META'08

Ant Colony Optimization for Testing Concurrent Systems: Analysis of Scalability

LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE MÁLAGA

UNIVERSIDAD DE MÁLAGA

Francisco Chicano and Enrique Alba

Motivation

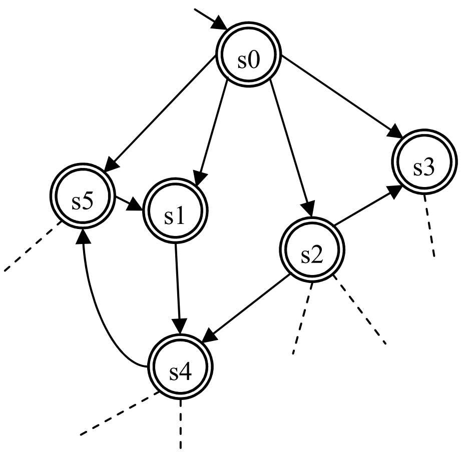
- Concurrent software is difficult to test ...
- ... and it is in the heart of a lot of critical systems

- Techniques for proving the correctness of concurrent software are required
- Model checking → fully automatic
- Traditional techniques for this purpose have problems with large models
- We analyze here the scalability of a new proposal: ACOhg-mc

Explicit State Model Checking

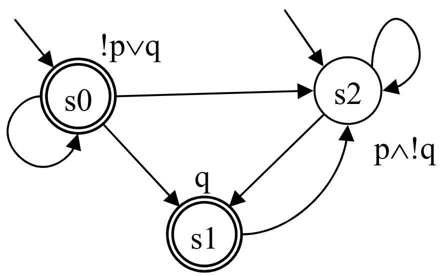
- **Objective: Prove that model M satisfies the property f: $M \models f$**
- **HSF-SPIN: the property *f* is an LTL formula**

Model M



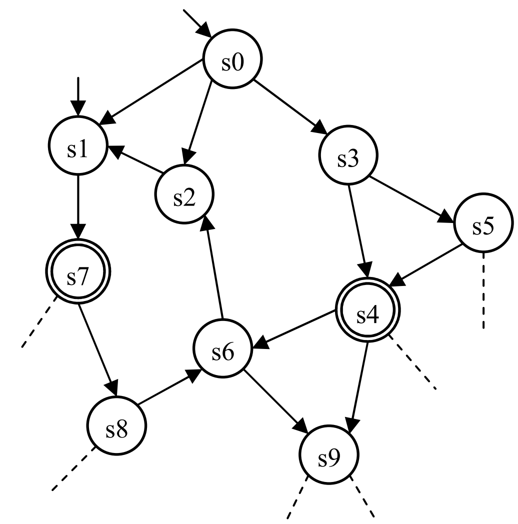
$\cap$

LTL formula $\neg f$ (never claim)



