Finding Safety Errors with ACO

LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE MÁLAGA

UNIVERSIDAD DE MÁLAGA

Enrique Alba and Francisco Chicano

Motivation

- Nowadays software is very complex

- An error in a software system can imply the loss of lot of money …

… and even human lifes

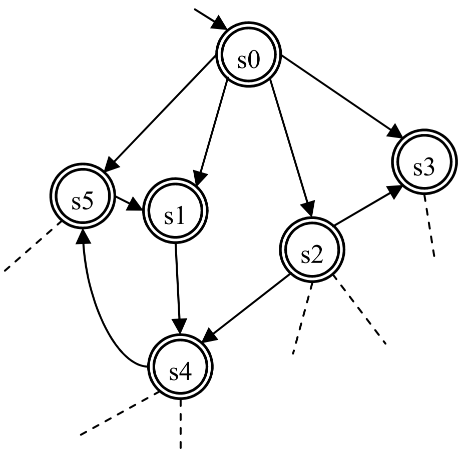
- Techniques for proving the correctness of the software are required

- Model checking → fully automatic

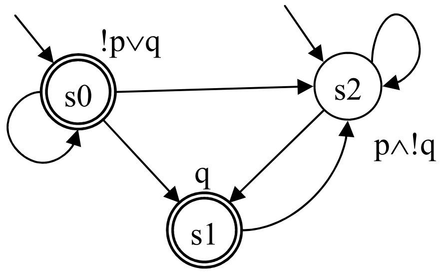
Explicit State Model Checking

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

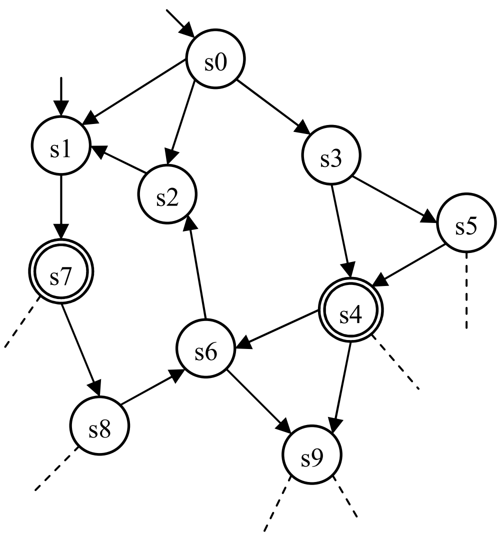
Model *M* **LTL formula $\neg f$** **Intersection Büchi automaton**



