading Heuristic)

Require: Labelling L with at least one UNDEC argument, framework $F = (\mathcal{A}, \mathcal{R})$
Ensure: Argument v and initial label $\ell \in \{\texttt{IN}, \texttt{OUT}\}$

function SELECTVARIABLE(L)
  $U \leftarrow \{a \in \mathcal{A} \mid L(a) = \texttt{UNDEC}\}$
  ▷ **Stage 1: Coverage-first**
  for each $a \in U$ **do**
    $\mathit{cov}(a) \leftarrow |\{b \in \mathit{targets}(a) \mid L(b) \neq \texttt{IN} \wedge \text{no IN attacker of } b\}|$
  if $\max_{a \in U} \mathit{cov}(a) > 0$ **then**
    return $\arg\max_{a \in U} \mathit{cov}(a)$ ▷ Ties broken by degree, then index
  ▷ **Stage 2: SCC-aware selection**
  $C^* \leftarrow$ first SCC in topological order of condensation DAG with $C^* \cap U \neq \emptyset$ and all predecessor SCCs decided
  $U \leftarrow U \cap C^*$
  ▷ **Stage 3: Degree-based fallback**
  for each $a \in U$ **do**
    $\mathit{score}(a) \leftarrow \mathit{out\text{-}degree}(a)$
    for each $b \in \mathit{targets}(a)$ with $L(b) = \texttt{OUT}$ and no IN attacker of b **do**
      $k \leftarrow |\{u \in \mathit{attackers}(b) \mid L(u) = \texttt{UNDEC}\}|$ ▷ $k > 0$ by propagation
      $\mathit{score}(a) \leftarrow \mathit{score}(a) + 1/k$ ▷ Discretised in practice
  return $\arg\max_{a \in U} \mathit{score}(a)$ ▷ Ties broken by index

function CHOOSELABEL(v, L)
  $g_{\texttt{IN}} \leftarrow |\{b \in \mathit{targets}(v) \mid L(b) = \texttt{UNDEC}\}|$ ▷ Targets forced OUT if v is IN
  $g_{\texttt{OUT}} \leftarrow |\{b \in \mathit{attackers}(v) \mid L(b) = \texttt{UNDEC}\}|$ ▷ Attackers affected if v is OUT
  if $g_{\texttt{IN}} \geq g_{\texttt{OUT}}$ **then**
    return IN first ▷ Larger propagation effect
  else
    return OUT first

Stage 1: Coverage-first. Based upon the stability condition, each rejected (OUT) argument must have at least one accepted (IN) attacker [Dun95, p. 328]. In other words, coverage counts how many uncovered OUT arguments a candidate would justify if it were set to IN. For each candidate, we count how many uncovered rejected arguments still need an IN attacker (line 4). If at least one candidate has a count greater than zero, the candidate with the highest count is chosen (lines 5–6). For example, assume that through previous branching steps, both x and y are not IN and uncovered. Each one therefore needs an IN attacker. If argument p attacks both x and y while argument q only attacks x, then $cov(p)=2$ and $cov(q)=1$, so the coverage-first strategy selects p before q.

Stage 2: SCC-aware selection. After the previous step has finished, we determine whether the coverage scores of all the candidates are 0. All candidates can have a coverage score of 0 because all of the attack targets they cover already have an accepted attacker assigned to them, or the runtime gate has turned off coverage-based scoring on the graph. If all candidates have a coverage score of 0, then the next step is the SCC-recursive view described by Baroni et al. [BG07, pp. 685–687]. This time the candidate pool will be reduced to the first SCC in the topologically ordered condensation DAG that contains arguments which have not yet been decided but whose predecessors have all already been decided (lines 7–8). The branching process also focuses on the largest connected component of the undecided subgraph. Therefore, instead of deciding a single node in a completely disconnected area, the search will likely decide the whole cluster. In the running example shown in Figure 7, it is preferable to branch in SCC_1 before continuing with SCC_2.

Stage 3: Degree-based fallback. Now that all of the possible candidates have been narrowed down to those inside the selected SCC, the solver calculates a score for each of the candidates by applying the following scoring function. The scoring function used here is a combination of the fail-first approach of constraint satisfaction [HE80] and the impact-based variable selection method of Refalo [Ref04]. The most important factor used to determine the score assigned to a candidate is its out-degree (line 10). Candidates with higher out-degrees generally propagate further with fewer moves than candidates with lower out-degrees. Therefore, candidates with higher out-degrees will generally have higher scores. For each candidate, the base score is increased for each of its uncovered rejected neighbours (lines 11–13). The amount of the increase is inversely proportional to the number of undecided attackers for that neighbour. Rather than continuous weights, three discrete integer tiers are used: $+15$ for neighbours with $k{=}1$ undecided attacker, $+5$ for $k{=}2$, and $+1$ for $k \geq 3$. The highest weight is for $k{=}1$ since that neighbour is the closest to being resolved through unit propagation (since the last undecided attacker of that neighbour must be set IN to cover it). These values were empirically determined utilising the ICCMA 2023 benchmark suite. Table 1 shows the cascade scores for the running example. Algorithm 3 presents the whole cascade in a simplified form. In practice, coverage scoring is turned off on dense graphs: it is only performed if the average out-degree is at least $\max(5,\ 2\sqrt{|\text{candidates}|})$ and at least 30% of the

candidates are not covered. On extremely dense graphs, the above conditions are never met and therefore the cascade proceeds to Stage 2. Additionally, a 20-point bonus is awarded to candidates that have an undecided mutual attacker (i.e., a attacks b and b attacks a), provided there are at least 100 such bidirectional pairs in the graph. Otherwise, the cascade may try to set one of the pair `IN`, only to find later that the mutual attacker must also be `IN`, since both are candidates. This would violate conflict-freeness and result in backtracking. We always set the lower-indexed partner `IN` first, thus fixing the order of symmetric pairs. Section 6.1.3 examines the influence of this bonus.

The function CHOOSELABEL (lines 15–21) selects the direction of the first attempted label. Lines 16–17 determine how many undecided neighbours will be immediately affected by each direction. Whichever direction affects the most undecided neighbours is chosen first [Ref04]. During the parallel search (Section 4.4), a lookahead may select a different direction when both directions will affect an equal number of undecided neighbours.

Running example (continued). When SCC_2 of the running example starts, each of the four arguments, c, d, e, and f, have undecided status. Table 1 displays the cascade scores that would be generated by the pseudocode (some of these stages are omitted in the actual implementation for smaller SCCs, but the ordering is always preserved). Once the candidates are examined during the coverage stage, if argument c were accepted, then d and e would be labelled `OUT`, and therefore $cov(c) = 2$, while both d and f would receive a coverage score of 2, and only argument e would receive a coverage score of 1. Arguments c, d, and f have the same coverage scores, and they each have an out-degree of 2, so the cascade uses its final vertex-order tiebreaker to resolve this, and selects argument c because c has the lowest index among the remaining candidates. The branching result of this decision is displayed in Figure 7, with the focus on the decision-making process for selecting the next branching variable at this point.

Table 1: Heuristic scores for SCC_2. Stage 1 (coverage) produces a three-way tie at score 2. The degree tiebreaker also ties at 2, so c wins by vertex order.

Argument	Coverage Score	Degree	Selected
c	2 (d, e)	2	✓
d	2 (c, f)	2	
e	1 (f)	1	
f	2 (c, e)	2	

Figure 9 illustrates two very small frameworks. On the left side of the figure, repeated branching and propagation can exhaust all possibilities until there are no longer any stable extensions. The right side of the figure illustrates a case where propagation alone can be used to determine the complete labelling without using

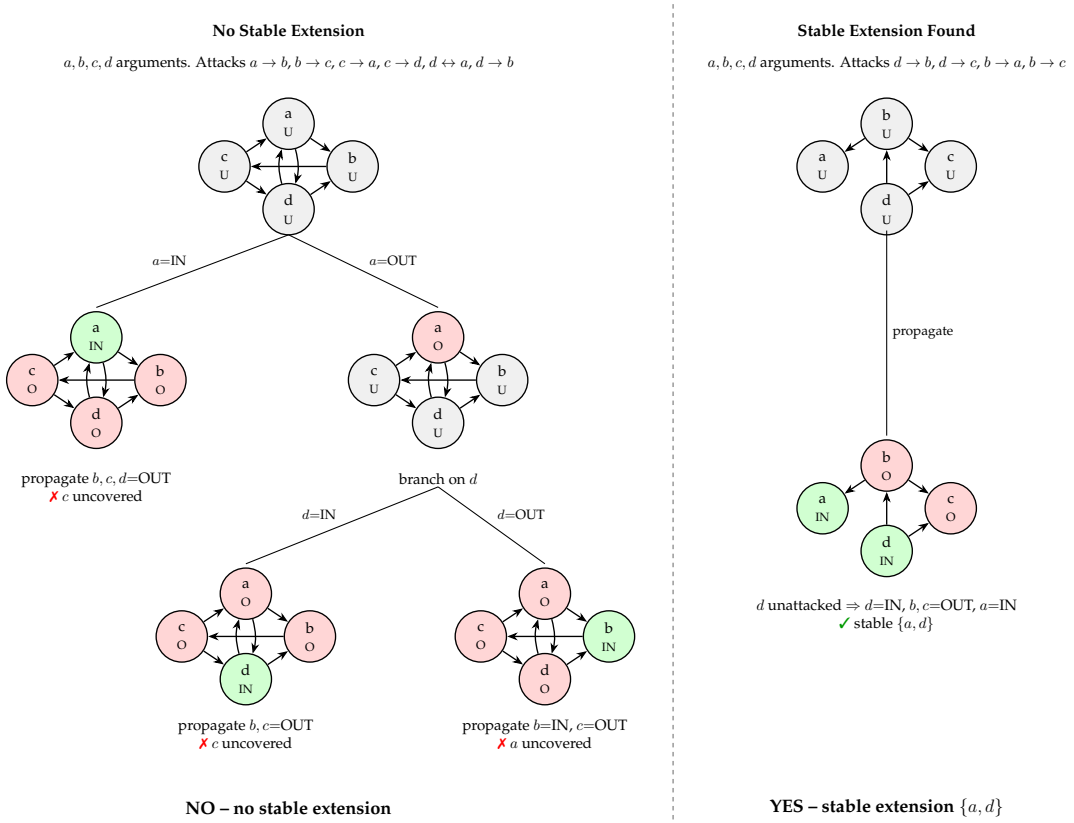


Figure 9: Backtracking search for stable extensions.
Left: every branch ends in a conflict, so no stable extension exists.
Right: propagation alone determines the complete labelling without branching.
Colours indicate labels: green = IN, red = OUT, gray = undecided.

any branching.

In the left side of the figure, every branch results in a conflict, so the solver must refute each of the conflicts before reporting that there is no stable extension.[2] On the right side of the figure, the unattacked argument d triggers a chain of propagation that determines the complete labelling without any branching.

4.3 Source SCC Precheck

Importantly, the same backtracking search that solves the problem is also the basis for the source SCC precheck. Before the main search begins, the solver looks through

[2] A similar decision would choose c over d as the second branching variable (Section 4.2); d was chosen here because its two branches illustrate how both the `IN` and `OUT` labels can result in a coverage conflict.

the source SCCs, i.e., SCCs that do not have incoming edges from other SCCs in the condensation DAG. If there is a source SCC that has no stable extension, then the entire instance can be immediately rejected (Lemma 1).

The preliminary check is only useful for small SCCs, because checking a large SCC takes about as long as solving the whole instance. For this reason, SCCs larger than a certain size limit are skipped during the precheck. Although the preliminary check for each of the small SCCs may slightly slow the solver down on some instances that could be solved very quickly, the cost of checking larger SCCs is likely to greatly exceed the possible speedup. Additionally, since the preliminary check is dependent upon the type of graph structure, the solver can make an adaptive choice regarding whether or not to use the preliminary check at all.

Lemma 1 (Source-SCC Check Soundness)**.** *Let F be an argumentation framework and C a source SCC. If $F|_C$ has no stable extension, then F has no stable extension.*

Proof. Since C is a source SCC with no incoming attacks from outside, any stable extension S of F restricts to a stable extension of $F|_C$ by the SCC recursion result [BG07, pp. 685–687]. If even this single source component has no stable extension, then F cannot have one either. □

Section 4.5 discusses the threshold values for this adaptive choice, and Section 5 evaluates how much influence the preliminary check has on the solving process for each benchmark family.

4.4 Parallelisation

Some graphs produce a search tree that can exceed millions of nodes, and a single core could take hours to solve the problem even on very fast hardware. That is why we focused on distributing the search across the available processors. A first simple idea would be to let several threads start simultaneously with different heuristics; however, they will still explore the search tree in a different order and could end up duplicating a large portion of the work. It is at this point that our method differs, as Figure 10 illustrates for two threads: we divide the search tree into subtrees and assign a different subtree to each thread, so that no thread repeats any of the work.

Running example (continued). As shown in Figure 11, only the scheduling portions of the running example are shown here, and the propagation is omitted as it is identical to what was explained above. On the left side of Figure 11, Thread 1 continues along the branch on c as the owner of the task and puts the sibling subtree on the back of its deque for some other thread to pick up. On the right-hand side, Thread 2 removes the first task from Thread 1's deque, makes an independent copy of the labelling state ($a = \texttt{IN}$, $b = \texttt{OUT}$), and starts its own search for the sibling branch. As soon as either of the two threads finds a witness labelling, it sets the atomic stop flag, which stops the other thread when it checks for termination at the next iteration.

