Computing Extensions of Abstract Argumentation Frameworks by Enumerating Closed Sets

Sergei Obiedkov[1,2] Barış Sertkaya[3]

KR 2026

[1]Knowledge-Based Systems Group, TU Dresden [2]ScaDS.AI Dresden/Leipzig
[3]Frankfurt University of Applied Sciences

Closing the Argument

Abstract argumentation models arguments and conflicts as a directed graph.

An extension is a set of arguments that can jointly be accepted.

For an AF $\mathcal{F} = (A, R)$, an argument $a \in A$, and a semantics σ:

DC-σ (credulous acceptance): is a in *some* σ-extension?
SE-σ: find *one* σ-extension (if any) EE-σ: enumerate *all* σ-extensions

Intractable for most interesting semantics [Dunne, Wooldridge 2009; Kröll et al. 2017]

Closing the Argument

Abstract argumentation models arguments and conflicts as a directed graph.

An extension is a set of arguments that can jointly be accepted.

For an AF $\mathcal{F} = (A, R)$, an argument $a \in A$, and a semantics σ:

DC-σ (credulous acceptance): is a in *some* σ-extension?
SE-σ: find *one* σ-extension (if any) EE-σ: enumerate *all* σ-extensions

Intractable for most interesting semantics [Dunne, Wooldridge 2009; Kröll et al. 2017]

Established approach

Reduce reasoning tasks to, e.g., SAT or CSP and use back-end solvers.

Closing the Argument

Abstract argumentation models arguments and conflicts as a directed graph.

An extension is a set of arguments that can jointly be accepted.

For an AF $\mathcal{F} = (A, R)$, an argument $a \in A$, and a semantics σ:

DC-σ (credulous acceptance): is a in *some* σ-extension?
SE-σ: find *one* σ-extension (if any) EE-σ: enumerate *all* σ-extensions

Intractable for most interesting semantics [Dunne, Wooldridge 2009; Kröll et al. 2017]

Established approach

Reduce reasoning tasks to, e.g., SAT or CSP and use back-end solvers.

This talk: compute extensions directly

- Main idea: extensions live inside a closure system.
- Enumerate closed sets until a desired extension is found.

Contributions

1. Introduce closure systems of semi-complete, quasi-complete, pseudo-complete, and P-pseudo-complete sets.

Contributions

1. Introduce closure systems of semi-complete, quasi-complete, pseudo-complete, and P-pseudo-complete sets.
2. Show that complete, preferred, and stable extensions are contained inside these closure systems.

Contributions

1. Introduce closure systems of semi-complete, quasi-complete, pseudo-complete, and P-pseudo-complete sets.
2. Show that complete, preferred, and stable extensions are contained inside these closure systems.
3. Design algorithms for efficiently exploring closure systems to obtain a required extension.

Contributions

1. Introduce closure systems of semi-complete, quasi-complete, pseudo-complete, and P-pseudo-complete sets.
2. Show that complete, preferred, and stable extensions are contained inside these closure systems.
3. Design algorithms for efficiently exploring closure systems to obtain a required extension.
4. Exploit the component structure of the argumentation framework to improve the performance (for preferred semantics).

Closure Systems

Definition

$\mathcal{C} \subseteq 2^A$ is a closure system if $A \in \mathcal{C}$ and $\mathcal{C}$ is closed under intersection.

It induces a closure operator: $S \mapsto$ smallest closed superset of S.

Closure Systems

Definition

$\mathcal{C} \subseteq 2^A$ is a closure system if $A \in \mathcal{C}$ and $\mathcal{C}$ is closed under intersection.
It induces a closure operator: $S \mapsto$ smallest closed superset of S.

- Central in formal concept analysis (FCA) and frequent itemset mining
- Well-studied algorithms (e.g., for enumerating concepts of a formal context in FCA)

Closure Systems

Definition

$\mathcal{C} \subseteq 2^A$ is a closure system if $A \in \mathcal{C}$ and $\mathcal{C}$ is closed under intersection.
It induces a closure operator: $S \mapsto$ smallest closed superset of S.

- Central in formal concept analysis (FCA) and frequent itemset mining
- Well-studied algorithms (e.g., for enumerating concepts of a formal context in FCA)

Closure systems meet argumentation

- AF $\rightarrow$ context; stable extensions are among the concept intents [Obiedkov, Sertkaya 2023]
- AF $\rightarrow$ implications; closed sets $=$ complements of self-defending sets [Elaroussi et al. 2023]

Closure Systems

Definition

$\mathcal{C} \subseteq 2^A$ is a closure system if $A \in \mathcal{C}$ and $\mathcal{C}$ is closed under intersection.
It induces a closure operator: $S \mapsto$ smallest closed superset of S.