$\cap$

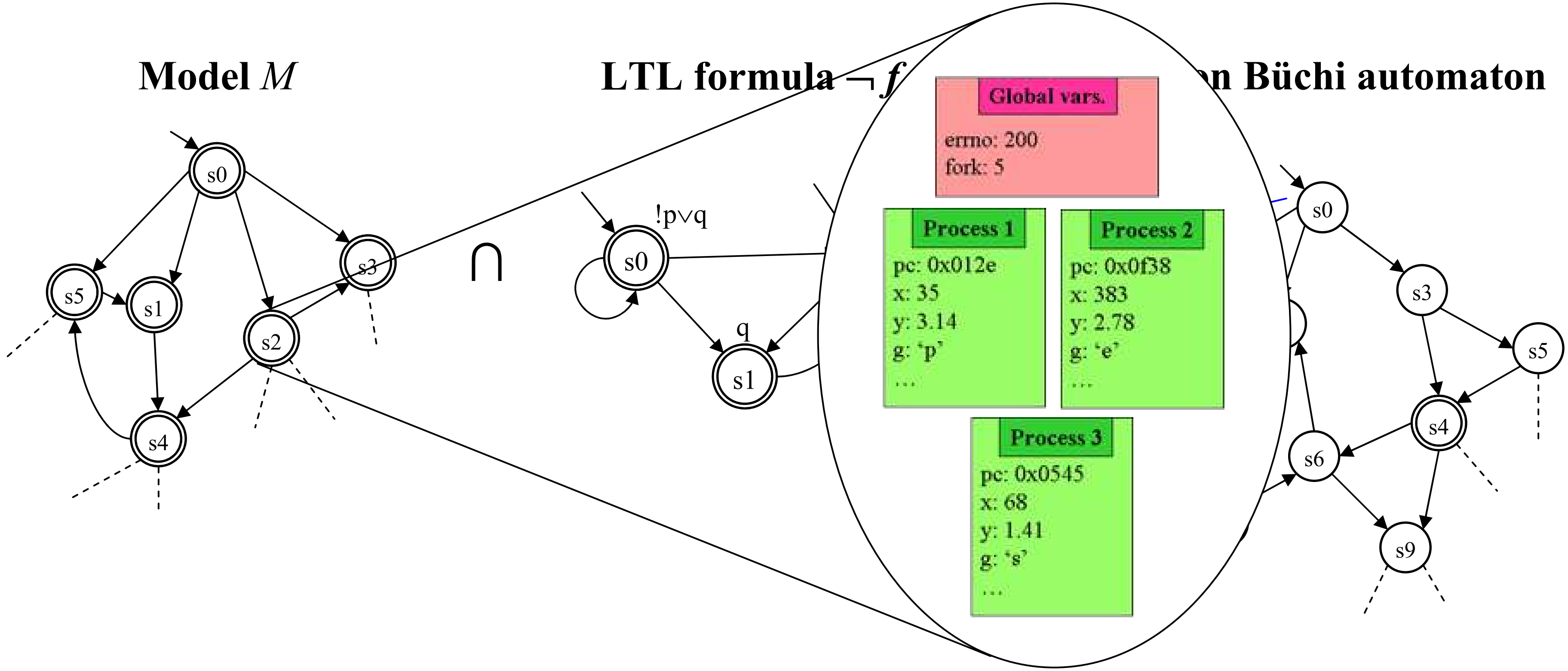


$=$



Explicit State Model Checking

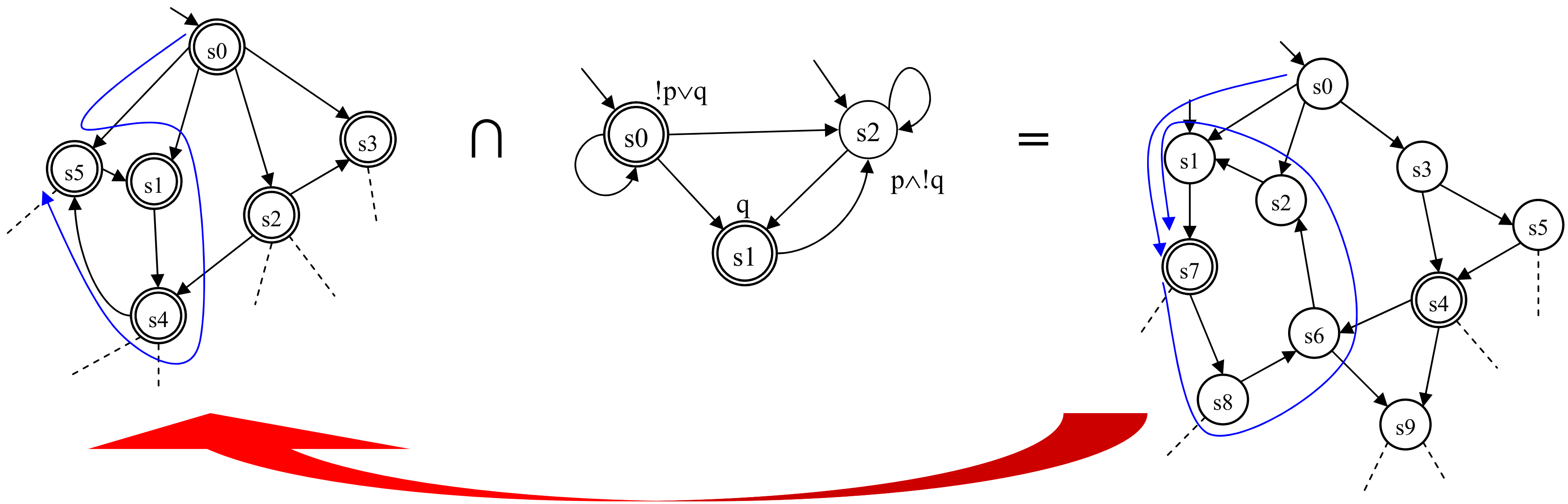
- **Objective: Prove that model M satisfies the property f: $M \models f$**
- **SPIN: the property *f* is an LTL formula**



Explicit State Model Checking

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

Model *M* **LTL formula ¬*f*** **Intersection Büchi automaton**



Using Nested-DFS

Safety Properties

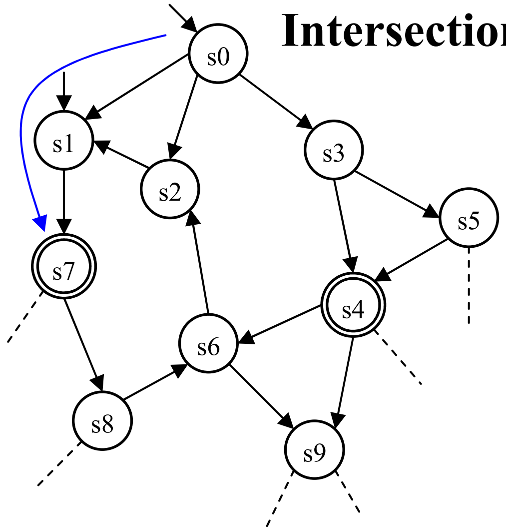
- **Safety properties are those expressed by an LTL formula of the form:**

