Parallel Nested Depth-First Searches for LTL Model Checking

Sami Evangelista Laure Petrucci Samir Youcef

LIPN — Université Paris 13

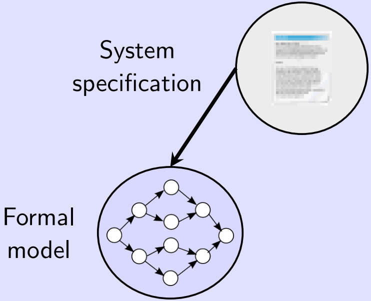
Vendredi 11 mars 2011
MeFoSyLoMa

Overview

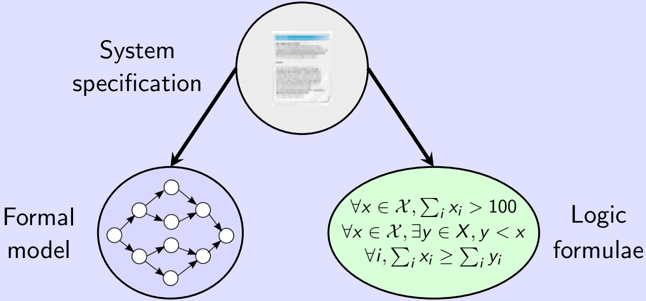
The model checking approach

System specification

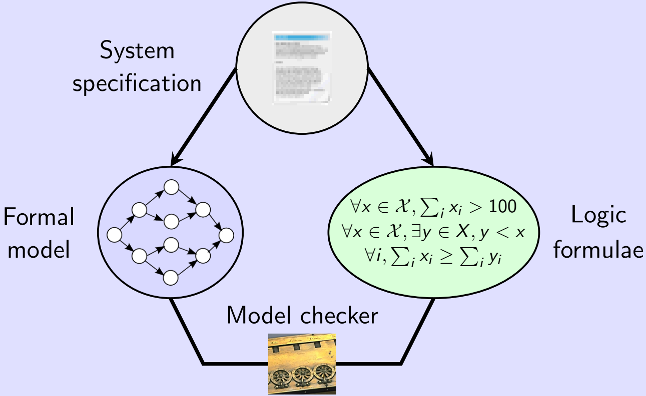
The model checking approach



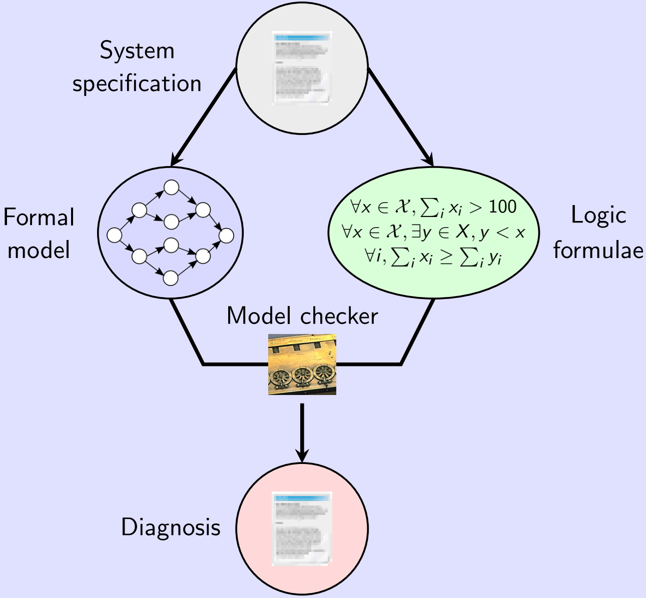
The model checking approach



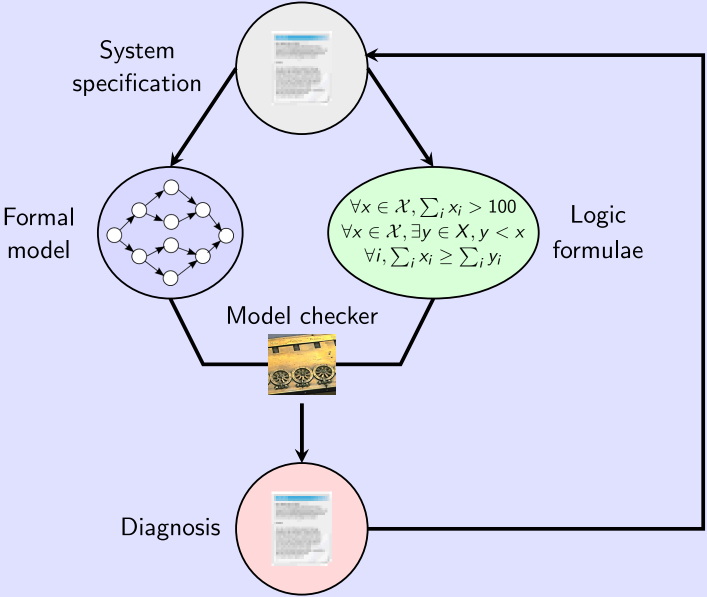
The model checking approach



The model checking approach



The model checking approach



The automata theoretic approach to LTL model checking

- formal model $=$ directed graph $\mathcal{S}$ representing system's dynamic
- logic formula $=$ a property ϕ with temporal operators (e.g., until, next)

The automata theoretic approach to LTL model checking

- formal model = directed graph $\mathcal{S}$ representing system's dynamic
- logic formula = a property ϕ with temporal operators (e.g., until, next)
- the model checker
  1. builds the Büchi automaton $\mathcal{B}_{\neg\phi}$ of the negation of ϕ
  2. builds the synchronized product $\mathcal{G} = \mathcal{S} \times \mathcal{B}_{\neg\phi}$
  3. checks for the emptiness of $\mathcal{G}$
     - Does $\mathcal{G}$ have a cycle going through an accepting state of $\mathcal{B}_{\neg\phi}$?

