

I/O Efficient Model Checking

Pavel Šimeček

Jiří Barnat, Luboš Brim, Michael Weber



Masaryk University Brno



University of Twente
Enschede - The Netherlands

University of Twente

May 19, 2008

Introduction

- Model checking:
 - large-scale,
 - enumerative,
 - temporal properties (LTL),
 - finite state systems.
- Possible solution: **external memory** (hard disks) utilization in model checking

Pioneering Work

- *Ulrich Stern, David L. Dill: Using Magnetic Disk instead of Main Memory in the Mur ϕ Verifier (CAV'98)*

LTL Model Checking

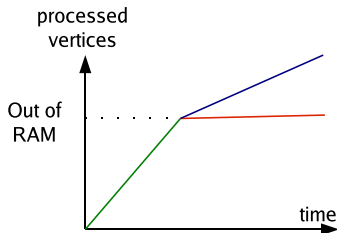
- *Stefan Edelkamp, Shahid Jabbar: Large-Scale Directed Model Checking LTL (SPIN'06)*
- *J. Barnat, L. Brim, P. Šimeček: I/O Efficient Accepting Cycle Detection (CAV'07)*
- *J. Barnat, L. Brim, P. Šimeček, M. Weber: Revisiting Resistance Speeds Up I/O-Efficient LTL Model Checking (TACAS'08)*

Outline

- I/O efficient BFS (reachability analysis)
- I/O efficient LTL model checking
- MAP algorithm – in detail
- Merge omission – heuristic
- Analytical results
- Experimental results

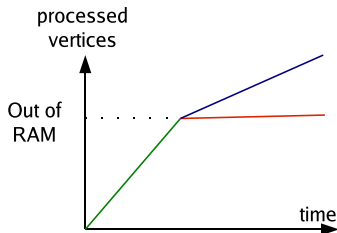
Need for I/O Efficiency

- Enumerative model checking \approx graph traversal
- Default OS swapping fails:
- An ordinary (BFS, DFS, ...) graph traversal does not have any locality in access to the graph
 \Rightarrow a lot of random access operations



Need for I/O Efficiency

- Enumerative model checking \approx graph traversal
- Default OS swapping fails:
- An ordinary (BFS, DFS, ...) graph traversal does not have any locality in access to the graph
 \Rightarrow a lot of random access operations
- Need to transfer data by **blocks**



I/O Complexity

- Standard two-level I/O-model with one disk (by Aggarwal and Vitter)
- Model attributes:
 - B ... block size
 - M ... main memory size

I/O Complexity

- Standard two-level I/O-model with one disk (by Aggarwal and Vitter)
- Model attributes:
 - B ... block size
 - M ... main memory size
- I/O operation = transfer of a data block between RAM and disk

I/O Complexity

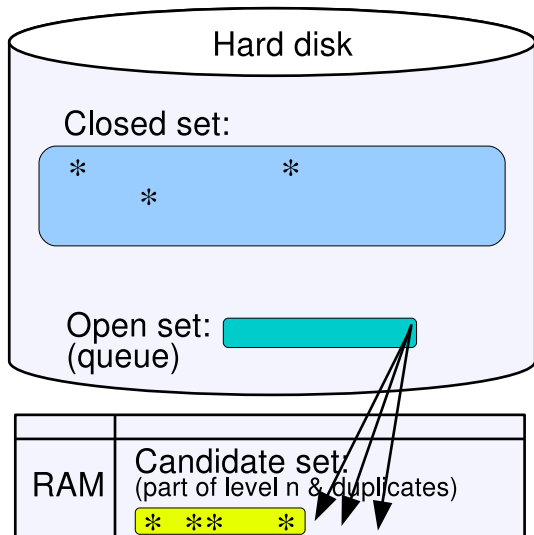
- Standard two-level I/O-model with one disk (by Aggarwal and Vitter)
- Model attributes:
 - B ... block size
 - M ... main memory size
- I/O operation = transfer of a data block between RAM and disk
- I/O complexity = number of I/Os the algorithm performs

I/O Complexity

- Standard two-level I/O-model with one disk (by Aggarwal and Vitter)
- Model attributes:
 - B ... block size
 - M ... main memory size
- I/O operation = transfer of a data block between RAM and disk
- I/O complexity = number of I/Os the algorithm performs
- Typical operations in algorithms using disks:
 - Random access operation: $\theta(1)$
 - Linear pass through N items: $scan(N) = \theta(\frac{N}{B})$
 - Sort of N items: $sort(N) = \theta(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$

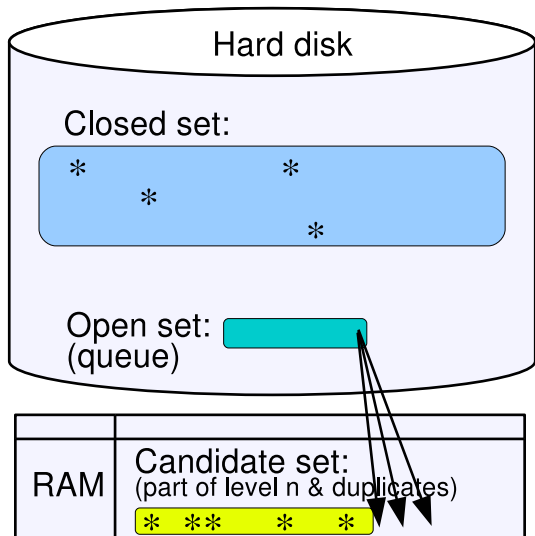
I/O Efficient BFS + Delayed Duplicate Detection

BFS (demo A)



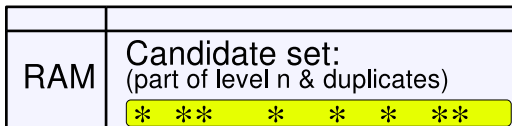
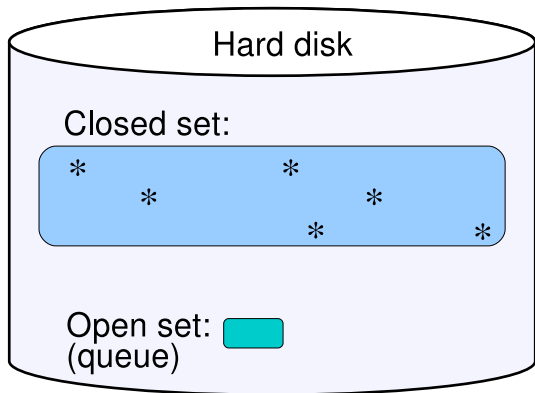
I/O Efficient BFS + Delayed Duplicate Detection

BFS (demo A)



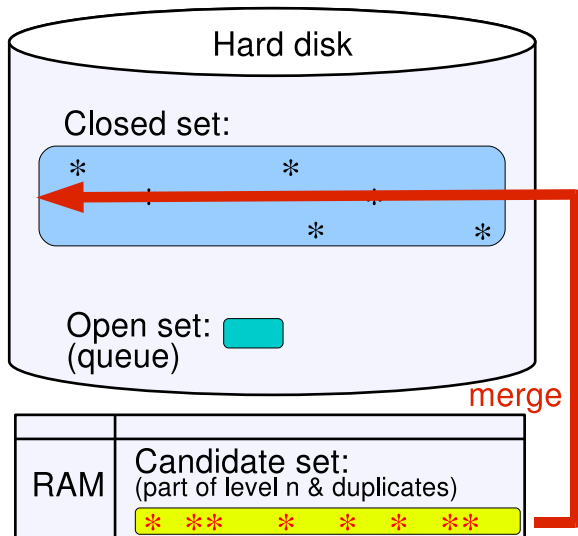
I/O Efficient BFS + Delayed Duplicate Detection

BFS (demo A)



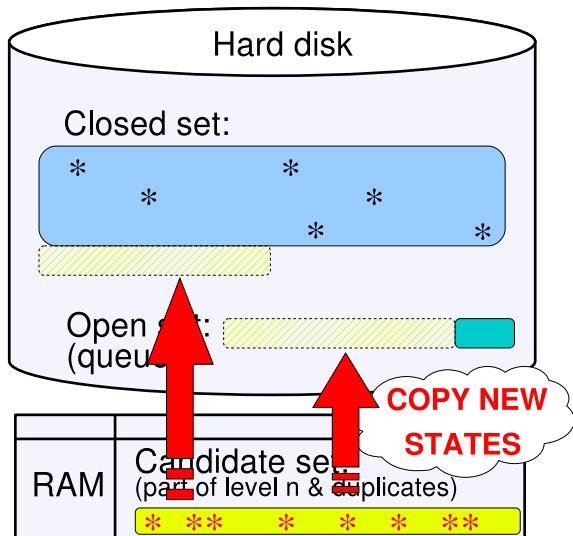
I/O Efficient BFS + Delayed Duplicate Detection

