Searching for Liveness Property Violations in Concurrent Systems with ACO

LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE MÁLAGA

UNIVERSIDAD DE MÁLAGA

Francisco Chicano and Enrique Alba

Motivation

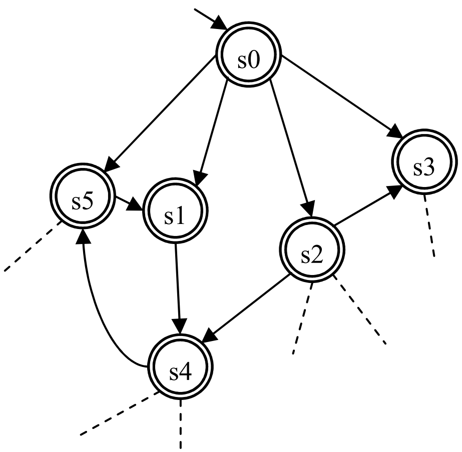
- Concurrent software is difficult to test ...
- ... and it is in the heart of a lot of critical systems

- Techniques for proving the correctness of concurrent software are required
- Model checking → fully automatic
- In the past the work using metaheuristics focused on safety properties
- In this work we focus on liveness properties

Explicit State Model Checking

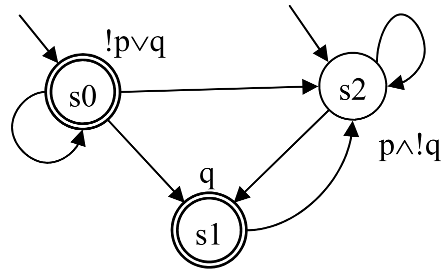
- **Objective: Prove that model M satisfies the property f: $M \models f$**
- **HSF-SPIN: the property f is an LTL formula**

Model M



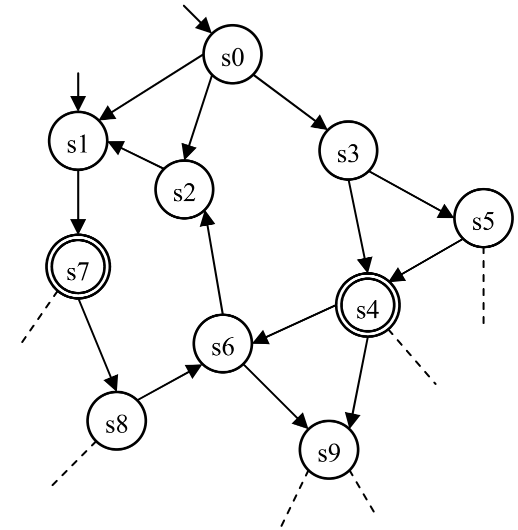
$\cap$

LTL formula $\neg f$ (never claim)



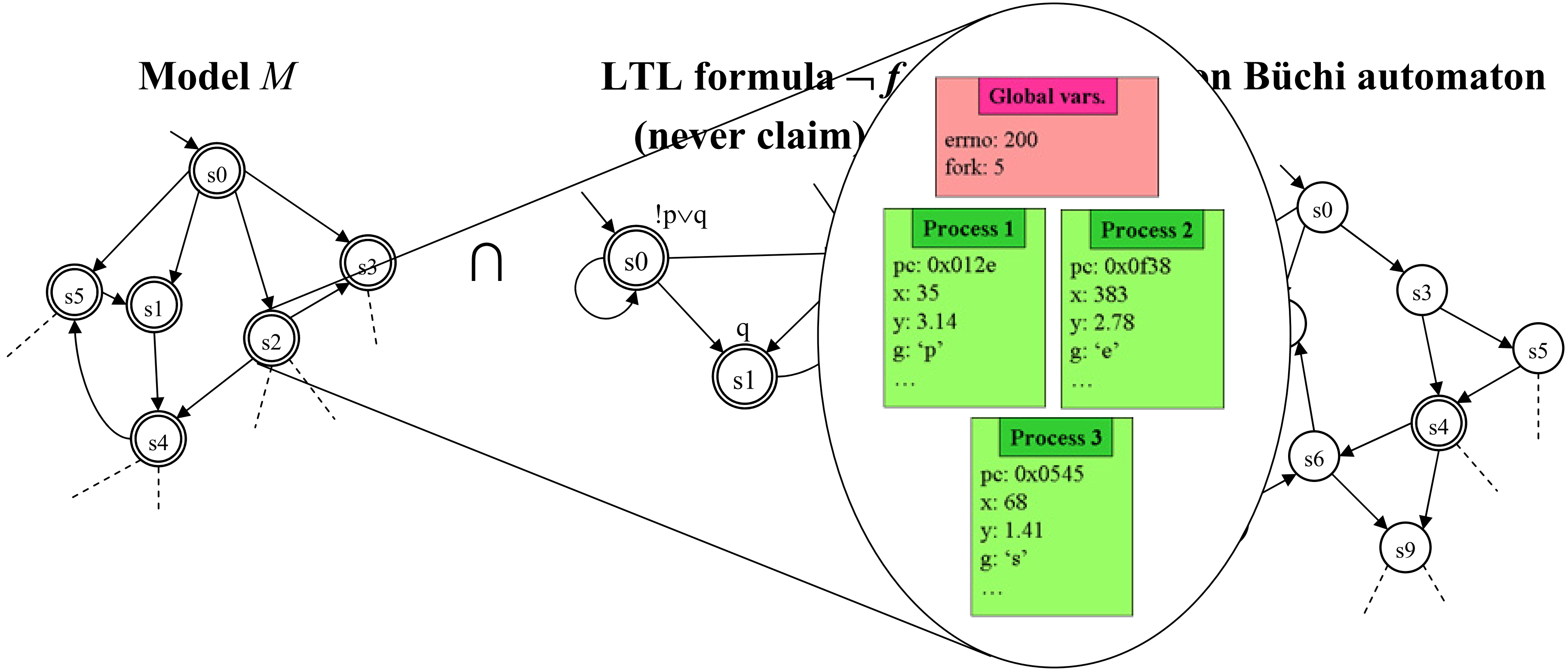
$=$

Intersection Büchi automaton



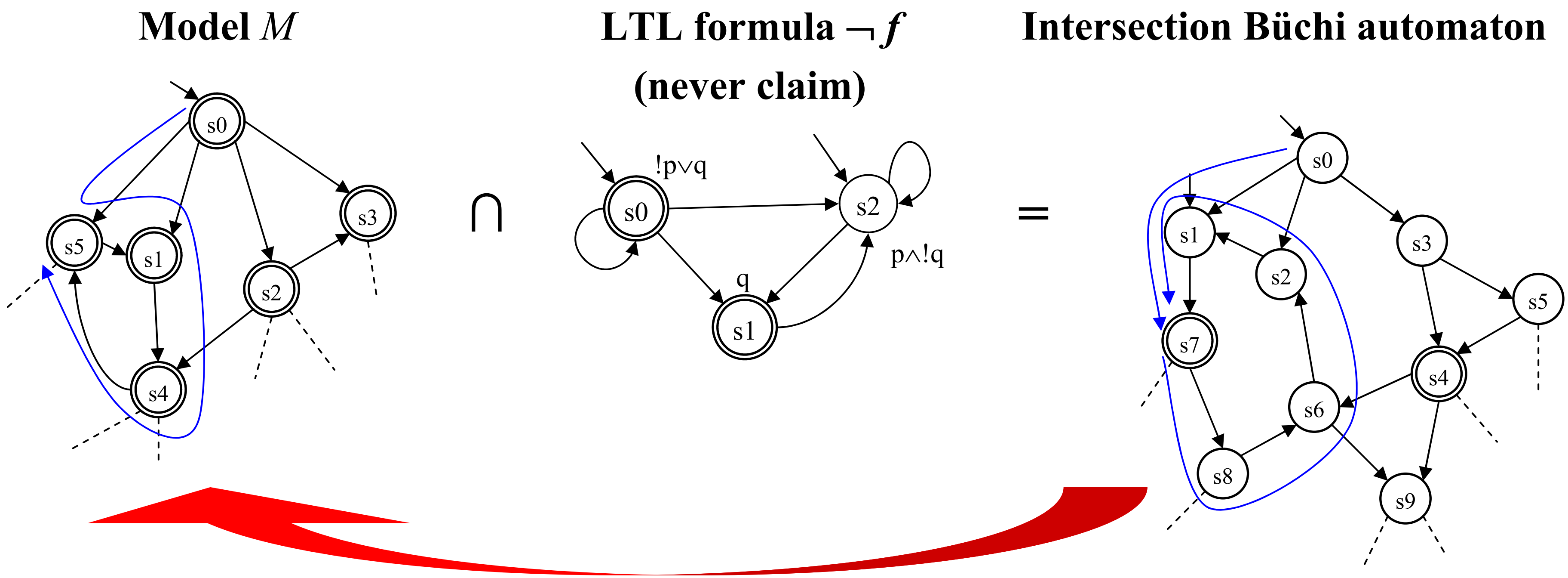
Explicit State Model Checking

- **Objective: Prove that model M satisfies the property f: $M \models f$**
- **HSF-SPIN: the property f is an LTL formula**