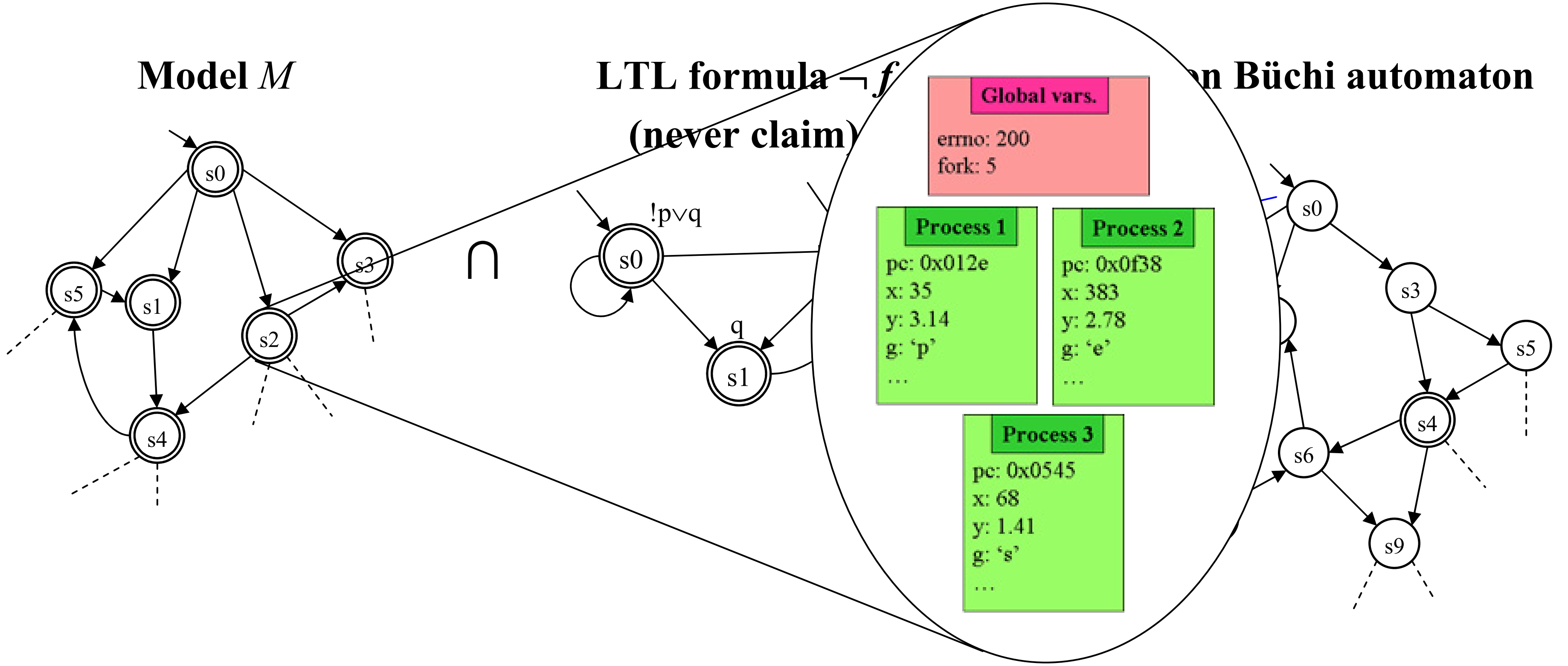
$=$

Intersection Büchi automaton



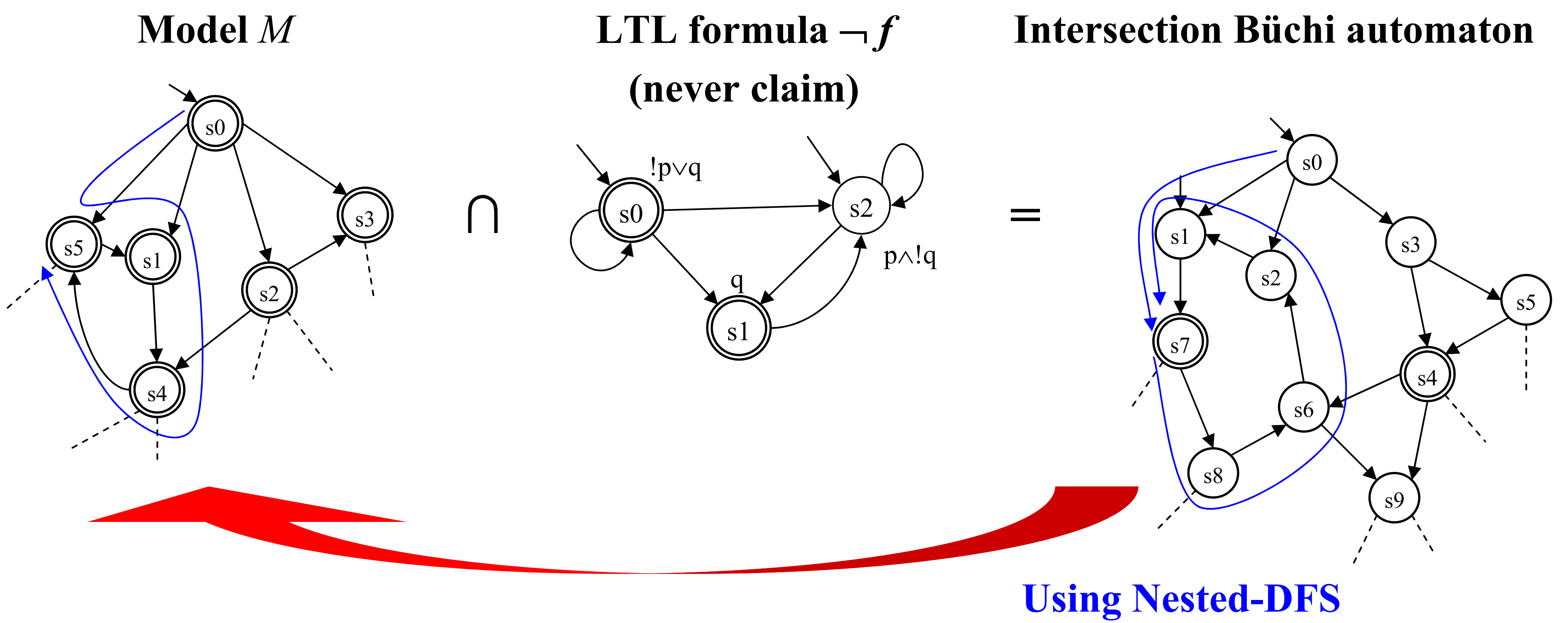
Explicit State Model Checking

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



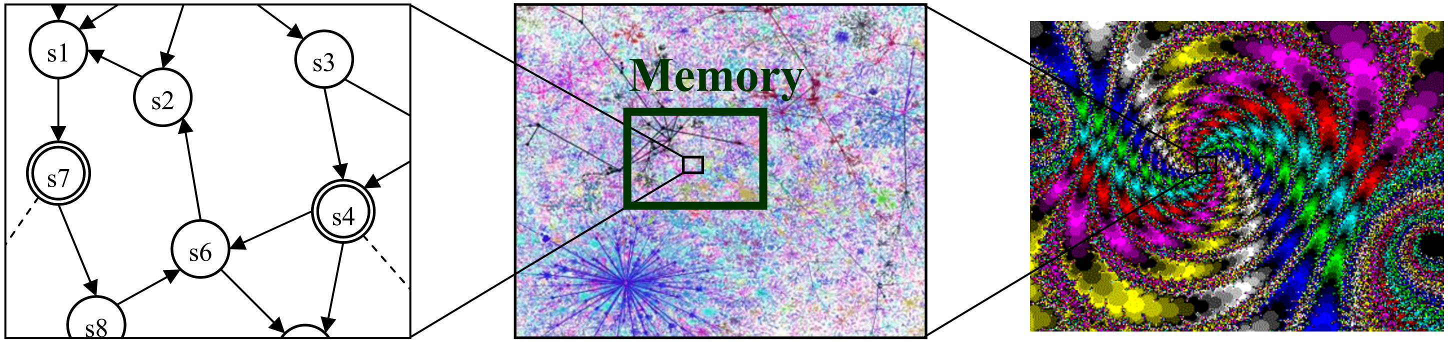
Explicit State Model Checking

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



State Explosion Problem

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



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

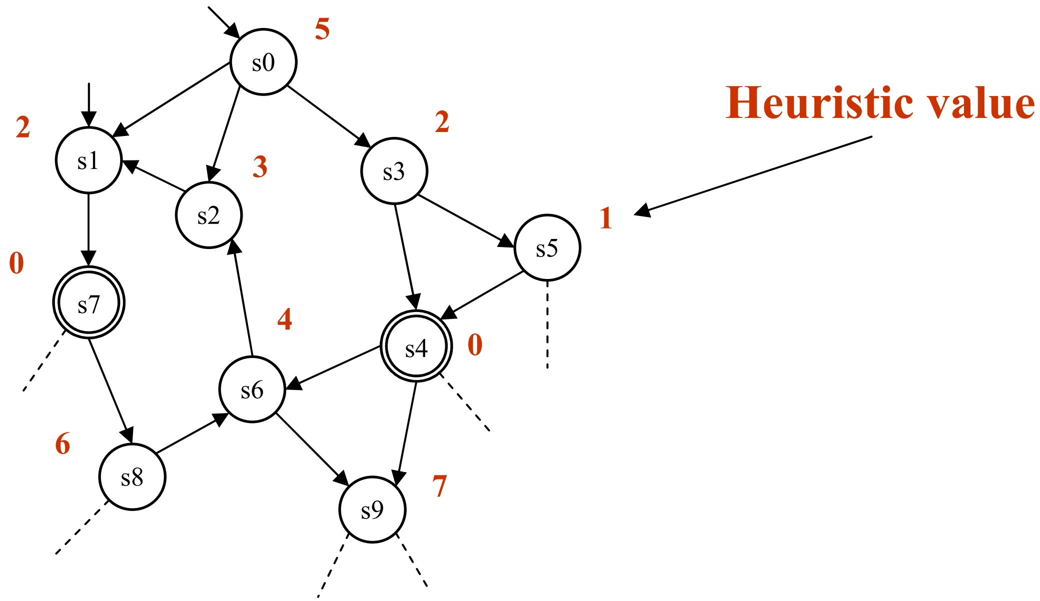
  20 philosophers → 1039 GB for storing the states

- **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

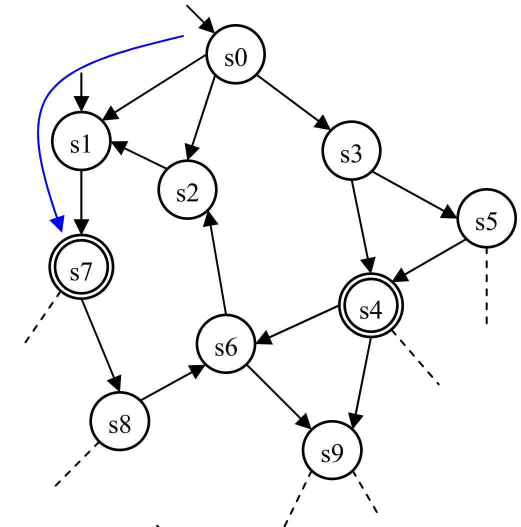


- Different kinds of heuristic functions have been proposed in the past:
  - Formula-based heuristics
  - Structural heuristics
  - Deadlock-detection heuristics
  - State-dependent heuristics

Safety and Liveness Properties

Safety property

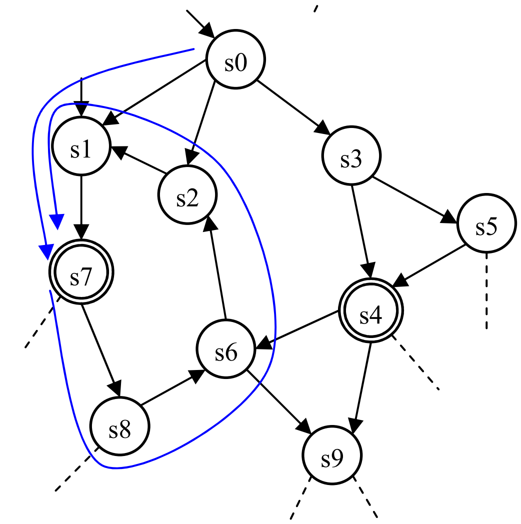
$$\forall \sigma \in S^{\omega} : \sigma \nvdash \mathcal{P} \Rightarrow (\exists i \geq 0 : \forall \beta \in S^{\omega} : \sigma_i \beta \nvdash \mathcal{P})$$