BFS (demo A)



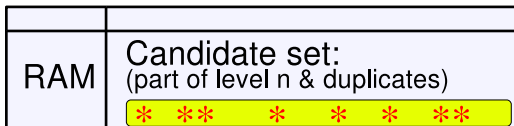
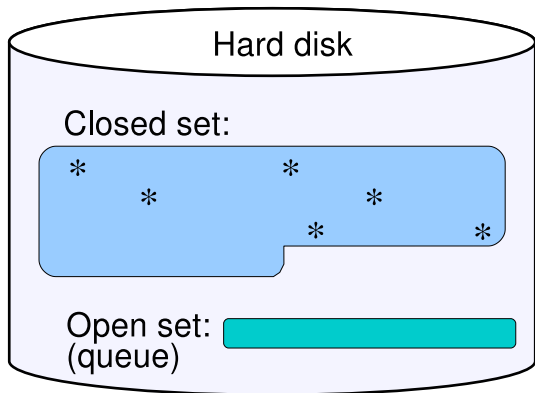
I/O Efficient BFS + Delayed Duplicate Detection

BFS (demo A)



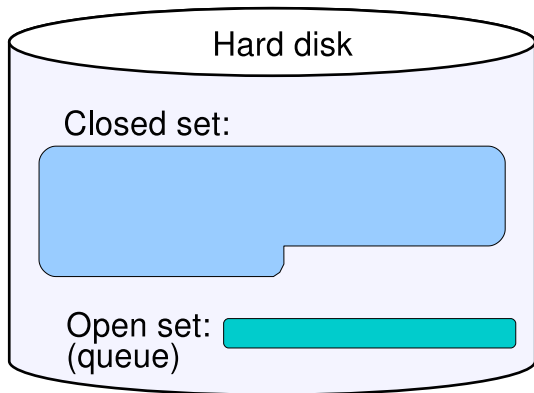
I/O Efficient BFS + Delayed Duplicate Detection

BFS (demo A)



I/O Efficient BFS + Delayed Duplicate Detection

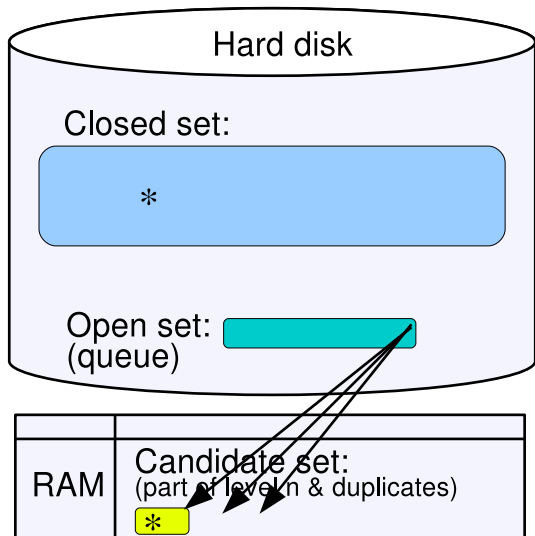
BFS (demo A)



RAM	Candidate set: (part of level n & duplicates)

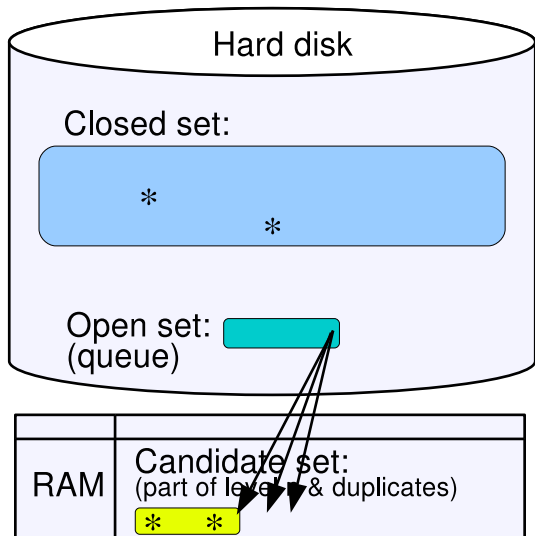
I/O Efficient BFS + Delayed Duplicate Detection

BFS (demo B)



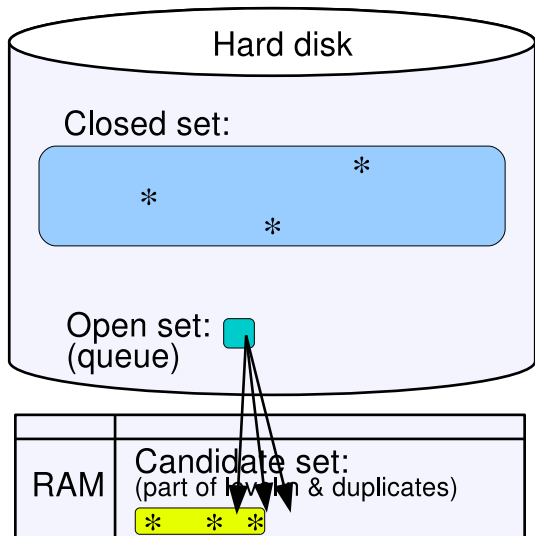
I/O Efficient BFS + Delayed Duplicate Detection

BFS (demo B)



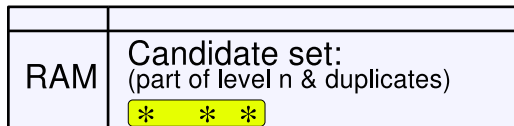
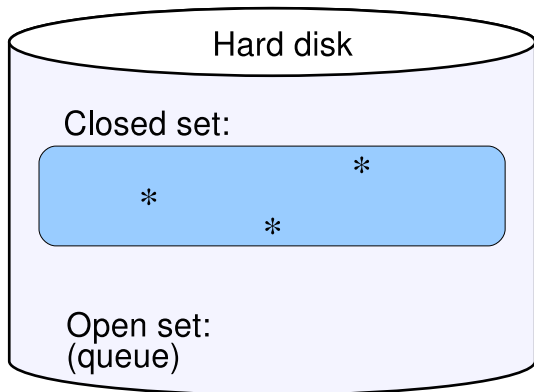
I/O Efficient BFS + Delayed Duplicate Detection

BFS (demo B)



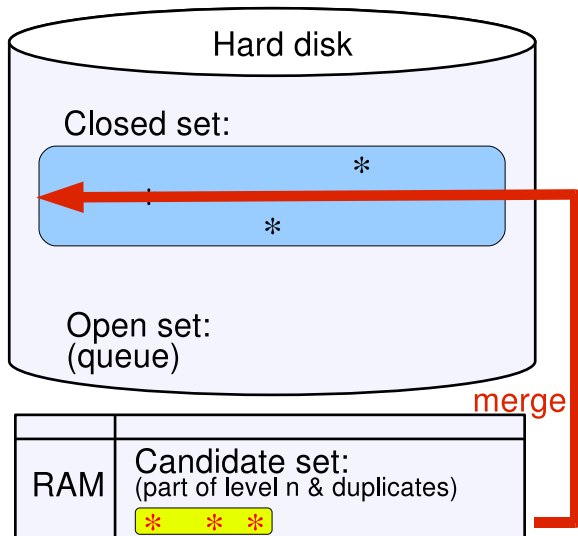
I/O Efficient BFS + Delayed Duplicate Detection

BFS (demo B)



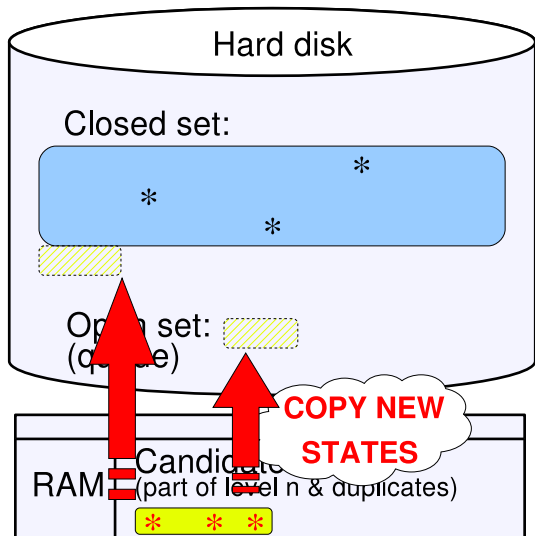
I/O Efficient BFS + Delayed Duplicate Detection