Explicit State Model Checking

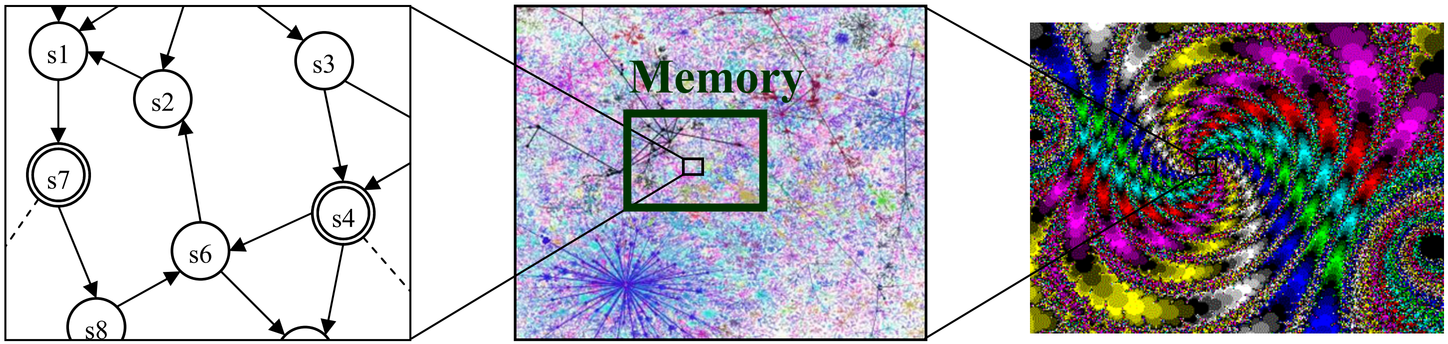
- **Objective: Prove that model M satisfies the property f: $M \models f$**
- **HSF-SPIN: the property f is an LTL formula**



Using Nested-DFS

State Explosion Problem

- **Number of states very large even for small models**



- **Example: Dining philosophers with *n* philosophers → 3^n states**

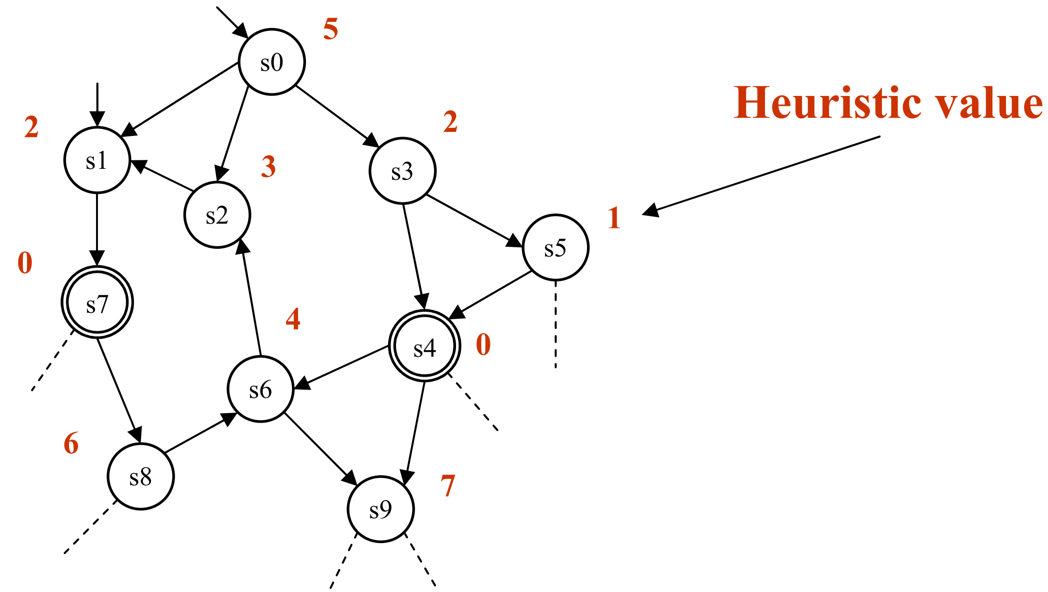
  20 philosophers → 1039 GB for storing the states

- **Solutions: collapse compression, minimized automaton representation, bitstate hashing, partial order reduction, symmetry reduction**

- **Large models cannot be verified but errors can be found**

Heuristic Model Checking

- The search for errors can be directed by using heuristic information



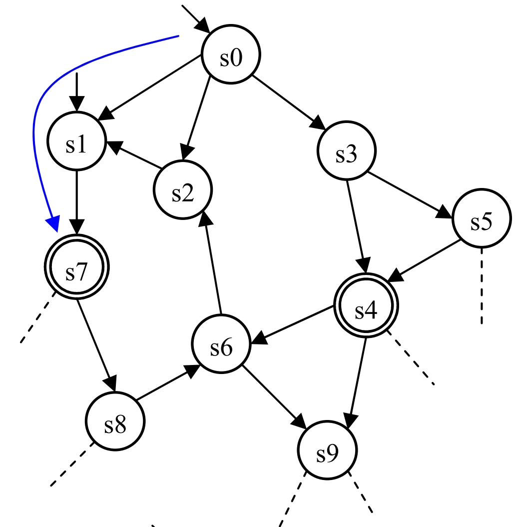
- Different kinds of heuristic functions have been proposed in the past:
  - Formula-based heuristics
  - Structural heuristics
  - Deadlock-detection heuristics
  - State-dependent heuristics

Safety and Liveness Properties

Safety property

$$\forall \sigma \in S^\omega : \sigma \nvdash \mathcal{P} \Rightarrow (\exists i \geq 0 : \forall \beta \in S^\omega : \sigma_i \beta \nvdash \mathcal{P})$$

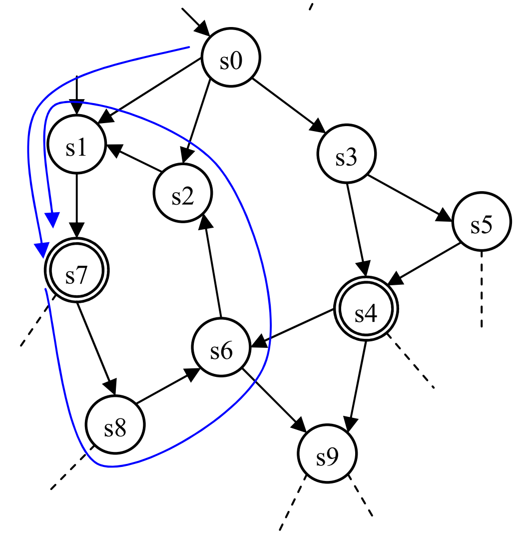
- **Counterexample ≡ path to accepting state**
- **Graph exploration algorithms can be used: DFS and BFS**



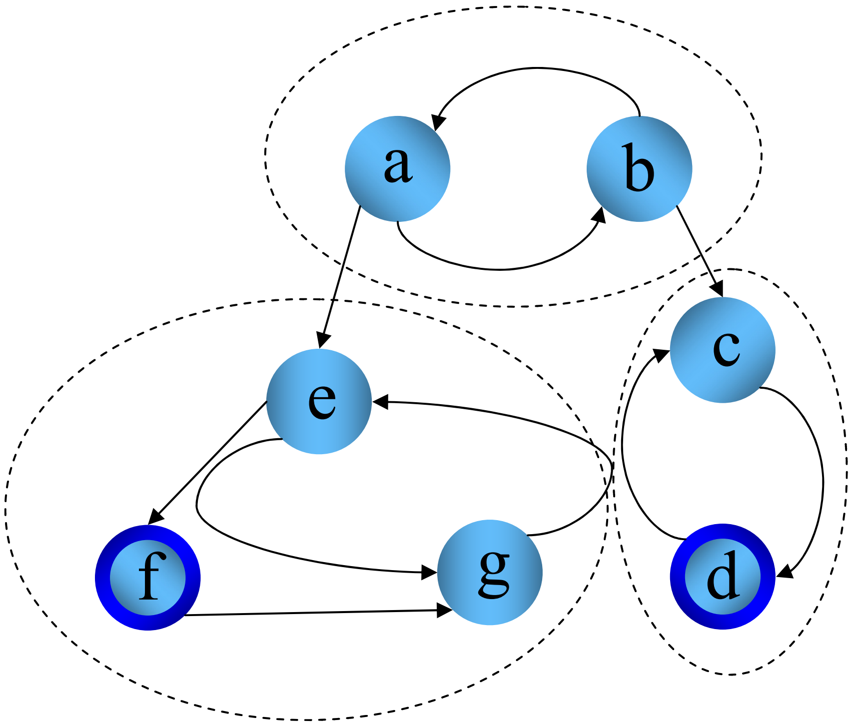
Liveness property

$$\forall \alpha \in S^* : \exists \beta \in S^\omega, \alpha\beta \vdash \mathcal{P}$$