- Central in formal concept analysis (FCA) and frequent itemset mining
- Well-studied algorithms (e.g., for enumerating concepts of a formal context in FCA)

Closure systems meet argumentation

- AF $\rightarrow$ context; stable extensions are among the concept intents [Obiedkov, Sertkaya 2023]
- AF $\rightarrow$ implications; closed sets $=$ complements of self-defending sets [Elaroussi et al. 2023]

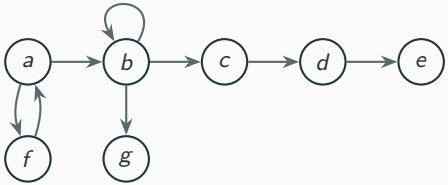
Here: a closure system that contains all complete (and, thus, preferred and stable) extensions.

Abstract Argumentation Frameworks [Dung 1995]

- An argumentation framework (AF) is a directed graph $\mathcal{F} = (A, R)$:
  - A is a finite set of *arguments*
  - $R \subseteq A \times A$ — the *attack relation*

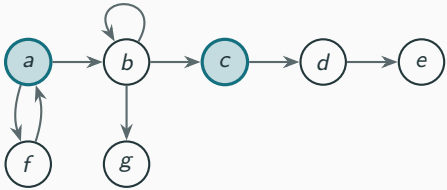
- $R(S)$ — arguments attacked by $S \subseteq A$;
  $R^{-1}(S)$ — arguments attacking $S \subseteq A$

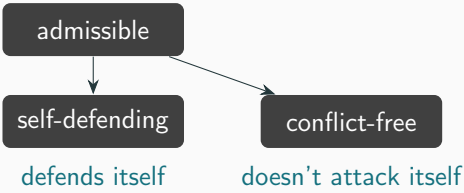
- S defends x if S attacks every its attacker:
  $$R^{-1}(x) \subseteq R(S)$$



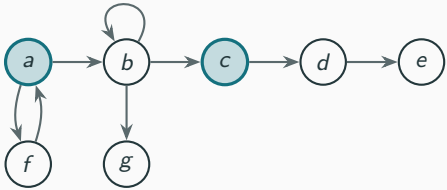
a attacks b and f;
$\{a, e\}$ defends c and g
(it attacks their only attacker, b)

Admissible Sets





Admissible Sets and Extensions

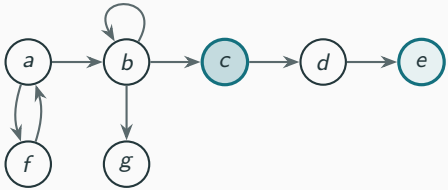


Semi-complete Sets

attacks everything it doesn't contain

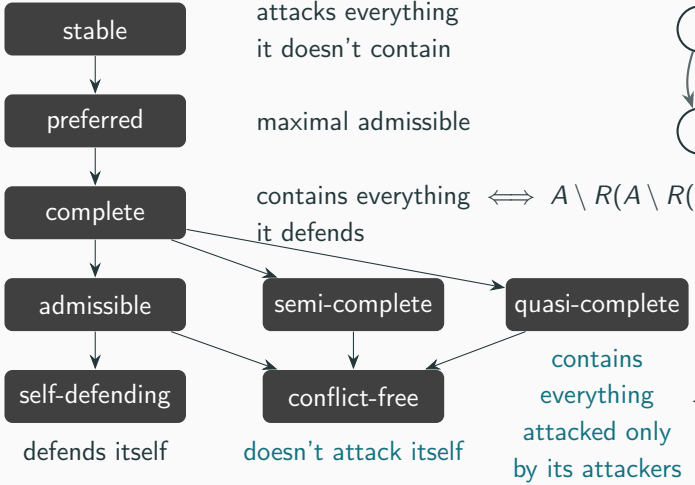
maximal admissible

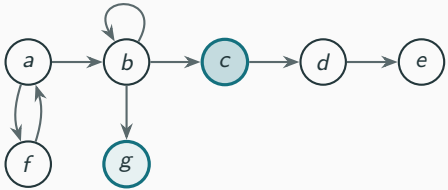
contains everything it defends $\iff A \setminus R(A \setminus R(S)) \subseteq S$



Semi-complete subsets of A, together with A, form a closure system.