$$f = \Box p$$

where *p* is a past formula

- **Finding one counterexample ≡ finding one accepting state**

Intersection automaton

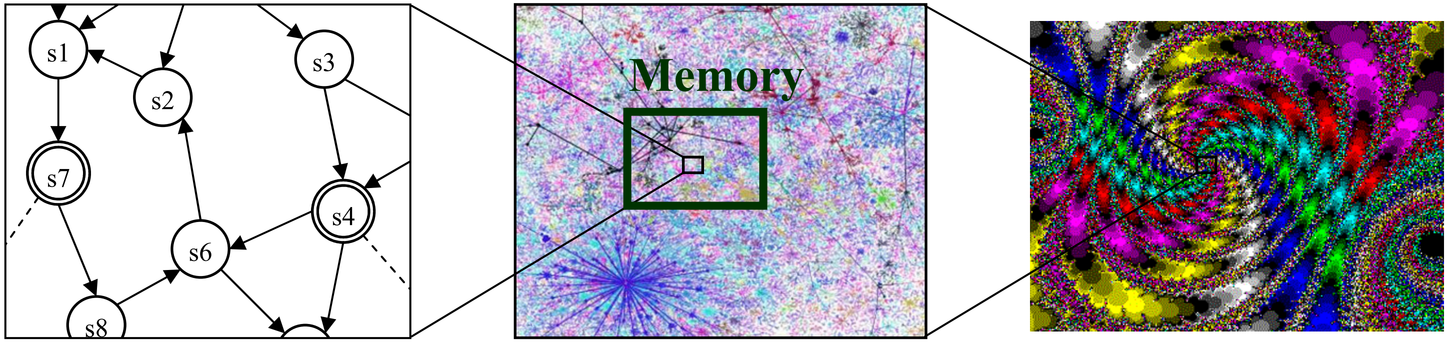


Safety Properties
Deadlocks
Invariants
Assertions
…

- **Classical algorithms for graph exploration can be used: DFS and BFS**

State Explosion Problem

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



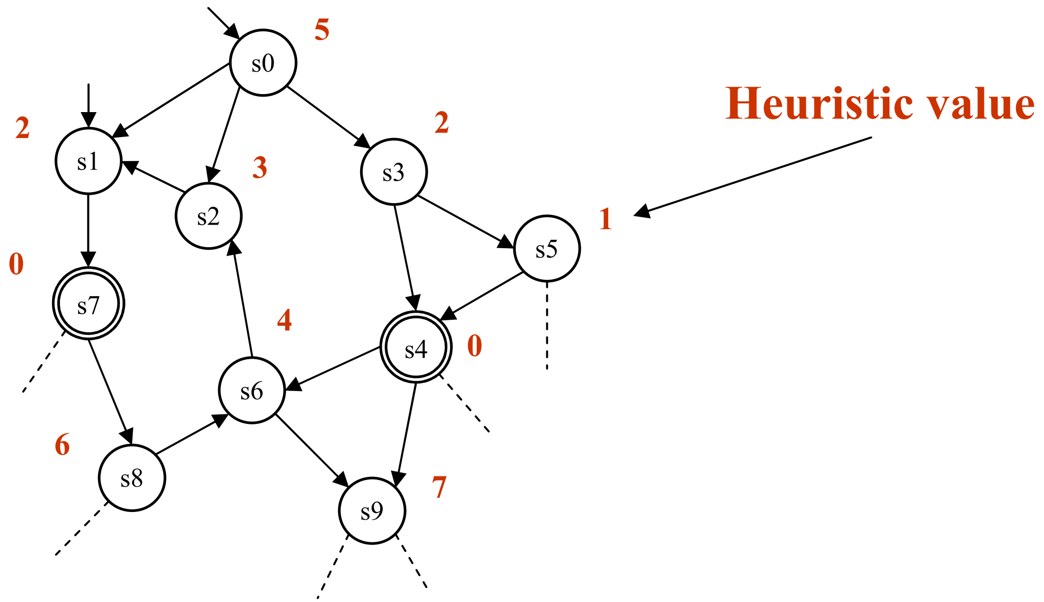
- **Example: Dining philosophers with *n* philosophers → 3^n states**
  20 philosophers → 1039 GB for storing the states

- **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 heuristics using algorithms like A*, IDA*, WA* and Best-First



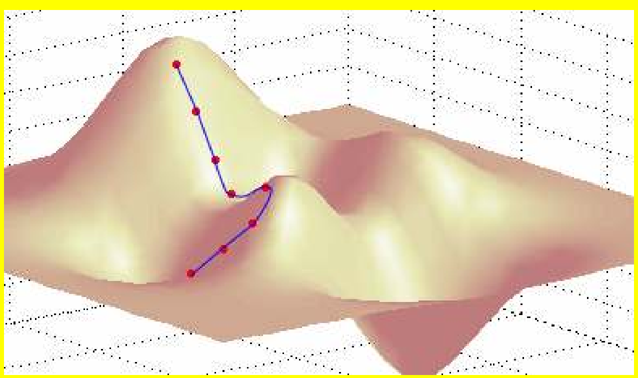
- Different kinds of heuristic functions have been proposed in the past:
  - Formula-based heuristics
  - Structural heuristics
  - Deadlock-detection heuristics
  - State-dependent heuristics

Metaheuristic Algorithms

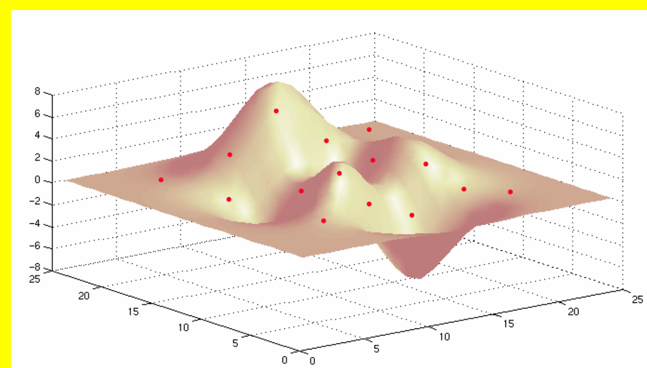
- **Designed to solve optimization problems**
  - **Maximize or minimize a given function: the fitness function**
- **They can find "good" solutions with a "reasonable" amount of resources**