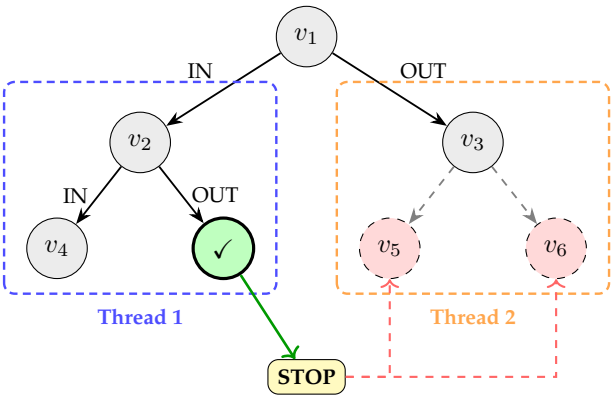


Figure 10: Parallel search tree exploration. Thread 1 explores the left subtree and finds a stable extension (green node with check mark). The stop flag terminates Thread 2 early (dashed red nodes). Each thread works on its own state copy to avoid synchronisation overhead.

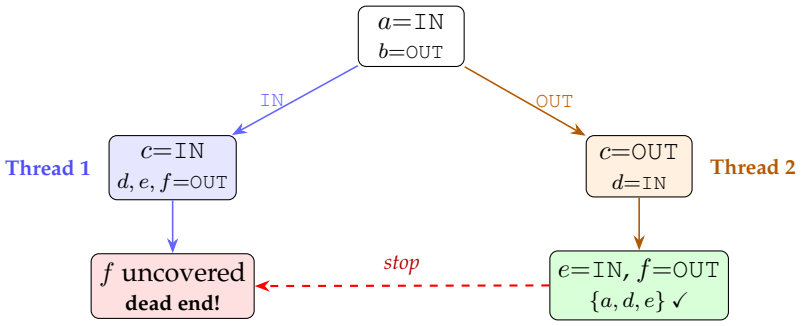


Figure 11: Work-stealing on the running example. Thread 2 steals the $c = \texttt{OUT}$ subtree, finds $\{a, d, e\}$, and sets the stop flag. Thread 1 terminates the dead-end branch.

Since the amount of synchronisation is minimised, the graph is read-only and shared between all threads. The only shared, writeable objects are the atomic stop flag, the lock-protected deques for moving tasks from one thread to another and a small number of atomic counters for termination and spawning logic. Because each thread only examines a completely different section of the search tree, there is no reason to synchronise the access to each thread's local labels, counters, and trail stacks. If all of the aforementioned data structures were to be shared among the threads, then each of them would have to be accessed sequentially (i.e., in lock-step), thus removing the benefit of using multiple threads.

Work-stealing For parallelism, the search algorithm uses the Blumofe-Leiserson strategy [BL99] to provide a randomised work-stealing approach. According to Blumofe and Leiserson, there exists an upper bound for the average-case execution time of their randomised victim selection strategy:

$$T_P \leq \frac{T_1}{P} + O(T_\infty).$$

In plain terms, this means that adding more processors reduces the runtime roughly in proportion to their count, with a small overhead that depends on how long the critical path is. Hence, assuming that the total work to be performed is greater than the length of the critical path, the scheduler should achieve linear speedup. Blumofe and Leiserson proved the bound for randomised victim selection, whereas our solver utilises cyclic scanning (Algorithm 4). Our experimental results presented in Section 5 show that the estimate of the average-case execution time also holds in practice.

Our work-stealing strategy is a simplified version of the one described in Algorithm 4. Each thread has a deque from which it pulls tasks in a depth-first manner from the back. An idle thread will steal from the front of another worker's deque in a cyclic manner, always choosing the oldest (and therefore typically the largest) subtree. Figure 12 shows this for two threads.

To support the parallelisation of the search, the algorithm must distinguish between task-specific data and shared data. At the start of a new traversal of a frontier or component-split task initiated by a thread, the thread will have an empty history for its trail. Shared data objects in Algorithm 4 include the read-only graph, the atomic stop flag, the lock-protected deques, and the atomic counters representing the number of active workers, the budget, and the spawning controls. Also, at the start of the computation, one thread is responsible for propagating the root state and placing the resulting initial frontier state in the deques on a round-robin basis (Algorithm 4).

Each thread calls the SEARCH function (Algorithm 1) and creates a new task (representing a sibling branch) to be added into its deque or do nothing based on whether the spawn guards permit it. Four limitations control the creation of a new task for representing a sibling branch. The first limitation is the minimum split

depth constraint. This ensures that recursion has occurred at least a certain depth, as most shallow branches will have insufficient data to warrant duplication of the state. The second limitation is the backlog constraint. It prevents a single busy thread from adding many tasks to the deque while the other threads are processing their own tasks. The third limitation is the capacity limitation. It limits the total number of tasks in the deques, so they do not grow indefinitely as a result of an overly large branching factor in the graph. The fourth limitation is a global budget counter that restricts the overall number of tasks created. Once the budget has been used up, the throttle mechanism refills it at regular intervals so that new tasks can be created again.

Algorithm 4 Parallel Work-Stealing Search

Require: Framework F, thread count P, deques $D[1 \dots P]$
Ensure: status flag indicating whether a stable extension was found

```
1:  function ParallelSearch(F, P)
2:      L_0 ← labelling with all arguments UNDEC
3:      frontier ← BuildFrontier(L_0)
4:      for i ← 1 to |frontier| do
5:          push frontier[i] onto D[((i−1) mod P) + 1]                ▷ Round-robin
6:      stopped ← false; active ← 0

7:      for each thread t ∈ {1, ..., P} in parallel do
8:          while not stopped do
9:              if D[t] ≠ ∅ then
10:                 task ← pop from back of D[t]                  ▷ LIFO, depth-first order
11:             else
12:                 stolen ← false
13:                 for o ← 1 to P do
14:                     t' ← ((t + o − 1) mod P) + 1
15:                     if t' ≠ t and D[t'] ≠ ∅ then
16:                         task ← steal from front of D[t']          ▷ Oldest subtree
17:                         stolen ← true; break
18:                 if not stolen then
19:                     if all D[1], ..., D[P] = ∅ and active = 0 then
20:                         break                                     ▷ No work left
21:                     continue
22:             active ← active + 1
23:             if Search(task.labelling) = YES then                  ▷ Algorithm 1
24:                 stopped ← true                                    ▷ Atomic flag
25:             active ← active − 1
26:     return stopped
```

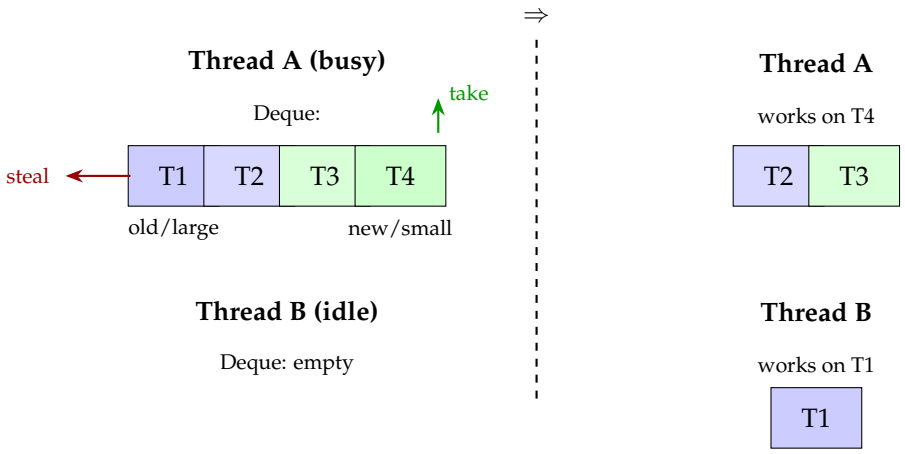


Figure 12: Work-stealing between two threads. Thread A takes task T4 from the end of its deque (small subtree, depth-first). Thread B steals task T1 from the beginning (large subtree, much work). This avoids conflicts and balances load automatically.

Termination relies on two atomic counters, *pending* and *active* (Algorithm 4). A thread that becomes idle will terminate the loop when both counters reach zero, which indicates that there are no threads currently being executed and there are no new tasks being generated.

The BUILDFRONTIER function on line 3 of Algorithm 4 uses a single thread to perform propagation (Rules 1–5) and defence forcing prior to starting the parallel loop. If undecided arguments remain, the single propagated state is assigned to one of the thread deques (lines 4–5). All further tasks are then created dynamically during the search by sibling-branch spawning and component splitting. Should propagation alone solve the problem, the parallel loop is skipped entirely.

Each deque is implemented as a `std::deque` and is protected by an `atomic_flag` spinlock. Since the spawn guards restrict the number of tasks per thread, contention between the owner and the stealing thread is low.

Another form of parallelisation of the search is illustrated in Figure 13. If propagation assigns a label to a node that connects two different components, the edges linking that node leave the undecided subgraph and the components split apart. The current thread then continues with the larger component, while the smaller ones are placed into the deque as separate tasks.

Parallel label lookahead. Since the heuristic scores for `IN` and `OUT` are often very close at the root of the search tree, the choice of which direction to take first is almost random. To make this less sensitive, we perform a simple lookahead at the beginning of the parallel search. In this lookahead, each direction receives a full propagation pass on a saved copy of the state. The direction that leaves the most undecided arguments is chosen first, because a larger remaining subtree contains

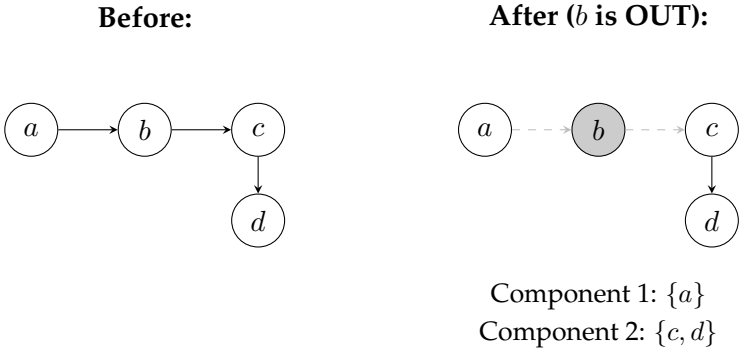


Figure 13: Component decomposition. When node b is marked OUT, its edges are removed from the UNDEC subgraph, causing it to split into two independent components.

more work that can be distributed to idle threads via work-stealing. We also process the second direction, and if the spawn guards allow it, we add the second direction to the deque as a separate task. If the spawn guards do not allow it, the thread will continue to process the second direction locally once the first direction is complete.

SCC-Level Parallelism The second phase of the heuristic cascade (Algorithm 3) tends to pick an option within the current SCC, while the first phase based on coverage will occasionally select an option outside of the current SCC when there are multiple SCCs included in the undecided subgraph. However, because of the way dependencies are defined by the DAG associated with the condensation of the graph, cross-SCC parallelism is not used in WallEParBt, because each SCC depends on choices made in preceding SCCs.

To see why, we consider a case where A has two distinct local labellings (i.e., $a_1 \mapsto$ IN and $a_1 \mapsto$ OUT); similarly, B has two distinct local labellings (i.e., $b_1 \mapsto$ IN and $b_1 \mapsto$ OUT). Therefore, the successor SCC C can only be solved using either $(a_1^{in} \wedge b_1^{out})$ or $(a_1^{out} \wedge b_1^{in})$. Therefore, even though the two threads have selected a_1^{in} and b_1^{in}, respectively—which makes both of them locally correct—C is still unsolvable. In a naive parallel scheme that solved SCCs independently, this situation would lead to the answer NO even though a globally valid extension exists. This is why WallEParBt does not parallelise across SCC boundaries.

As shown in Figure 14, the primary challenge is that a single SCC can produce many different local labellings. However, this does not mean that all combinations of the local labellings of the predecessors can create a globally valid solution. If the successor SCC determines that no labelling exists that is consistent with the labellings inherited from its predecessors, the labellings that were selected for the predecessors must be rejected and another combination must be attempted. This is the exact kind of overhead that we wish to avoid.

In practice, WallEParBt is primarily using SCC information as a preprocessing

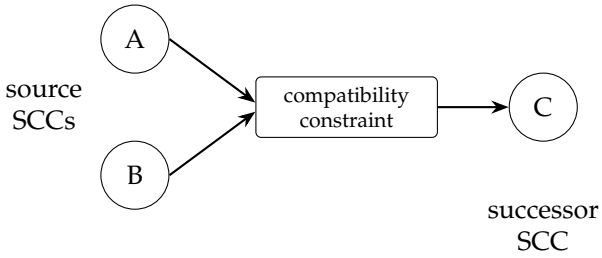


Figure 14: SCC-level parallelism can pick locally valid labellings for source SCCs A and B, but the successor C may admit only specific combinations. The gate marks this global compatibility constraint, and WallEParBt therefore derives most of its parallelism from splitting the search itself rather than from solving mutually dependent SCC choices independently.

signal and as a branching preference within the search.

Some families contain many small SCCs, for example crusti (31–511 SCCs), BA (10–200 SCCs) and planning (12–1237 SCCs). In these cases, the potential improvement in performance is going to be very limited. Brent's theorem [Bre74] states that the total speedup achievable with multi-threading will eventually plateau when the total execution time is determined primarily by the length of the longest sequential dependency chain. For the sequential part of the solver, the largest contributors to this chain are the grounded propagation step, the Tarjan step and those SCC dependencies that have fewer opportunities for concurrent work to be assigned to threads. Therefore, whether the work-stealing mechanism can achieve, or even surpass, the Amdahl limit depends upon how large the search tree is for each individual SCC. Once a thread finds the first witness extension before the search terminates, it may also mean that less work was done in total than would be accomplished by the sequential version.

Running example (continued). The SCCs produced by Tarjan's algorithm in the running example are $\mathrm{SCC}_1 = \{a, b\}$ and $\mathrm{SCC}_2 = \{c, d, e, f\}$, as shown in Figure 6. Because the DAG after condensation only contains the edge $\mathrm{SCC}_1 \to \mathrm{SCC}_2$, the choices made in SCC_1 will determine what options remain available in SCC_2. Because of the cross-boundary attack $b \to c$, the boundary condition transfers from SCC_1 to SCC_2: if b is labelled `IN` in SCC_1, then because of conflict-freeness c must be labelled `OUT`, and the labelling of c is dependent upon the results of SCC_1. As a consequence, SCC dependencies will impact the search process and thus provide an upper limit on the amount of concurrency that can be realised across SCC boundaries.