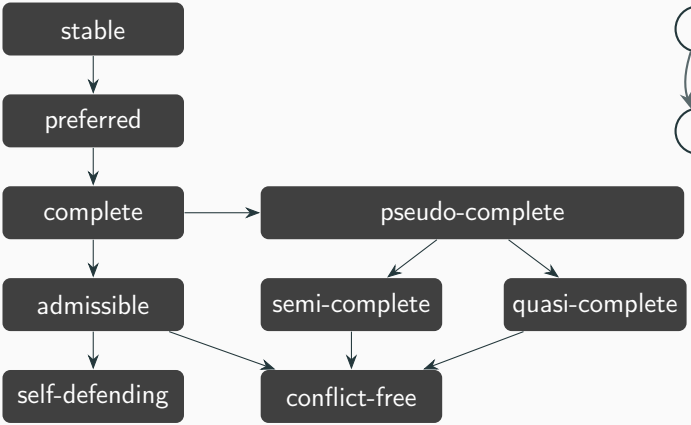
Quasi-complete Sets

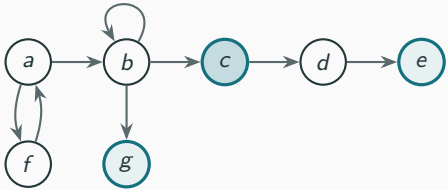




Quasi-complete subsets of A, together with A, form a closure system.

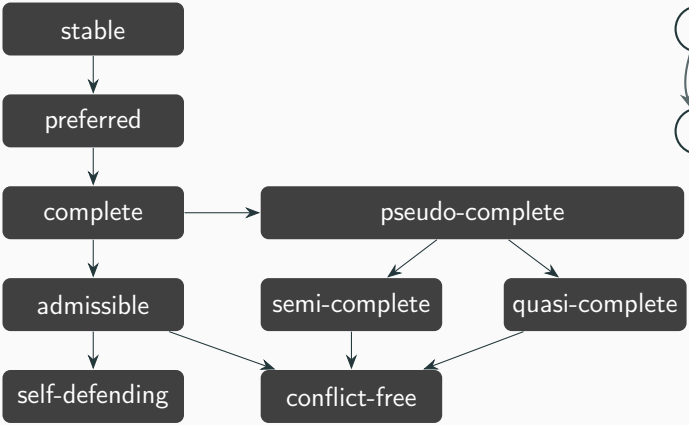
Pseudo-complete Sets

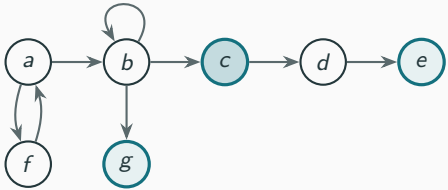




Pseudo-complete subsets of A, together with A, form a closure system.

Pseudo-complete Sets





Pseudo-complete subsets of A, together with A, form a closure system.

Plan: enumerate pseudo-complete sets until the required extension shows up.

The Closure Operator

`pseudo-complete`$(\mathcal{F}, S, P)$: expands S applying two rules until a fixpoint:

- if some a is defended by S, add it; $R^{-1}(a) \subseteq R(S)$
- if some a is attacked only by attackers of S, add it; $R^{-1}(a) \subseteq R^{-1}(S)$

returns $\perp$ if this forces a conflict or hits a *prohibited* argument in P.

The Closure Operator

`pseudo-complete`$(\mathcal{F}, S, P)$: expands S applying two rules until a fixpoint:

- if some a is defended by S, add it; $R^{-1}(a) \subseteq R(S)$
- if some a is attacked only by attackers of S, add it; $R^{-1}(a) \subseteq R^{-1}(S)$

returns $\perp$ if this forces a conflict or hits a *prohibited* argument in P.

To find a complete extension containing c: DC-σ

$$S_0 := \texttt{pseudo-complete}(\mathcal{F}, \{c\}, \emptyset)$$

then enumerate pseudo-complete *supersets* of S_0 until one is self-defending (σ = complete),
or also attacks everything outside (σ = stable),
or until all are enumerated (σ = preferred).

Enumeration with Pruning

σ-extension$(\mathcal{F}, S, P)$

1. if $S \neq \emptyset$ is a σ-extension **return** S

Enumeration with Pruning

σ-extension$(\mathcal{F}, S, P)$

1. if $S \neq \emptyset$ is a σ-extension **return** S
2. build a *(hopefully, small)* candidate set $C \subseteq A \setminus (R(S) \cup R^{-1}(S) \cup P)$

Enumeration with Pruning

σ-extension$(\mathcal{F}, S, P)$

1. if $S \neq \emptyset$ is a σ-extension **return** S
2. build a *(hopefully, small)* candidate set $C \subseteq A \setminus (R(S) \cup R^{-1}(S) \cup P)$
3. **for** each $d \in C$:

Enumeration with Pruning

σ-extension$(\mathcal{F}, S, P)$

1. if $S \neq \emptyset$ is a σ-extension **return** S
2. build a *(hopefully, small)* candidate set $C \subseteq A \setminus (R(S) \cup R^{-1}(S) \cup P)$
3. **for** each $d \in C$:
   - $T := \texttt{pseudo-complete}(\mathcal{F}, S \cup \{d\}, P)$