Metaheuristic Algorithms

Single solution



Population



Metaheuristics Classification

Single solution

- Greedy Randomized Adaptive Search Procedure
- Iterated Local Search
- Variable Neighborhood Search
- Tabu Search
- Simulated Annealing
- Iterative Improvement
- Guided Local Search

Population

- Estimation of Distribution Algorithms
- Evolutionary Computation
- Scatter Search
- Ant Colony Optimization
- Particle Swarm Optimization

Metaheuristics Classification

Single solution

- Greedy Randomized Adaptive Search Procedure
- Iterated Local Search
- Variable Neighborhood S
- Tabu Search
- Simulated Annealing
- Iterative Improvement
- Gu

Genetic Algorithms

Alba & Troya, 1996

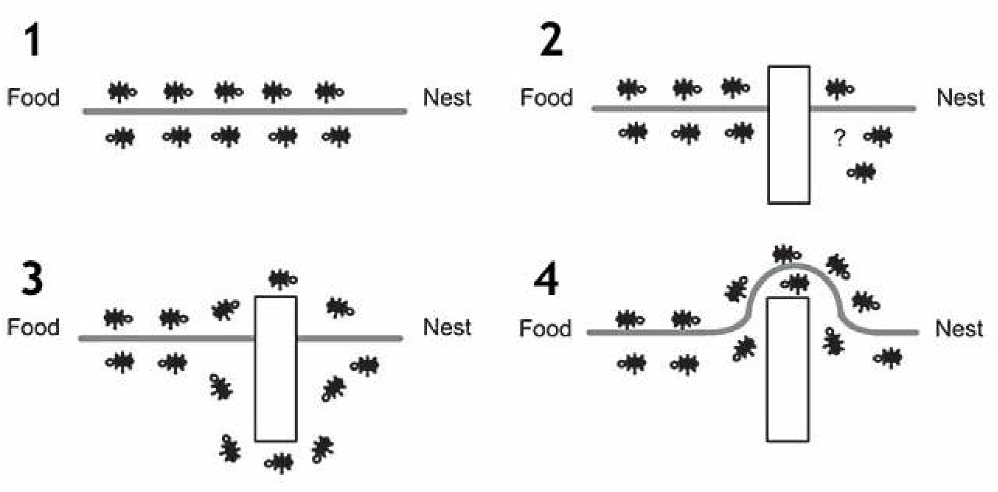
Godefroid & Khurshid, 2002, 2004

Population

- Estimation of Distribution Algorithms
- Evolutionary Computation
- Scatter Search
- Ant Colony Optimization
- Particle Swarm Optimization