The automata theoretic approach to LTL model checking

- formal model = directed graph $\mathcal{S}$ representing system's dynamic
- logic formula = a property ϕ with temporal operators (e.g., until, next)
- the model checker
  1. builds the Büchi automaton $\mathcal{B}_{\neg\phi}$ of the negation of ϕ
  2. builds the synchronized product $\mathcal{G} = \mathcal{S} \times \mathcal{B}_{\neg\phi}$
  3. checks for the emptiness of $\mathcal{G}$
     - Does $\mathcal{G}$ have a cycle going through an accepting state of $\mathcal{B}_{\neg\phi}$?

$\mathcal{G}$ is empty $\Leftrightarrow$ no execution validates $\neg\phi$ and the property holds

The automata theoretic approach to LTL model checking

- formal model = directed graph $\mathcal{S}$ representing system's dynamic
- logic formula = a property ϕ with temporal operators (e.g., until, next)
- the model checker
  1. builds the Büchi automaton $\mathcal{B}_{\neg\phi}$ of the negation of ϕ
  2. builds the synchronized product $\mathcal{G} = \mathcal{S} \times \mathcal{B}_{\neg\phi}$
  3. checks for the emptiness of $\mathcal{G}$
     - Does $\mathcal{G}$ have a cycle going through an accepting state of $\mathcal{B}_{\neg\phi}$?

  $\mathcal{G}$ is empty $\Leftrightarrow$ no execution validates $\neg\phi$ and the property holds
- **on-the-fly** model checking $\Leftrightarrow$ steps 2–3 are performed simultaneously