Enumeration with Pruning

σ-extension$(\mathcal{F}, S, P)$

1. if $S \neq \emptyset$ is a σ-extension **return** S
2. build a *(hopefully, small)* candidate set $C \subseteq A \setminus (R(S) \cup R^{-1}(S) \cup P)$
3. **for** each $d \in C$:
   - $T := \texttt{pseudo-complete}(\mathcal{F}, S \cup \{d\}, P)$
   - if $T \neq \bot$: recurse; return result if it succeeds

Enumeration with Pruning

σ-extension$(\mathcal{F}, S, P)$

1. if $S \neq \emptyset$ is a σ-extension **return** S
2. build a *(hopefully, small)* candidate set $C \subseteq A \setminus (R(S) \cup R^{-1}(S) \cup P)$
3. **for** each $d \in C$:
   - $T := \texttt{pseudo-complete}(\mathcal{F}, S \cup \{d\}, P)$
   - if $T \neq \bot$: recurse; return result if it succeeds
   - else add d to P *(prune dead-end branches)*

Enumeration with Pruning

σ-extension$(\mathcal{F}, S, P)$

1. if $S \neq \emptyset$ is a σ-extension **return** S
2. build a *(hopefully, small)* candidate set $C \subseteq A \setminus (R(S) \cup R^{-1}(S) \cup P)$
3. **for** each $d \in C$:
   - $T := \texttt{pseudo-complete}(\mathcal{F}, S \cup \{d\}, P)$
   - if $T \neq \bot$: recurse; return result if it succeeds
   - else add d to P *(prune dead-end branches)*
4. **return** $\bot$

Enumeration with Pruning

σ-extension$(\mathcal{F}, S, P)$

1. if $S \neq \emptyset$ is a σ-extension **return** S
2. build a *(hopefully, small)* candidate set $C \subseteq A \setminus (R(S) \cup R^{-1}(S) \cup P)$
3. **for** each $d \in C$:
   - $T := \texttt{pseudo-complete}(\mathcal{F}, S \cup \{d\}, P)$
   - if $T \neq \bot$: recurse; return result if it succeeds
   - else add d to P *(prune dead-end branches)*
4. **return** $\bot$

- *SE / DC / EE* are small variations (initialization / termination)

Enumeration with Pruning

σ-extension$(\mathcal{F}, S, P)$

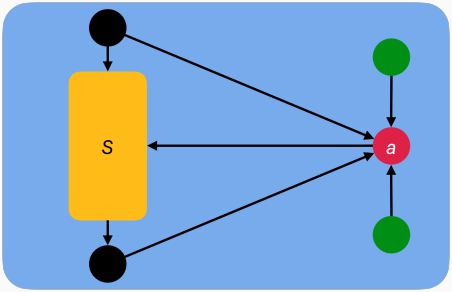
1. if $S \neq \emptyset$ is a σ-extension **return** S
2. build a *(hopefully, small)* candidate set $C \subseteq A \setminus (R(S) \cup R^{-1}(S) \cup P)$
3. **for** each $d \in C$:
   - $T :=$ `pseudo-complete`$(\mathcal{F}, S \cup \{d\}, P)$
   - if $T \neq \bot$: recurse; return result if it succeeds
   - else add d to P *(prune dead-end branches)*
4. **return** $\bot$

- *SE / DC / EE* are small variations (initialization / termination)
- For preferred extensions, do not perform the check in line 1 but return S in line 4 if it is self-defending

Candidate Selection

How to form the candidate set C?

Complete: pick an attacker a of S not attacked by S;
$C :=$ attackers of a (not in P, not in conflict with S)

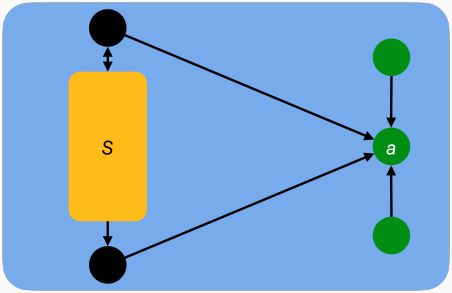


if no candidate works, S cannot be defended against $a \Rightarrow$ prune the whole branch

Candidate Selection

How to form the candidate set C?

Stable: a stable extension meets every set $\{a\} \cup R^{-1}(a)$;
once S is complete but not stable, use such a set as C

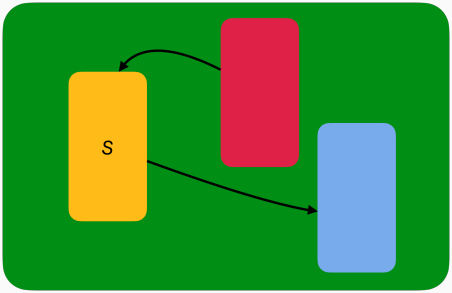


among several candidate sets, choose the smallest $\Rightarrow$ fail early

Candidate Selection

How to form the candidate set C?