4.5 Adaptive Configuration

Not all graph structures react uniformly to the preprocessing step. Accordingly, the solver in AUTO mode evaluates the graph metrics at the beginning of each run and selects the configuration. Both of the most important decisions made during a run are directly dependent upon the same set of graph metrics. First, as also investigated in H3, the solver determines whether to apply the source SCC precheck at all, and if so, which source SCCs are small enough to precheck. Second, the solver determines whether to run a grounded propagation pass before the search, which fixes as many labels as possible without any branching. Both of these decisions only impact the quantity of work required for the search. Neither of these decisions affects the correctness of the result.

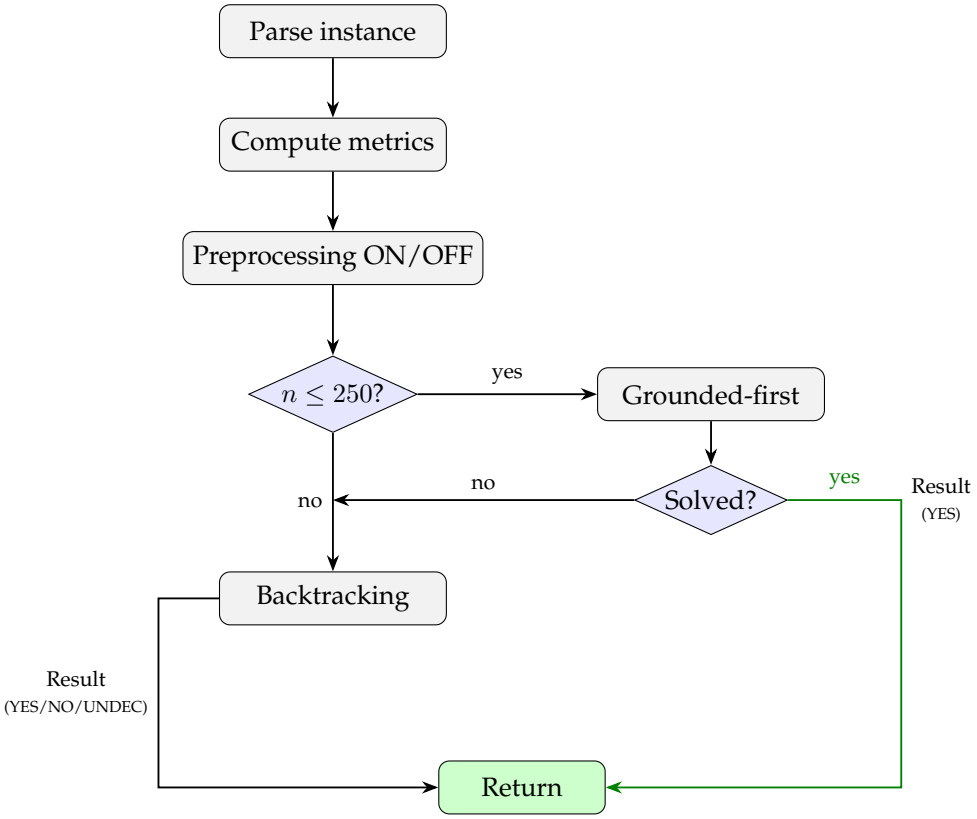


Figure 15: Adaptive configuration decision flow in AUTO mode. Thresholds are listed in Table 2.

Figure 15 illustrates the process for dynamically adapting to the input graph. The solver gathers five metrics from the first pass through Tarjan's algorithm: (1) the ratio of maximum SCC size to the total number of vertices $|\text{maxSCC}|/n$, (2) the total number of strongly connected components, (3) the edge density ρ, (4) the average SCC size $\bar{s}$, and (5) the proportion of tiny SCCs f_{tiny}. Together, these metrics de-

termine whether the graph is structured or random-like. For smaller graphs (i.e., where $n \leq 250$), the grounded-first heuristic is activated, since these graphs require much less branching than larger graphs. All threshold values are listed in Table 2.

Prior to beginning a search, the solver establishes a node budget. When working with a graph that contains many SCCs, the solver enters a restart loop. Once there is no budget left, the solver aborts the current search and resets the node budget back to its original value. It then restarts from the root node without retaining any of the information generated during the previous search. Although the stealing order itself is cyclic rather than randomised, a restart can still lead to a different search path because task creation and thread timing may change which subtrees are explored first.

In most cases, a single restart is enough. After that, the search continues without any budget limit.

Following every search that was aborted due to exhaustion of the node budget, the solver performs a top-SCC fallback check. This check examines only the source components of the SCC decomposition to determine if the instance can already be rejected. Because the fallback runs after the main search has already been abandoned, it has a bigger budget than the initial precheck, this can be seen as a last chance rescue action.

Decision / Limit	**Condition**	**Value**
Preprocessing	$\|\text{SCCs}\| > 20 \wedge \rho < 0.01$	ON (transit-like)
	giant SCC ($\|\text{maxSCC}\|/n > 0.5$)	OFF (random)
	$\|\text{SCCs}\| \leq 5 \wedge \bar{s} > 0.45n \wedge \rho < 0.12 \wedge f_{\text{tiny}} < 0.2$	OFF (random-like fallback)
	otherwise	ON (structured)
Grounded-first	AUTO, $n \leq 250$	ON
	$n > 250$	OFF
Source-SCC precheck	preprocessing ON $\wedge$ $n \geq 200$	Enabled
	$\|C\| \leq 500$	Component eligible
	node limit	200,000 per component
	time limit	2.0 s per component
	scan budget	$\min(0.30 \cdot \text{timeout},\ 60\,\text{s})$

Table 2: Adaptive configuration thresholds and precheck limits. $\rho = m/n^2$ (density), $\bar{s} = n/|\text{SCCs}|$ (mean SCC size), f_{tiny} = fraction of SCCs with ≤ 4 nodes. All values were calibrated on the ICCMA 2023 benchmark and transferred to the ICCMA 2025 suite without retuning.

4.6 Implementation

We implemented the solver in C++20. We use OpenMP for threading. We wrote our own scheduler for the work-stealing deque and task execution logic, since the

default OpenMP task model does not provide enough control over the subtree splitting described in Section 4.4. In addition to implementing a CLI compatible with `probo2`, the solver also includes an extended CLI, with the options in Table 3. We compiled and tested the solver under both Linux and macOS. The full implementation can be viewed and downloaded on GitHub.[3]

Table 3: Command line flags and output format.

Flag / Output	Values	Description
ICCMA mode		
`-p <task>`	SE-ST, DC-ST, DS-ST	Problem to solve
`-f <file>`	path	Input file in `.af` format
`-a <arg>`	argument id	Query argument (DC-ST, DS-ST only)
Extended mode		
`--threads`	N	Number of OpenMP threads
`--timeout`	seconds	Wall-clock time limit
`--pre`	auto | on | off	Preprocessing mode
`--heuristic`	full | no-coverage | degree-only	Heuristic ablation
`--csv`	--	Write CSV summary line
Output (SE-ST)		
`w <ids>`		Witness stable extension found
`NO`		No stable extension exists
`UNKNOWN`		Timeout or node budget reached

4.7 Correctness and Complexity

All proofs in this section use the following notation: $n = |\mathcal{A}|$ as the number of arguments in the framework; $m = |\mathcal{R}|$ as the number of attacks; T as the number of threads; and δ as the maximum depth in the SCC-DAG.

Proposition 1 (Termination)**.** *For any given input, the algorithm halts within a finite amount of time.*

Proof. In each call to the recursive function, at least one of the undecided arguments (`UNDEC`) is changed to either `IN` or `OUT`. Hence, the number of undecided arguments is decreased by at least one with each call. Thus, if we have n arguments in our framework, it can take us at most n calls to reach our maximum recursion level. Therefore, the search tree contains at most $O(2^n)$ nodes. Each node performs an amount of polynomial work during propagation and conflict detection, so the total amount of work performed is $O(2^n(n+m))$. □

Proposition 2 (Soundness)**.** *If the algorithm returns YES, then the framework has a stable extension.*

[3] `https://github.com/eckodo/WallEParBt`

Proof. Define ℓ to be the labelling produced by Algorithm 1. When the algorithm terminates, it checks to see if there are any undecided arguments (UNDEC). Then, using IsStable, it tests to see if the labelling ℓ is conflict-free, and if every argument is labelled $\ell(a) \in \{\texttt{IN}, \texttt{OUT}\}$. In other words, the set $\{a \mid \ell(a) = \texttt{IN}\}$ is conflict-free, and every a with $\ell(a) = \texttt{OUT}$ has an attacker b with $\ell(b) = \texttt{IN}$. By Definition 5, this means that ℓ is a stable extension [Dun95, p. 328]. □

Proposition 3 (Completeness)**.** *If the framework has a stable extension, and the search completes successfully (i.e., neither a timeout nor a node budget causes the search to terminate early), then the algorithm returns YES.*

Proof. Algorithm 1 tries all possible ways of assigning IN and OUT to each undecided $a \in \mathcal{A}$. Algorithm 1 only ever prunes a branch in two situations: a contradiction was found by propagation rules 1–5 in the current partial labelling, or Lemma 1 shows that a source SCC has no stable extension. Since both conditions are sound, no branch that may lead to a stable extension will be eliminated. □

Lemma 2 (Subtree Independence)**.** *Search tasks correspond to disjoint subtrees of the global search tree, and the order in which the tasks are executed has no bearing on the final result.*

Proof. When a task is assigned to another thread, the solver creates a duplicate copy of the current labelling state, which contains all of the labels and counters, and sets the trail back to the start. After that, each thread has its own separate labelling state, and they both backtrack separately. Together, all tasks look at the same set of possible assignments that would be looked at by a sequential depth-first search. □

Proposition 4 (Parallel Correctness)**.** *If the search does not terminate due to a timeout or a node budget, then the decision (*YES *or* NO*) produced by the parallel and sequential executions of the algorithm is identical.*

Proof. Subtree independence says that the parallel version looks at the same assignments as the sequential version, but in a different order, using work-stealing. An atomic flag guarantees that all of the threads stop when any of them finds a stable extension. If no threads find a solution, and the search did not terminate because of a timeout or node limit, then all of the subtrees have been completely searched, and the correct answer is NO. □

For the ICCMA 2025 experiments, every solver invocation is subject to a maximum wall-clock time of 600 seconds. If the wall-clock time limit is reached before the search tree of Algorithm 1 is completely explored, the solver reports UNKNOWN rather than YES or NO.

Complexity analysis To provide evidence that at least the three graph-processing phases (parsing, decomposition, and propagation) continue to be efficient, we prove that they can be completed in polynomial time, even though SE-ST has been proven to be NP-complete and there will be no polynomial-time algorithm for the backtracking phase of Algorithm 1 unless P = NP.

Theorem 4 (Parsing)**.** *The time complexity of the parsing phase is $O(n + \sum_v d_v \log d_v)$ and the space complexity is $O(n + m)$, where $n = |A|$, $m = |R|$, and d_v is the out-degree of v. The worst-case time complexity is $O(n + m \log m)$ if all the edges start from one node.*

Proof. First, the parser reads the m lines of the `.af` file one by one ($O(m)$). Either from the header or from the largest identifier that it encounters during parsing, it obtains the number of arguments n. Then, it creates adjacency lists for each of the arguments v, sorts them (for example, via mergesort in $O(d_v \log d_v)$ time per list), and finally removes duplicates. The parsed data is stored in CSR format, which requires $O(n + m)$ space. □

Theorem 5 (Complexity of SCC Decomposition)**.** *Using Tarjan's algorithm, the SCC decomposition of the graph takes $O(n + m)$ time and $O(n)$ auxiliary space.*

Proof. As Tarjan demonstrated [Tar72], the depth-first search of the graph visits each node and each edge exactly once. Therefore, the time taken by the algorithm is $O(n+m)$. Additionally, each node stores three values of constant size: the discovery time, the smallest reachable index in the subtree rooted at this node, and whether this node is currently on the recursion stack. Thus, the auxiliary space used by Tarjan's algorithm is $O(n)$. Furthermore, a traversal of the SCC-DAG for topological sorting takes linear time. Therefore, we conclude that the time taken by the algorithm is $O(n + m)$, and the auxiliary space is $O(n)$. □

Theorem 6 (Complexity of Grounded Propagation)**.** *Grounded propagation takes $O(n + m)$ time and $O(n)$ auxiliary space.*

Proof. For grounded propagation, a worklist is used, and each argument v appears at most twice in the worklist based on the labelling rules of Section 4.2. Once for acceptance (`IN`), and once for rejection (`OUT`). Each time the routine processes an argument v in the worklist, it checks its attackers and targets, i.e., the $\deg(v)$ edges, so that the total number of edges worked on will be $\sum_v \deg(v) = O(m)$, and the three per-argument counter arrays (the non-rejected attackers, undecided attackers, and coverage status) plus the worklist that can hold at most n items at any time will require $O(n)$ additional space. Therefore, from these two bounds, the time and space requirements of $O(n + m)$ and $O(n)$ respectively, for grounded propagation are immediately concluded. □

Lemma 3 (One Propagation Call)**.** *A single call to the propagation routine after assigning labels takes $O(n + m)$ time.*

Proof. We have proven in Theorem 6 that the worklist propagation goes through a fixed sequence of phases, in each of which node arrays are scanned linearly, and predecessor or successor lists are traversed. At the same time, counters are updated, forced labels are found, and possible conflicts are checked. Because each node and each edge is examined a constant number of times across all of the phases in a single call to the propagation routine, the overall run time is $O(n + m)$ for a single call to the propagation routine. □

Since Lemma 3 is now available, the cost of the entire backtracking core of Algorithm 1 can now be summarised as shown in Table 4.