ACO: Introduction

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

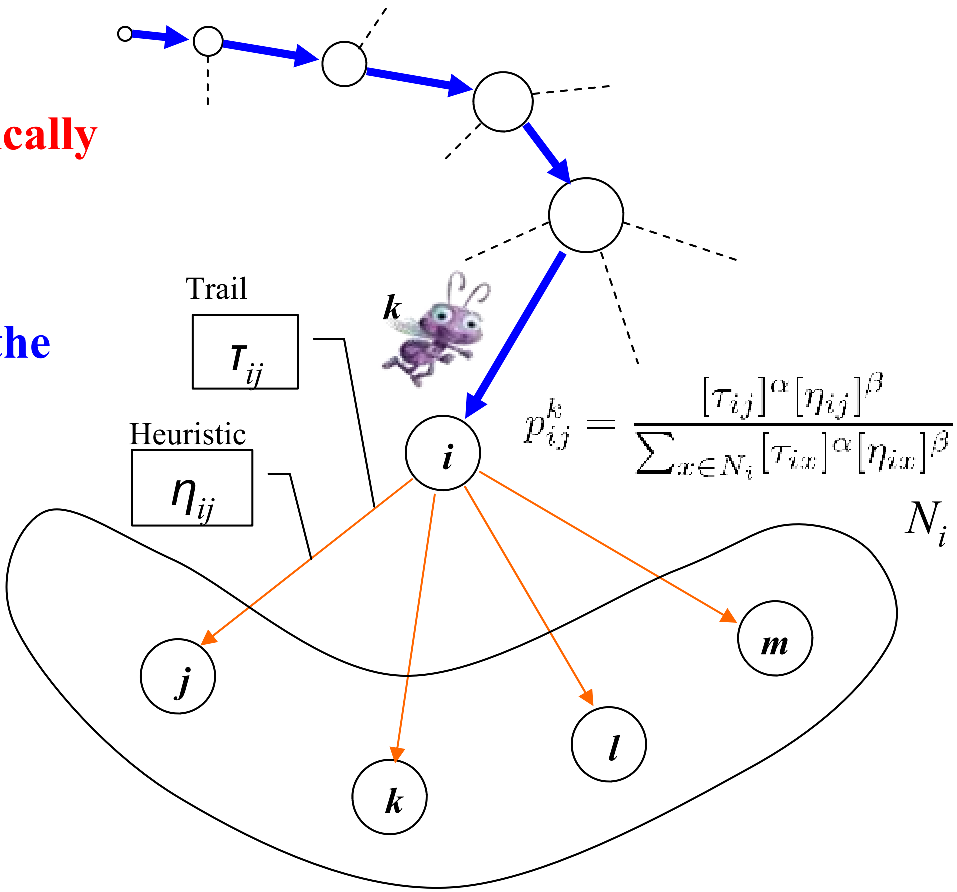


- **ACO Pseudo-code**

```
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
- 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**

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

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

$$\tau_{max} = \frac{Q}{1-\rho} \qquad \tau_{min} = \frac{\tau_{max}}{a}$$

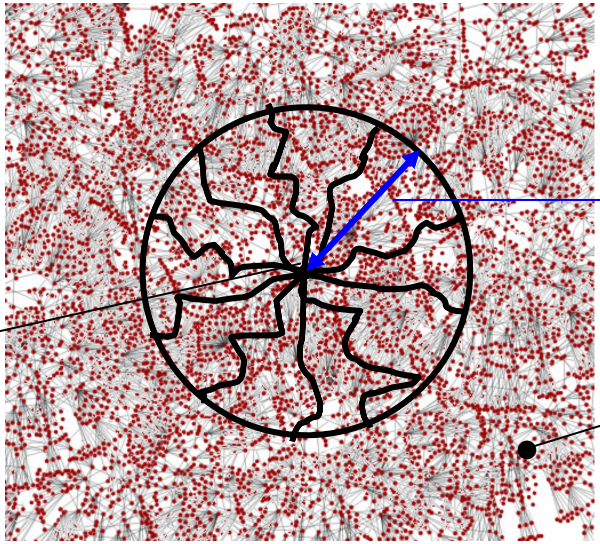
ACOhg: Motivation

- **Existing ACO models cannot be applied to the search for errors in concurrent programs**
  - **The graph is very large, the construction of a complete solution could require too much time and memory**
  - **In some models the number of nodes of the graph is used for computing the initial pheromone values**

- **We need a new model for tackling these problems: ACOhg (ACO for Huge Graphs)**
  - **Constructs the ant paths and updates the pheromone values in the same way as the traditional models**
  - **Allows the construction of partial solutions**
  - **Allows the exploration of the graph using a bounded amount of memory**
  - **The pheromone matrix is never completely stored**

ACOhg: Huge Graphs Exploration

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

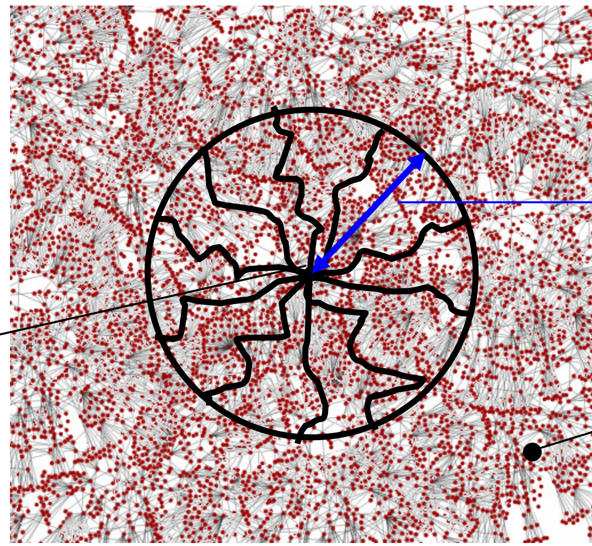


What if…?

ACOhg: Huge Graphs Exploration

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

Initial node



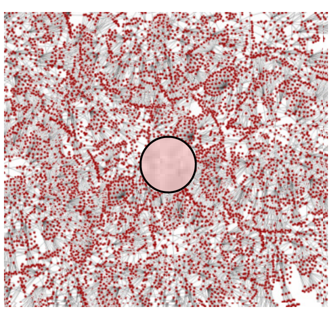
λ_{ant}

What if…?

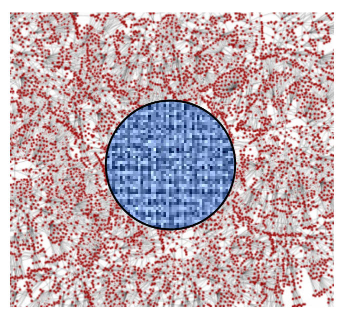
Objective node

Two alternatives

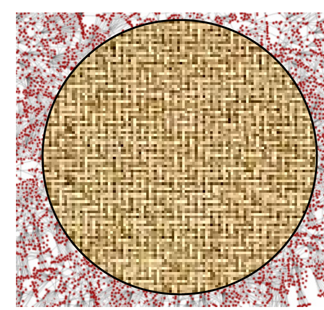
Expansion Technique: λ_{ant} changes



After σ_i steps

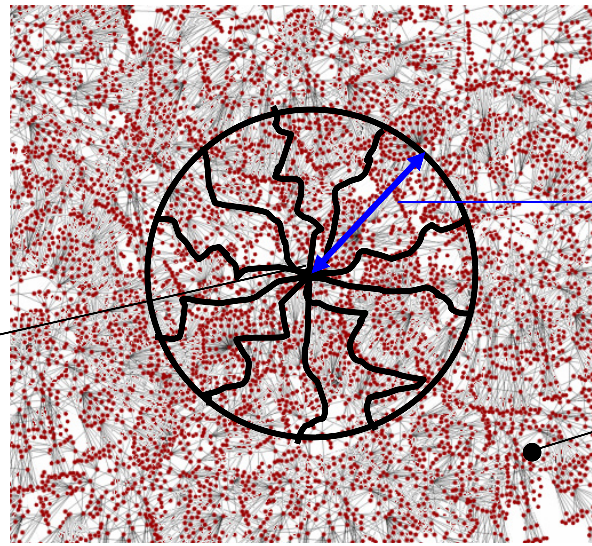


$\lambda_{ant} = \lambda_{ant} + \delta_l$



ACOhg: Huge Graphs Exploration

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



λ_{ant}