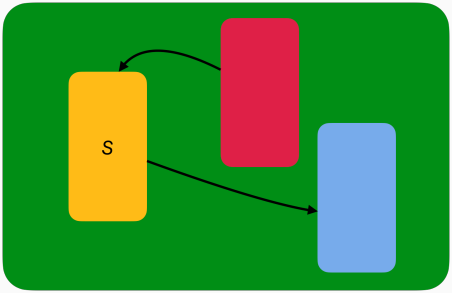
Preferred: do not stop at a complete S—keep extending;
if no extension is possible and S is admissible, S is preferred



Candidate Selection

How to form the candidate set C?

Preferred: do not stop at a complete S—keep extending;
if no extension is possible and S is admissible, S is preferred



Unlike in generic closure-enumeration algorithms with their fixed order on the ground set, candidates are selected dynamically, resulting in much smaller search trees on sparse AFs.

Optimization: P-pseudo-complete Sets

Observation: search trees contain long paths of forced moves—arguments with a *single* allowed defender, added one closure at a time.

Optimization: P-pseudo-complete Sets

Observation: search trees contain long paths of forced moves—arguments with a *single* allowed defender, added one closure at a time.

Definition

S is P-pseudo-complete if it is pseudo-complete and every attacker of S is attacked by S or has ≥ 2 attackers in $A \setminus (R(S) \cup R^{-1}(S) \cup P)$.

Optimization: P-pseudo-complete Sets

Observation: search trees contain long paths of forced moves—arguments with a *single* allowed defender, added one closure at a time.

Definition

S is P-pseudo-complete if it is pseudo-complete and every attacker of S is attacked by S or has ≥ 2 attackers in $A \setminus (R(S) \cup R^{-1}(S) \cup P)$.

- Still a closure system for each P

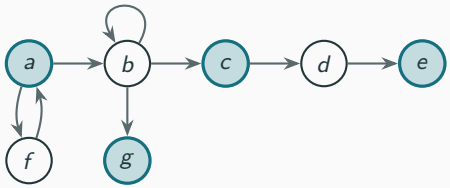
Optimization: P-pseudo-complete Sets

Observation: search trees contain long paths of forced moves—arguments with a *single* allowed defender, added one closure at a time.

Definition

S is P-pseudo-complete if it is pseudo-complete and every attacker of S is attacked by S or has ≥ 2 attackers in $A \setminus (R(S) \cup R^{-1}(S) \cup P)$.

- Still a closure system for each P
- New closure rule: if attacker a of S has a single available attacker b, add b immediately



$\emptyset$-pseudo-completion of $\{c\}$: add e, g, then the *forced* a (b's only available attacker) $\Rightarrow$ complete extension in one step

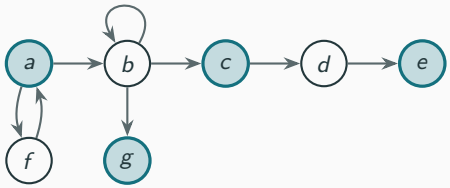
Optimization: P-pseudo-complete Sets

Observation: search trees contain long paths of forced moves—arguments with a *single* allowed defender, added one closure at a time.

Definition

S is P-pseudo-complete if it is pseudo-complete and every attacker of S is attacked by S or has ≥ 2 attackers in $A \setminus (R(S) \cup R^{-1}(S) \cup P)$.

- Still a closure system for each P
- New closure rule: if attacker a of S has a single available attacker b, add b immediately
- Forced chains collapse into one closure computation; as P grows, the search space shrinks



$\emptyset$-pseudo-completion of $\{c\}$: add e, g, then the *forced* a (b's only available attacker) $\Rightarrow$ complete extension in one step

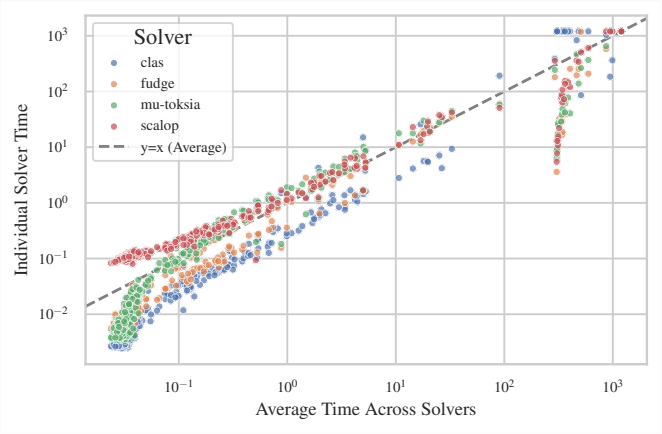
Experimental Evaluation: CLAS

CLAS (Closure-based Argumentation Solver in C) `github.com/sertkaya/afca/tree/CLAS`