BFS (demo B)



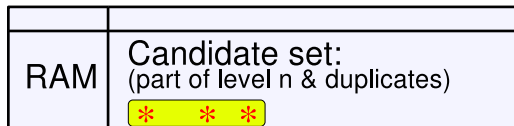
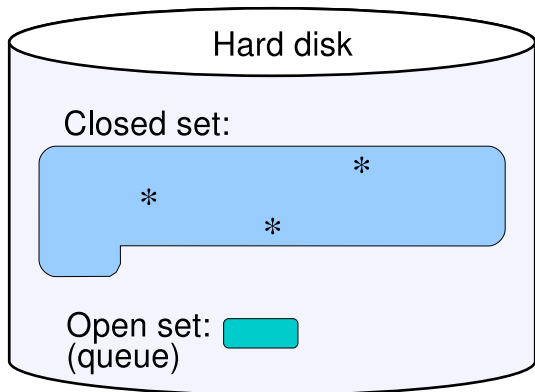
I/O Efficient BFS + Delayed Duplicate Detection

BFS (demo B)



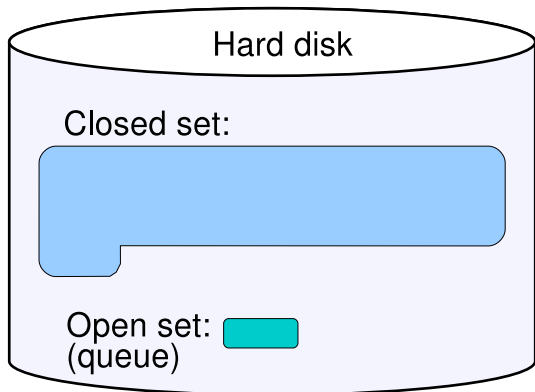
I/O Efficient BFS + Delayed Duplicate Detection

BFS (demo B)



I/O Efficient BFS + Delayed Duplicate Detection

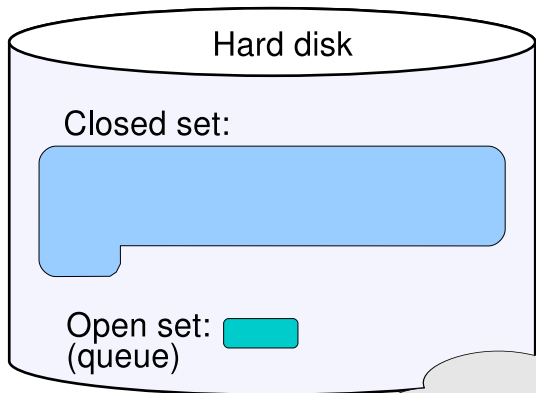
BFS (demo B)



RAM	Candidate set: (part of level n & duplicates)

I/O Efficient BFS + Delayed Duplicate Detection

BFS



RAM	Candidate set: (part of level n & duplicates)

$$O((h_{BFS} + |E|/M) \cdot scan(|V|))$$

Our Setting

- Implicitly given graph (*implicit graph*): initial state + successor function
- Advantage of implicit graphs: no disk operations performed when successors of a state needed
- (Disadvantage explicit graphs: at least $|V|$ I/O operations to explore the entire graph)

LTL Model Checking

- **Our task:** Check for existence of an accepting cycle

I/O Efficient Accepting Cycle Detection

- Naive solution – DFS-based algorithms (NDFS, DDFS)

I/O Efficient Accepting Cycle Detection

- Naive solution – DFS-based algorithms (NDFS, DDFS)
 - No existing efficient external DFS-based algorithm on **implicit graphs**

I/O Efficient Accepting Cycle Detection

- Naive solution – DFS-based algorithms (NDFS, DDFS)
 - No existing efficient external DFS-based algorithm on **implicit graphs**
- Existing solutions:

I/O Efficient Accepting Cycle Detection

- Naive solution – DFS-based algorithms (NDFS, DDFS)
 - No existing efficient external DFS-based algorithm on **implicit graphs**
- Existing solutions:
 - **EJ**: I/O efficient 'liveness as safety' (Edelkamp, Jabbar)
 - Reducing accepting cycle detection to reachability
 - Potentially quadratic state space growth:
 $V \xrightarrow{\text{reduction}} V \times F$
 - On-the-fly

I/O Efficient Accepting Cycle Detection

- Naive solution – DFS-based algorithms (NDFS, DDFS)
 - No existing efficient external DFS-based algorithm on **implicit graphs**
- Existing solutions:
 - **EJ**: I/O efficient 'liveness as safety' (Edelkamp, Jabbar)
 - Reducing accepting cycle detection to reachability
 - Potentially quadratic state space growth:
$$V \xrightarrow{\text{reduction}} V \times F$$
 - On-the-fly
 - **OWCTY**: I/O eff. OWCTY (Barnat, Brim, Šimeček)
 - Quite efficient in practice – especially for verification of valid properties
 - NOT on-the-fly

I/O Efficient Accepting Cycle Detection

- Naive solution – DFS-based algorithms (NDFS, DDFS)
 - No existing efficient external DFS-based algorithm on **implicit graphs**
- Existing solutions:
 - **EJ**: I/O efficient 'liveness as safety' (Edelkamp, Jabbar)
 - Reducing accepting cycle detection to reachability
 - Potentially quadratic state space growth:
$$V \xrightarrow{\text{reduction}} V \times F$$
 - On-the-fly
 - **OWCTY**: I/O eff. OWCTY (Barnat, Brim, Šimeček)
 - Quite efficient in practice – especially for verification of valid properties
 - NOT on-the-fly
- Most recent solution: **MAP** algorithm

MAP – Basics

- MAP = Maximal Accepting Predecessors
- Internal memory algorithm introduced by L. Brim, I. Černá, P. Moravec and J. Šimša, FMCAD'04

MAP – Basics

- MAP = Maximal Accepting Predecessors
- Internal memory algorithm introduced by L. Brim, I. Černá, P. Moravec and J. Šimša, FMCAD'04
- Cycle detection based on labeling of vertices with their maximal accepting predecessors

MAP – Basics

- MAP = Maximal Accepting Predecessors
- Internal memory algorithm introduced by L. Brim, I. Černá, P. Moravec and J. Šimša, FMCAD'04
- Cycle detection based on labeling of vertices with their maximal accepting predecessors
- Maximum is taken with respect to arbitrary given linear ordering on accepting vertices

MAP – Basics

- MAP = Maximal Accepting Predecessors
- Internal memory algorithm introduced by L. Brim, I. Černá, P. Moravec and J. Šimša, FMCAD'04
- Cycle detection based on labeling of vertices with their maximal accepting predecessors
- Maximum is taken with respect to arbitrary given linear ordering on accepting vertices
- Advantages:

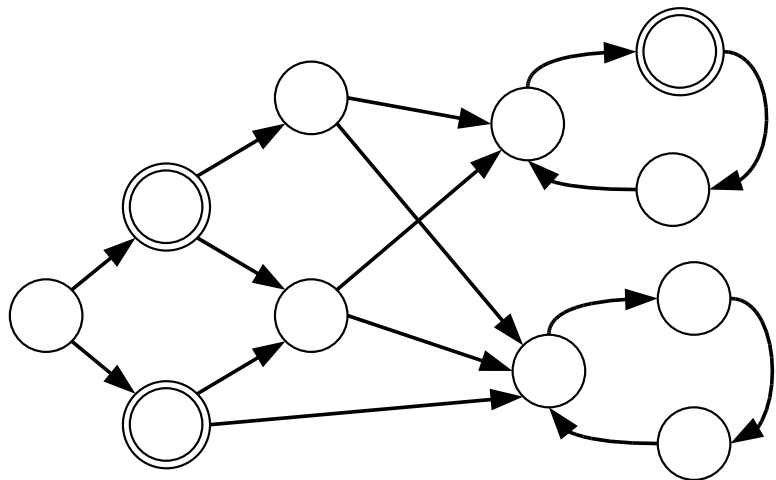
MAP – Basics

- MAP = Maximal Accepting Predecessors
- Internal memory algorithm introduced by L. Brim, I. Černá, P. Moravec and J. Šimša, FMCAD'04
- Cycle detection based on labeling of vertices with their maximal accepting predecessors
- Maximum is taken with respect to arbitrary given linear ordering on accepting vertices
- Advantages:
 - BFS-based