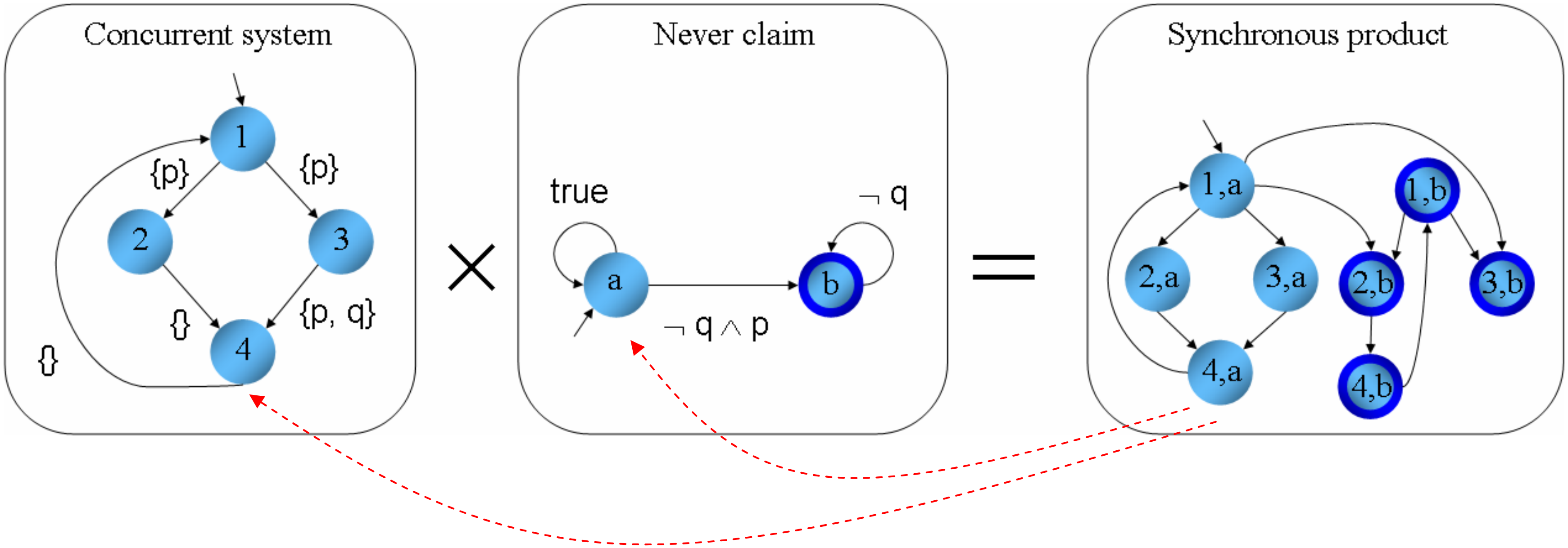
- **Counterexample ≡ path to accepting cycle**
- **It is not possible to apply DFS or BFS**



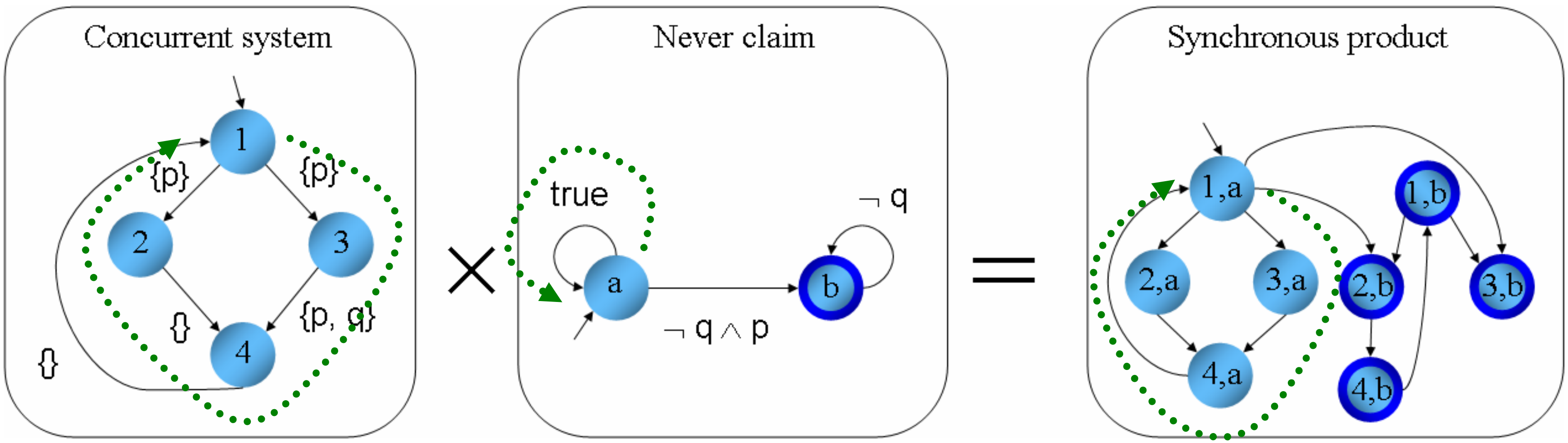
Strongly Connected Components



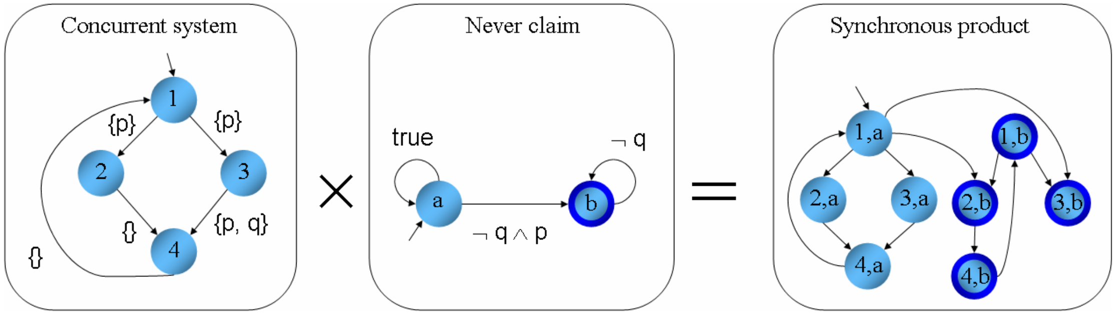
Strongly Connected Components

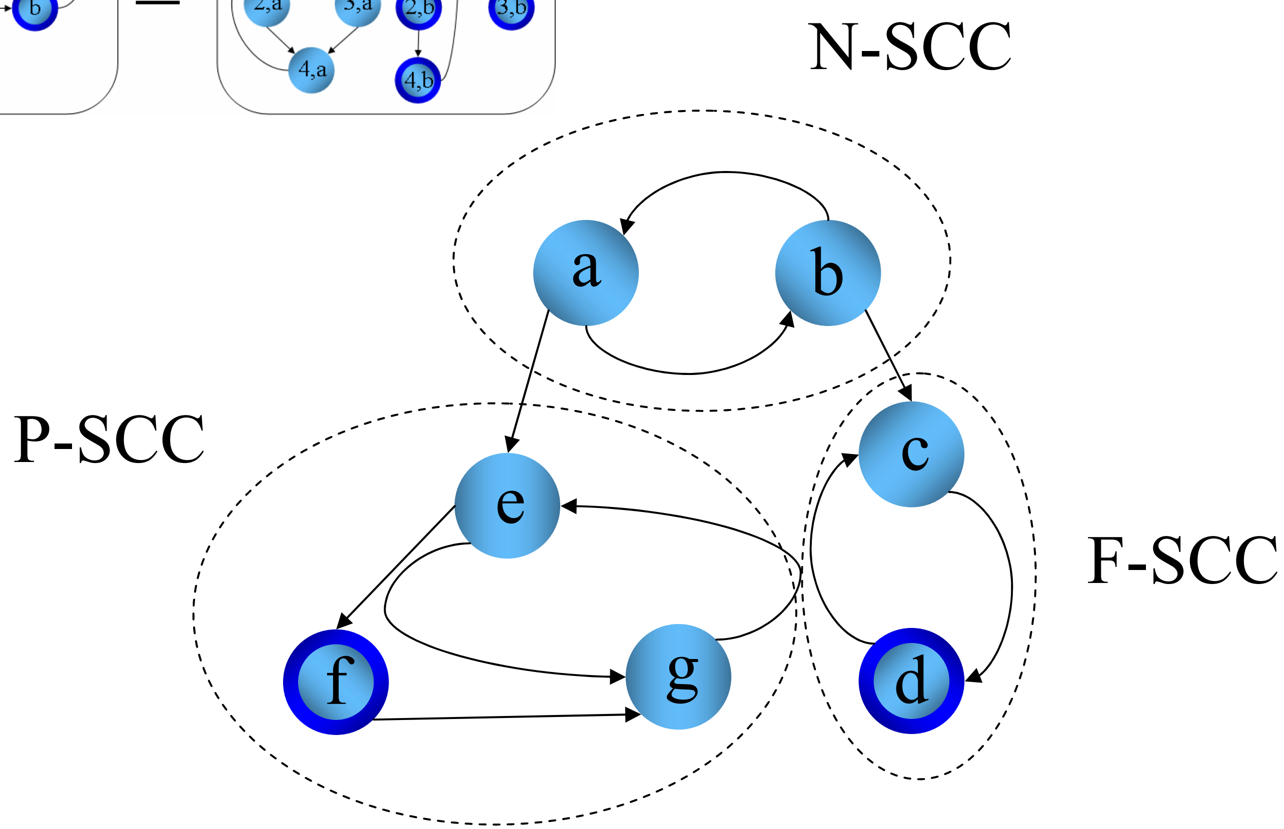


Strongly Connected Components



Strongly Connected Components





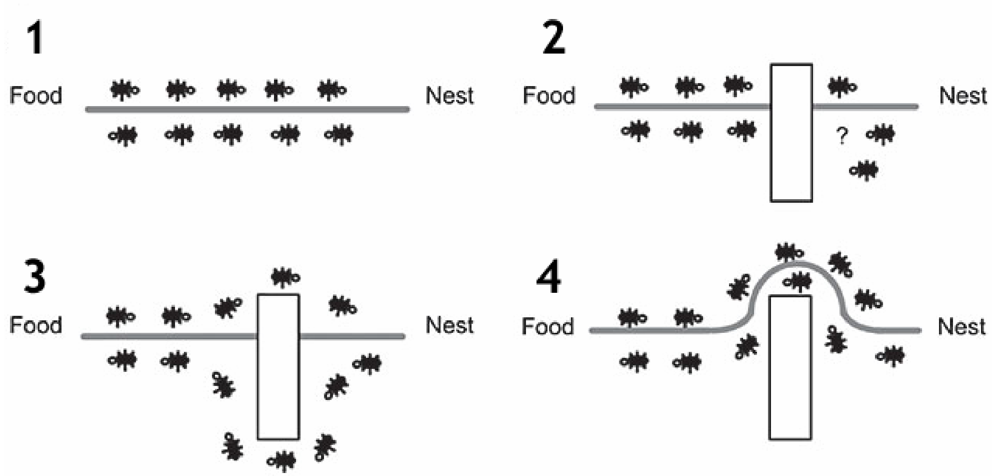
Optimization/Search Techniques

OPTIMIZATION/SEARCH TECHNIQUES

- EXACT
  - Based on Calculus
    - Newton
    - Gradient
  - Enumeratives
    - Depth First Search
    - Branch and Bound
- APPROXIMATED
  - Ad Hoc Heuristics
  - METAHEURISTICS
    - Trayectory-based
      - SA
      - VNS
      - TS
    - Population-based
      - EA
      - ACO
      - PSO