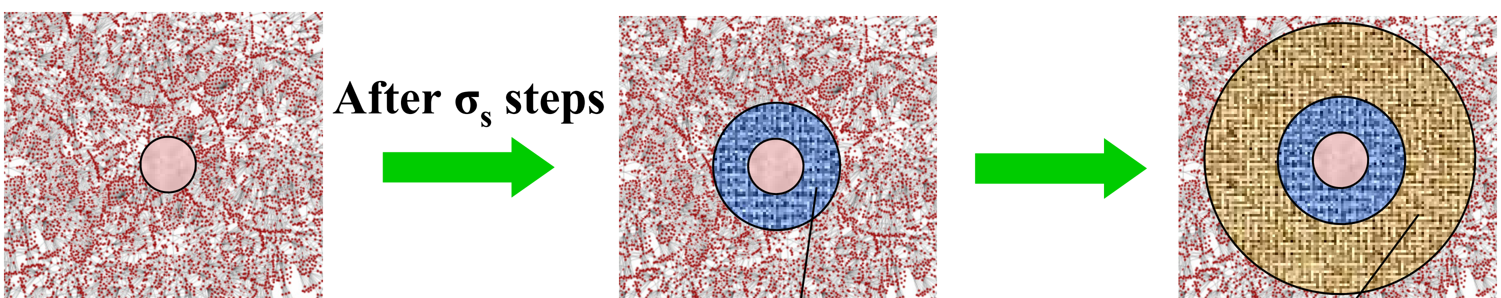
Initial node

What if…?

Objective node

Two alternatives

Missionary Technique: starting nodes for path construction change



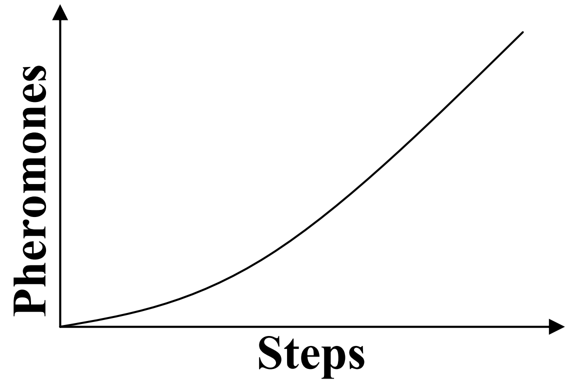
After σ_s steps

Second stage

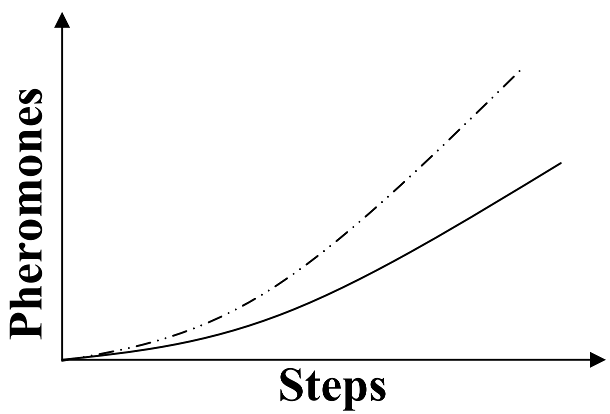
Third stage

ACOhg: Pheromones

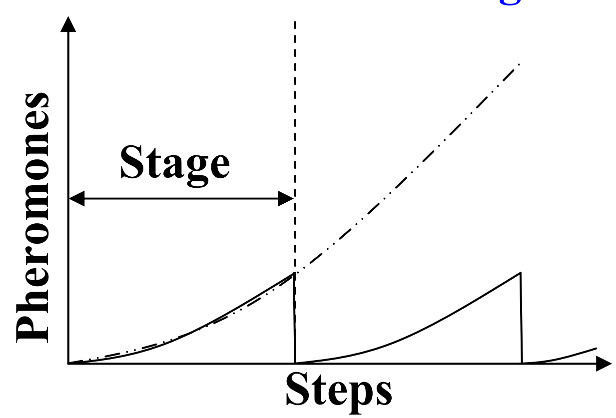
- The number of pheromone trails increases during the search
- This leads to memory problems
- We must remove some pheromone trails from memory



Remove pheromone trails τ_{ij} below a given threshold τ_{θ}

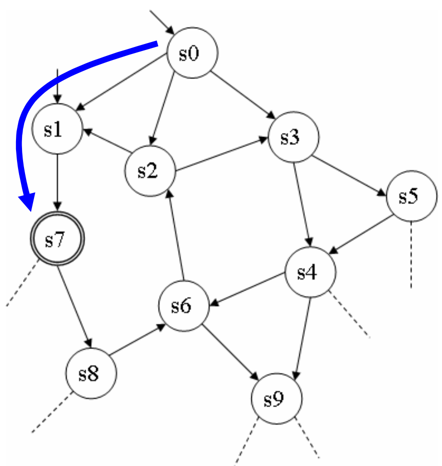


In the missionary technique, remove all pheromone trails after one stage



ACOhg: Fitness Function

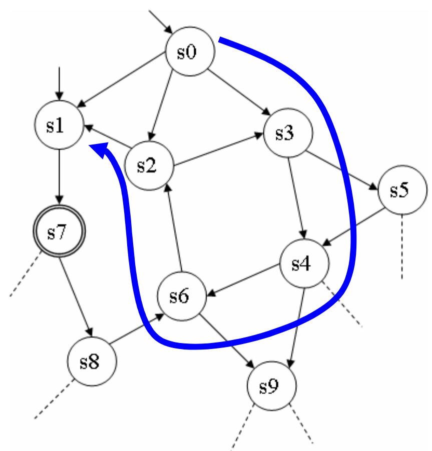
- The fitness function must be able to evaluate partial solutions
- Penalties are added for partial solutions and solutions with cycles



Complete solution

$$p = 0$$

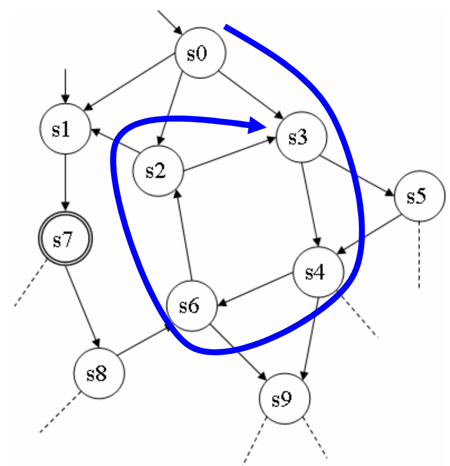
Total penalty



Partial solution without cycle

$$p = p_p$$

Penalty constant for partial solutions



Partial solution with cycle

$$p = p_p + p_c \frac{\lambda_{ant} - l}{\lambda_{ant} - 1}$$

Penalty constant for solutions with cycles

Path length

Promela Models

- We selected **5 Promela models** for the experiments

Model	LoC	States	Processes	Safety Property
giop22	717	*unknown*	11	Deadlock
marriers4	142	*unknown*	5	Deadlock
needham	260	18242	4	LTL formula
phi16	34	43046721*	17	Deadlock
pots	453	*unknown*	8	Deadlock

* Theoretical result

- For all except **needham**, the states do not fit into the main memory of the computer

Parameters for ACOhg

- **The ACOhg model was implemented inside the MALLBA library and then included into the HSF-SPIN model checker**

Parameter	**Value**	**Parameter**	**Value**
Steps	100	ξ	0.5
Colony size	10	a	5
λ_{ant}	10	ρ	0.8
σ_s	2	α	1.0
s	10	β	2.0