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

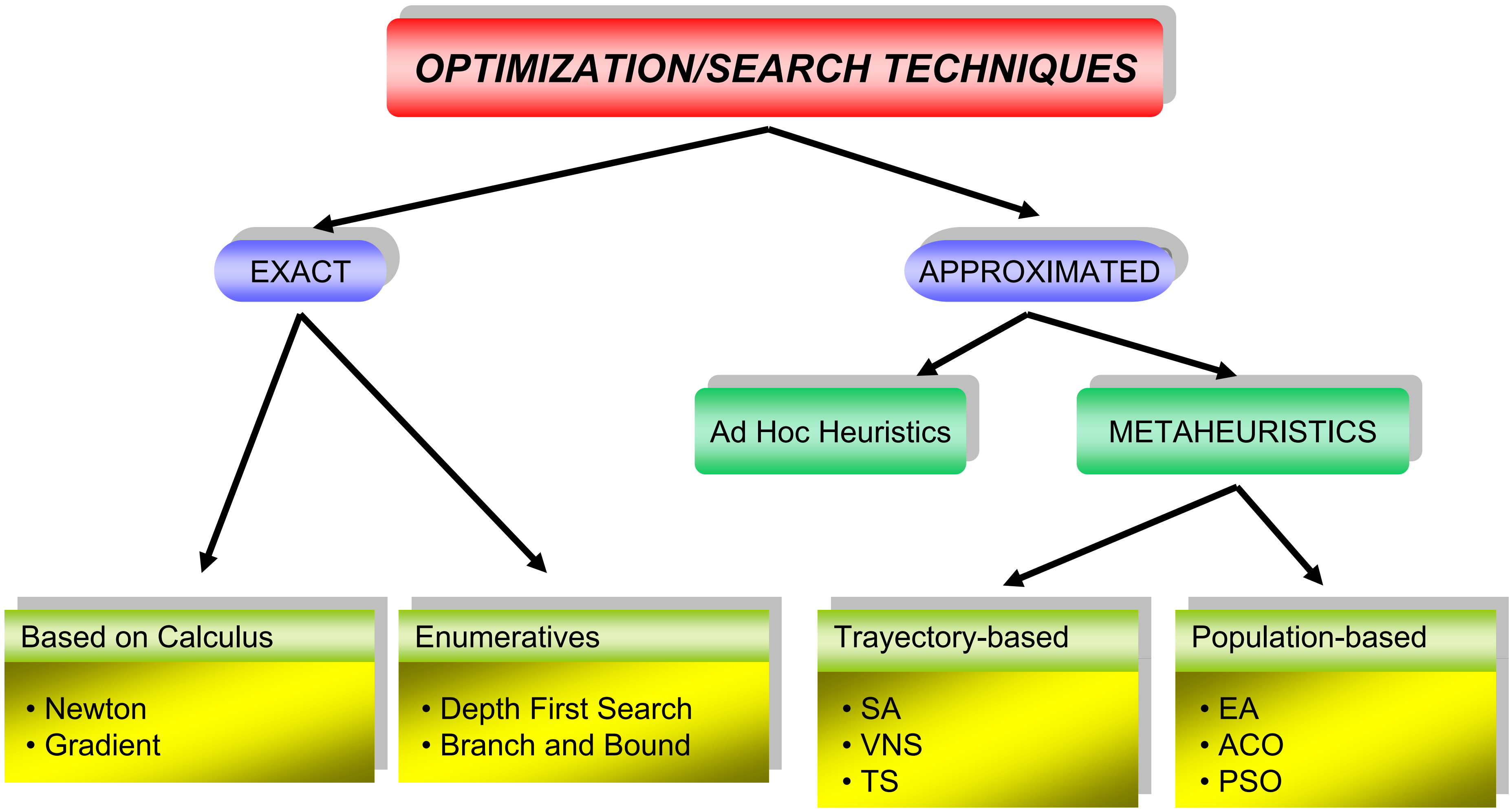
Liveness property

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



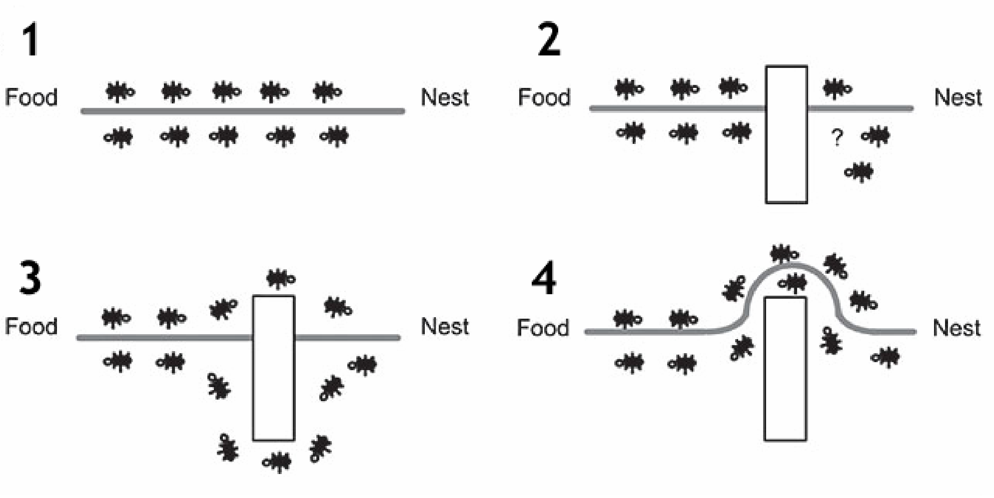
- **Counterexample ≡ path to accepting cycle**
- **It is not possible to apply DFS or BFS**