MAP – Basics

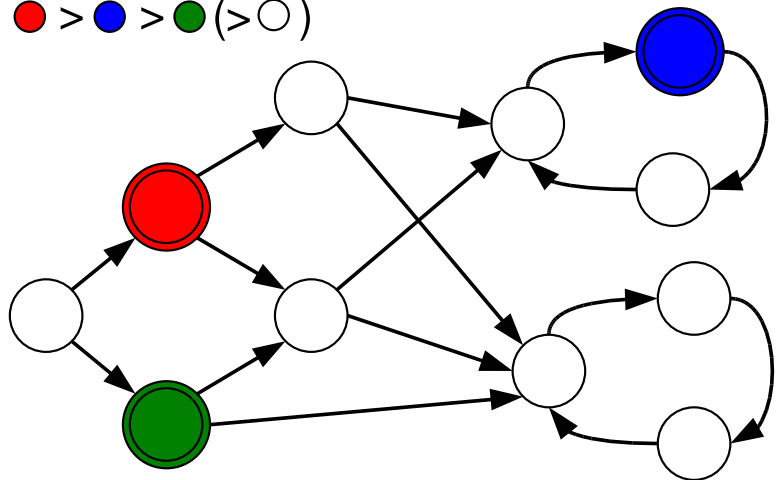
- MAP = Maximal Accepting Predecessors
- Internal memory algorithm introduced by L. Brim, I. Černá, P. Moravec and J. Šimša, FMCAD'04
- Cycle detection based on labeling of vertices with their maximal accepting predecessors
- Maximum is taken with respect to arbitrary given linear ordering on accepting vertices
- Advantages:
 - BFS-based
 - On-the-fly

MAP Demonstration



MAP Demonstration

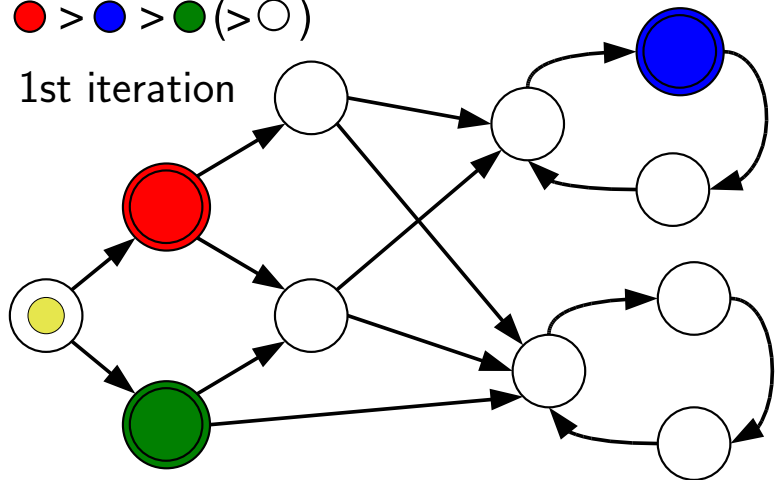
● > ● > ● (> ○)



MAP Demonstration

● > ● > ● (> ○)

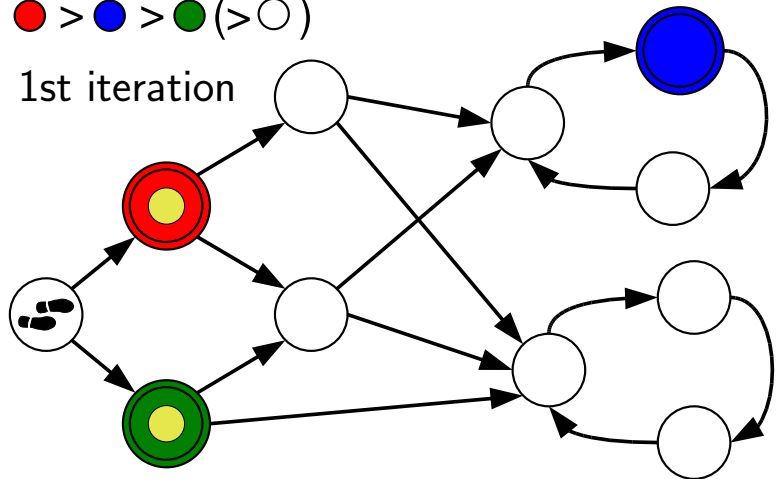
1st iteration



MAP Demonstration

● > ● > ● (> ○)

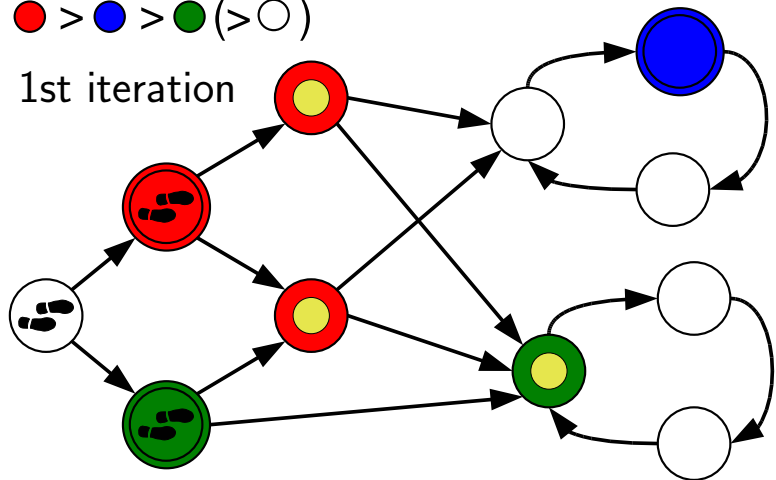
1st iteration



MAP Demonstration

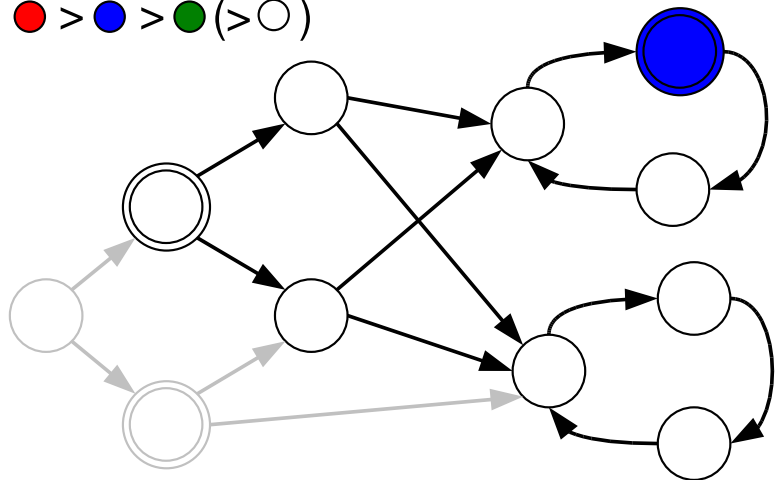
● > ● > ● (> ○)

1st iteration



MAP Demonstration

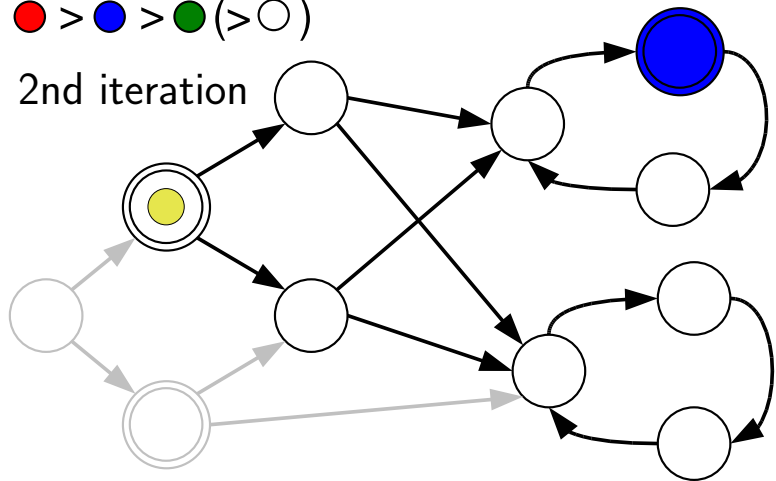
● > ● > ● (> ○)



MAP Demonstration

● > ● > ● (> ○)