ACO: Introduction

- **Ant Colony Optimization (ACO) metaheuristic is inspired by the foraging behaviour of real ants**

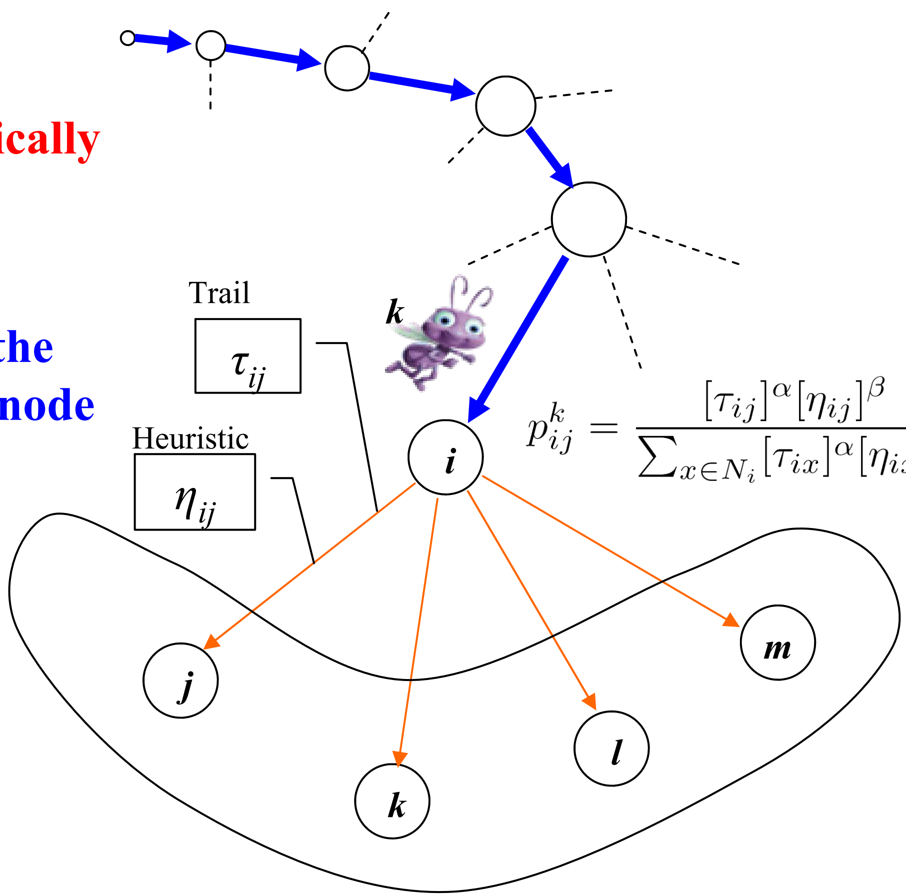


- **ACO Pseudo-code**

```
procedure ACOMetaheuristic
    ScheduleActivities
        ConstructAntsSolutions
        UpdatePheromones
        DaemonActions // optional
    end ScheduleActivities
end procedure
```

ACO: Construction Phase

- **The ant selects its next node stochastically**
- **The probability of selecting one node depends on the pheromone trail and the heuristic value (optional) of the edge/node**
- **The ant stops when a complete solution is built**



$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{x \in N_i} [\tau_{ix}]^\alpha [\eta_{ix}]^\beta}$$

ACO: Pheromone Update

- **Pheromone update**
  - **During the construction phase**

$$\tau_{ij} \leftarrow (1-\xi)\tau_{ij} \quad \text{with} \quad 0 \leq \xi \leq 1$$

  - **After the construction phase**

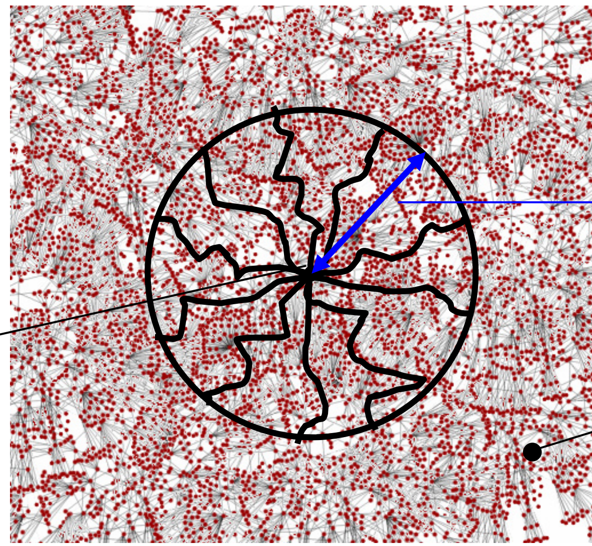
$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij} + \Delta\tau_{ij}^{bs} \quad \text{with} \quad 0 \leq \rho \leq 1$$

- **Trail limits (particular of *MM*AS)**
  - **Pheromones are kept in the interval [τ_{min}, τ_{max}]**

$$\tau_{max} = \frac{Q}{\rho} \qquad \tau_{min} = \frac{\tau_{max}}{a}$$

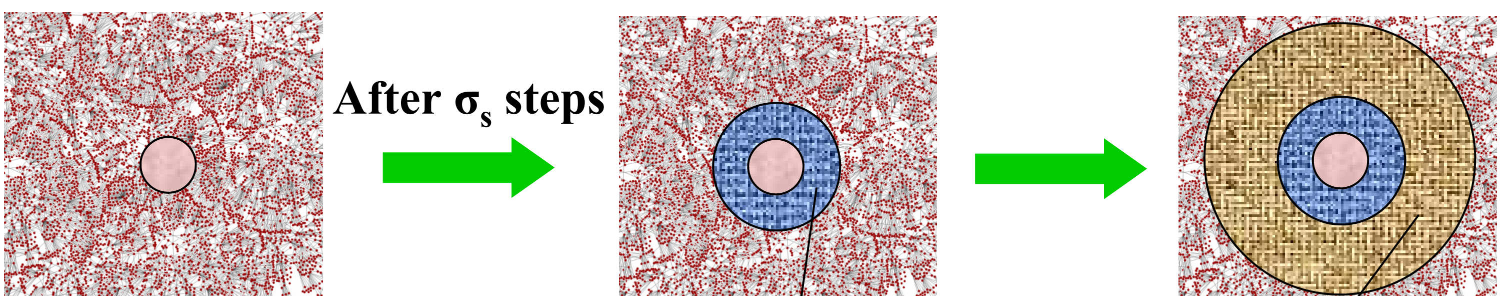
ACOhg: Huge Graphs Exploration

The length of the ant paths is limited by λ_{ant}



What if…?

Starting nodes for path construction change

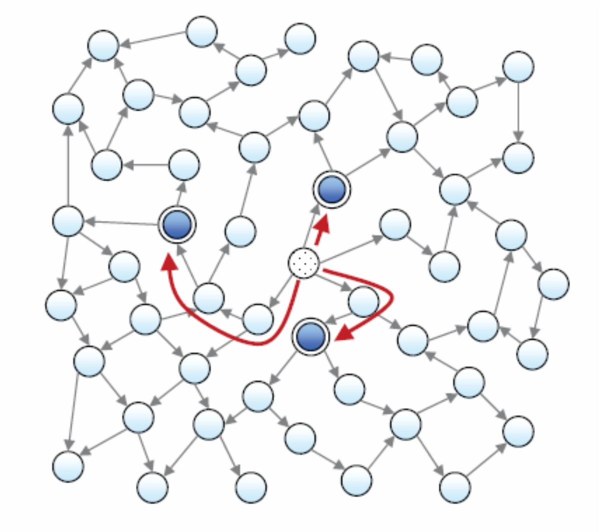


ACOhg-live

- **The search is an alternation of two phases**
  - **First phase: search for accepting states**
  - **Second phase: search for cycles from the accepting states**

ACOhg-live Pseudocode

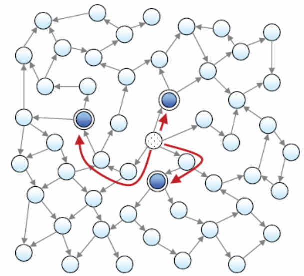
```
repeat
  accpt = acohg1.findAcceptingStates(); {First phase}
  for node in accpt do
    acohg2.findCycle(node); {Second phase}
    if acohg2.cycleFound() then
      return acohg2.acceptingPath();
    end if
  end for
  acohg1.insertTabu(accpt);
until empty(accpt)
return null;
```