The automata theoretic approach to LTL model checking

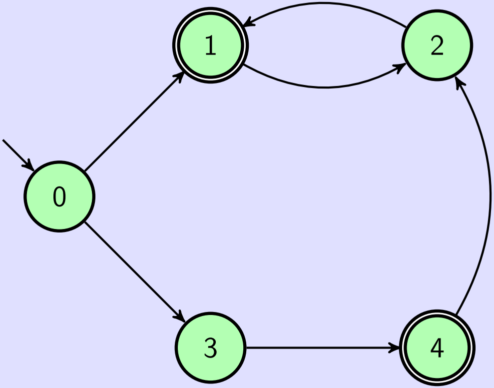
- formal model $=$ directed graph $\mathcal{S}$ representing system's dynamic
- logic formula $=$ a property ϕ with temporal operators (e.g., until, next)
- the model checker
  1. builds the Büchi automaton $\mathcal{B}_{\neg\phi}$ of the negation of ϕ
  2. builds the synchronized product $\mathcal{G} = \mathcal{S} \times \mathcal{B}_{\neg\phi}$
  3. checks for the emptiness of $\mathcal{G}$
     - Does $\mathcal{G}$ have a cycle going through an accepting state of $\mathcal{B}_{\neg\phi}$?

  $\mathcal{G}$ is empty $\Leftrightarrow$ no execution validates $\neg\phi$ and the property holds
- **on-the-fly** model checking $\Leftrightarrow$ steps 2–3 are performed simultaneously

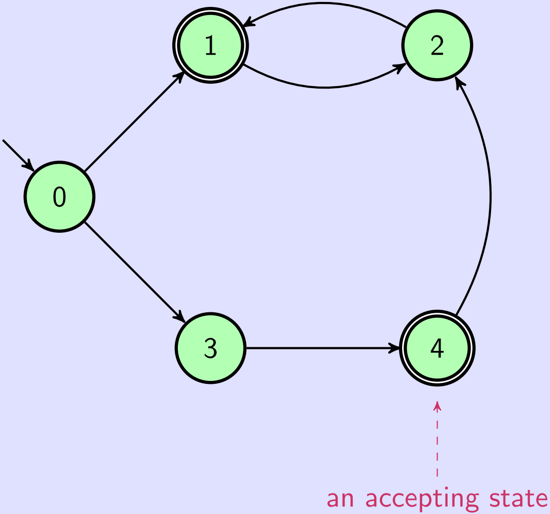
This talk

- an on-the-fly model checking algorithm (focussing on steps 2–3)
- for **multi-core** architectures with a **shared memory**
- adapted from classical nested depth-first search (used in SPIN)