- 322 benchmarks from ICCMA'25; tracks DC-CO, DC-ST, SE-ST, SE-PR
- vs. top-3 solvers: FUDGE, μ-TOKSIA, SCALOP
- timeout 1200 s; PAR-2: failures count as 2×1200 s
- Linux / 2.9 GHz / 256 GB

	DC-CO			DC-ST			SE-ST			SE-PR		
	fastest	fail	PAR2	fastest	fail	PAR2	fastest	fail	PAR2	fastest	fail	PAR2
CLAS	**280**	26	204	**268**	32	243	**228**	32	246	**223**	43	327
FUDGE	13	10	86	29	8	71	44	12	100	37	11	100
μ-TOKSIA	14	10	85	9	8	71	29	11	98	38	10	100
SCALOP	7	11	93	8	9	78	11	11	94	15	10	95
all failed		8			8			10			9	

What the Numbers Mean



Each point: one solver on one framework (log–log);
x = average time across solvers $\approx$ instance difficulty.

- Fastest on the vast majority of the 322 benchmarks
- Solves instances where all other solvers time out
  (two hardest solved DC-CO instances: 363 s and 184 s — CLAS only)
- But: more failures $\Rightarrow$ highest PAR-2
- Failures concentrate on `crusti*` / `scc*`: frameworks with many strongly connected components

$\Rightarrow$ exploit that structure!

Leveraging Strongly Connected Components

Focus: preferred extensions (SE-PR).

Definitions

- A source component of $\mathcal{F}$: an SCC (C, R_C) with no attacks coming into C from outside
- $\mathcal{F} - S$: remove the arguments $S \subseteq A$
- $\mathcal{F} \dot{-} S$: additionally add self-loops to all remaining arguments attacked by S

Find a complete extension in a source component, then recurse on a smaller residual framework

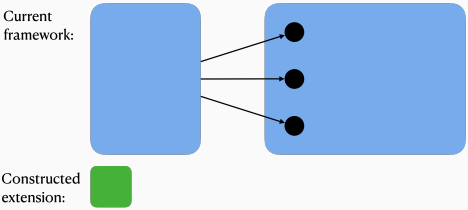
Leveraging Strongly Connected Components

Focus: preferred extensions (SE-PR).

Definitions

- A source component of $\mathcal{F}$: an SCC (C, R_C) with no attacks coming into C from outside
- $\mathcal{F} - S$: remove the arguments $S \subseteq A$
- $\mathcal{F} \dot{-} S$: additionally add self-loops to all remaining arguments attacked by S

Find a complete extension in a source component, then recurse on a smaller residual framework



- If the only complete extension of a source component $\mathcal{F}_C = (C, R_C)$ of $\mathcal{F}$ is $\emptyset$,

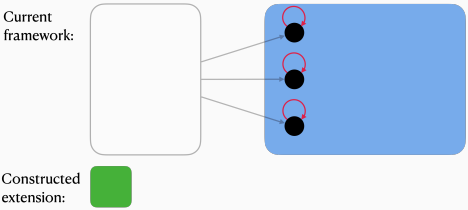
Leveraging Strongly Connected Components

Focus: preferred extensions (SE-PR).

Definitions

- A source component of $\mathcal{F}$: an SCC (C, R_C) with no attacks coming into C from outside
- $\mathcal{F} - S$: remove the arguments $S \subseteq A$
- $\mathcal{F} \dot{-} S$: additionally add self-loops to all remaining arguments attacked by S

Find a complete extension in a source component, then recurse on a smaller residual framework



- If the only complete extension of a source component $\mathcal{F}_C = (C, R_C)$ of $\mathcal{F}$ is $\emptyset$,
- then the preferred extensions of $\mathcal{F}$ are the preferred extensions of $\mathcal{F} \dot{-} C$.

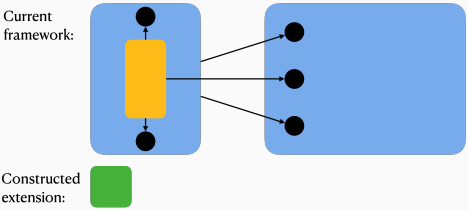
Leveraging Strongly Connected Components

Focus: preferred extensions (SE-PR).

Definitions

- A source component of $\mathcal{F}$: an SCC (C, R_C) with no attacks coming into C from outside
- $\mathcal{F} - S$: remove the arguments $S \subseteq A$
- $\mathcal{F} \dot{-} S$: additionally add self-loops to all remaining arguments attacked by S

Find a complete extension in a source component, then recurse on a smaller residual framework



- If the source component contains a non-empty complete extension,

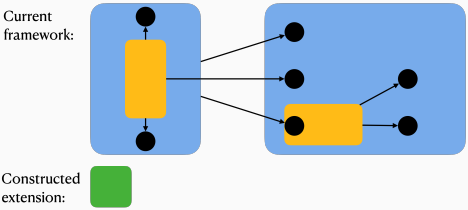
Leveraging Strongly Connected Components

Focus: preferred extensions (SE-PR).