Optimization/Search Techniques



ACO: Introduction

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

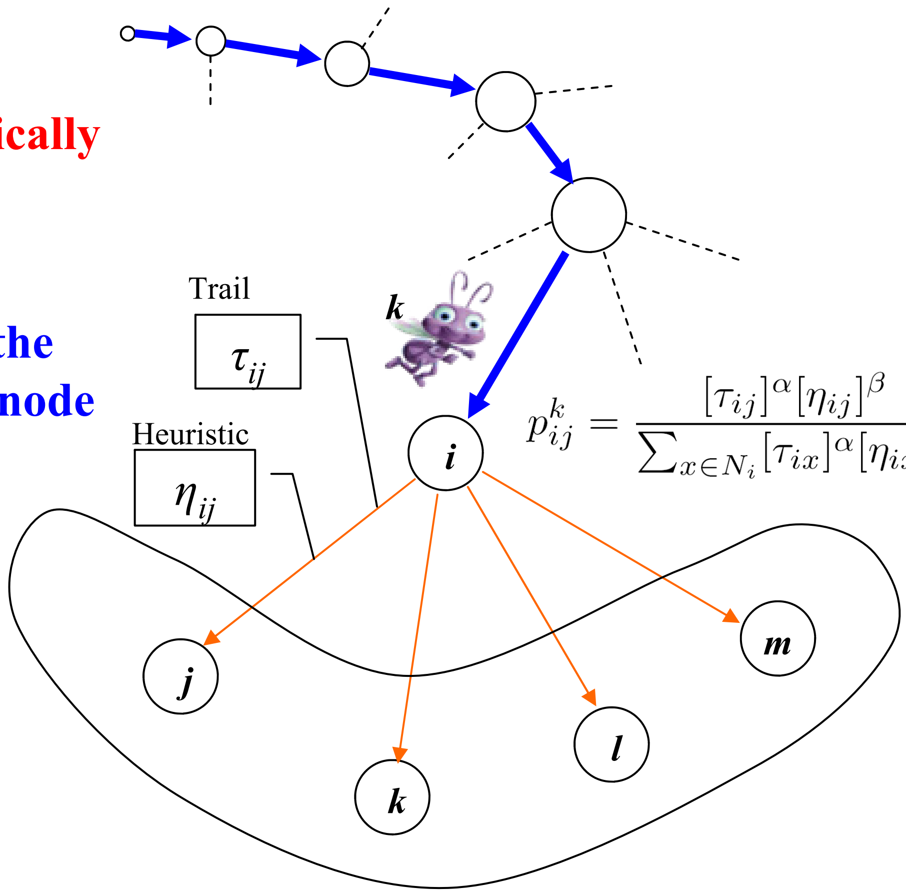


- **ACO Pseudo-code**

```
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**

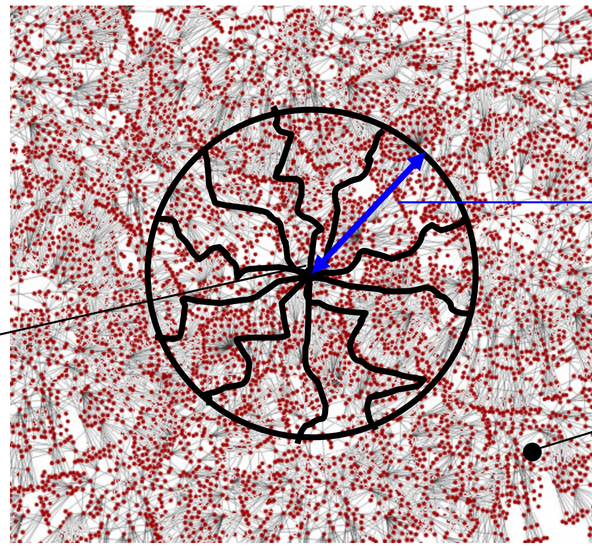
$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij} + \Delta\tau_{ij}^{bs} \quad \text{with} \quad 0 \leq \rho \leq 1$$

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

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

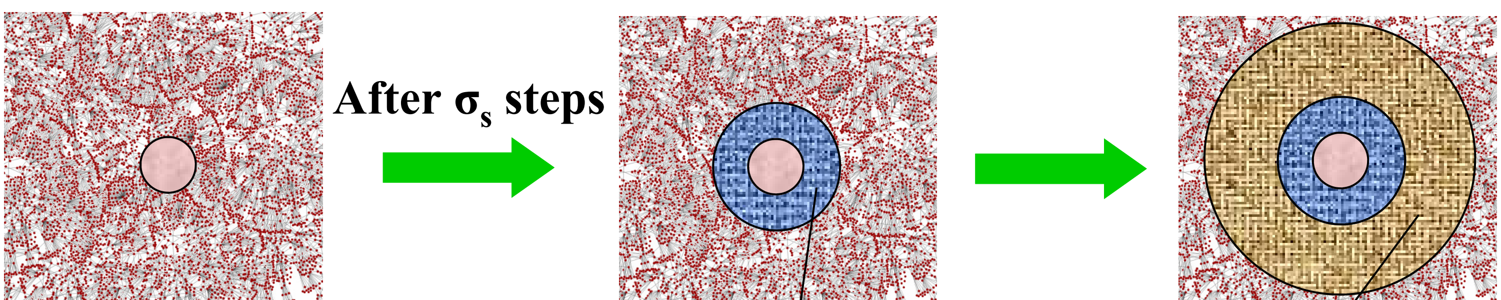
ACOhg: Huge Graphs Exploration

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



What if…?

Starting nodes for path construction change

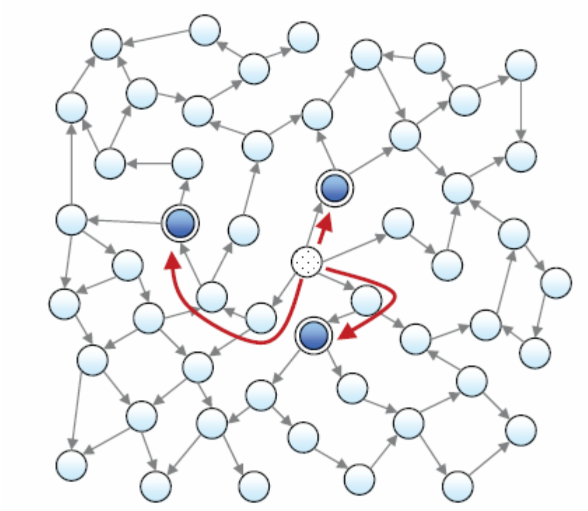


ACOhg-mc

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

ACOhg-mc Pseudocode

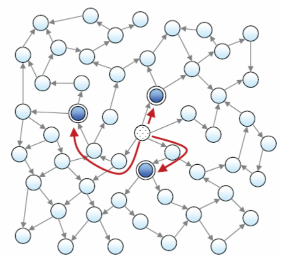
```
repeat
   accpt = acohg1.findAcceptingStates(); {First phase}
   for node in accpt do
      acohg2.findCycle(node); {Second phase}
      if acohg2.cycleFound() then
         return acohg2.acceptingPath();
      end if
   end for
   acohg1.insertTabu(accpt);
until empty(accpt)
return null;
```



First phase

ACOhg-mc

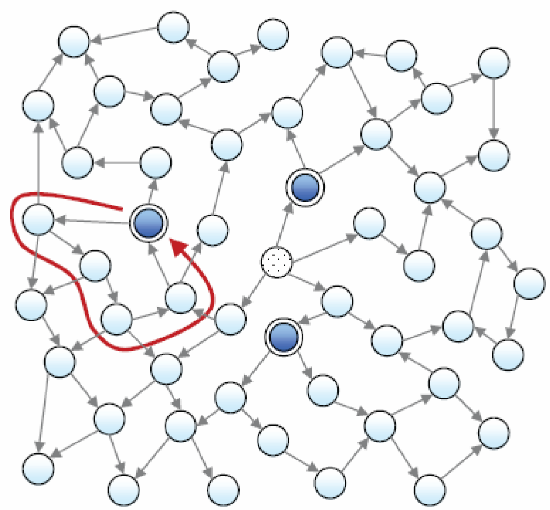
- **The search is an alternation of two phases**



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

ACOhg-mc Pseudocode

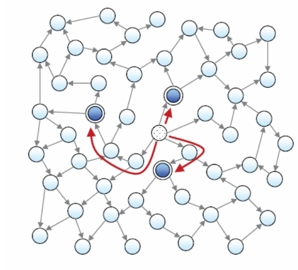
```
repeat
   accpt = acohg1.findAcceptingStates(); {First phase}
   for node in accpt do
      acohg2.findCycle(node); {Second phase}
      if acohg2.cycleFound() then
         return acohg2.acceptingPath();
      end if
   end for
   acohg1.insertTabu(accpt);
until empty(accpt)
return null;
```

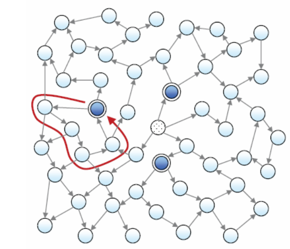


Second phase

ACOhg-mc

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

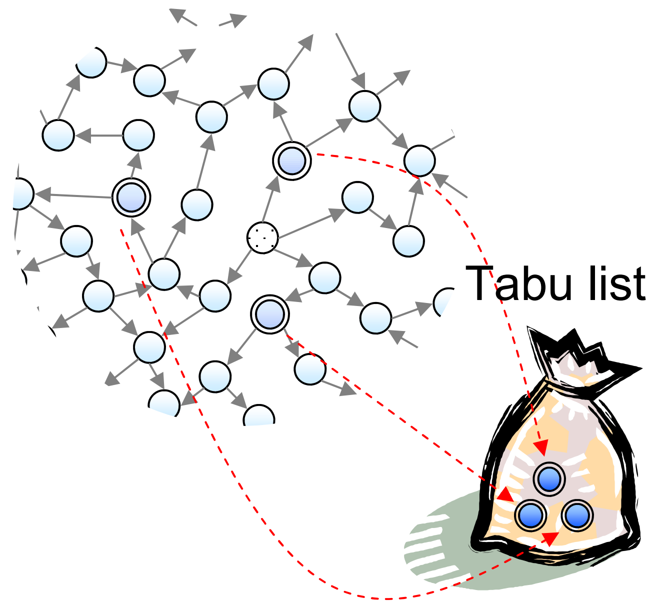




  - **First phase:** **search for accepting states**
  - **Second phase:** **search for cycles from the accepting states**

ACOhg-mc 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;
```