- **Fitness function: length of the path + heuristic + penalty for partial solutions**
- **Two variants: using no heuristic (ACOhg-b) and using it (ACOhg-h)**
- **Machine: Pentium 4 at 2.8 GHz with 512 MB**

Results I: Efficacy

- We compare the results of ACOhg algorithms against state-of-the-art model checker algorithms: DFS, BFS, A*, and BF

Which algorithm finds errors?

Models	BFS	DFS	A*	BF	ACOhg
giop22		●	●	●	🐜
needham	●	●	●	●	🐜
phi16			●	●	🐜
pots	●	●	●	●	🐜
marriers4				●	🐜
marriers20					🐜

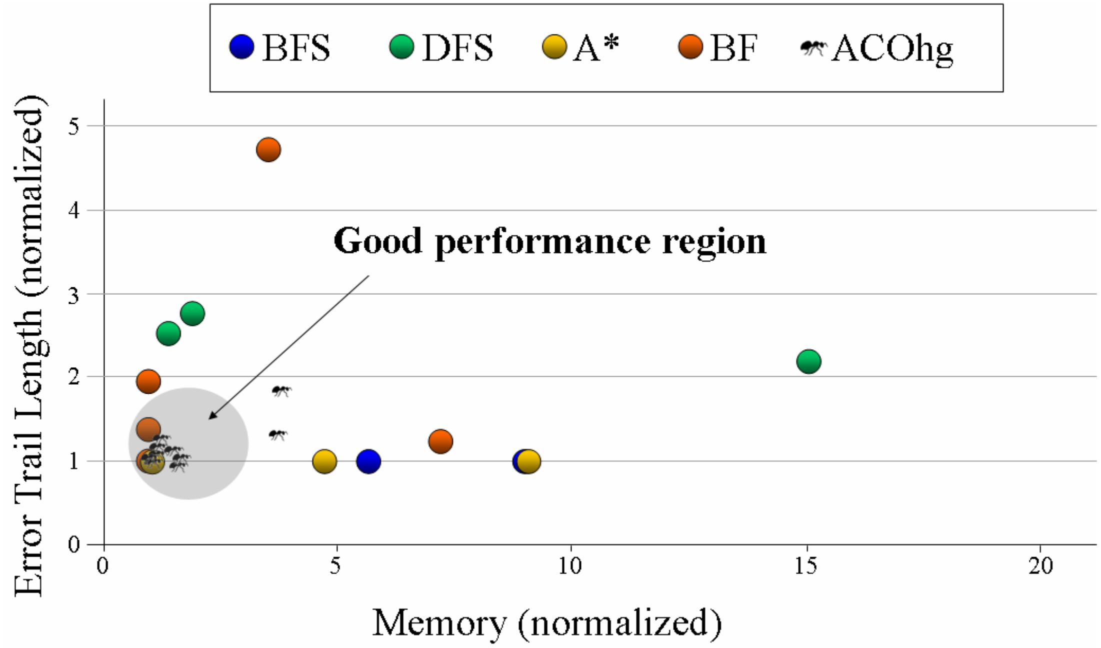
- ACOhg algorithms are the only ones that are able to find errors in very large models (marriers20).

Results II: Details

Models	Measurements	BFS	DFS	ACOhg-b	A*	BF	ACOhg-h
giop22	hit rate	0/1	1/1	100/100	1/1	1/1	100/100
	len (states)	-	112.00	45.80	44.00	44.00	44.20
	mem (KB)	-	3945.00	4814.12	417792.00	2873.00	4482.12
	exp (states)	-	220.00	1048.52	83758.00	168.00	1001.78
	cpu (ms)	-	30.00	113.60	46440.00	10.00	112.40
marriers4	hit rate	0/1	0/1	57/100	0/1	1/1	84/100
	len (states)	-	-	92.18	-	108.00	86.65
	mem (KB)	-	-	5917.91	-	41980.00	5811.43
	exp (states)	-	-	2045.84	-	9193.00	1915.30
	cpu (ms)	-	-	257.19	-	190.00	233.33
needham	hit rate	1/1	1/1	100/100	1/1	1/1	100/100
	len (states)	5.00	11.00	6.39	5.00	10.00	6.12
	mem (KB)	23552.00	62464.00	5026.36	19456.00	4149.00	4865.40
	exp (states)	1141.00	11203.00	100.21	814.00	12.00	87.47
	cpu (ms)	1110.00	18880.00	262.00	810.00	20.00	229.50
phi16	hit rate	0/1	0/1	100/100	1/1	1/1	100/100
	len (states)	-	-	31.44	17.00	81.00	23.08
	mem (KB)	-	-	10905.60	2881.00	10240.00	10680.32
	exp (states)	-	-	832.08	33.00	893.00	587.53
	cpu (ms)	-	-	289.40	10.00	40.00	243.80
pots	hit rate	1/1	1/1	49/100	1/1	1/1	99/100
	len (states)	5.00	14.00	5.73	5.00	7.00	5.44
	mem (KB)	57344.00	12288.00	9304.67	57344.00	6389.00	6974.56
	exp (states)	2037.00	1966.00	176.47	1257.00	695.00	110.48