Definitions

- A source component of $\mathcal{F}$: an SCC (C, R_C) with no attacks coming into C from outside
- $\mathcal{F} - S$: remove the arguments $S \subseteq A$
- $\mathcal{F} \dot{-} S$: additionally add self-loops to all remaining arguments attacked by S

Find a complete extension in a source component, then recurse on a smaller residual framework



- If the source component contains a non-empty complete extension,
- semi-complete it in $\mathcal{F}$ to obtain a complete extension E of $\mathcal{F}$,

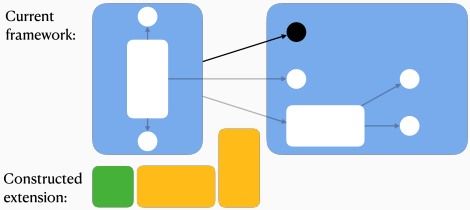
Leveraging Strongly Connected Components

Focus: preferred extensions (SE-PR).

Definitions

- A source component of $\mathcal{F}$: an SCC (C, R_C) with no attacks coming into C from outside
- $\mathcal{F} - S$: remove the arguments $S \subseteq A$
- $\mathcal{F} \dot{-} S$: additionally add self-loops to all remaining arguments attacked by S

Find a complete extension in a source component, then recurse on a smaller residual framework



- If the source component contains a non-empty complete extension,
- semi-complete it in $\mathcal{F}$ to obtain a complete extension E of $\mathcal{F}$,
- accumulate the result, and continue on the residual $\mathcal{F} - E - R(E)$.

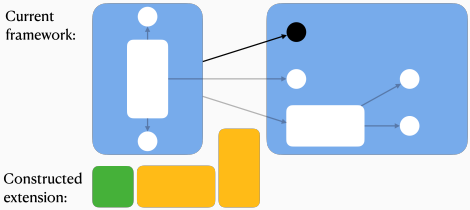
Leveraging Strongly Connected Components

Focus: preferred extensions (SE-PR).

Definitions

- A source component of $\mathcal{F}$: an SCC (C, R_C) with no attacks coming into C from outside
- $\mathcal{F} - S$: remove the arguments $S \subseteq A$
- $\mathcal{F} \dot{-} S$: additionally add self-loops to all remaining arguments attacked by S

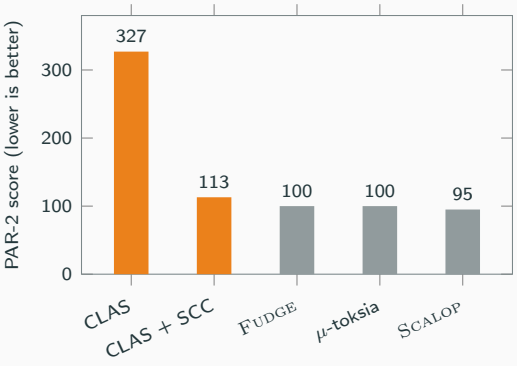
Find a complete extension in a source component, then recurse on a smaller residual framework



- . . . accumulate the result, and continue on the residual $\mathcal{F} - E - R(E)$.
- Why this works:

  If E is a complete extension of $\mathcal{F}$ and D is a preferred extension of $\mathcal{F} - E - R(E)$, then $D \cup E$ is a preferred extension of $\mathcal{F}$

SE-PR with Decomposition



Same 322 benchmarks, SE-PR track:

- failures: 43 → 12
  (others: 10–11; 9 unsolved by everyone)
- still fastest on 167 instances
  (FUDGE 80, μ-toksia 49, SCALOP 17)
- remaining failures: single SCC or a huge source component

Decomposition is cheap and pays off broadly.

Conclusion

Summary

- A direct approach: no SAT/CSP translation
- Complete, preferred, stable extensions live in the closure system of pseudo-complete sets: enumerate until one is found
- Dynamic candidate selection + P-pseudo-completeness prune the search
- CLAS: fastest on most ICCMA'25 benchmarks, but sometimes fail where others succeed; SCC decomposition (almost) fixes its weak spot for preferred semantics

Conclusion

Summary

- A direct approach: no SAT/CSP translation
- Complete, preferred, stable extensions live in the closure system of pseudo-complete sets: enumerate until one is found
- Dynamic candidate selection + P-pseudo-completeness prune the search
- CLAS: fastest on most ICCMA'25 benchmarks, but sometimes fail where others succeed; SCC decomposition (almost) fixes its weak spot for preferred semantics

Future work

- SCC decomposition for complete and stable semantics
- CDCL-style learning of indirect conflicts between arguments
- Skeptical reasoning by enumeration