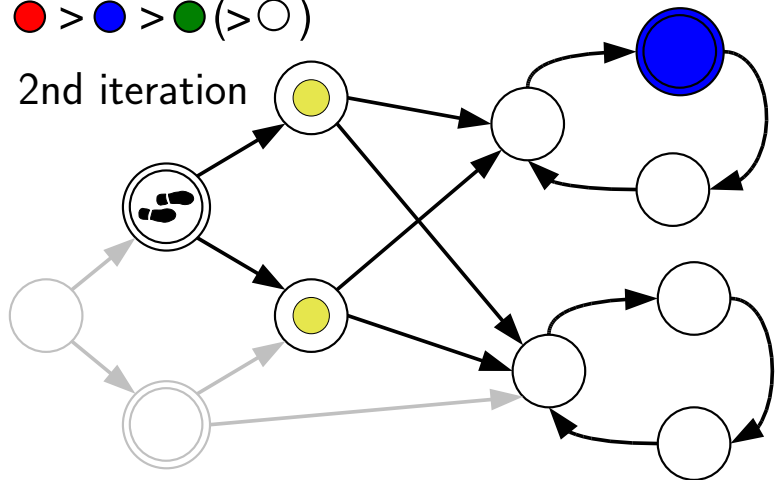
2nd iteration



MAP Demonstration

● > ● > ● (> ○)

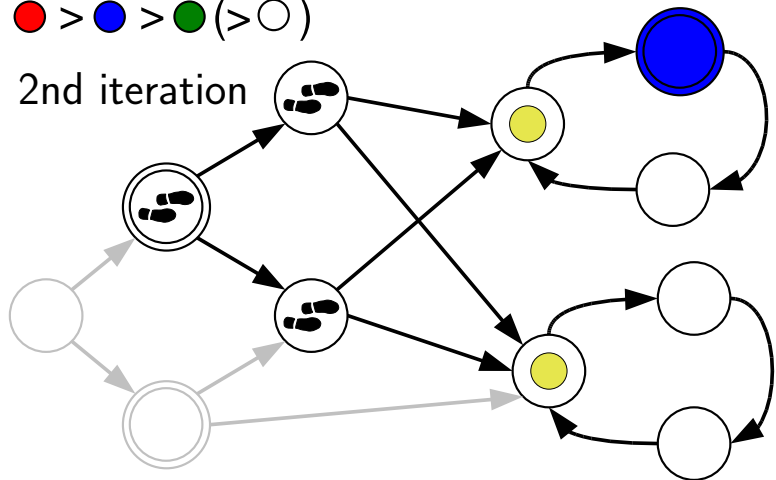
2nd iteration



MAP Demonstration

● > ● > ● (> ○)

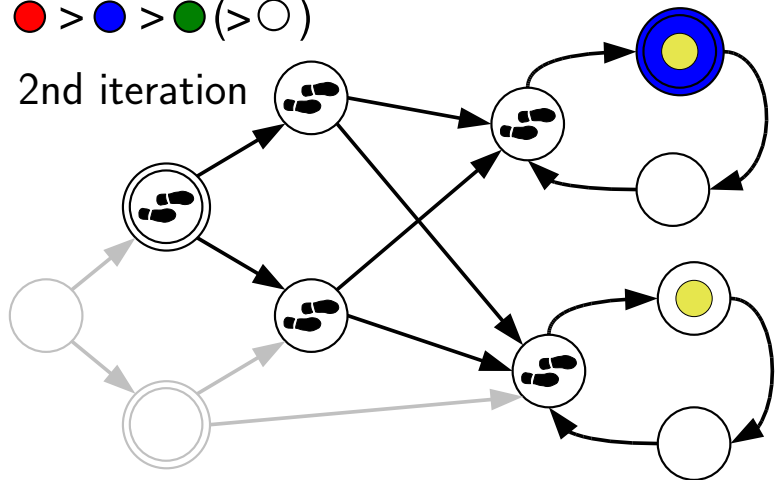
2nd iteration



MAP Demonstration

● > ● > ● (> ○)

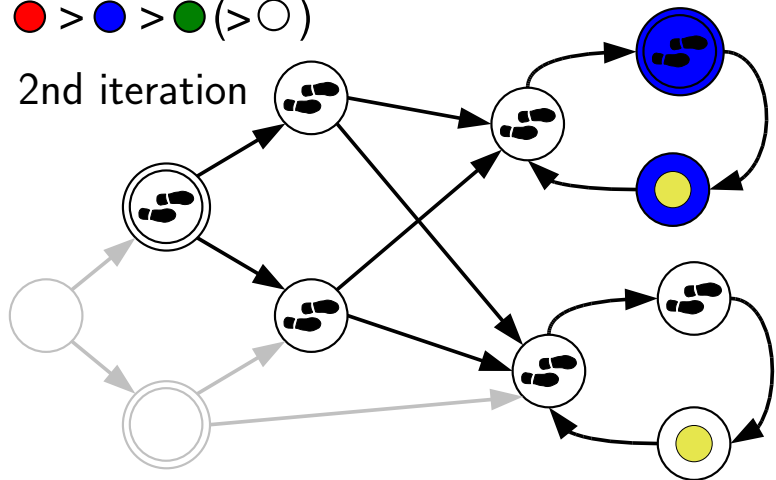
2nd iteration



MAP Demonstration

● > ● > ● (> ○)

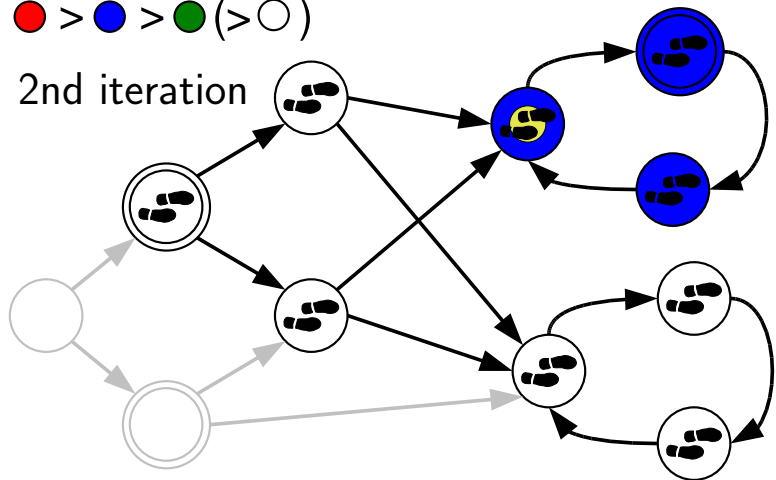
2nd iteration



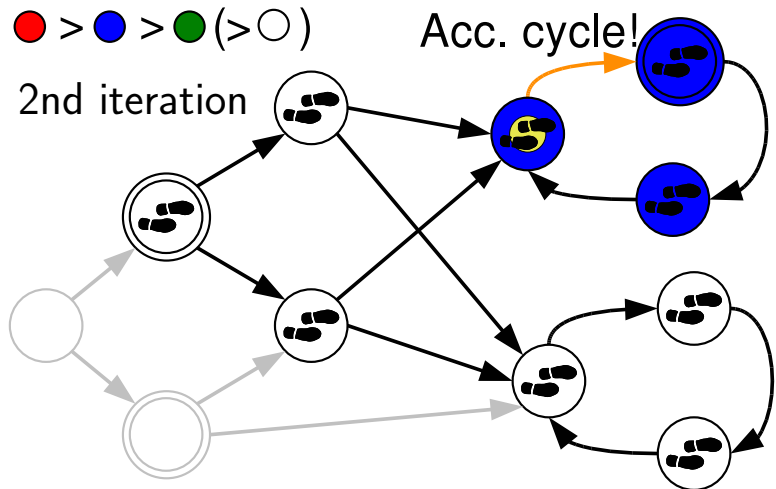
MAP Demonstration

● > ● > ● (> ○)

2nd iteration



MAP Demonstration



I/O Efficient MAP

- 1 iteration = I/O efficient BFS with recoloring
 - recoloring = reopening of states, which obtain higher color

I/O Efficient MAP

- 1 iteration = I/O efficient BFS with recoloring
 - recoloring = reopening of states, which obtain higher color
- If partition is small, run internal memory NDFS on it

I/O Efficient MAP

- 1 iteration = I/O efficient BFS with recoloring
 - recoloring = reopening of states, which obtain higher color
- If partition is small, run internal memory NDFS on it
- **In practice: In second iteration all partitions fit in RAM**

I/O Complexity Comparison

MAP:

- $\mathcal{O}(|F| \cdot ((d + |E|/M + |F|) \cdot \text{scan}(|V|) + \text{sort}(|V|)))$
 - (V, E) ... graph
 - F ... set of accepting vertices
 - d ... diameter of the graph