	cpu (ms)	4190.00	140.00	441.63	6640.00	50.00	319.49

Results III: Graphical Comparison

- **Error trail length vs. memory graph**



ACOhg algorithms require less memory than BFS

They also get shorter (better) error trails than DFS

- **In general, unlike exhaustive algorithms, ACOhg algorithms keep all the results in a good performance region (high accuracy and efficiency)**

Previous Results with Metaheuristics

- **GA is the previous metaheuristic algorithm applied to this problem**
- **Godefroid & Khurshid (2002), found errors in phi17 and needham models with GA**
- **To the best of our knowledge, this is the most recent result for this problem using metaheuristics**

Model	Algorithm	Hit (%)	Time (s)	Mem. (KB)
phi17	GA	52	197.00	n/a
	ACOhg-h	**100**	**0.28**	11274
needham	GA	3	3068.00	n/a
	ACOhg-h	**100**	**0.23**	4865

- **The results state that ACOhg has higher efficacy and efficiency than GA (even taking into account the differences in the machines)**
- **But we cannot do a fair comparison because the models and the model checkers are different (Verisoft against HSF-SPIN)**

Conclusions and Future Work

Conclusions

- ACOhg is able to outperform state-of-the-art algorithms used nowadays in current model checkers for finding safety errors
- ACOhg is able to explore really large concurrent models for which traditional model checking techniques fail
- This represents a promising starting point for the use of metaheuristic algorithms in model checking and an interesting subject in SBSE

Future Work

- Combine ACOhg algorithms with other techniques for reducing the amount of memory: Partial Order Reduction and Symmetry Reduction (in progress)
- Include ACOhg into JavaPathFinder for finding errors in Java programs (in progress)
- Parallel implementation of ACOhg for this problem (parallel model checkers)

Thanks for your attention !!!