First phase

ACOhg-live

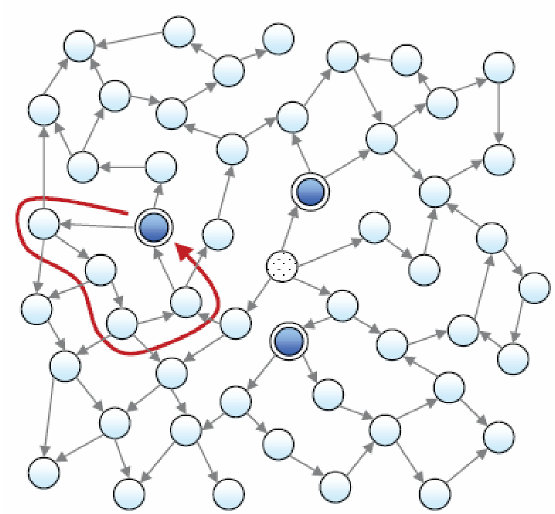
- **The search is an alternation of two phases**



  - **First phase: search for accepting states**
  - **Second phase: search for cycles from the accepting states**

ACOhg-live Pseudocode

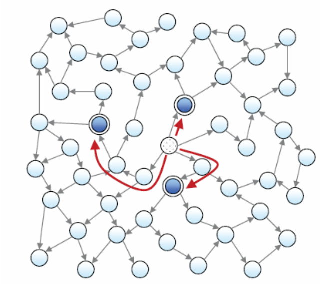
```
repeat
  accpt = acohg1.findAcceptingStates(); {First phase}
  for node in accpt do
    acohg2.findCycle(node); {Second phase}
    if acohg2.cycleFound() then
      return acohg2.acceptingPath();
    end if
  end for
  acohg1.insertTabu(accpt);
until empty(accpt)
return null;
```

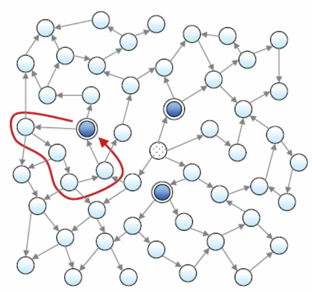


Second phase

ACOhg-live

- **The search is an alternation of two phases**

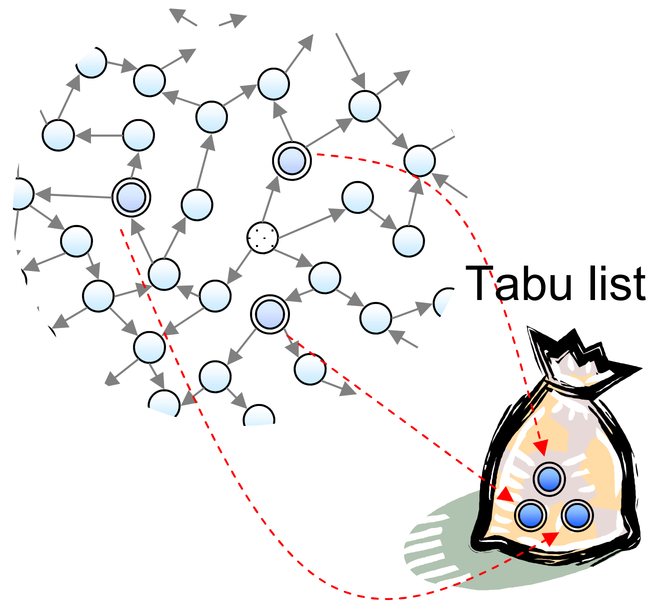




  - **First phase: search for accepting states**
  - **Second phase: search for cycles from the accepting states**

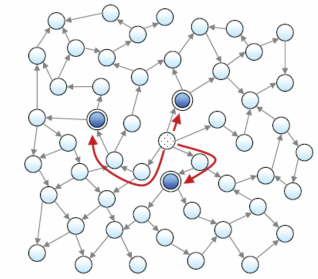
ACOhg-live Pseudocode

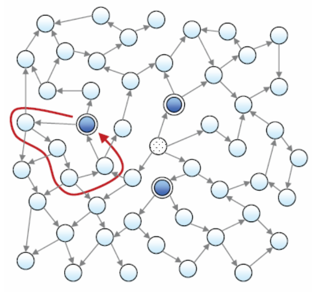
```
repeat
  accpt = acohg1.findAcceptingStates(); {First phase}
  for node in accpt do
    acohg2.findCycle(node); {Second phase}
    if acohg2.cycleFound() then
      return acohg2.acceptingPath();
    end if
  end for
  acohg1.insertTabu(accpt);
until empty(accpt)
return null;
```



ACOhg-live

- **The search is an alternation of two phases**



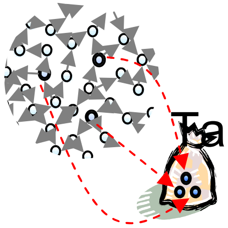


  - **First phase: search for accepting states**
  - **Second phase: search for cycles from the accepting states**

ACOhg-live Pseudocode

```
repeat
  accpt = acohg1.findAcceptingStates(); {First phase}
  for node in accpt do
    acohg2.findCycle(node); {Second phase}
    if acohg2.cycleFound() then
      return acohg2.acceptingPath();
    end if
  end for
  acohg1.insertTabu(accpt);
until empty(accpt)
return null;
```

- **Improvement using SCCs**



  - **First phase: accepting states in N-SCC are ignored**
  - **Both phases: cycle detected in F-SCC is a violation**

Promela Models

- **We used 11 Promela models for the experiments**

Model	LoC	Scalable	Processes	LTL formula (liveness)
`alter`	64	no	2	$\Box(p \rightarrow \Diamond q) \wedge \Box(r \rightarrow \Diamond s)$
`giopj`	740	yes	j+6	$\Box(p \rightarrow \Diamond q)$
`phij`	57	yes	j+1	$\Box(p \rightarrow \Diamond q)$
`elevj`	191	yes	j+3	$\Box(p \rightarrow \Diamond q)$
`sgc`	1001	no	20	$\Diamond p$

- **Parameters for ACOhg-live**

<table>
<tr><td>Parameter</td><td>msteps</td><td>colsize</td><td>λ_{ant}</td><td>σ_s</td><td>ξ</td><td>a</td><td>ρ</td><td>α</td><td>β</td></tr>
<tr><td>1st phase</td><td rowspan="2">100</td><td>10</td><td>20</td><td rowspan="2">4</td><td>0.7</td><td rowspan="2">5</td><td rowspan="2">0.2</td><td rowspan="2">1.0</td><td rowspan="2">2.0</td></tr>
<tr><td>2nd phase</td><td>20</td><td>4</td><td>0.5</td></tr>
</table>

- **Formula-based and finite state machine heuristics**
- **ACOhg-live implemented in HSF-SPIN**
- **100 independent executions and statistical validation**

Promela Models

- We used 11 Promela models for the experiments

Model	LoC	Scalable	Processes	LTL formula (liveness)
`alter`	64	no	2	$\Box(p \rightarrow \Diamond q) \wedge \Box(r \rightarrow \Diamond s)$
`giopj`	740	yes	j+6	$\Box(p \rightarrow \Diamond q)$
`phij`	57	yes	j+1	$\Box(p \rightarrow \Diamond q)$
`elevj`	191	yes	j+3	$\Box(p \rightarrow \Diamond q)$
`sgc`	1001	no	20	$\Diamond p$