I/O Complexity Comparison

MAP:

- $\mathcal{O}(|F| \cdot ((d + |E|/M + |F|) \cdot \text{scan}(|V|) + \text{sort}(|V|)))$
 - (V, E) ... graph
 - F ... set of accepting vertices
 - d ... diameter of the graph

EJ:

- $\mathcal{O}((l + |F||E|/M)\text{scan}(|F||V|))$
 - l ... length of the shortest counterexample
($l = h_{BFS}$ in case of no counterexample)

I/O Complexity Comparison

MAP:

- $\mathcal{O}(|F| \cdot ((d + |E|/M + |F|) \cdot \text{scan}(|V|) + \text{sort}(|V|)))$
 - (V, E) ... graph
 - F ... set of accepting vertices
 - d ... diameter of the graph

EJ:

- $\mathcal{O}((l + |F||E|/M)\text{scan}(|F||V|))$
 - l ... length of the shortest counterexample
($l = h_{BFS}$ in case of no counterexample)

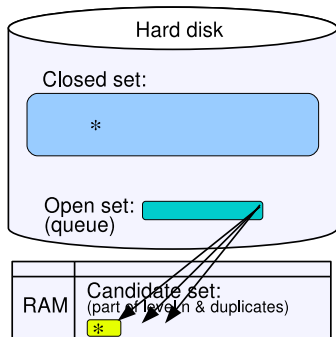
OWCTY:

- $\mathcal{O}(|I_{SCC}| \cdot (h_{BFS} + |p_{max}| + |E|/M)\text{scan}(|V|))$
 - I_{SCC} ... the longest path in the SCC graph
 - h_{BFS} ... height of BFS tree
 - p_{max} ... the longest path through trivial SCCs

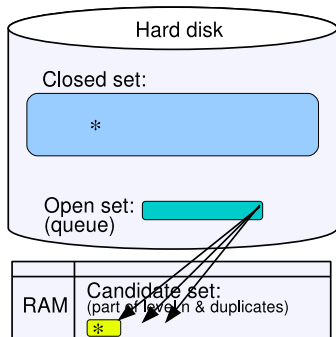
Merge Omission and Revisiting

Merge omission

Standard Travers.



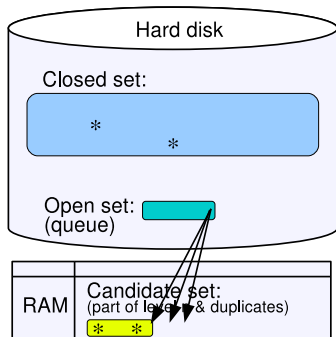
Modified Travers.



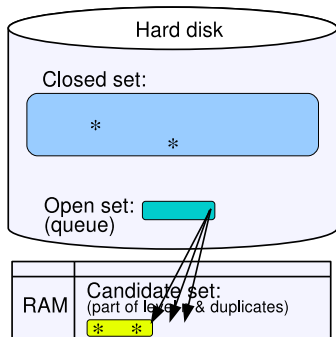
Merge Omission and Revisiting

Merge omission

Standard Travers.



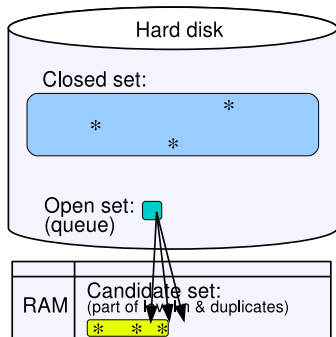
Modified Travers.



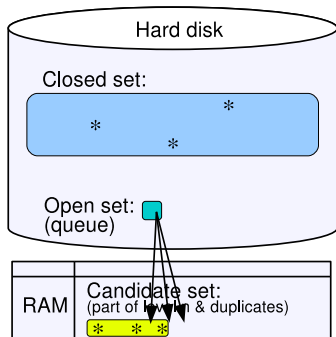
Merge Omission and Revisiting

Merge omission

Standard Travers.



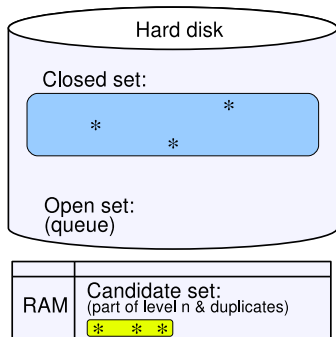
Modified Travers.



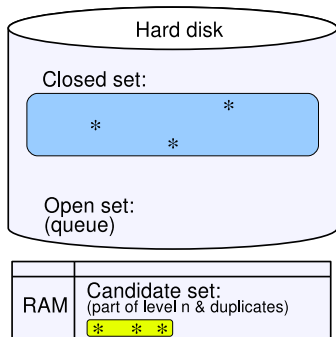
Merge Omission and Revisiting

Merge omission

Standard Travers.



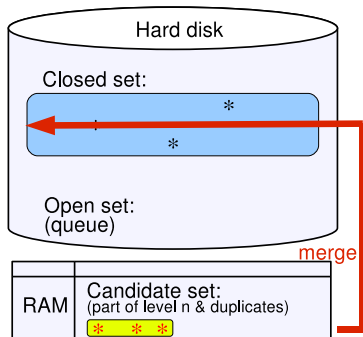
Modified Travers.



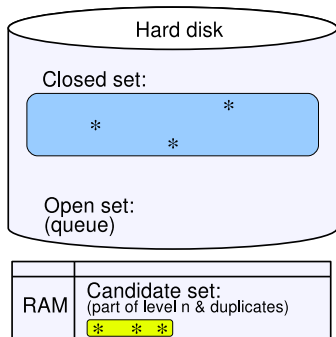
Merge Omission and Revisiting

Merge omission

Standard Travers.



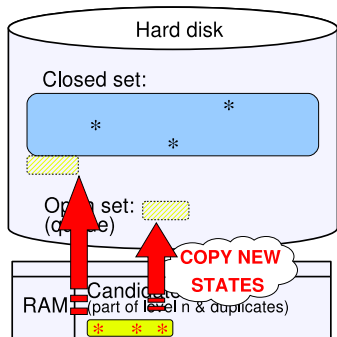
Modified Travers.



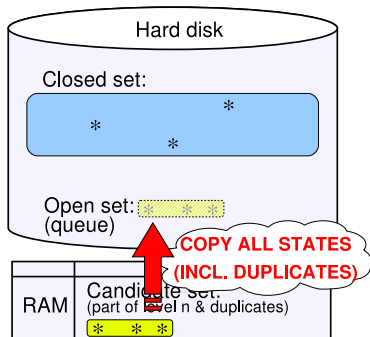
Merge Omission and Revisiting

Merge omission

Standard Travers.



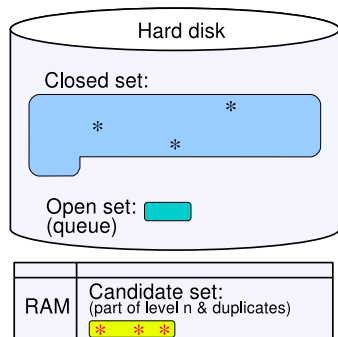
Modified Travers.



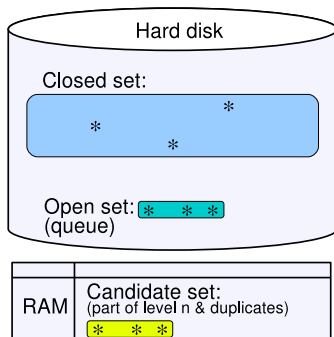
Merge Omission and Revisiting

Merge omission

Standard Travers.



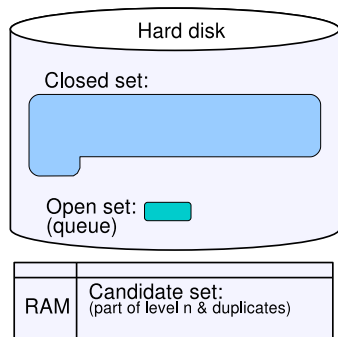
Modified Travers.



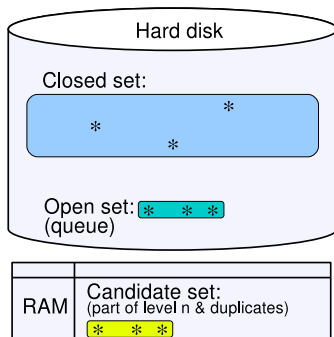
Merge Omission and Revisiting

Merge omission

Standard Travers.