Conclusion

Summary

- A direct approach: no SAT/CSP translation
- Complete, preferred, stable extensions live in the closure system of pseudo-complete sets: enumerate until one is found
- Dynamic candidate selection + P-pseudo-completeness prune the search
- CLAS: fastest on most ICCMA'25 benchmarks, but sometimes fail where others succeed; SCC decomposition (almost) fixes its weak spot for preferred semantics

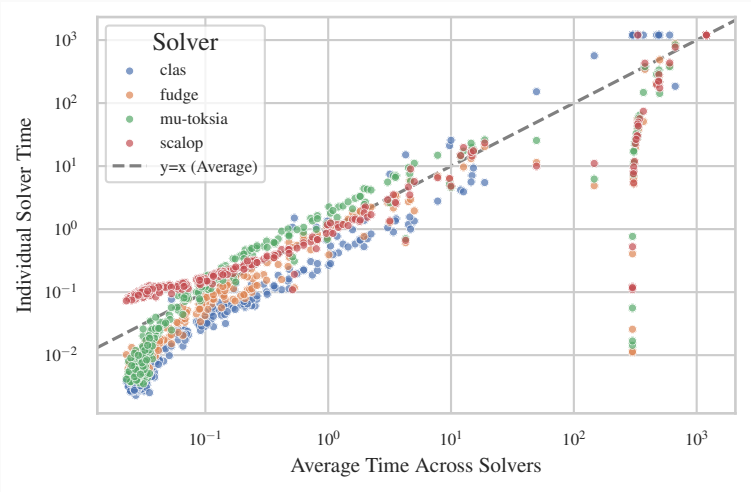
Future work

- SCC decomposition for complete and stable semantics
- CDCL-style learning of indirect conflicts between arguments
- Skeptical reasoning by enumeration

Thank you!

CLAS: `github.com/sertkaya/afca/tree/CLAS`

Backup: DC-ST Per-Instance Runtimes



Backup: Full Results (Algorithm 1), SE-PR

	SE-PR (Algorithm 1)		
	fastest	failures	PAR2
CLAS	223	43	327
FUDGE	37	11	100
μ-toksia	38	10	100
SCALOP	15	10	95
all failed		9	

PAR-2: per instance, runtime in seconds, or 2×1200 on timeout/memory-out; final score is the average over all instances.

Algorithm 2: Preferred Extension by Decomposition

preferred-extension($\mathcal{F}$)

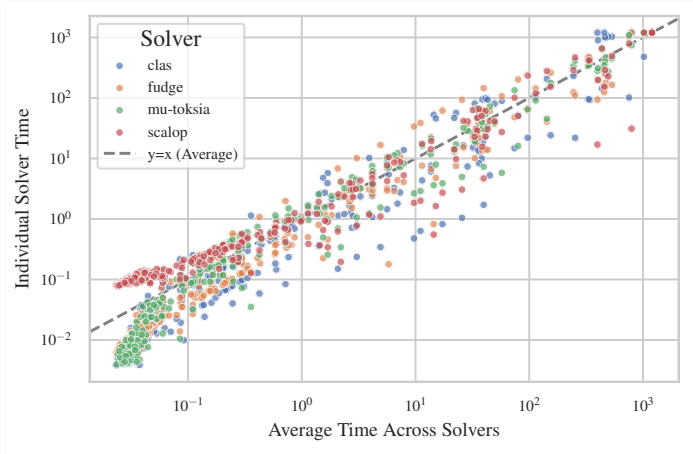
1: $X := \emptyset$
2: **while** $\mathcal{F}$ is not empty **do**
3: find a source component $\mathcal{F}_C = (C, R_C)$
4: $S :=$ `semi-complete`$(\mathcal{F}_C, \emptyset)$
5: $E :=$ `complete-extension`$(\mathcal{F}_C, S, \emptyset)$
6: **if** $E = \bot$ **then**
7: $\mathcal{F} := \mathcal{F} \dot{-} C$
8: **else**
9: $E :=$ `semi-complete`$(\mathcal{F}, E)$
10: $X := X \cup E$
11: $\mathcal{F} := \mathcal{F} - E - R(E)$
12: **return** X

- Line 5: Algorithm 1, searching for a *non-empty* complete extension in a source component.
  - Any procedure computing complete extensions can be plugged in.
- If none exists, drop C and add self-loops to $R(C)$ (work in $\mathcal{F} \dot{-} C$)
- Otherwise, semi-complete E in $\mathcal{F}$ to obtain a complete extension of $\mathcal{F}$, accumulate the result, and continue on the residual $\mathcal{F} - E - R(E)$.

Theorem

Algorithm 2 outputs a preferred extension of $\mathcal{F}$.

SE-PR: Per-Instance Runtimes (with Decomposition)



With decomposition, CLAS keeps its speed advantage while its timeouts drop to the level of the other solvers.