giopj: j=10,15,20

phij: j=20,30,40

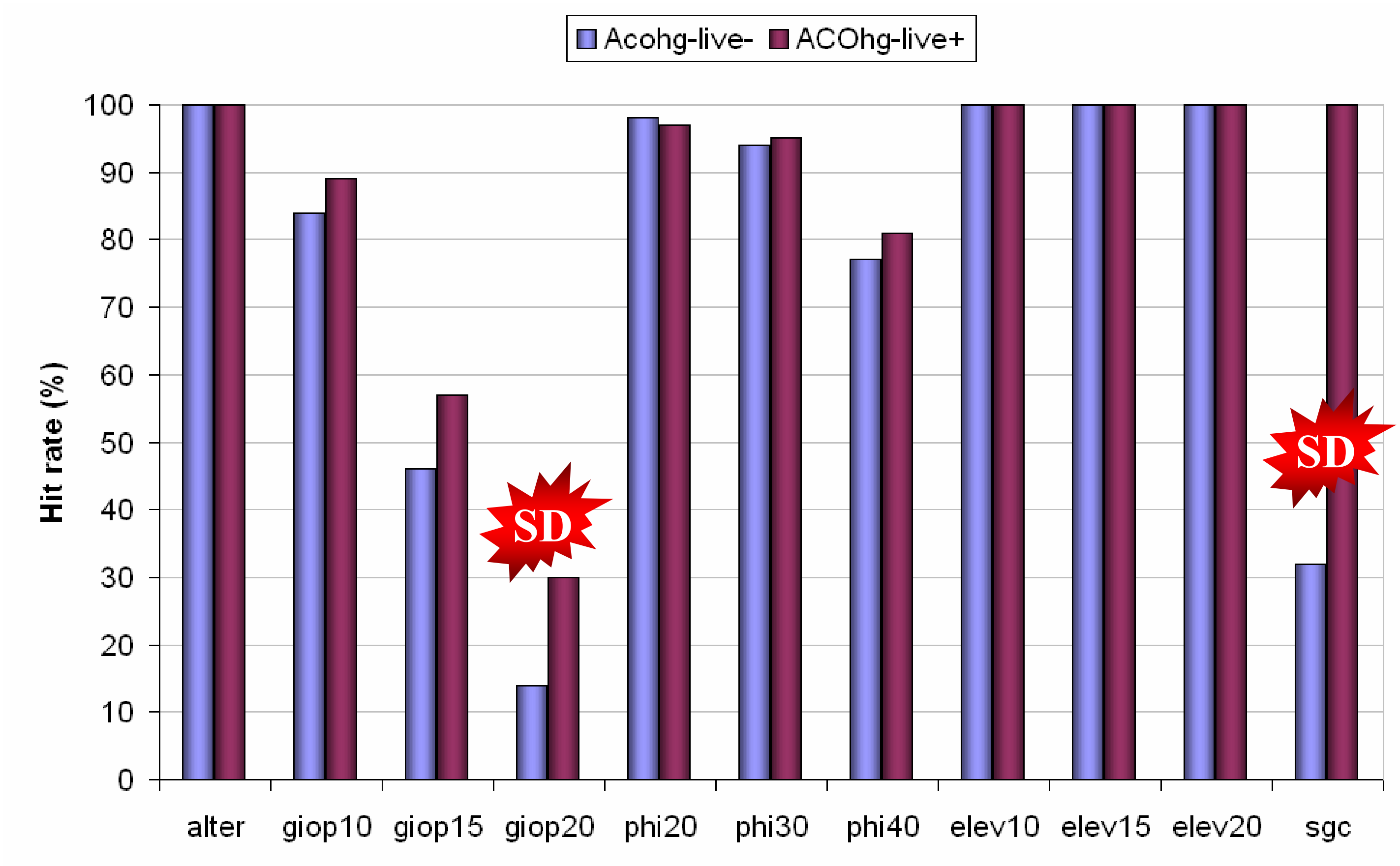
elevj: j=10,15,20

- Parameters for ACOhg-live

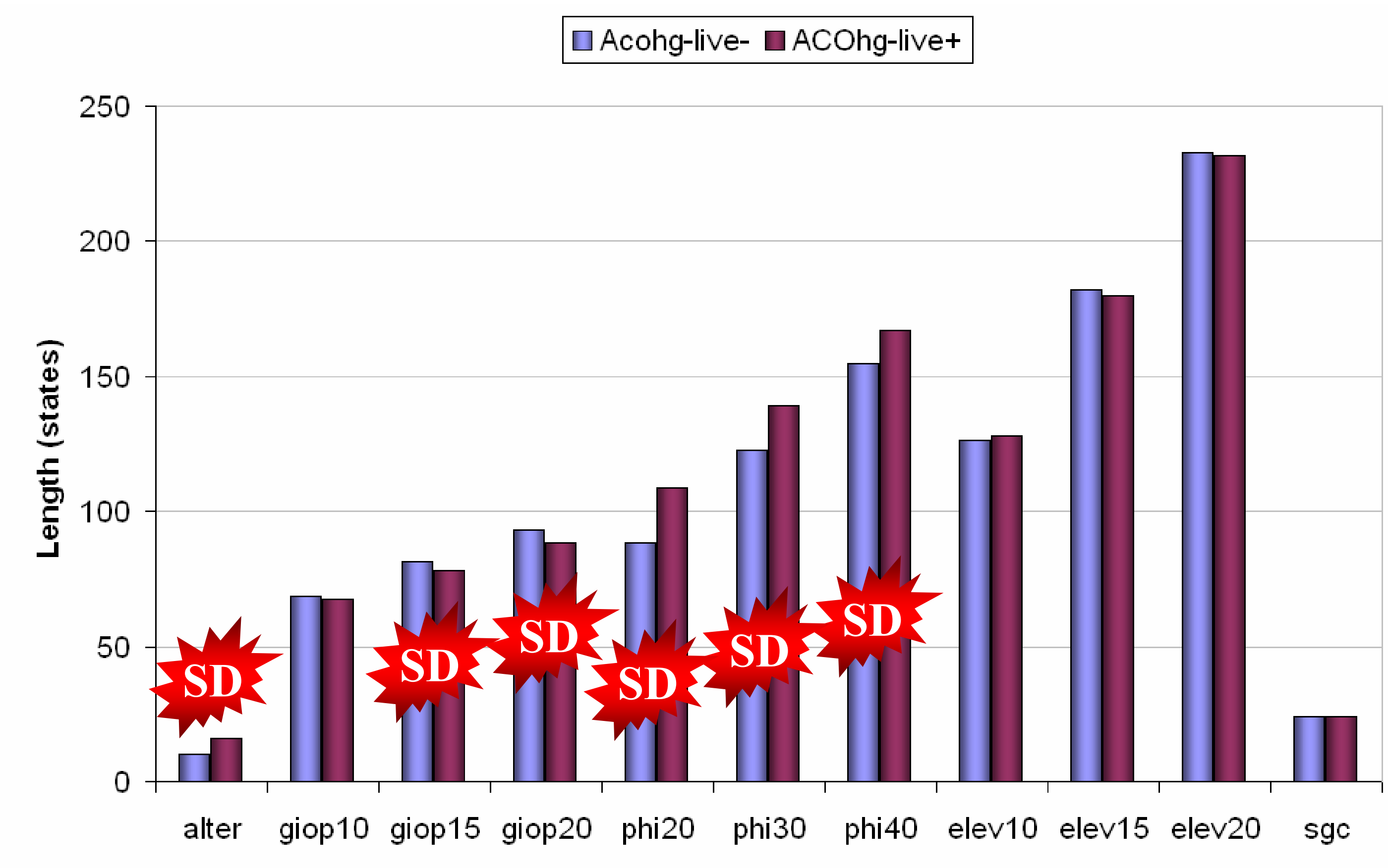
Parameter	*msteps*	*colsize*	λ_{ant}	σ_s	ξ	a	ρ	α	β
1st phase	100	10	20	4	0.7	5	0.2	1.0	2.0
2nd phase		20	4		0.5				

- Formula-based and finite state machine heuristics
- ACOhg-live implemented in HSF-SPIN
- 100 independent executions and statistical validation

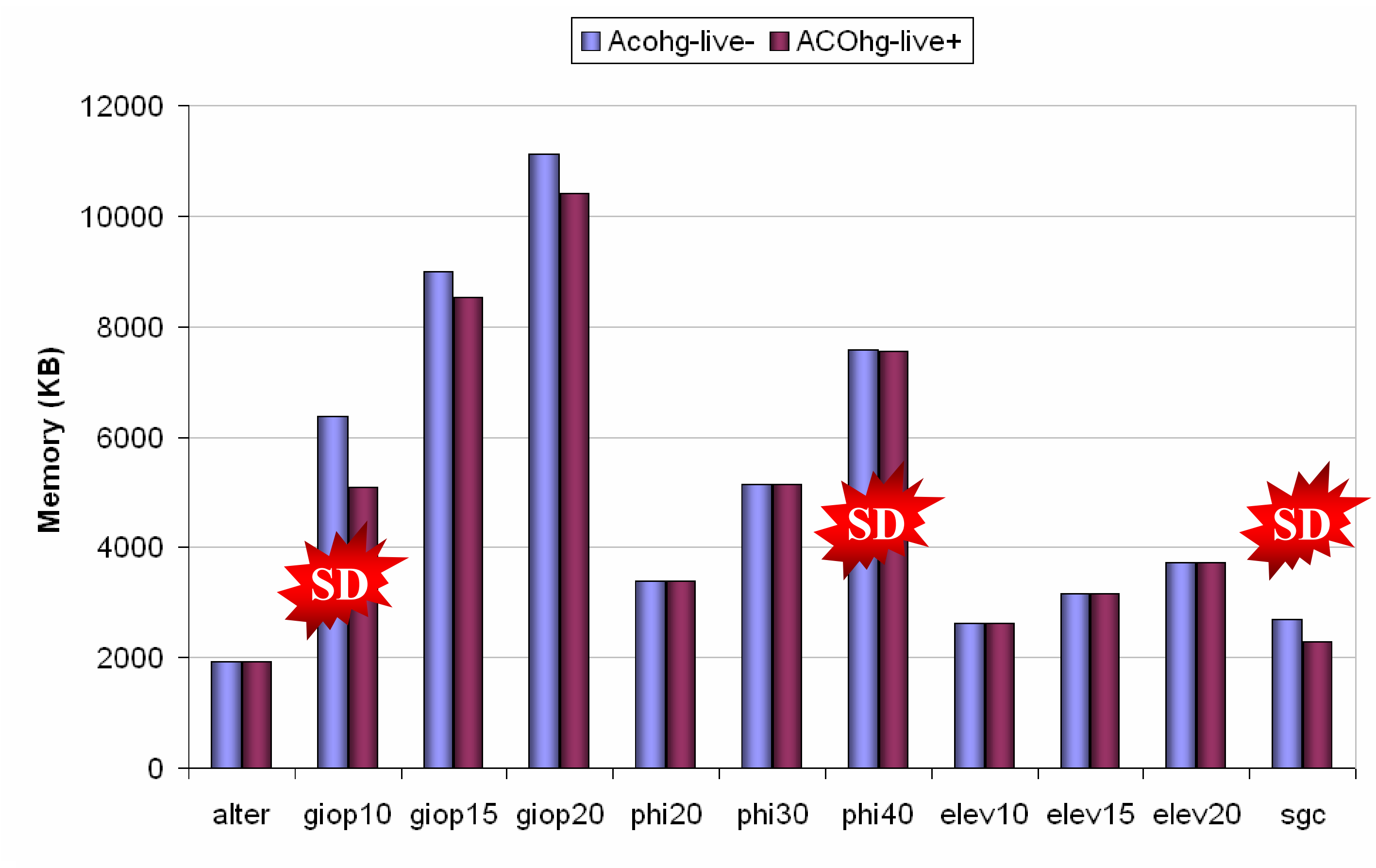
Results I: Influence of the SCC Improvement