Promela Models

- We used 4 scalable Promela models for the experiments

Model	LoC	Processes	Property
`marriersj`	64	j+1	deadlock
`elevj`	191	j+3	$\square(p \rightarrow \Diamond q)$
`giopj`	740	j+6	deadlock and $\square(p \rightarrow \Diamond q)$
`phij`	57	j+1	deadlock and $\square(p \rightarrow \Diamond q)$

- Parameters for ACOhg-mc

<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>40</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-mc implemented in HSF-SPIN
- 100 independent executions

Promela Models

- **We used 4 scalable Promela models for the experiments**

j=2 to 30

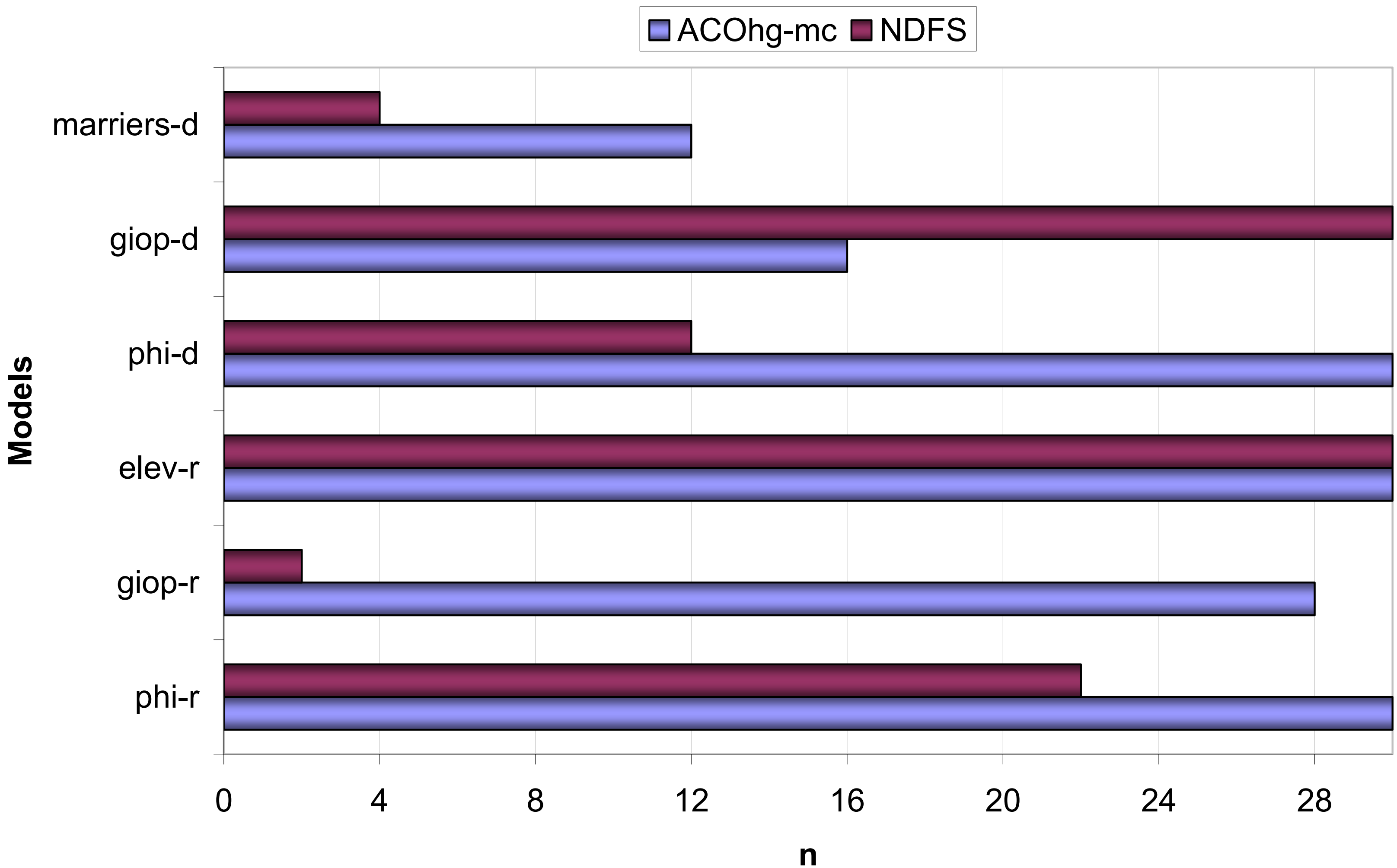
Model	LoC		Property
`marriersj`	64	j+1	deadlock
`elevj`	191	j+3	□(p → ◊q)
`giopj`	740	j+6	deadlock and □(p → ◊q)
`phij`	57	j+1	deadlock and □(p → ◊q)

- **Parameters for ACOhg-mc**

Parameter	*msteps*	*colsize*	λ_{ant}	σ_s	ξ	a	ρ	α	β
1st phase	100	10	40	4	0.7	5	0.2	1.0	2.0
2nd phase		20	4		0.5				