Metric	**Bound**	**Source**
Per-node cost	$O(n + m)$	Lemma 3
Search tree	$\leq 2^n$ leaves	Binary `IN`/`OUT` branching
Total runtime	$O(2^n(n + m))$	Product of above
Space (T threads)	$O(n + m + Tn)$	Graph + T thread copies

Table 4: Complexity of the backtracking core (Algorithm 1).

At every node it visits, the call to PROPAGATE in line 4 of Algorithm 1 takes $O(n + m)$ time per Lemma 3, and the call to SELECTVARIABLE goes through each of the candidates once, so this is also bounded by the propagation cost. The for-loop in lines 13–17 of Algorithm 1 has a maximum of two child nodes (i.e., one for `IN` and one for `OUT`), so the search tree can grow up to 2^n leaves, and thus the total worst-case running time is

$$T(n, m) = O(2^n \cdot (n + m)),$$

which matches the NP-hardness result of Dvořák and Dunne [DD18]. Typically, the propagation in line 4 of Algorithm 1 will fix a significant number of the labels near the root of the tree, and thus the actual number of nodes that the solver visits for the ICCMA 2025 instance set is significantly smaller than 2^n. The copy operation in line 14 of Algorithm 1 uses trail snapshots to make the copy, and therefore only the changed labels are copied into the trail snapshot. Therefore, the solver never makes a complete copy of the labelling state until a task is transferred to the deques in Algorithm 4; the lookahead probes in lines 13–17 use the trail-based save/restore mechanism on the same labelling object. In addition to storing the graph itself ($O(n + m)$) and the T thread-local copies of the current labelling state ($O(Tn)$), in addition to the trail stack, each thread has its own set of counters. The two most extreme examples in terms of memory in the ICCMA 2025 suite are the densest instance ($n \approx 11\,000$, $m \approx 24\,000\,000$), for which the forward and backward CSR together use about 182 MB, and the largest instance ($n = 2\,500\,000$, $m \approx 3\,750\,000$), for which the CSR is about 48 MB but the thread-local copies use by far the most space. Each thread-local labelling copy contains n bytes for labels, two integer arrays of size n for the attacker counters, and n bytes for coverage flags, totalling

approximately $10n$ bytes. For the largest instance, that amounts to roughly 25 MB per thread, so 32 copies add about 800 MB. At the time of the experiments, 72 GB of RAM were available on the hardware, and therefore memory was not a concern. In practice, scalability is limited by Amdahl's law and the SCC-chain serialisation as described below. Amdahl's law [Amd67] defines the maximum speedup S of an N-threaded system to be

$$S(N) \;=\; \frac{1}{(1-p) \;+\; \frac{p}{N}}$$

where p is the portion of the program that may be executed in parallel. As can be seen from this equation, even if a very high proportion of the program (95%) may be executed in parallel, a low number of threads (32), results in a relatively small speedup, $S \approx 12.5$. If the number of threads is increased without limit, the maximum possible speedup is given by

$$\textit{speedup} \;=\; \frac{1}{(1-p)}.$$

In the case of the sequential part of the solver, this corresponds primarily to the grounded propagation step, the Tarjan step, and the SCC dependencies that can reduce the amount of work available to threads at the same time. Whether the work-stealing mechanism will be able to reach or possibly surpass the Amdahl limit depends on the size of the search trees within each strongly connected component. If the search is terminated early due to a single thread having found the first witness extension, the total amount of work performed will be lower than the sequential baseline.

Limitations and trade-offs There are three main reasons why the parallel search in `WallEParBt` is limited in scalability on specific types of graphs. Firstly, there is no information maintained by the solver about potential conflicts between the branches of the search tree. Therefore, any dead end found in one branch of the search tree must be rediscovered in every other branch of the search tree. Secondly, we chose not to implement conflict learning or watched literal management. On the small-to-medium instances in the ICCMA benchmark, the synchronisation overhead would negate any benefit. Thirdly, SCC dependencies make it difficult to expose large amounts of independent work across component boundaries, so the implementation obtains most of its parallelism from splitting search subtrees rather than from solving predecessor and successor SCCs independently. All of the heuristic weights in SELECTVARIABLE were tuned for the ICCMA 2023 instances, and we investigate in Section 6 whether they generalise to the ICCMA 2025 benchmark. Since the solver performs its computations directly on CSR arrays and reads `.af` files without any intermediate backends, it gives a straightforward basis against which researchers can compare their argumentation solvers.

Quality of the code and testing To test the quality of the code, the codebase was statically checked by the static code analysis tool Cppcheck [Mar24] and the thread sanitiser [SI09] while developing Algorithm 2 and Algorithm 4. We identified a few errors, such as unintentional pass-by-value copies of the labelling array and an uninitialised counter. We corrected these errors before running any experimental trials. The unit tests are run via `cd build && ctest --output-on-failure`.

5 Evaluation

The solver as described in Section 4 is a combination of work-stealing parallelism, SCC-based preprocessing, and an adaptive heuristic. Since the contribution of each mechanism to the overall performance cannot be isolated from the design alone, we test the three hypotheses from the introduction in separate experiments on the full ICCMA 2025 suite (322 argumentation frameworks from various graph families). Experiment 1 shows how the solver scales with thread count and which graph families benefit from work-stealing. Experiment 2 compares the FORCEPRE and NOPRE preprocessing configurations to determine on which families the SCC precheck helps or hurts. Experiment 3 tests if the adaptive AUTO mode can select the optimal preprocessing strategy for each input instance compared to using fixed preprocessing strategies as well as an oracle. Experiment 4 compares WallEParBt-32T with the top-performing SE-ST solvers in the ICCMA 2025 main track (Scalop, FUDGE, and μ-toksia) on the same AWS hardware using a 600-second time limit. Scalop, FUDGE, and μ-toksia are single-threaded, therefore ASPARTIX was included as a parallel baseline (clingo backend, 32 threads). No parallel backtracking solver was present in the ICCMA 2025 solver pool for SE-ST, therefore the baselines represent the best solvers currently available for this task. The results of the four experiments are discussed in Section 6.

5.1 Experimental Setup

Metric	Definition
Coverage	Instances solved within 600 s
Wall-clock time	Elapsed real time per instance
PAR-2	Timeouts count as 2×600 s
Speedup $S(t)$	T_1/T_t (T_1 = sequential, $T_t = t$ threads)
Efficiency $E(t)$	$S(t)/t$ (1.0 = ideal)
Parallel fraction p	From Amdahl's law [Amd67]

Table 5: Metrics used in the evaluation.

All binaries are based upon the same source revision (x86-64 for AWS, ARM for M1 Pro). All experiments utilised wall-clock time, following the ICCMA No-Limits

convention. The **AWS c5.9xlarge** machine (Intel Xeon Platinum 8124M, 36 vCPUs, 72 GB RAM, Linux) provided sufficient capacity for the operating system alongside the 32 solver threads. We also ran the H1 scaling experiment on a MacBook Pro with an Apple M1 Pro (see Section 6.3 for the platform comparison).

The time limit for each solver call was 10 minutes (600 s), shorter than the 20 minutes used in the ICCMA competition. This allowed testing all six thread levels for both preprocessing modes. It also made targeted repeated-run checks feasible. Table 5 lists the metrics used. Table 6 lists the tested configurations. `WallEParBt-`x`T` denotes the solver running with x worker threads, i.e., `WallEParBt-1T` is the sequential baseline and `WallEParBt-32T` is the 32-thread configuration.

Table 6: Evaluated solver configurations.

Solver	Type	T	Pre	Purpose
WallEParBt-1T	Direct BT	1	AUTO	Sequential baseline (H1)
WallEParBt-2T	Direct BT	2	AUTO	Scaling (H1)
WallEParBt-4T	Direct BT	4	AUTO	Scaling (H1)
WallEParBt-8T	Direct BT	8	AUTO	Scaling (H1)
WallEParBt-16T	Direct BT	16	AUTO	Scaling (H1)
WallEParBt-32T	Direct BT	32	AUTO	Full system (H1, Exp. 4)
WallEParBt-32T-forcepre	Direct BT	32	ON	Preprocessing ablation (H2, H3)
WallEParBt-32T-nopre	Direct BT	32	OFF	Preprocessing ablation (H2, H3)
FUDGE [TCV23]	SAT/CaDiCaL [BFFH20]	1	–	ICCMA 2025 baseline (Exp. 4)
μ-toksia [NJ20]	SAT/CaDiCaL	1	–	ICCMA 2025 baseline (Exp. 4)
Scalop [LLM25]	SAT/IPASIR	1	–	ICCMA 2025 baseline (Exp. 4)
ASPARTIX [EGW08]	ASP/clingo [GKKS14]	32	–	ASP baseline (Exp. 4)

5.2 Statistical Treatment

Since solver coverage alone does not tell the whole story, the supplementary metrics (runtime distributions, PAR-2 scores, family-wise win rates, and head-to-head tables) help to identify where each solver has its strengths.

Each instance was run once per solver configuration, the same practice as in the ICCMA competition. This means that runtimes close to the 600 s timeout boundary have some degree of noise. For this reason, a repeated-run check was conducted on 290 of the 291 solved instances (Table 7), and small runtime differences near the timeout boundary are treated with caution in all experiments below.

Subset	n	Median CV	Max CV	P90 CV
All solved instances	290	1.17%	26.3%	3.84%
Runtime $>$ 1 s	30	0.26%	12.4%	4.04%
Runtime $>$ 10 s	9	1.85%	12.4%	7.40%

Table 7: Runtimes are stable: the median CV across all solved instances is below 2 %.

For each solver pair, only instances that both solvers completed are counted. We

report two views, one over all commonly solved instances and one over a hard subset where at least one solver took more than 1 s. Without this filter, trivial instances solved in milliseconds would dominate the comparison.

5.3 Benchmark Graph Families

Family	n	Generator	Structure	# SCCs	Pre.	W-S.
ER	20	uniform random	1 giant SCC	1	fixes some	eff.
WS	20	ring + rewire	1 giant SCC	1 + periph.	fixes some	eff.
BA	20	pref. attach.	many tiny SCCs	10–200	fixes all	n/a
ST	20	semi-acyclic	dominant SCC, shallow DAG	7–260	subtasks	very eff.
crusti	20	binary SCC tree	deep chain, small SCCs	31–511	chained	limited
GML	22	transit networks	varies	varies	varies	varies
SCC	20	many small SCCs	fragmented	2–50	subtasks	limited
admbuster	15	admiss. tests	acyclic	Trivially solved		
sembuster	15	semistable tests	acyclic	Trivially solved		

Table 8: Benchmark families and their structural profiles as seen by the solver.

Table 8 shows the nine graph families that cover 172 of the 322 instances. The remaining 150 instances are grouped as "other." ER, WS, and BA are based on the classic random-graph models [ER59, WS98, BA99]. GML is the only family derived from real-world transit networks.

As Table 8 shows, the families differ widely in their SCC structure. This has a direct impact on whether parallelism helps or not. Section 6 explains the details.

5.4 Results

5.4.1 Experiment 1: Thread Scaling for H1

Setup WallEParBt was run in AUTO mode at $T \in \{1, 2, 4, 8, 16, 32\}$ on the entire 322 ICCMA 2025 benchmark with a 600-second timeout on the AWS c5.9xlarge instance described in Section 5.1. All other parameters were set to their default values as specified in Section 4.6.

Result As can be seen in Figure 16 and Table 9, the families of ST, ER, and WS benefit the most from an increase in the number of threads. Each of these three families contains one SCC that accounts for the vast majority of the search effort. Section 6 explains why the SCC structure of these families benefits from work-stealing. Total coverage in Table 9 increases as the thread count doubles until it plateaus at four threads. Above that point, no additional instances are solved. However, the PAR-2 score continues to decline because the already-solvable hard instances finish more quickly when more threads are available. Several instances that could not be solved in sequential mode are now solvable at four threads and above (Table 10).

Config	Solved	Coverage	vs 1T	PAR-2
WallEParBt-1T	286	89%	(base)	45 009
WallEParBt-2T	287	89%	+1	43 348
WallEParBt-4T	291	90%	+5	39 616
WallEParBt-8T	291	90%	+5	38 477
WallEParBt-16T	291	90%	+5	37 902
WallEParBt-32T	291	90%	+5	37 740

Table 9: Coverage plateaus at four threads; PAR-2 keeps dropping as solved instances finish faster.

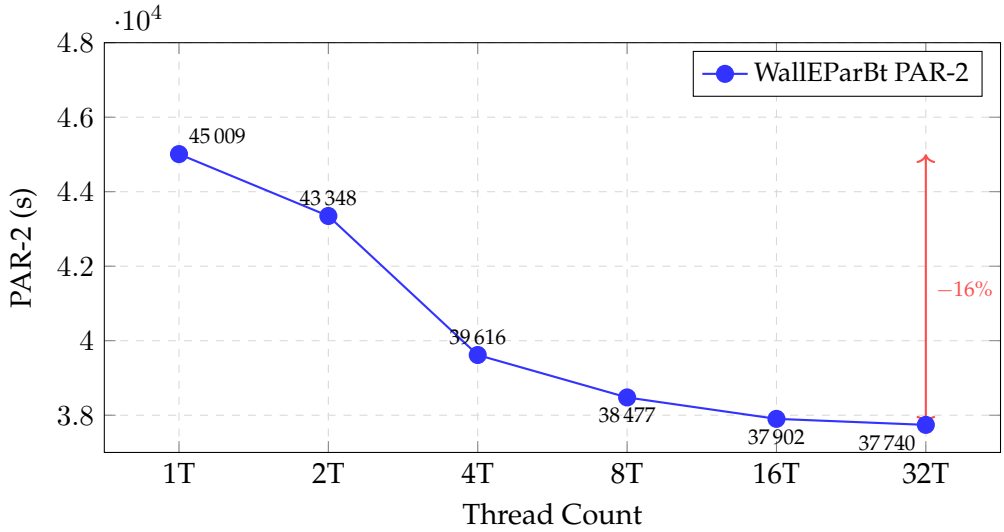


Figure 16: PAR-2 drops steadily with more threads, even after coverage plateaus at $T{=}4$.