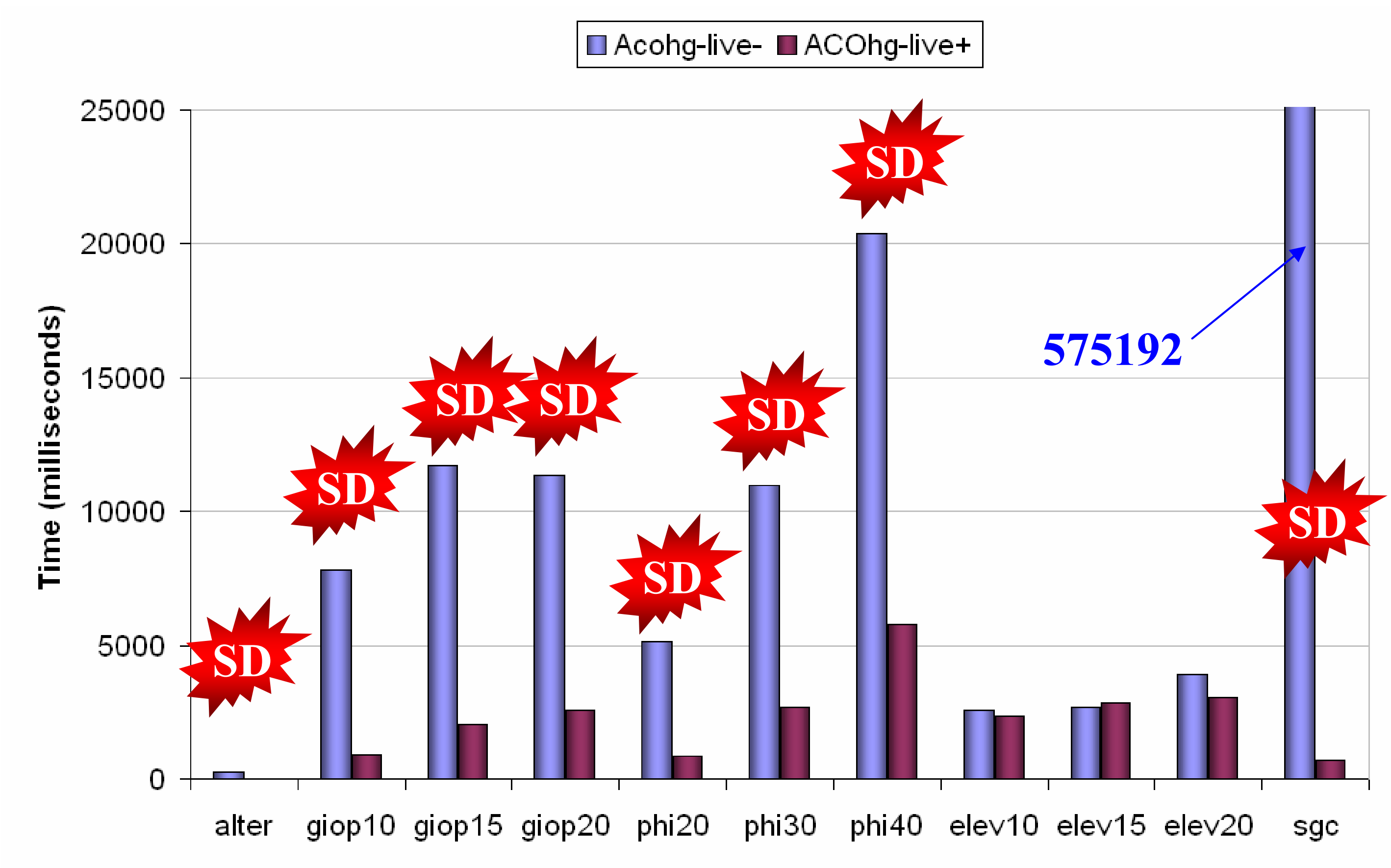
Results I: Influence of the SCC Improvement



Results I: Influence of the SCC Improvement



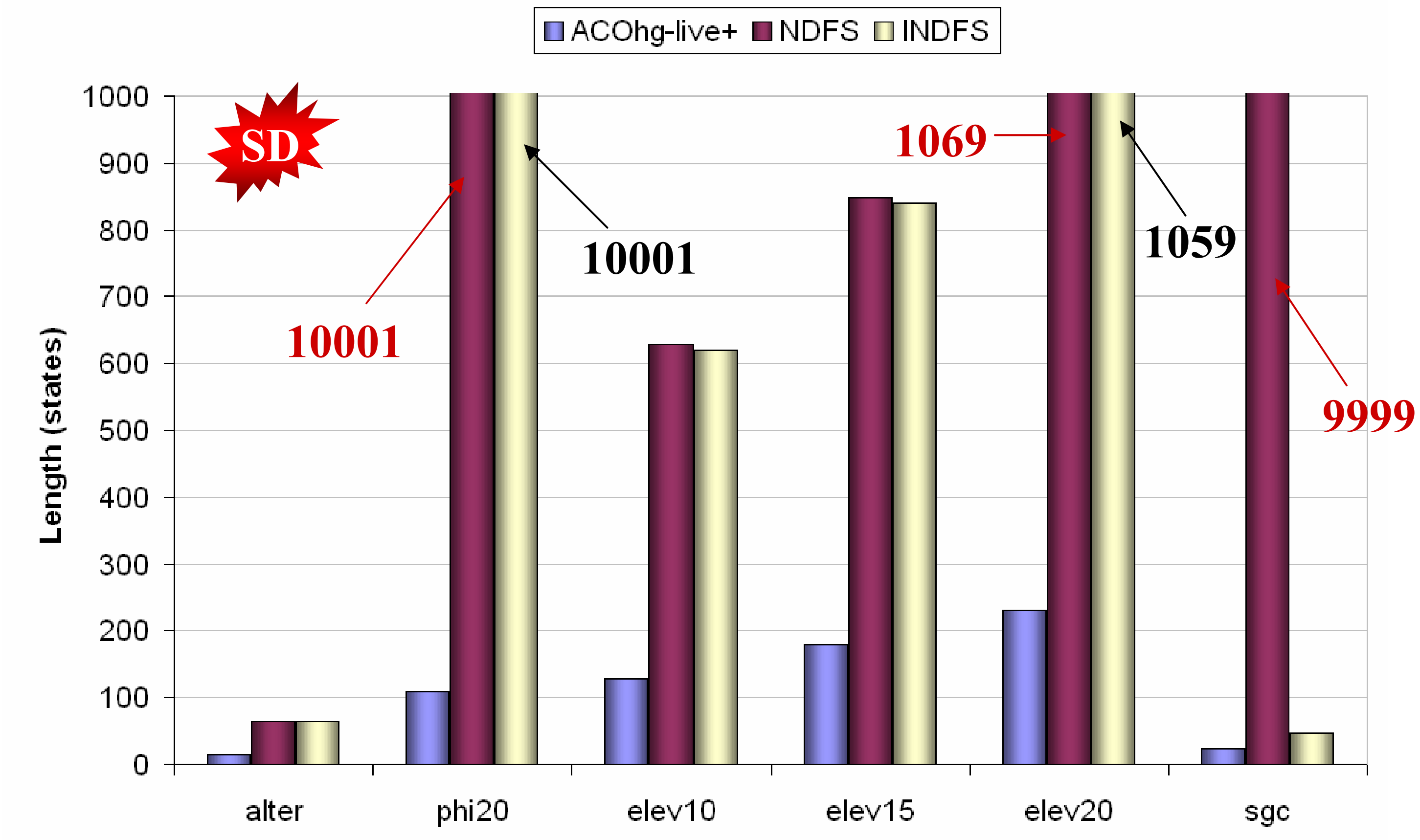
Results I: Influence of the SCC Improvement