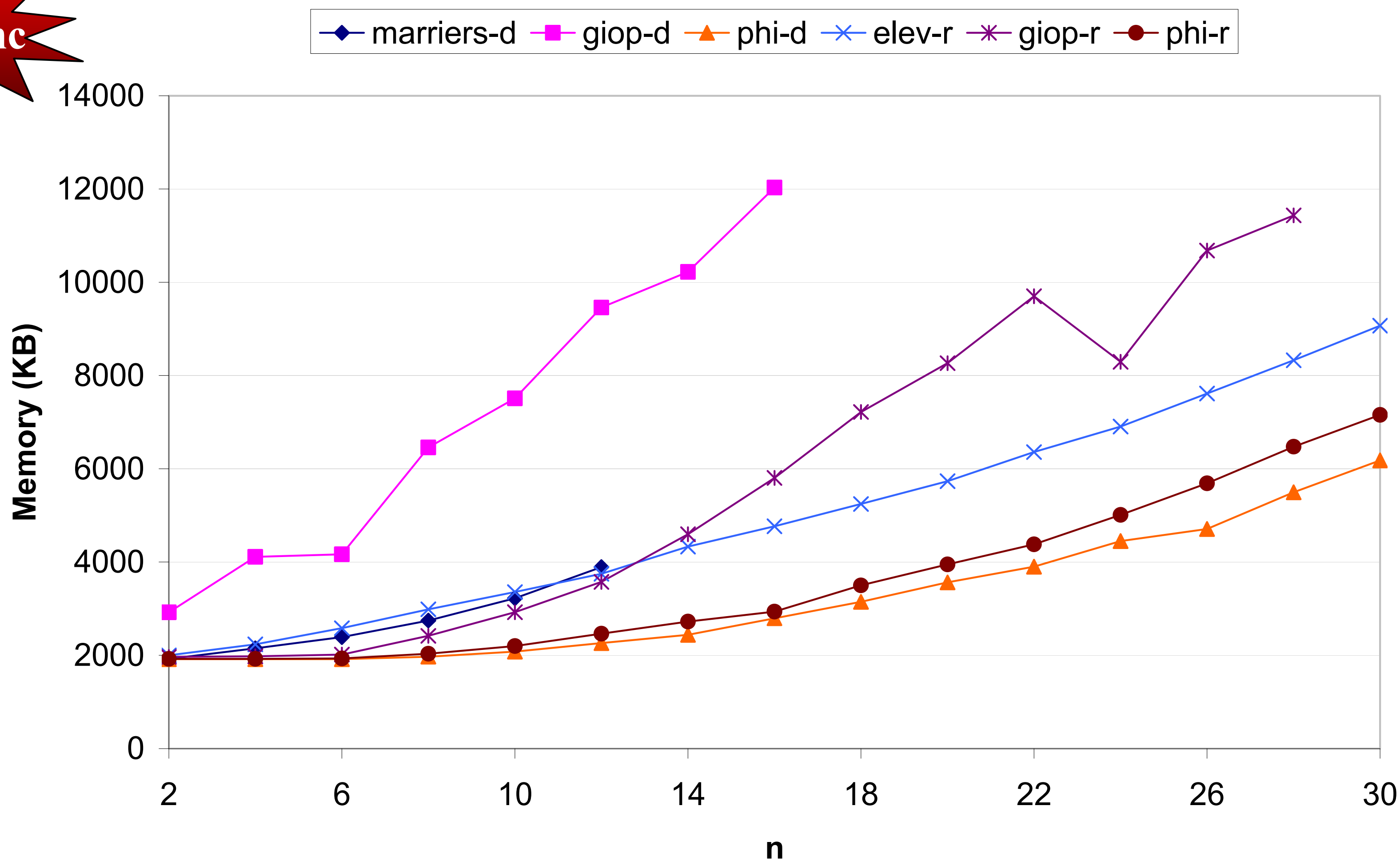
- **Formula-based and finite state machine heuristics**
- **ACOhg-mc implemented in HSF-SPIN**
- **100 independent executions**

Efficacy

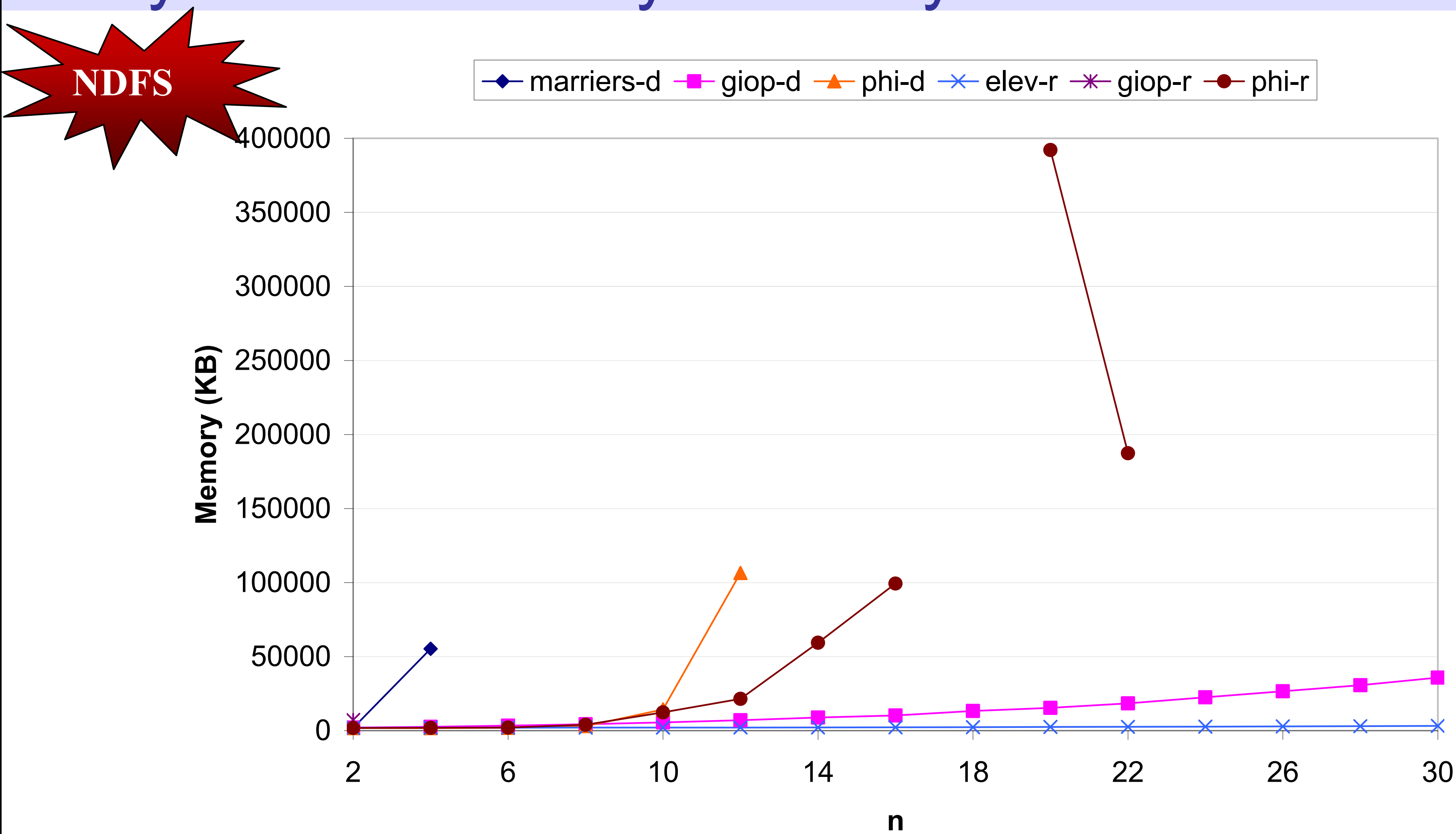

ACOhg-mc
NDFS
marriers-d
giop-d
phi-d
elev-r
giop-r
phi-r
Models
0
4
8
12
16
20
24
28
n