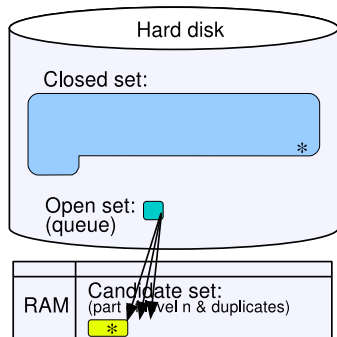
Modified Travers.



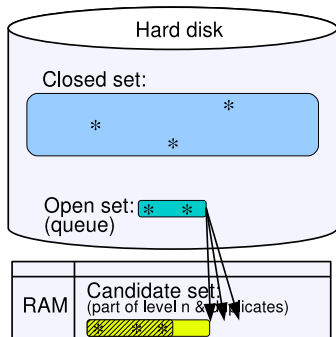
Merge Omission and Revisiting

Merge omission

Standard Travers.



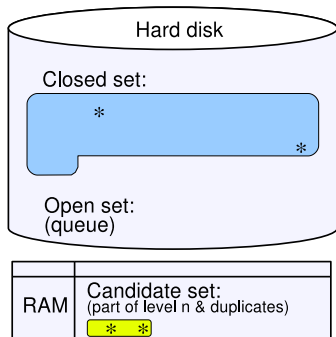
Modified Travers.



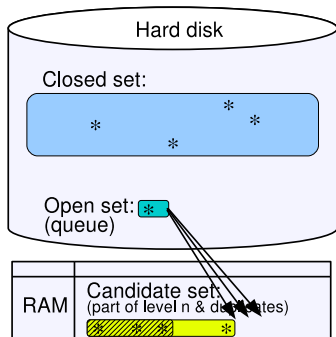
Merge Omission and Revisiting

Merge omission

Standard Travers.



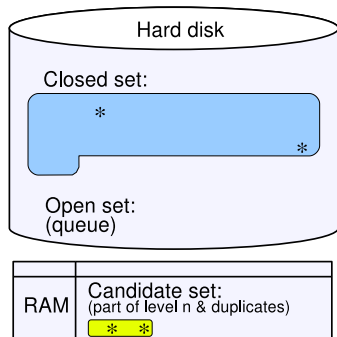
Modified Travers.



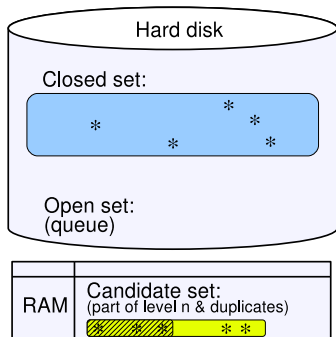
Merge Omission and Revisiting

Merge omission

Standard Travers.



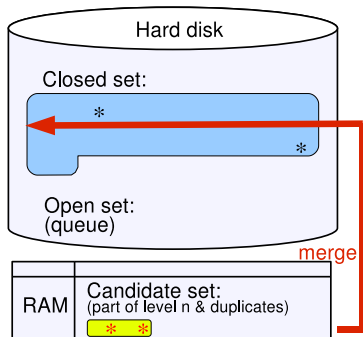
Modified Travers.



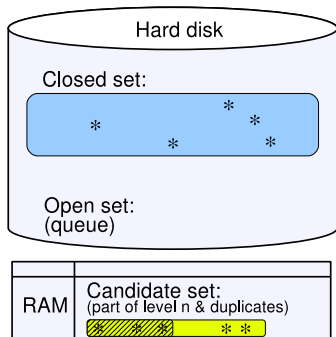
Merge Omission and Revisiting

Merge omission

Standard Travers.



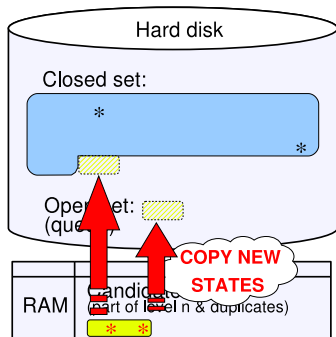
Modified Travers.



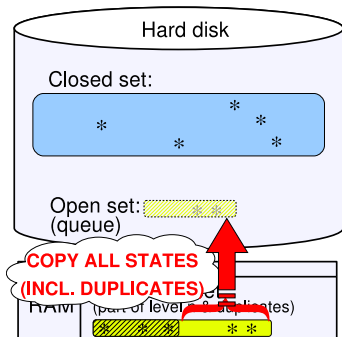
Merge Omission and Revisiting

Merge omission

Standard Travers.



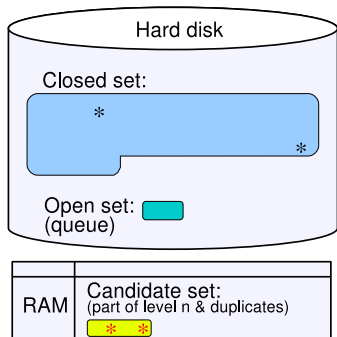
Modified Travers.



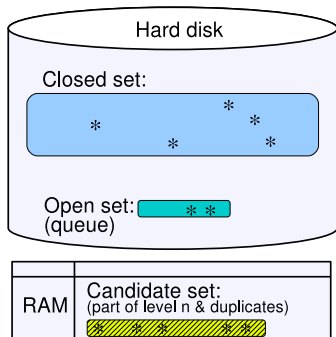
Merge Omission and Revisiting

Merge omission

Standard Travers.



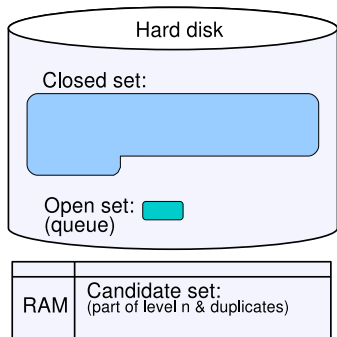
Modified Travers.



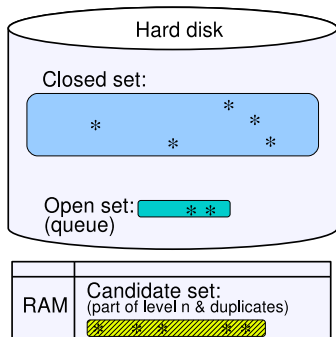
Merge Omission and Revisiting

Merge omission

Standard Travers.



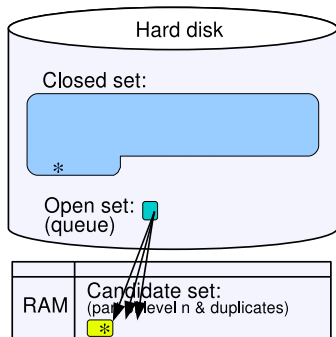
Modified Travers.



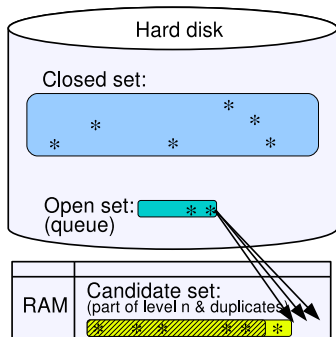
Merge Omission and Revisiting

Merge omission

Standard Travers.



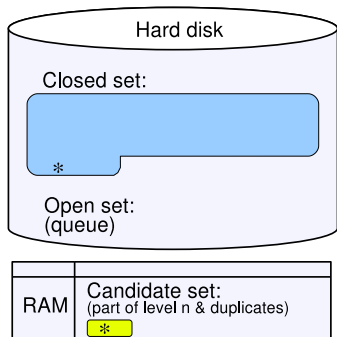
Modified Travers.



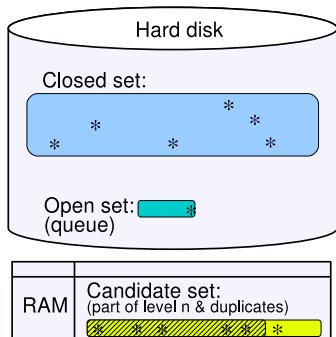
Merge Omission and Revisiting

Merge omission

Standard Travers.



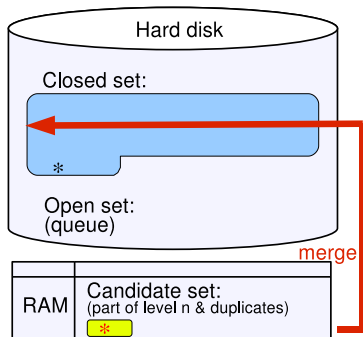
Modified Travers.