Parallelism therefore not only reduces the time to solve problems, but also makes previously unsolvable problems solvable within the given timeout. BA and admbuster, on the other hand, are trivial. Propagation alone resolves every instance at $T{=}1$. With more threads, the SCC family is able to pick up a couple of additional solves (Figure 17). By contrast, crusti remains flat. Most of the timeouts at $T{=}32$ are due to crusti instances. Section 6 explains why.

Interpretation As shown in Table 10, representative instances reach speedups as high as 23.7 times. In a few cases, the measured speedup is even higher than the Amdahl prediction for 32 threads. Instance `st_395` is the clearest example, dropping from 278.5 s at $T{=}1$ to 0.3 s at $T{=}32$. Results like this indicate that one of the parallel threads reached a witness branch much earlier than the sequential search. Similar superlinear effects have been observed in parallel SAT solving as well [HJS09]. ER scales equally well, reaching a maximum of 20.2 times. Instance `st_542` is another useful example: it times out at $T{=}1$, but WallEParBt-32T finishes it in 48.4 s. Here, the single large SCC is broad enough for all 32 threads to stay busy.

Instance	T1	T4	T8	T16	T32	S_{32}	E_{32}
st_395	278.5s	1.1s	0.6s	0.4s	0.3s	1113×	≫100%
WS_400_16_90_50	476.8s	122.0s	61.4s	31.0s	20.2s	23.7×	74%
ER_300_30_4	163.5s	42.2s	21.6s	10.9s	8.1s	20.2×	63%
ER_500_50_1	249.6s	65.1s	33.5s	17.3s	13.5s	18.4×	58%
ER_500_50_3	244.7s	64.4s	33.1s	17.0s	13.3s	18.4×	58%
Timeout-to-solved (T1 > 600 s → solved at T4)							
st_542	>600s	272.0s	136.1s	69.9s	48.4s	≥12×	≥39%
WS_300_32_90_70	>600s	352.8s	179.1s	89.5s	58.6s	≥10×	≥32%
scc_1554	>600s	292.5s	154.1s	82.7s	56.0s	≥11×	≥33%
ER_300_20_2	>600s	465.7s	231.5s	118.1s	78.6s	≥8×	≥24%
ER_500_40_1	>600s	555.6s	284.3s	145.4s	111.8s	≥5×	≥17%

Table 10: Selected instances with high speedups. $S_{32} = T_1/T_{32}$, $E_{32} = S_{32}/32$. Timeout-to-solved entries are lower bounds.

Amdahl comparison By comparing our measured speedups against the Amdahl curves in Figure 18, the ST, ER, and WS families land close to the 95% line, indicating that around 95% of their search work is parallelisable. At higher thread counts, the measured values actually exceed the prediction. This is attributable to the early-termination effect, where one thread finds the witness and the remaining threads terminate. At $T{=}32$ the efficiency values in Table 10 decrease, which is plausible because spinlock contention on the work-stealing deque grows with the number of competing threads.

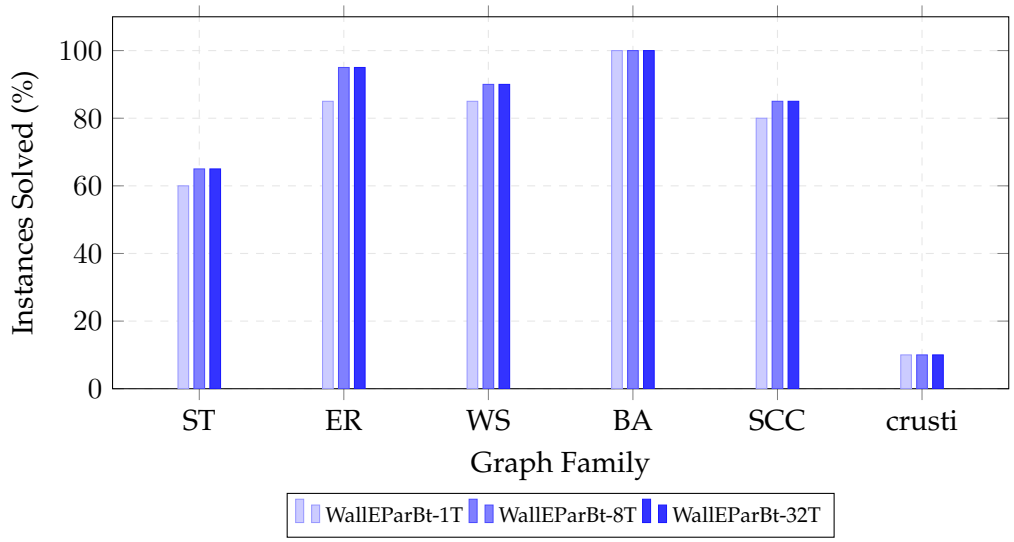


Figure 17: ER, WS, and ST gain coverage with more threads; crusti stays flat regardless of thread count.

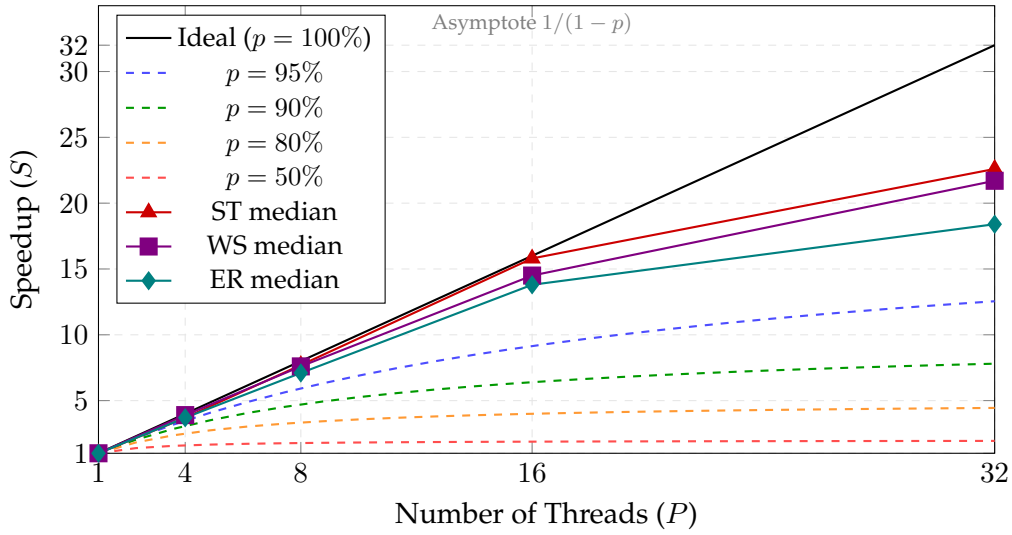


Figure 18: Measured per-family speedups (hard instances, $T_1 \geq 1\,\mathrm{s}$) exceed the Amdahl prediction at p=95% because work-stealing reduces total search work through early termination.

Answer to H1 H1 is supported. Based upon data from the ST, WS and ER families, the median speedup with 32 threads reaches up to 22.6× on ST, 21.7× on WS, and 18.4× on ER. In comparison to T=1, we saw a decrease in the PAR-2 score of 16%, and at the same time an increase in the number of problems that could be solved (Table 9). Of particular interest was the fact that there were several instances where a solution was found more quickly than the theoretical maximum as defined by Amdahl's bound. This is best explained by the order in which searches are made rather than by actual linear scaling of performance (Table 10). The crusti family however remained a clear outlier since there was no measurable benefit realised by using parallelisation (Section 6).

5.4.2 Experiment 2: Preprocessing Ablation for H2

Setup The goal of this experiment was to evaluate how FORCEPRE (SCC prechecking turned on) compares to NOPRE (no preprocessing, but Tarjan still runs for metadata). Both configurations were tested on all 322 ICCMA 2025 problems with 32 threads and the same 600-second time limit. Specifically, we wanted to find out whether the overhead of breaking down the graph into SCCs using Tarjan would be offset by the benefits seen across different types of graph families. Also, we needed to ensure that breaking down the graph would not have a negative impact on those graph families that create mostly sequential barriers. Table 11 provides the results broken down by graph family.

Family	n	Pre=ON	Pre=OFF	Speedup	Only PRE / NOP
scc_*	20	1 904.9s	6 098.2s	3.2×	7 / 0
GML	22	0.7s	0.7s	1.0×	0 / 0
g_*	20	2.4s	2.4s	1.0×	0 / 0
ER	20	927.7s	833.0s	0.9×	0 / 0
WS	20	1 295.2s	1 279.8s	1.0×	0 / 0
st_*	20	4 236.0s	4 845.1s	1.1×	1 / 0
crusti	20	10 801.0s	10 047.9s	0.9×	0 / 2
BA	20	0.1s	0.1s	1.0×	0 / 0
other	160	79.6s	70.7s	0.9×	0 / 0
Total	322	19 247.6s	23 177.9s	1.2×	8 / 2

Table 11: Preprocessing impact by family. SCC benefits strongly; crusti is slightly hurt. "Only PRE/NOP" counts instances solved exclusively by that mode.

Result As expected, the effects of preprocessing were directly related to the type of graph family. We observed that the source-SCC precheck provided a speedup factor of 3.2× on the SCC family, as the precheck was able to resolve independent sub-problems before the search began (see Table 11). On the ER family, the effect was

slightly negative (0.9× in Table 11) because the single giant SCC gave the precheck relatively few arguments to resolve. On crusti, however, the source-SCC precheck clearly harmed performance (see Table 11).

Concerning the "Only PRE / NOP" column in Table 11, 7 SCC instances were solved only using the preprocessing, while crusti gained 2 exclusively without it. This is most likely due to the complex SCC structure of the crusti instances, where between 31 and 511 small SCCs form a deep chain. Additionally, the ST family gained one additional instance exclusively using the preprocessing.

Interpretation High speedup factors were achieved on the top instances in Table 12 as the SCC precheck resolved the instances before the backtracking engine began.

Instance	Family	t_{PRE} [s]	t_{NOP} [s]	Ratio
st_884_6_17_34_91	ST	0.01	>600	≥92,515×
scc_7481_39_0.4_0.1_14	scc	0.48	>600	≥1,248×
scc_2951_7_0.5_0.2_18	scc	0.54	>600	≥1,112×
ER_300_20_2	ER	80.7	79.2	1.0×
WS_300_16_90_30	WS	1.79	0.11	0.06×
crusti_g2io_150_0.1_255_16	crusti	>600	178.0	–
crusti_g2io_200_0.1_127_14	crusti	>600	269.0	–

Table 12: Preprocessing contrasts. Ratio above 1 means preprocessing helps.

Concerning the number of nodes in Table 13, the SCC entries show that there were no search nodes as the precheck resolved them. Concerning the dense Erdős–Rényi graphs with a single dominant SCC, these expanded the same number of nodes under both configurations, since the giant SCC left nothing to split. Therefore, FORCEPRE solved 291 of the 322 instances and NOPRE solved 285 of the 322 instances.

Answer to H2 H2 is supported for both SCC and ST because the source-SCC precheck may either fragment the graph or resolve grounded arguments prior to beginning the search (see Table 11). However, in the case of the crusti instances, the precheck should be disabled as the instances have a deep chain structure. Overall, the results are positive (291 vs. 285 solved instances) for the whole benchmark.

5.4.3 Experiment 3: Adaptive Preprocessing for H3

Setup We ran the same experimental setup as before using an AWS c5.9xlarge machine with 32 threads and the same 600-second time limit on all 322 ICCMA 2025 instances. AUTO determines whether to enable the source SCC precheck based upon the SCC count, the maximum SCC size and the graph density read from a single

Instance	n	m/n	SCCs	max SCC	t (s)	$\log_{10}(2^n)$	$\log_{10}$(nodes)
ER — same graph size, density alone determines difficulty							
ER_300_20_2	301	30	1	301	78.6	90.6	7.64
ER_300_60_4	301	91	1	301	0.20	90.6	4.96
ER_500_40_1	501	100	1	501	111.8	150.8	7.87
ER_500_90_2	501	225	1	501	1.00	150.8	4.11
ST — component size drives exponential growth							
st_412_21_34_36_17	412	9.9	17	379	0.24	124.0	5.23
st_542_25_17_36_39	542	12	34	507	48.4	163.2	7.39
SCC — Tarjan precheck resolves fragmented graphs without search							
scc_280_15…	280	13	16	30	<0.01	84.3	*precheck*
scc_7836_35…	7 836	210	35	253	0.60	2359	*precheck*
BA — grounded propagation resolves sparse components							
BA_200_0_3	201	1.0	200	2	<0.01	60.5	*grounded*

Table 13: Edge density (m/n) and component size drive search effort. Denser ER graphs at the same size need orders of magnitude fewer search nodes. *Precheck / grounded* means no backtracking was needed.

Tarjan pass. The two fixed baselines FORCEPRE and NOPRE from the previous experiment were used as references. Table 14 compares the resulting coverage.

Configuration	**Solved**	**Timeouts**	**Total Time**
AUTO (adaptive)	291 (90%)	31	**19 139s**
FORCEPRE (fixed ON)	291 (90%)	31	19 248s
NOPRE (fixed OFF)	285 (89%)	37	23 178s
Oracle (best per instance)	*293* (91%)	*29*	–

Table 14: AUTO matches FORCEPRE in coverage and is marginally faster. The oracle gap is two instances.