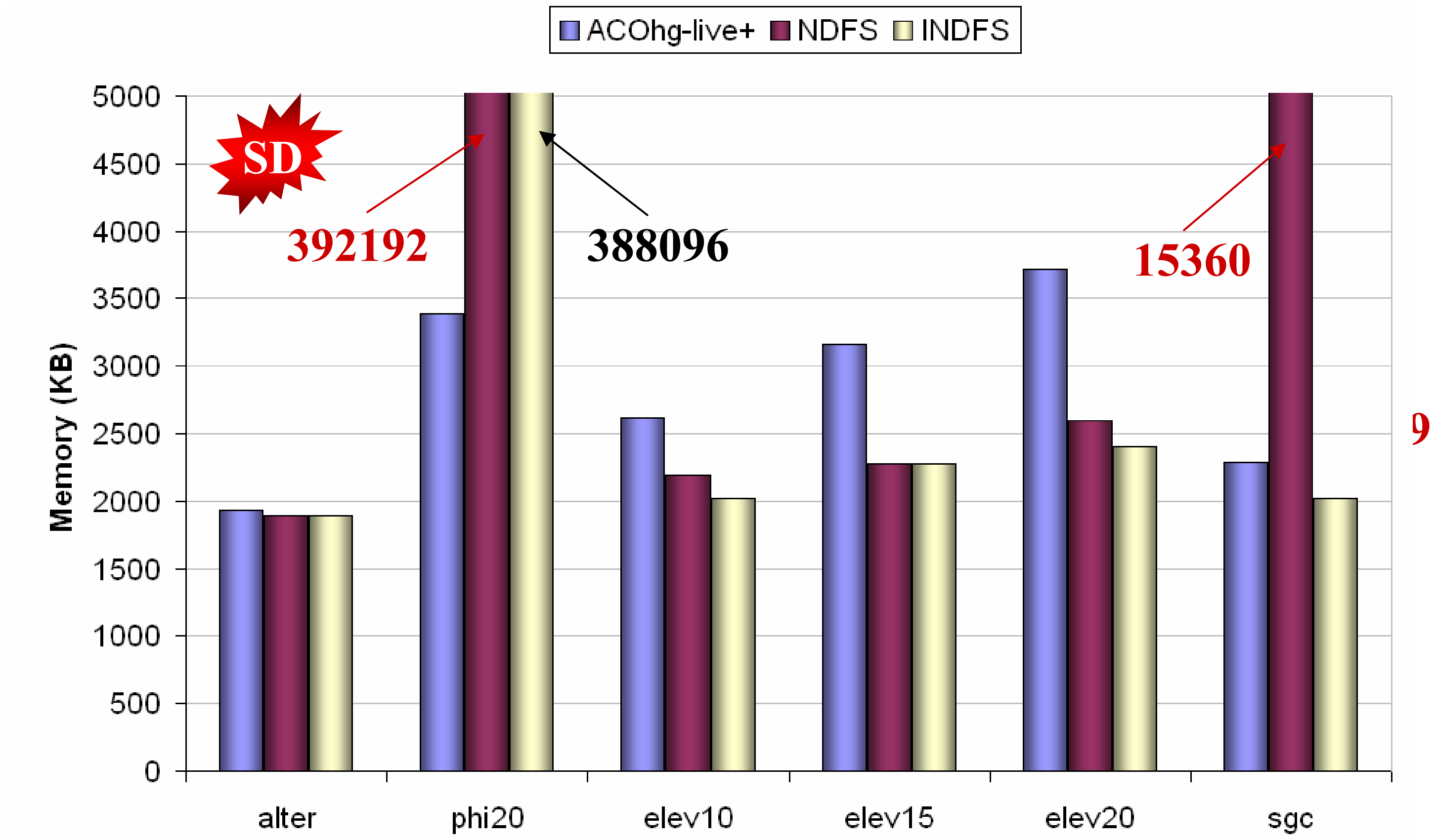
Results II: ACOhg-live^{+} vs. NDFS and INDFS

Models	ACOhg-live^{+}	NDFS	INDFS
alter	●	●	●
giop10	●	✗	✗
giop15	●	✗	✗
giop20	●	✗	✗
phi20	●	●	●
phi30	●	✗	✗
phi40	●	✗	✗
elev10	●	●	●
elev15	●	●	●
elev20	●	●	●
sgc	●	●	●

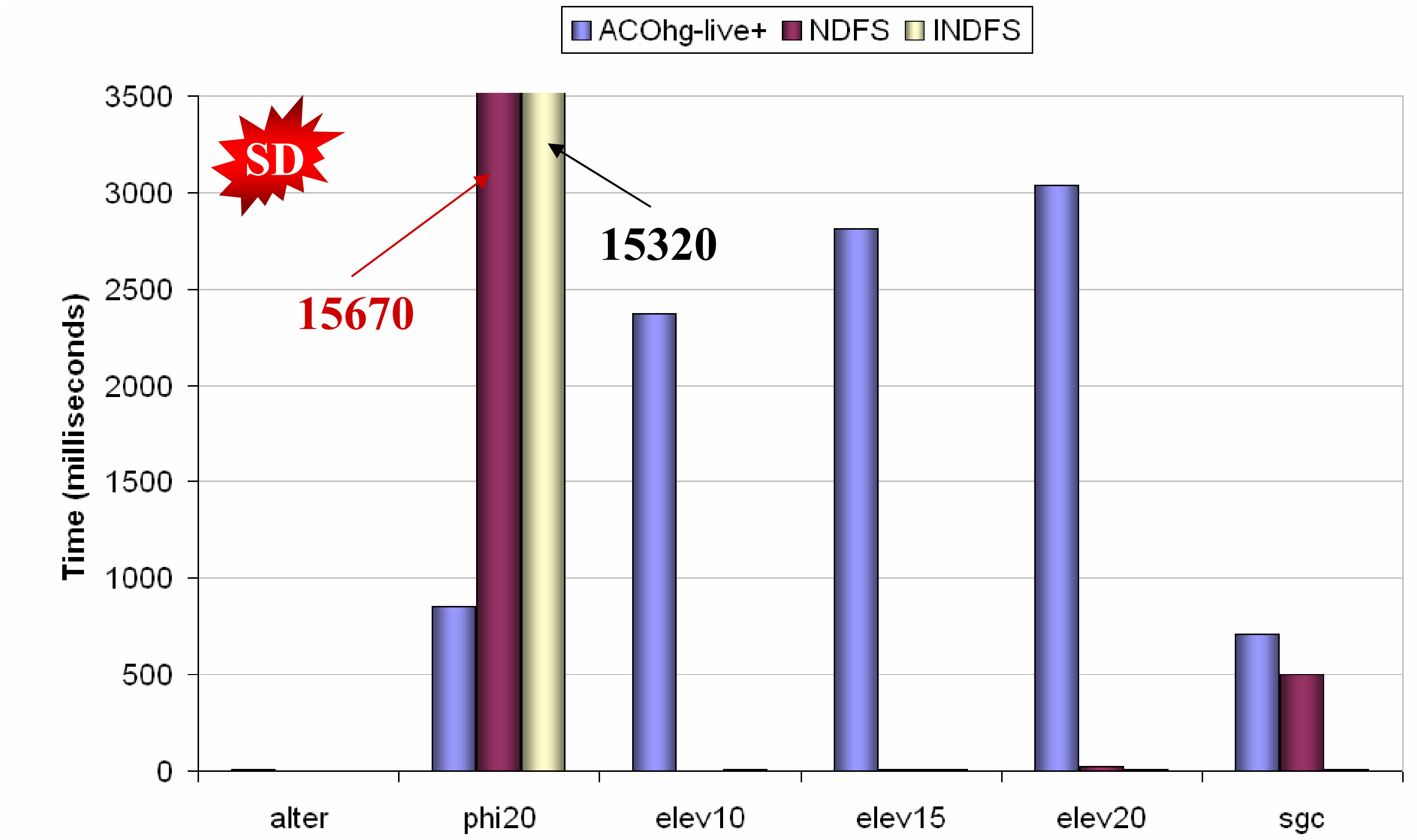
Results II: ACOhg-live^{+} vs. NDFS and INDFS



Results II: ACOhg-live^{+} vs. NDFS and INDFS

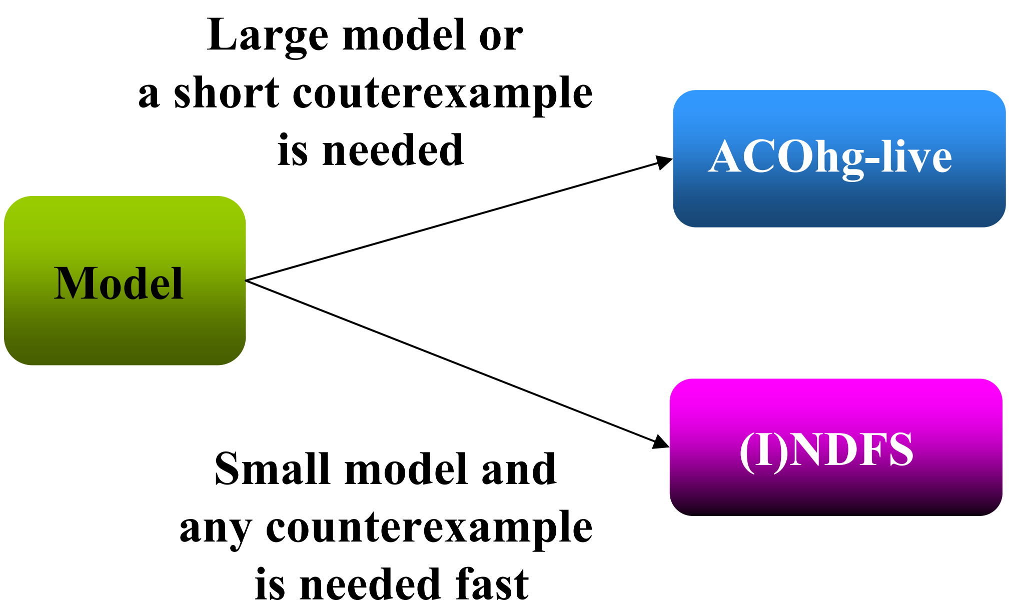


Results II: ACOhg-live^{+} vs. NDFS and INDFS



How to use ACOhg-live

- **ACOhg-live should be used in the first/middle stages of the software development, when software errors are expected**
- **ACOhg-live can also be used in other phases of the software development for testing concurrent software**



Conclusions & Future Work

Conclusions

- ACOhg-live is the first algorithm based on metaheuristics (to the best of our knowledge) applied to the search for liveness errors in concurrent models
- The improvement based on the SCCs of the never claim outperforms the efficacy of ACOhg-live
- ACOhg-live is able to outperform (Improved) Nested-DFS in efficacy and efficiency in the search for liveness errors

Future Work

- Analysis of parameterization for reducing the parameters
- Include ACOhg-live into JavaPathFinder for finding liveness errors in Java programs
- Combine ACOhg-live with techniques for reducing the memory required for the search such as partial order reduction (work in progress)

Thanks for your attention !!!