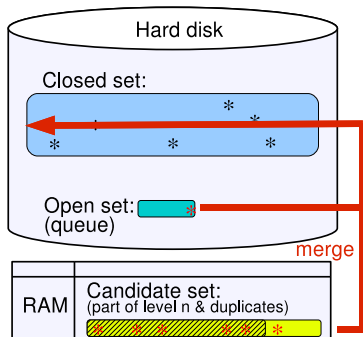
Merge Omission and Revisiting

Merge omission

Standard Travers.



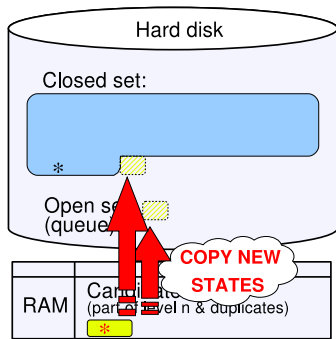
Modified Travers.



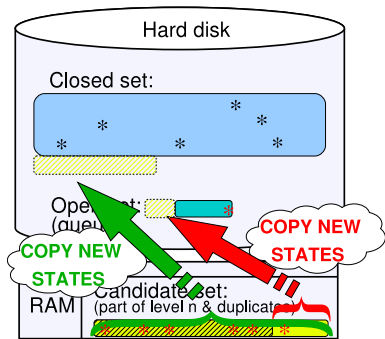
Merge Omission and Revisiting

Merge omission

Standard Travers.



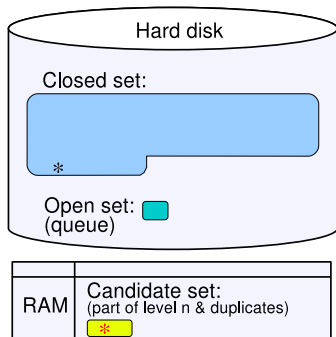
Modified Travers.



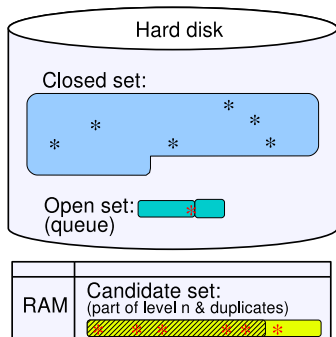
Merge Omission and Revisiting

Merge omission

Standard Travers.



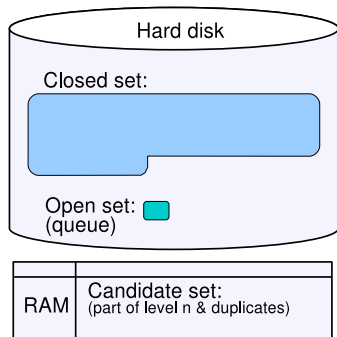
Modified Travers.



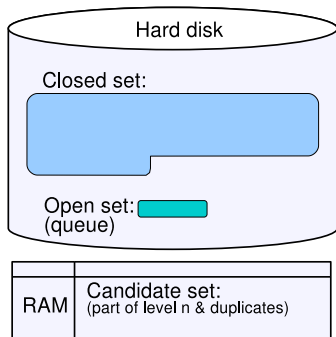
Merge Omission and Revisiting

Merge omission

Standard Travers.



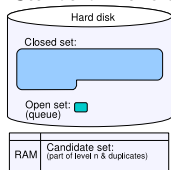
Modified Travers.



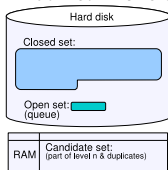
Merge Omission and Revisiting

Merge omission

Standard Travers.



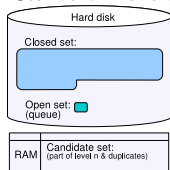
Modified Travers.



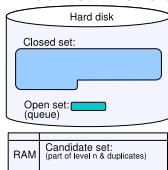
Merge Omission and Revisiting

Merge omission

Standard Travers.



Modified Travers.

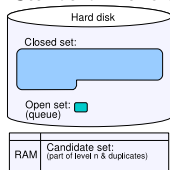


- Standard Travers.: Each state exactly **once** in *open set*

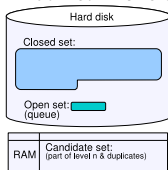
Merge Omission and Revisiting

Merge omission

Standard Travers.



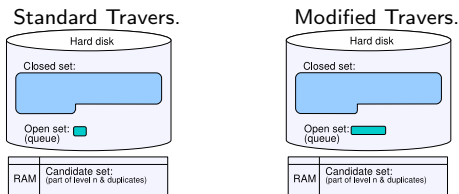
Modified Travers.



- Standard Travers.: Each state exactly **once** in *open set*
- Modified Travers.: Some states **more times** in *open set*

Merge Omission and Revisiting

Merge omission

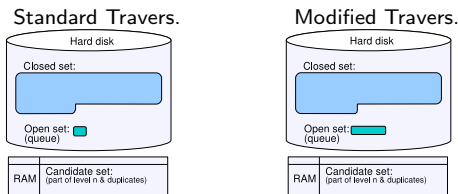


- Standard Travers.: Each state exactly **once** in *open set*
- Modified Travers.: Some states **more times** in *open set*
- But the following invariant holds in both versions:

Closed set contains exactly all explored states.

Merge Omission and Revisiting

Merge omission



- Standard Travers.: Each state exactly **once** in *open set*
- Modified Travers.: Some states **more times** in *open set*
- But the following invariant holds in both versions:
 Closed set contains exactly all explored states.
- \Rightarrow Heuristic omitting merge operations

Experimental Setting

- A single PC: Intel Pentium4 2 GHz (muticore), 2 GB RAM, 60 GB hard disc space available
- Implementation of the variant using RAM for delayed operations
- Comparison of external versions of MAP, OWCTY and EJ
- MAP-rr = modified version of MAP (omitting merge ops.)

Experimental Comparison

Valid properties.

	V	EJ	OWCTY	MAP	MAP-rr
Lamport	$7 \cdot 10^7$	OOS	02:37:17	03:16:36	02:37:56
Peterson	$3 \cdot 10^8$	OOS	18:20:03	25:09:35	15:24:29
Szymanski1	$4 \cdot 10^8$	OOS	45:52:25	59:35:25	29:09:12

Invalid properties.

	V	EJ	OWCTY	MAP	MAP-rr
Szymanski2	$4 \cdot 10^6$	00:00:50	00:20:07	00:00:04	00:00:02
Elevator	$4 \cdot 10^4$	00:01:02	00:00:25	00:00:05	00:00:01
Bakery	$5 \cdot 10^8$	00:25:59	68:23:34	00:00:09	00:00:23

Times are given in hh:mm:ss format

OOS = out of hard disk space

Conclusions

- Several I/O efficient LTL model checking algorithms exist:
 - **EJ** – on-the-fly, but bad practical complexity
 - **OWCTY** – not on-the-fly, but good complexity upper-bound and fast on models with valid properties
 - **MAP** – on-the-fly, bad complexity upper-bound, fast in practice (comparable to OWCTY on valid, winner on invalid)

Conclusions

- Several I/O efficient LTL model checking algorithms exist:
 - **EJ** – on-the-fly, but bad practical complexity
 - **OWCTY** – not on-the-fly, but good complexity upper-bound and fast on models with valid properties
 - **MAP** – on-the-fly, bad complexity upper-bound, fast in practice (comparable to OWCTY on valid, winner on invalid)
- Performing large-scale model checking on cheap HW

Conclusions

- Several I/O efficient LTL model checking algorithms exist:
 - **EJ** – on-the-fly, but bad practical complexity
 - **OWCTY** – not on-the-fly, but good complexity upper-bound and fast on models with valid properties
 - **MAP** – on-the-fly, bad complexity upper-bound, fast in practice (comparable to OWCTY on valid, winner on invalid)
- Performing large-scale model checking on cheap HW
- Ways to do better:
 - Heuristics applicable (merge omission, Bloom filters, compression)
 - Better HW would bring better performance

Conclusions

- Several I/O efficient LTL model checking algorithms exist:
 - **EJ** – on-the-fly, but bad practical complexity
 - **OWCTY** – not on-the-fly, but good complexity upper-bound and fast on models with valid properties
 - **MAP** – on-the-fly, bad complexity upper-bound, fast in practice (comparable to OWCTY on valid, winner on invalid)
- Performing large-scale model checking on cheap HW
- Ways to do better:
 - Heuristics applicable (merge omission, Bloom filters, compression)
 - Better HW would bring better performance
- Ongoing research – *P. Šimeček, S. Edelkamp and P. Sanders*: Semi-External LTL Model Checking