Result AUTO solves 291 of the 322 instances with 31 timeouts (Table 14), while FORCEPRE reaches 291 and NOPRE reaches 285. AUTO does not solve more instances than FORCEPRE on this benchmark because every instance that FORCEPRE can solve is also classified as suitable for preprocessing by the heuristic. Compared to an ON/OFF oracle that always chooses the optimal configuration, AUTO falls short by only two instances across the entire benchmark (Table 14).

Interpretation As expected, the decision made by the AUTO heuristic was heavily influenced by the number of SCCs and the density of the graph. On the easy

instances, all three modes finish with roughly the same runtime. For the harder instances, FORCEPRE has lost 31 instances to timeout whereas NOPRE has lost 37 (Table 14). AUTO avoids the NOPRE shortfall by reading all three values (the SCC count, the largest SCC size, and the density) from a single Tarjan pass. As can be seen in Table 15, preprocessing wins on SCC and ST but loses on ER and WS, while BA shows no measurable effect.

Family	n	ON wins	OFF wins	Heuristic
scc_*	12	11	1	ON (multi-SCC)
GML	0	0	0	neutral
g_*	0	0	0	neutral
ER	11	0	11	OFF (giant SCC)
WS	7	0	7	OFF (giant SCC)
BA	0	0	0	neutral
st_*	3	3	0	ON (multi-SCC)

Table 15: Preprocessing wins on SCC and ST, loses on ER and WS. Wins include timeout-to-solved cases and $\geq$100 ms differences.

Answer to H3 The gap of two instances between AUTO and the ON/OFF oracle indicates that the adaptive heuristic captures the relevant structural features of the graph. FORCEPRE and AUTO achieve the same coverage on this benchmark. AUTO is slightly faster and the safer choice when the graph structure is unknown. AUTO also avoids the penalty associated with NOPRE on multi-SCC graphs (Table 14), which is why H3 is seen as supported.

5.4.4 Experiment 4: Solver Comparison Between Systems

Setup To compare the solver with existing ICCMA competitors, WallEParBt-32T, FUDGE, μ-toksia, Scalop, and ASPARTIX were run on the same AWS c5.9xlarge instance (36 vCPUs, Intel Xeon Platinum 8124M), on all 322 ICCMA 2025 instances, and under the same 600-second timeout. Table 16 lists the versions used in this comparison. For each solver pair, the win rate was computed on the instances that both solvers completed before timeout. Following the hard-subset filter from Section 5.2, trivial cases such as the BA family did not dominate the result.

Result Concerning the hard-subset win rates in Table 17, WallEParBt-32T won 98 percent of the time against FUDGE, 95 percent of the time against μ-toksia, and 92 percent of the time against Scalop over instances that took at least 1 second to complete. Our experiments showed that WallEParBt-32T achieves near-full coverage over the ER and WS families (Table 19). On the other hand, the SAT-based solvers achieved nearly full coverage of the crusti family, while WallEParBt-32T did

Solver	Version	Paradigm	Threads
WallEParBt	1.0	Direct backtracking	32
FUDGE	3.2.8	SAT (CaDiCaL 1.3.1)	1
μ-toksia	2025.06.06	SAT (CaDiCaL 1.3.1)	1
Scalop	2.0.0	SAT/IPASIR	1
ASPARTIX	2021-2	ASP (clingo 5.6.2)	32

Table 16: Solver versions used in Experiment 4.

not (Table 19). The per-family breakdown of the hard-subset win rates is presented in Table 18.

Comparison	WallEParBt-32T	Opponent	Win-Rate
vs FUDGE (286)	**268**	18	**94%**
vs μ-toksia (290)	**208**	82	**72%**
vs Scalop (290)	**278**	12	**96%**
Instances with $\max(t_{WallEParBt\text{-}32T}, t_{opp}) \geq 1s$			
vs FUDGE (63)	**62**	1	**98%**
vs μ-toksia (61)	**58**	3	**95%**
vs Scalop (52)	**48**	4	**92%**

Table 17: Pairwise win rates on commonly solved instances. On the hard subset the advantage shifts toward WallEParBt-32T.

Interpretation In particular, the crusti family arranges between 31 and 511 small SCCs into a deep chain, which forced the backtracking solver to process each component sequentially and restricted the ability of work-stealing to split work within the current SCC. On the ER, WS, BA, and GML families, WallEParBt-32T reaches comparable or higher coverage than the SAT-based solvers, while on SCC and ST the gap is smaller but still present (Table 19).

Eighteen of the twenty-three solver-specific timeouts are due to the crusti family. We discuss the remaining SCC, WS, and ST cases in Section 6.1.3, where we look at symmetry and large-SCC effects.

However, there are a few examples of random graphs where parallel search makes a crucial difference. An example is `ER_500_40_1`. On this graph, WallEParBt-32T solved the problem in 111.8 s, while none of the three sequential SAT-based solvers finished before the time limit. Notably, the single giant SCC produced a search tree large enough for 32 threads to explore, but too large for a sequential backend to traverse within 600 s.

To investigate the effect of the architecture limits on performance, Table 20 repeats the comparison using only the 302 non-crusti instances. Without crusti, the cover-

Family	n	WallEParBt-32T wins	μ-toksia wins
ER	12	**100%**	0%
scc	10	**80%**	20%
admbuster	6	**100%**	0%
st_*	4	**75%**	25%
WS	4	**100%**	0%
other	25	**100%**	0%
Total ($\geq$1s)	61	**95%**	5%

Table 18: Per-family win rates against μ-toksia on non-trivial instances (runtime $\geq$1 s).

Solver	Solved	Timeouts	Unique solves	PAR-2
Scalop	**311** (97%)	**11**	0	**17 918**
μ-toksia	310 (96%)	12	0	19 509
ASPARTIX-32T	299 (93%)	23	**2**	32 867
FUDGE	305 (95%)	17	0	23 713
WallEParBt-32T	291 (90%)	31	0	37 740

Table 19: Solver coverage on the full benchmark. ASPARTIX runs on clingo with 32 threads; the three SAT-based solvers use a single thread.

age of all five solvers fell into the 95–96 % range. WallEParBt-32T rose from last to second place in the PAR-2 ranking, just 698 seconds slower than Scalop. Among the thirteen timeouts for these instances, three were solved by a SAT-based competitor (`scc_7216`, `scc_9431`, and `st_885`), two more were solved by ASPARTIX only (`WS_400` and `scc_3605`), while the remaining eight timeouts were experienced by all five solvers.

Solver	Solved	Timeouts	PAR-2
Scalop	**291** (96%)	**11**	**15 441**
WallEParBt-32T	289 (96%)	13	16 139
μ-toksia	290 (96%)	12	16 493
FUDGE	287 (95%)	15	19 464
ASPARTIX-32T	286 (95%)	16	22 632

Table 20: Solver coverage on the 302 non-crusti instances. Without the structurally disadvantaged family, WallEParBt-32T rises from last to second place in PAR-2.

Parallelism Strategy Comparison To allow for a direct comparison of the parallelism strategies used in the solvers, we ran WallEParBt-32T with work-stealing and ASPARTIX-32T with portfolio search in clingo with 32 threads each under the same time limit.

Solver	Solved	Timeouts	Unique (pairwise)
WallEParBt-32T	291 (90%)	31	**8**
ASPARTIX-32T	**299** (93%)	23	16

Table 21: Work-stealing vs. portfolio parallelism, both at 32 threads.

ASPARTIX-32T covered 299 instances and WallEParBt-32T covered 291 (see Table 21). However, WallEParBt-32T was unique in solving eight instances (for example, `admbuster` and `Large-result` graphs), where the graph-direct approach completely eliminated the need to encode the graph. It also allowed the labelling of a graph with a million arguments to be propagated in $O(n+m)$ and a solution to be found in less than a second, while clingo's grounding phase timed out. On the 283 instances in which both solvers completed, WallEParBt-32T was faster 93 percent of the time.

Answer to Experiment 4 A summary of results for Experiment 4. WallEParBt-32T solved 20 fewer instances than Scalop (291 vs. 311). However, in the head-to-head comparison on instances that both solvers were able to complete, WallEParBt-32T

performed better than Scalop in 96 percent of those cases (see Table 17). As mentioned earlier, the crusti results point to a limitation of WallEParBt, which does not yet track cross-SCC dependencies. If we exclude the crusti family, the performance difference between WallEParBt-32T and Scalop is reduced to only two instances. Under that view, WallEParBt-32T ranks second in the PAR-2 comparison of all tested solvers (see Table 20).

Regarding the runtime percentiles listed in Table 22, WallEParBt-32T had the lowest median runtime and the lowest 90th percentile runtime. Looking at the cumulative time-threshold curves in Figure 19, all four solvers dispatch a large share of the easy instances in less than 0.1 seconds. Between 10 and 60 seconds, FUDGE and Scalop gain ground because their clause-learning backends are able to solve moderately difficult instances that the backtracker cannot solve.

Figure 20 shows the coverage breakdown by graph family, while the amounts of speedup measured for each family on the hard instances can be found in Table 23.

Summary of Findings As noted previously, the findings presented here empirically support each of the three hypotheses.

Solver	Median	P90	P95	P99	Stdev
WallEParBt-32T	**0.02s**	**1.05s**	**3.64s**	56.24s	**10.02s**
FUDGE	0.06s	15.21s	47.27s	**310.42s**	44.60s
μ-toksia	0.09s	17.13s	71.87s	388.86s	69.14s
Scalop	0.10s	16.97s	63.33s	341.91s	63.44s

Table 22: Runtime distribution on solved instances. WallEParBt-32T has the lowest median and P90.

Table 23 shows the parallelisation speedup on hard instances per family.

5.5 Summary of Results

Overall, the four experiments provided support for all three hypotheses. Two structural features recur across all experiments, namely how much independent search work the graph provides for work-stealing, and whether the preprocessing method matches the SCC structure. For example, an ER graph at low density, and a WS graph at high rewiring probability, will likely produce a single giant SCC, and therefore will have similar speedup values, though they will come from different generators. It is noticeable that on the instances where both methods complete, the backtracker is faster (Experiment 4, Table 17). The SAT-based competitors, however, solve more instances overall since clause learning can handle the deep SCC chains that prevent the backtracker from making progress. Table 23 shows how much speedup was measured for each family on the hard instances. The next section explains where the differences come from.

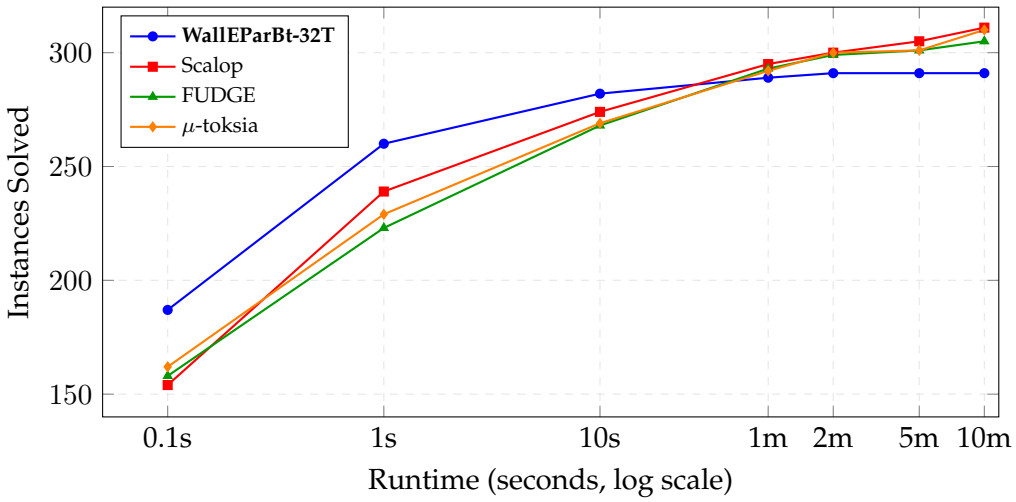


Figure 19: WallEParBt-32T leads below 10 s; SAT-based solvers overtake around 60 s.

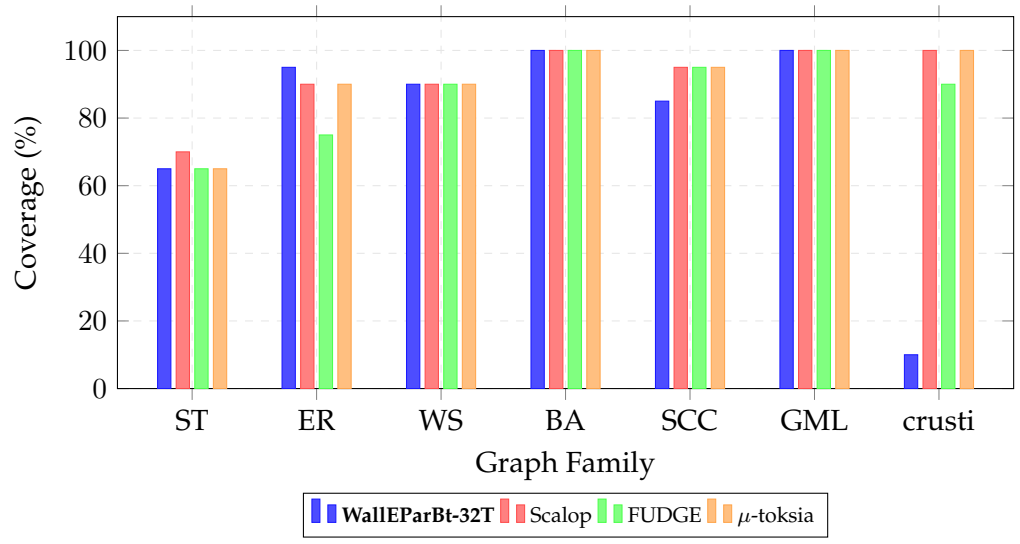


Figure 20: Coverage by graph family (322 instances, 600 s timeout). The crusti family exposes the main weakness: WallEParBt solves only 2 of 20, the SAT-based solvers 18–20.