Analysis of Scalability: Memory

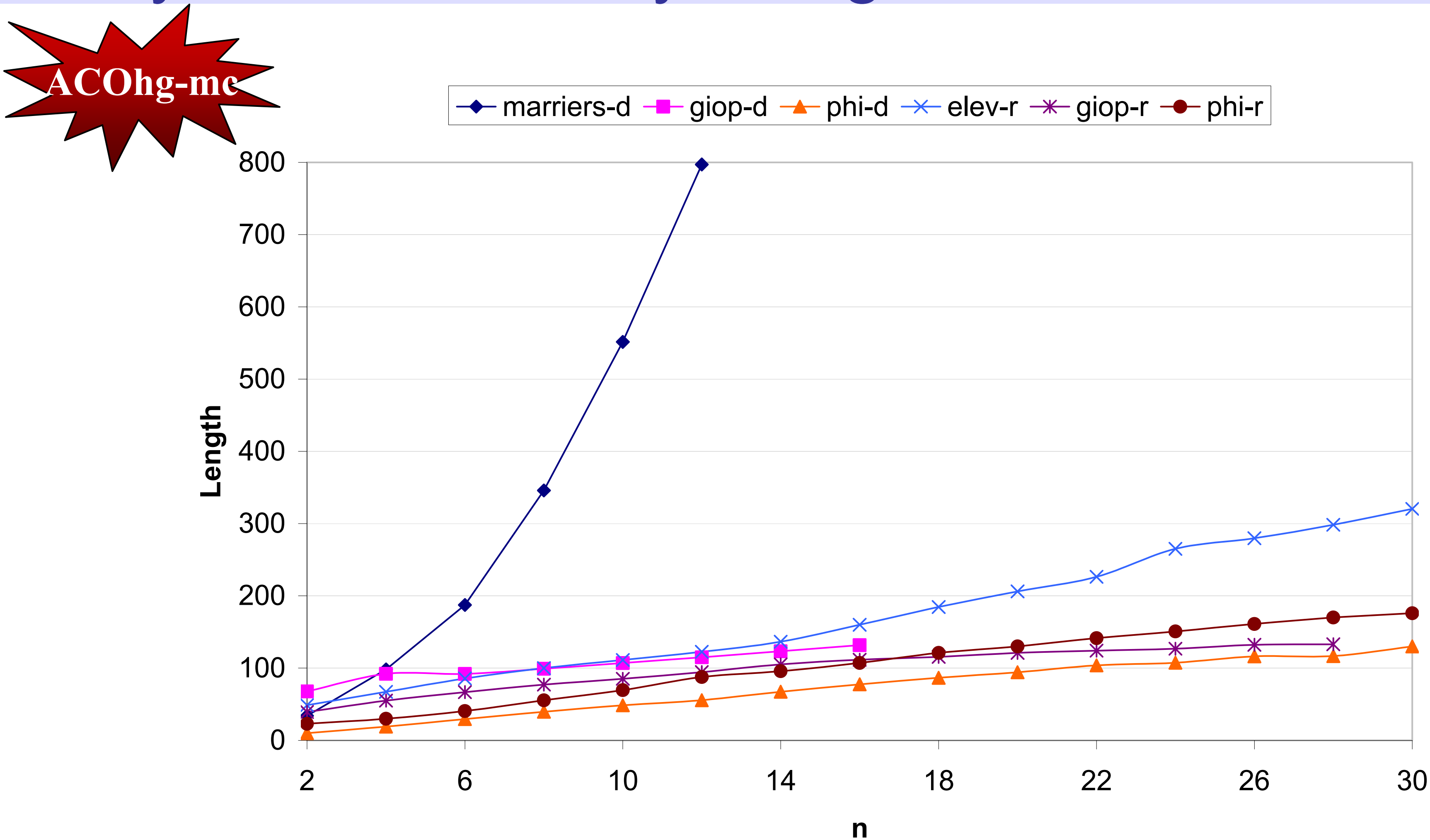
ACOhg-mc



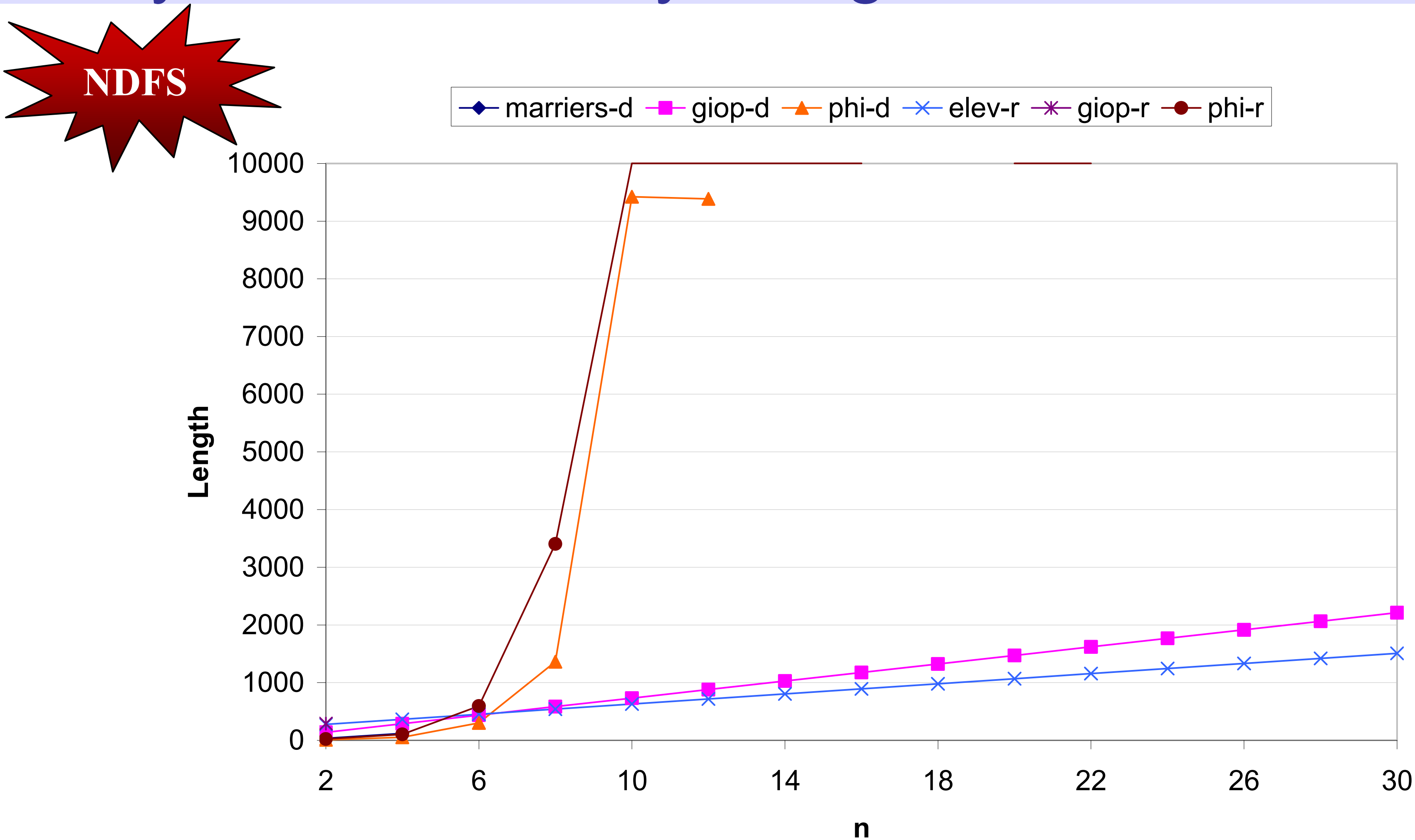
Analysis of Scalability: Memory



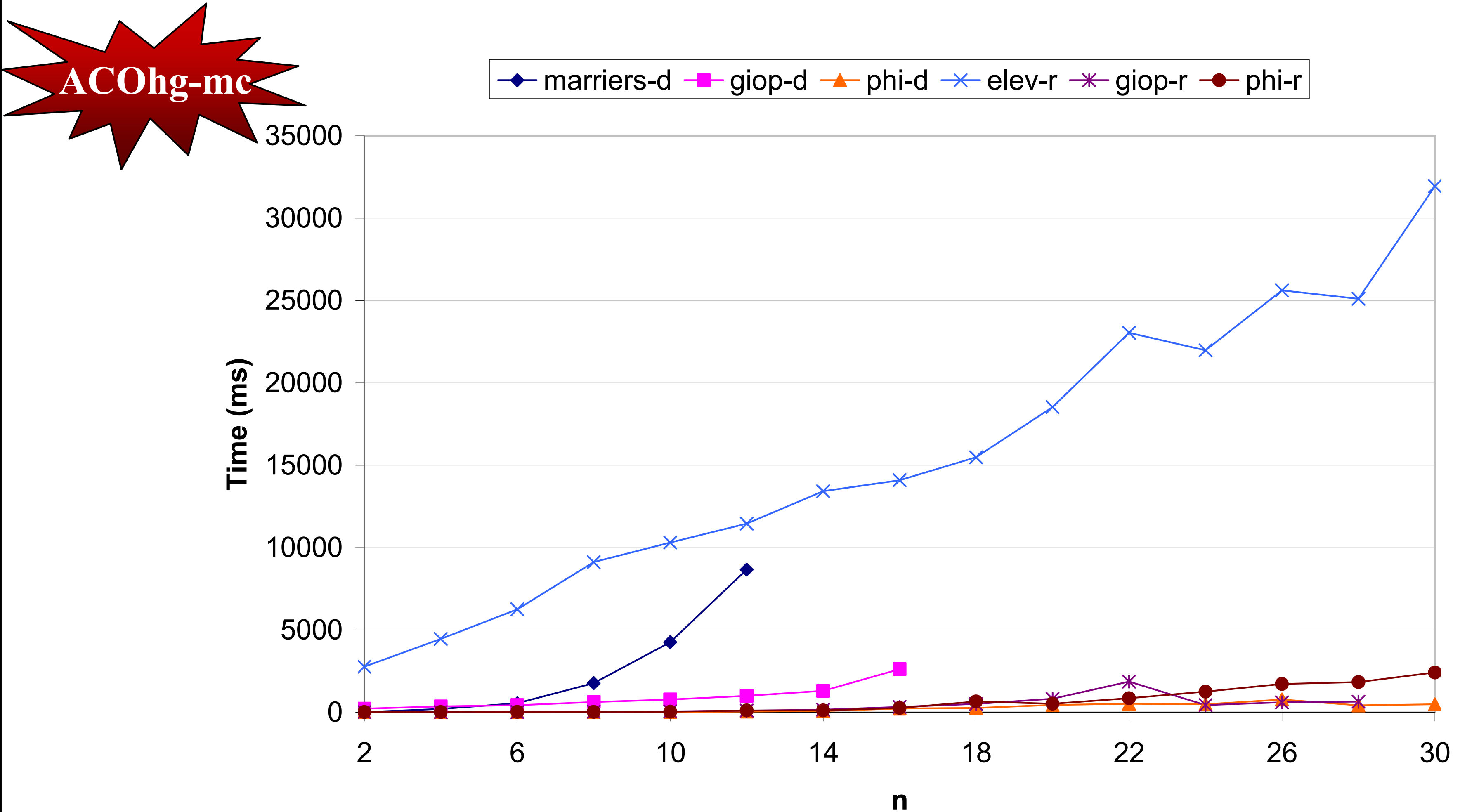
Analysis of Scalability: Length



Analysis of Scalability: Length

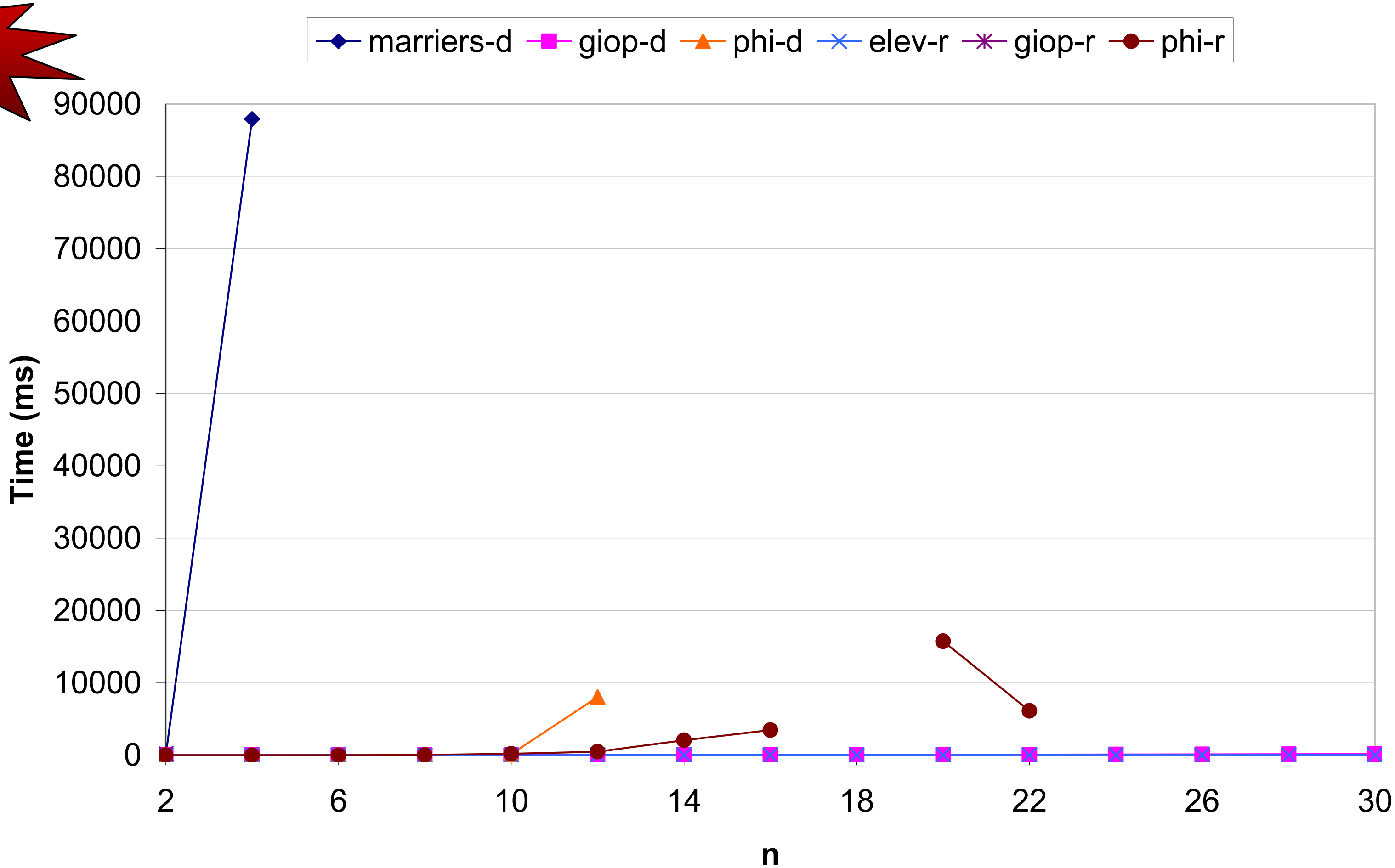


Analysis of Scalability: CPU time



Analysis of Scalability: CPU time

NDFS



Conclusions & Future Work

Conclusions

- ACOhg-mc is able to find errors in large models for which NDFS fails
- The memory required by ACOhg-mc for the search is small and grows very slowly
- The length of the error trails increases linearly in most of the cases
- Although ACOhg-mc is not always the fastest algorithm, the time required is small

Future Work

- Analysis of parameterization for reducing the parameters
- Include ACOhg-mc into Java PathFinder for finding errors in Java programs
- Combine ACOhg-mc with techniques for reducing the memory required for the search such as partial order reduction

Thanks for your attention !!!