Table 23: Parallelisation speedup on hard instances (1T $\geq$ 1 s). One ST outlier (1113×, superlinear due to early termination) is excluded.

Family	Median speedup	Range	Rescued	Coverage
ST	22.6×	20.6–28.1×	+1	13/20
WS	21.7×	9.1–23.7×	+1	18/20
ER	18.4×	5.0–20.4×	+2	19/20
SCC	1.0×	1.0–4.3×	+1	17/20
crusti	—	—	0	2/20

6 Discussion

6.1 Structural Interpretation

In order to understand how the solver performs over the six non-trivial graph families illustrated in Table 24, we need to look at three structural properties of the graph: the size of the largest strongly connected component (SCC), the ratio of edges to nodes (edge density m/n), and the depth of the SCC hierarchy.

Family	Inst.	Med. #SCCs	Med. max SCC	Med. m/n	Mode	Scaling
ER	20	1	351	83.0	backtracking (15/20)	18–25×
WS	20	1	300	8.0	backtracking (15/20)	18–25×
ST	20	28	470	9.2	backtracking (20/20)	18–25×
BA	20	61	35	1.6	grounded propagation	no need
crusti	20	255	225	87.8	timeout (18/20)	flat
SCC	20	18	286	193	preprocessing (15/20)	no need

Table 24: Structural profile of the six non-trivial families (all medians).

From the scaling curves shown in Figure 17, ST, ER, and WS clearly scale with the number of threads, whereas crusti does not scale at all. Broad search trees allow threads to steal subtrees from each other. Conversely, deep search trees cause the threads to wait at each SCC boundary until the current component is solved. Figure 21 illustrates these two extremes.

6.1.1 Parallelism

In Figure 22, the runtime and speedup for all 322 ICCMA 2025 instances are illustrated. Nearly all of the data points fall on or below the diagonal, which means that 32 threads are almost never slower than one. Fewer than ten instances had any sort of slowdown. Even the worst case produced a parallel runtime that was at most 1.5× longer than the sequential runtime. Part (a) of the figure plots 1T against 32T runtime. Part (b) plots search difficulty against speedup, where red crosses mark

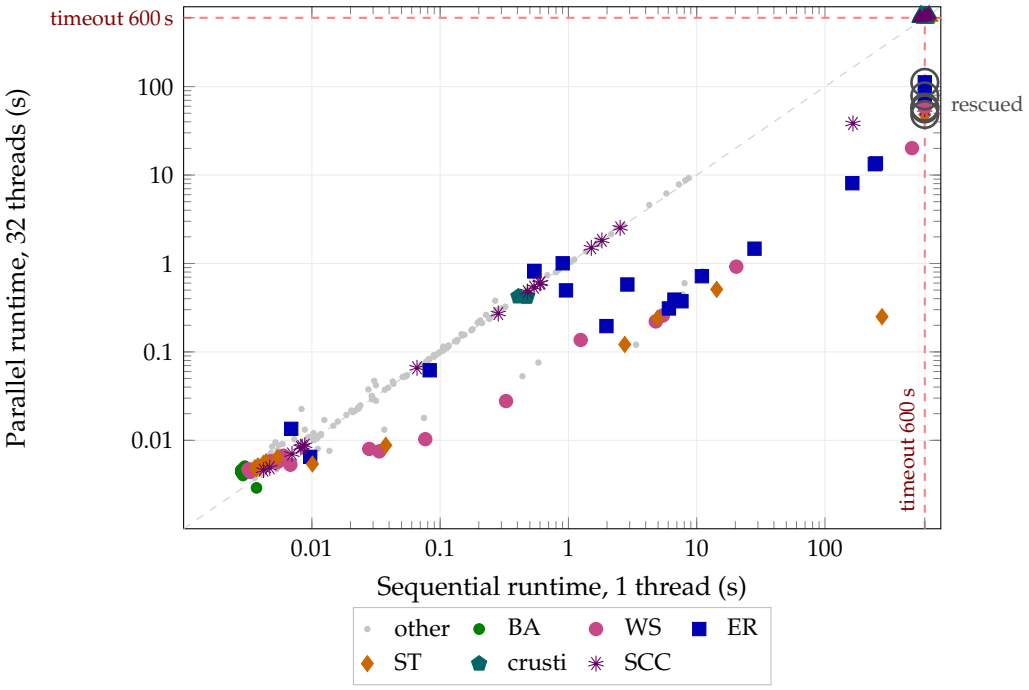
Figure 21: Single large SCC (left) vs. chain of small SCCs (right).

instances that timed out under both configurations.

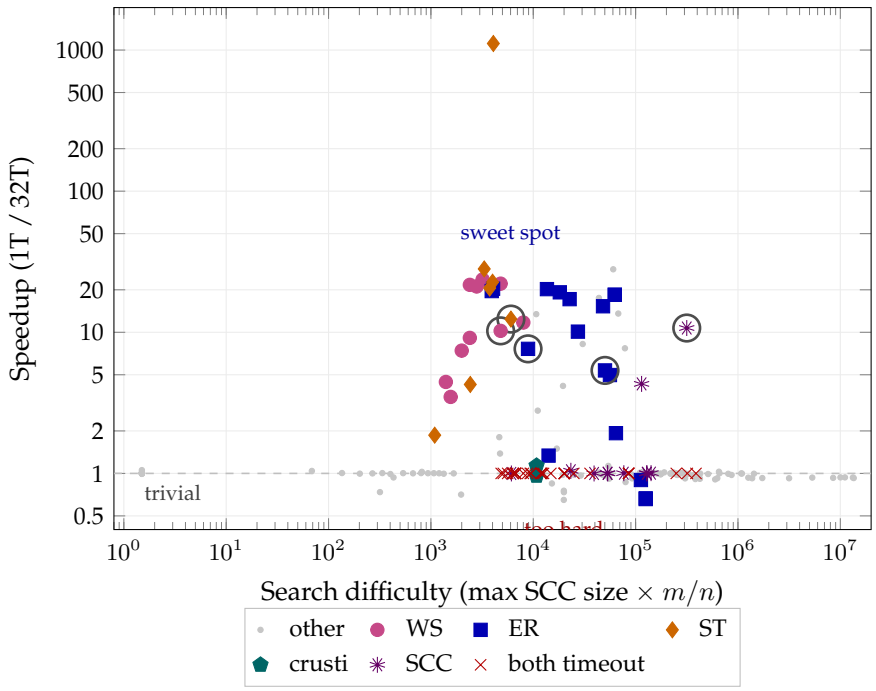
WallEParBt keeps the overhead low for two reasons. First, threads are started lazily. A thread is started only after the tree is large enough to split. Second, atomic operations are used to maintain a shared labelling array. Atomic operations avoid the use of locks and thus avoid the overhead of lock contention.

How much speedup we get depends mostly on the SCC structure of each graph family. The majority of instances on the left-hand side of the plot were relatively simple. Adding more threads has no real effect here, because grounded propagation resolves them without any backtracking. At the opposite end of the spectrum, both 1T and 32T timed out. Between the two extremes of the plot lies a region where WS, ER, and ST instances achieve speedups of 5–28×. These families all have one large connected component. The backtracker builds a large search tree within the component. The larger the component, the wider the search tree and the more subtrees are available for the 32 threads to steal from each other. As long as there are enough subtrees at the upper levels of the tree, all 32 threads remain active. The five rescued instances (circled in the plots) are at the boundary of the speedup zone. They are difficult enough to cause timeouts when run sequentially, but are sufficiently easy to allow the 32-threaded version to complete in time. The ST family exhibits similar behaviour when the largest SCC contains fewer than about 530 arguments. The single instance that was an exception is `st_395`. It achieved a speedup of 1 113×.

Weaknesses On the right-hand side of Figure 22b, we can see the structural limit. All 18 crusti timeouts and all 7 ST both-timeouts fall into the hard zone, which is not surprising. Both families contain deep SCC chains: crusti has up to 511 SCCs at an internal density of $m/n \approx 88$, while the harder ST instances have chains of 7–52 SCCs with individual SCC sizes greater than 500. Because WallEParBt processes SCCs in topological order, a predecessor SCC generally needs to be resolved before the parent SCC can proceed, and work-stealing cannot distribute work across SCC boundaries. With only one SCC being parallelised at a time, most of the threads will be idle whenever the current SCC is small compared to the chain length. Both BA and crusti have plenty of SCCs, but the two families end up on opposite ends of the performance spectrum. Due to an average density of $m/n = 1.6$ in BA, the



(a) 1T vs. 32T runtime (all 322 instances). Below the diagonal = faster with 32T. Circles = rescued (1T timeout, 32T solved).



(b) Search difficulty vs. speedup (169 instances with 1T > 0.01 s). Red crosses = both timeout. Circles = rescued.

Figure 22: Runtime and speedup across all 322 ICCMA 2025 instances.

grounded propagation process will resolve each SCC without the need for a backtrack. However, since crusti has a relatively high internal density of $m/n \approx 88$, the backtracker will have to perform a search at every level of the hierarchy. We regard this as the main limitation of the current design.

Role of graph structure As shown by the grey dots in Figure 22b, high search difficulty is clearly not enough to result in a timeout. Some instances in GML and Large-result have difficulty values greater than 100 000 yet were solved in under a second. In these cases, either grounded propagation or the symmetry-aware branching bonus (Section 6.1.3) cut the search tree down to a manageable size. We can see this clearly with `transit-authority` (n=3 324, 66% bidirectional edges): no matter how many threads were employed to solve the instance, it was solved in about 0.13 s. Whether an instance can be solved depends on three factors: the internal density of its SCCs, the depth of the SCC hierarchy, and the symmetry of the graph. No single number captures all three at once. Even two instances with the exact same difficulty value can end up with very different speedups, depending on how much of the tree propagation is able to cut away.

6.1.2 Deep SCC Structures and the Cross-Component Backtracking Gap

As is evident from Figure 23, for the crusti-like structures, the primary problem with which WallEParBt struggles is the SCC structure of the graph. The hierarchy of SCCs for these structures is extremely deep, which severely limits the ability of the work-stealing mechanism to divide the workload among the threads.

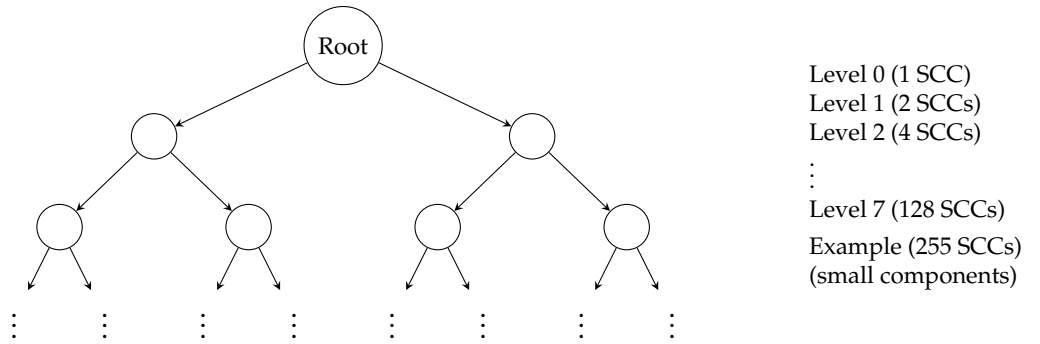


Figure 23: Schematic binary-tree-like SCC hierarchy for crusti. Each node represents an SCC. Arrows indicate attack direction (leaves attack toward root).

SCCs are processed in topological order, leaf SCCs first. A decision in a leaf SCC constrains its parent and all SCCs above it in the hierarchy. If that decision turns out to be wrong, some SCC higher up will eventually fail. The solver, however, will not notice until propagation actually fails at that higher SCC. It then backtracks to the leaf and tries a different decision. The current version of WallEParBt has no way

of tracking which prior SCC decisions led to the failure. The solver therefore keeps revisiting the same failed assignments. Adding more threads to the system simply adds more workers arriving at the same dead end. A SAT-based solver handles this differently. It records each failure as a learned clause and never revisits that decision. In the worst case, an error at the bottom of a hierarchy with up to 511 SCCs causes hundreds of components above it to fail one after another. Without cross-component dependency tracking, this can waste hours of search time. WallEParBt does not implement cross-SCC backjumping because it works directly on the argumentation graph and has no clause representation. Adapting such a technique to a parallel, non-SAT backtracker is beyond the scope of this thesis, but we consider it a promising direction for future work.

6.1.3 Symmetric Attacks and the Branching-Factor Explosion

Apart from the SCC hierarchy issue described above, symmetric attacks between pairs of vertices also cause problems. For every stable extension S and for any such pair of vertices (a, b), there are only three choices: $a \in S$ and $b \notin S$; $b \in S$ and $a \notin S$; or neither $a \in S$ nor $b \in S$. As long as the only constraints on a and b are the two attacks between them, the backtracking algorithm will need to try at least two branches to find out which branch leads to a consistent assignment. If a graph has s structurally independent symmetric pairs of vertices, then the search tree has at least 2^s leaves.

An extreme case of this is the GML instance `transit-authority`, where 66% of the edges are bidirectional and produce up to $s = 1\,662$ vertex-disjoint symmetric pairs, resulting in at least

$$2^{1662} \approx 10^{500}$$

leaves in the search tree. Since there are 32 threads, each thread will also have to check around 10^{499} leaves, which is many orders of magnitude over any reasonable timeout. The search tree for an SAT-based solver does not explode like this. This is because the CNF encoding represents each symmetric pair as a single clause. Determining the assignment of one argument then causes the assignment of the other by unit propagation. For instance, Scalop completes `transit-authority` in less than one second. The backtracking algorithm would experience the full 2^s explosion without the symmetry-aware heuristic described below.

The two families ER and WS have a large strongly connected component, however the branching factors are not exponential like in crusti. At a density of $p = 0.05$ in an ER graph, the directed edges happen independently, therefore the probability that two nodes attack each other is very low ($p^2 = 0.0025$), and as such the search trees will be wide and will not double their branching factors. Wide search trees are exactly what work-stealing needs to be effective.

There are a few instances of the SCC family where this behaviour also occurs but at slightly lower symmetry ratios (20–30%). At least one of the baseline competitors can solve each of these instances while the backtracker runs out of time, which

confirms that it is the structure of the problem, not its size, that determines which approach performs better.

To counter this exponential increase, we built a partner-lookup table during graph construction in $O(m)$. Candidates with an undecided partner in a mutual attack receive a bonus of 20 points when the solver chooses the next variable. This bonus is usually sufficient to move the candidate with the undecided partner to the top score in Stage 3. We always start by evaluating the lower-indexed argument as `IN`, which establishes a canonical order for the pair and prevents the conflict-freeness violation that would occur if the partner were evaluated first. Sabharwal [Sab09] and Coste-Marquis et al. [CMDM05] show that symmetric branches can shrink the search tree exponentially. In fully symmetric argumentation frameworks, the problem even becomes tractable. Our heuristic makes use of this observation and applies it directly in the backtracking solver. `transit-authority` was completed in 0.13 s due to this heuristic, but the three SCC-family instances with lower symmetry ratios (20–30%) still timed out because the number of pairs was too low to offset the remaining search effort.

6.1.4 Summary of Timeout Causes

As already discussed in the preceding sections, the solver struggles with three types of graph structure. All 31 timeout conditions fall into one of these three structural categories: the crusti family (Section 6.1.2), symmetric attacks (Section 6.1.3), and large strongly connected components. For each of the 23 solver-specific timeouts, at least one baseline competitor was able to solve the instance. We interpret this as evidence that clause learning or ASP grounding handle these structural features better than direct backtracking does. The weights of SELECTVARIABLE and the adaptive thresholds listed in Table 2 were optimised on the ICCMA 2023 suite during development. The ICCMA 2025 suite only became available late in the project, and testing on this new benchmark showed that the solver produced competitive results without any changes to the weights or thresholds, including on the two new sembuster and transit families. We see this as a positive indication regarding the generalisability of our approach.

6.2 Comparison with other Solvers

One important question is how WallEParBt compares to solvers that use a different approach entirely. The ICCMA results indicate that there is no single best performing solver for all families of graphs. Scalop and FUDGE demonstrate much better performance than WallEParBt on the crusti family since they use clause learning to save failed assignments when traversing the boundaries of an SCC. In contrast, WallEParBt demonstrates generally better performance than Scalop and FUDGE on the ST instances that they also solve, since the work-stealing strategy keeps all 32 threads active during the entire width of the search tree generated by the ST family.

Table 17 shows the head-to-head comparison. For each pair of solvers, we only counted instances that both solvers completed. WallEParBt outperformed Scalop 96% of the time and FUDGE 94% of the time. Similarly, when comparing WallEParBt with μ-toksia, we found that WallEParBt outperformed μ-toksia 72% of the time across all problem instances, and 95% of the time on the more difficult problem instances. From these numbers, we believe that SAT-based solving and graph-based backtracking are failing due to differing structural properties of the problem, rather than one method being better suited to solve the problem than the other. If we take the crusti family out of the picture, the situation looks quite different. The combined solution rate for all five solvers then falls into the range of 95–96%, and WallEParBt had the second-lowest PAR-2 score (see Table 20).

Even though 32 cores are available on AWS, Scalop still achieves greater total coverage than WallEParBt by 20 instances (291 vs. 311), which implies that brute-force parallelisation is insufficient to overcome the lack of cross-SCC dependency tracking. At the same time, WallEParBt requires all 32 cores to reach this coverage level, since it solves only 286 instances with a single thread.

As noted in Section 3.4, current parallel solvers use either a SAT backend or an ASP backend and do not parallelise the argumentation graph. Our results indicate that work-stealing (graph-based parallelisation) and clause learning (SAT-based solving) are complementary rather than competing. A portfolio of both approaches would likely cover the greatest number of graph structures. When only a single solver is deployed, which method is faster will depend on the SCC structure of the input graph, and the AUTO mode from H3 represents a reasonable first approximation for choosing how to proceed without user configuration.

6.3 Threats to Validity

The primary experiment was run once per platform, which is the standard practice in the ICCMA competition. To make sure our results are reliable, we ran a repeated-run check on 290 of the 291 solved instances (Table 7). The runtimes turned out to be stable across the board, with the exception of the crusti family where runtimes vary quite a bit between runs.

Nearly all instances in the ICCMA suite are produced synthetically via a generator. The GML family is the only exception here, as it comes from real transit networks. Consequently, the behaviour of actual argumentation graphs could differ from what was seen in the synthetic instances investigated in this thesis.

Our experiments were focused on SE-ST. The decision variants DC-ST and DS-ST may react differently to parallelisation and preprocessing. We did, however, run DC-ST and DS-ST on the Apple M1 Pro and observed the same structural patterns as for SE-ST. In other words, the family-level effects appear to carry over, although the M1 Pro runtimes are not directly comparable to those reported for AWS.

One of the pruning rules uses a bitset. The problem is that this bitset will need to grow its memory by quadratic order as the number of nodes increases. Therefore,

when there are more than 100 000 nodes in an instance, the solver does not use this rule at all. Raw data and solver outputs are available in the `benchmarks/` directory of the repository.

7 Conclusion

In this bachelor thesis we investigated whether parallelising the backtracking search on argumentation graphs could provide real benefits in terms of performance when more CPU cores are added. To answer this question, we built the solver WallEParBt, which combines SCC decomposition, a backtracking engine with label propagation, and a work-stealing scheduler that distributes subtrees across threads. The SCC decomposition turned out to be important, as it gave us structural information that helped organise the search and expose parallel work. By comparison of the six thread configurations for all 322 ICCMA 2025 instances, we could confirm that the benefit of parallelisation depends upon the size and the structure of the SCCs in the input graph. We then observed the strongest speedup on the ER, WS, and ST families, all of which have one or a few very large SCCs. Each such component gives the work-stealing scheduler enough subtrees to keep all threads busy. For these families, WallEParBt produced a median speedup of approximately 18–23$\times$ when run with 32 threads. Here, the measured values actually exceed the Amdahl prediction at p=95%. Some threads find the solution early and the search terminates before all branches are explored (H1). As for the source-SCC precheck, we found that it was beneficial for more families than it harmed. If the graph structure did not benefit from the precheck, it could determine this from the SCC counts and densities generated by Tarjan's pass and skip it (H2). Finally we considered the adaptive mode, which picked the right configuration for almost all graphs. It missed only two instances when compared against an oracle that always chose the best option (H3). We consider this a reasonable result.

In all, WallEParBt solved 291 of the 322 instances of the ICCMA 2025 problem set using 32 threads. Among the five solvers tested, this is the lowest coverage and the highest PAR-2 score. The raw numbers, however, do not tell the whole story. 18 of the 31 timeouts occurred on the crusti family, where the deep SCC hierarchies and the lack of cross-SCC dependency tracking limited the backtracker (Section 6.1.2). Moreover, if we remove the crusti family from the comparison, WallEParBt covered 96% of the remaining 302 instances. It came in second in the PAR-2 ranking, only 698 seconds behind Scalop (see Table 20). On instances that both solvers finished, WallEParBt was faster than Scalop 96% of the time and faster than FUDGE 94% of the time (Table 17). From these numbers, we believe that the coverage gap is structural rather than an issue with the solver design. Given that the crusti family is the main source of timeouts, this gap could be closed by addressing the cross-SCC tracking problem.

One idea for future work would be to track dependencies across SCC boundaries. With such a mechanism in place, the backtracker could skip back to the SCC where

the bad decision was made. A portfolio approach might also be worth examining. Wide-SCC graphs would go to our backtracker, while deep hierarchies would go to a SAT or ASP solver.

Looking back at what we have achieved, we are confident that parallel backtracking directly on the argumentation graph has earned its place alongside SAT- and ASP-based solvers for the task of finding stable extensions.

References

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18–20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485. ACM, 1967.

[BA99] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[BFFH20] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximilian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 50–53. University of Helsinki, 2020. Online, accessed on February 8, 2026: `https://helda.helsinki.fi/server/api/core/bitstreams/1c2f0021-5b65-42fa-9c65-4d03ffcdb51d/content`.

[BG07] Pietro Baroni and Massimiliano Giacomin. On principle-based evaluation of extension-based argumentation semantics. *Artificial Intelligence*, 171(10-15):675–700, 2007.

[BL99] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[Bre74] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[Cam06] Martin Caminada. On the issue of reinstatement in argumentation. In *Logics in Artificial Intelligence (JELIA 2006)*, volume 4160 of *Lecture Notes in Computer Science*, pages 111–123. Springer, 2006.

[CMDM05] Sylvie Coste-Marquis, Caroline Devred, and Pierre Marquis. Symmetric argumentation frameworks. In *Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU 2005)*, volume 3571 of *Lecture Notes in Computer Science*, pages 317–328. Springer, 2005.

[CTV+15] Federico Cerutti, Ilias Tachmazidis, Mauro Vallati, Sotirios Batsakis, Massimiliano Giacomin, and Grigoris Antoniou. Exploiting parallelism for hard problems in abstract argumentation. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015), Austin, Texas, USA, January 25–30, 2015*, pages 1475–1481. AAAI Press, 2015.

[DD18] Wolfgang Dvořák and Paul E. Dunne. Computational problems in formal argumentation and their complexity. In Pietro Baroni, Dov Gabbay,

Massimiliano Giacomin, and Leendert van der Torre, editors, *Handbook of Formal Argumentation*, pages 631–688. College Publications, 2018.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DT96] Yannis Dimopoulos and Alberto Torres. Graph theoretical structures in logic programs and default theories. *Theoretical Computer Science*, 170(1-2):209–244, 1996.

[Dun95] Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77(2):321–357, 1995.

[EGW08] Uwe Egly, Sarah Alice Gaggl, and Stefan Woltran. ASPARTIX: Implementing argumentation frameworks using answer-set programming. In *Logic Programming*, volume 5366 of *Lecture Notes in Computer Science*, pages 734–738. Springer, 2008.

[ER59] Paul Erdős and Alfréd Rényi. On random graphs I. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

[GKKS14] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + control: Preliminary report. In *Technical Communications of the 30th International Conference on Logic Programming (ICLP 2014)*, 2014.

[GT18] Nils Geilen and Matthias Thimm. Heureka: A general heuristic backtracking solver for abstract argumentation. In *Theory and Applications of Formal Argumentation (TAFA 2017)*, volume 10757 of *Lecture Notes in Computer Science*, pages 143–149. Springer, 2018.

[HE80] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.

[HJS09] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. ManySAT: A parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:245–262, 2009.

[HKT25] Sandra Hoffmann, Isabelle Kuhlmann, and Matthias Thimm. SMART V1.0. In Iosif Apostolakis, Andrei Popescu, and Johannes P. Wallner, editors, *Solver and Benchmark Descriptions of ICCMA 2025: Sixth International Competition on Computational Models of Argumentation*, 2025. Competition solver description.

[HKWB12] Marijn J.H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Hardware and Software: Verification and Testing (HVC 2011)*, volume 7261 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2012.

[LLM25] Jean-Marie Lagniez, Emmanuel Lonca, and Jean-Guy Mailly. Scalop at iccma'25, 2025. Solver description from the ICCMA 2025 solver archive. Online, accessed on February 8, 2026: `https://www.argumentationcompetition.org/2025/solvers/iccma_25_final_submission.tar.gz`.

[Mar24] Daniel Marjamäki. Cppcheck: A tool for static C/C++ code analysis. `https://cppcheck.sourceforge.io/`, 2024. Online, accessed on February 8, 2026.

[NAD14] Samer Nofal, Katie Atkinson, and Paul E. Dunne. Algorithms for decision problems in argument systems under preferred semantics. *Artificial Intelligence*, 207:23–51, 2014.

[NAD16] Samer Nofal, Katie Atkinson, and Paul E. Dunne. Looking-ahead in backtracking algorithms for abstract argumentation. *International Journal of Approximate Reasoning*, 78:265–282, 2016.

[NJ20] Andreas Niskanen and Matti Järvisalo. μ-toksia: An efficient abstract argumentation reasoner. In *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning (KR 2020)*, pages 800–804, 2020. Winner of all 21 ICCMA 2019 Main Track categories.

[Org23] ICCMA 2023 Organizers. ICCMA 2023 rules. `https://argumentationcompetition.org/2023/rules.html`, 2023. Online, accessed on February 8, 2026.

[Org25] ICCMA 2025 Organizers. ICCMA 2025 rules. `https://www.argumentationcompetition.org/2025/rules.html`, 2025. Online, accessed on February 26, 2026.

[Ref04] Philippe Refalo. Impact-based search strategies for constraint programming. In *Principles and Practice of Constraint Programming – CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 557–571. Springer, 2004.

[Sab09] Ashish Sabharwal. SymChaff: Exploiting symmetry in a structure-aware satisfiability solver. *Constraints*, 14:478–505, 2009.

[SI09] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA 2009)*, pages 62–71. ACM, 2009.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[TCV23] Matthias Thimm, Federico Cerutti, and Mauro Vallati. FUDGE: A lightweight solver for abstract argumentation based on SAT reductions. In *Proceedings of the 5th International Competition on Computational Models of Argumentation (ICCMA 2023)*, 2023. Competition solver description. Online, accessed on February 8, 2026: `https://iccma2023.github.io/`. See also arXiv:2109.03106.

[Thi19] Matthias Thimm. Dredd – a heuristics-guided backtracking solver with information propagation for abstract argumentation. In *System Descriptions of the 3rd International Competition on Computational Models of Argumentation (ICCMA 2019)*, 2019. Competition solver description.

[WS98] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.