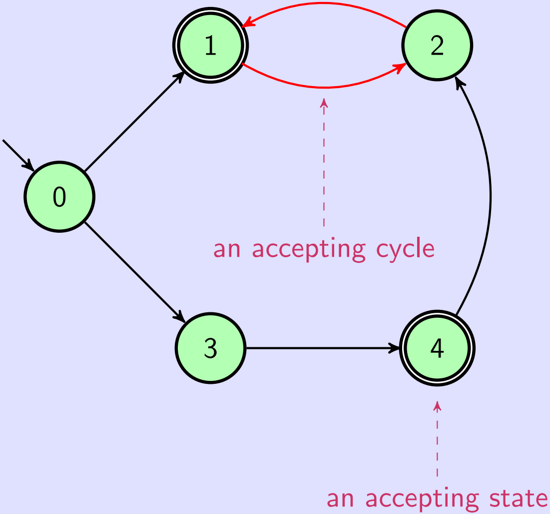
Synchronized graph — An example



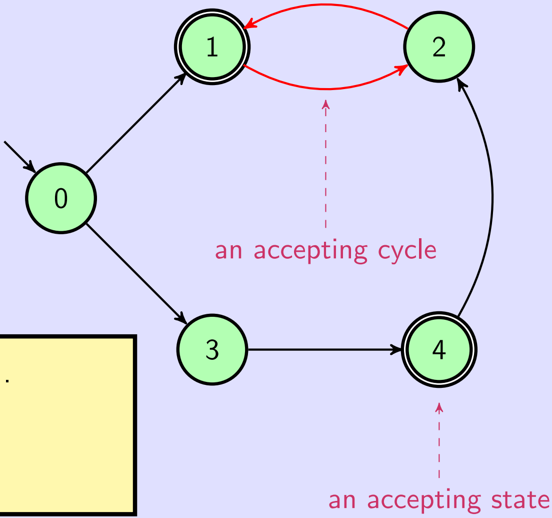
Synchronized graph — An example



Synchronized graph — An example

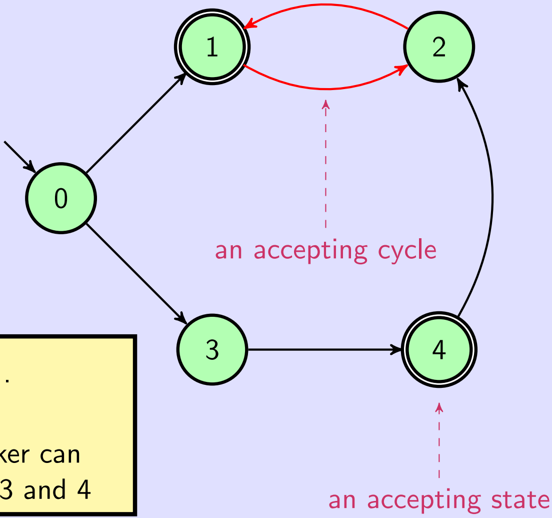


Synchronized graph — An example



- execution 0, 1, 2, 1, 2, . . . invalidates the property

Synchronized graph — An example



- execution 0, 1, 2, 1, 2, . . . invalidates the property
- an on-the-fly model checker can report it without visiting 3 and 4

Overview

Sequential algorithms for LTL model checking

Nested Depth-First search (ndfs)

Memory Efficient Algorithms for the Verification of Temporal Properties. CAV'1990. Courcoubetis, Vardi, Wolper and Yannakakis.

- Historical algorithm implemented by many model checkers, e.g., SPIN
- Principle: interleaving of two DFSs
  - a blue DFS that finds accepting states and launchs in DFS post-order
  - a red DFS that finds accepting cycles

Sequential algorithms for LTL model checking

Nested Depth-First search (ndfs)

Memory Efficient Algorithms for the Verification of Temporal Properties. CAV'1990. Courcoubetis, Vardi, Wolper and Yannakakis.

- Historical algorithm implemented by many model checkers, e.g., SPIN
- Principle: interleaving of two DFSs
  - a blue DFS that finds accepting states and launchs in DFS post-order
  - a red DFS that finds accepting cycles

Strongly connected component based (scc-ltl)

On-the-Fly Verification of Linear Temporal Logic. FM'1999. Couvreur

Tarjan's Algorithm Makes On-the-Fly LTL Verification More Efficient. TACAS'2004. Geldenhuys and Valmari.

- Principle: adaptation of Tarjan's algorithm for finding SCC

Comparison of NDFS and SCC-LTL

- linear complexities for both
- ndfs uses less memory: 2 bits / state vs. 1–2 integers for scc-ltl
- but scc-ltl usually reports counter-examples faster.

⇒ both have their strengths

Experimental comparison in :

- **A Note on On-the-Fly Verification Algorithms**. TACAS'2005. Schwoon and Esparza.
- **On-the-Fly Emptiness Checks for Generalized Bchi Automata**. SPIN'2005. Couvreur, Duret-Lutz and Poitrenaud.
- **Comparison of Algorithms for Checking Emptiness on Büchi Automata**. MEMICS'2009. Gaiser and Schwoon.

Parallel Algorithms for LTL Model Checking

- most algorithms are designed for distributed memory architectures
- but these can be easily adapted to shared memory architectures
- usually rely on a BFS because DFS is hard to parallelize:

  Depth-First Search is Inherently Sequential. IPL'1985. Reif.
- characterized by different "on-the-flyness" levels:
  0 off-line: first we build the synchronised graph then we check
  1 early termination possible but not guaranteed in the presence of an accepting cycle
  2 on-the-fly: early termination guaranteed in the presence of an accepting cycle

Multi-Core Algorithms for LTL Model Checking

Algo.	Source	Time Comp.	Proc.	Acceleration	OTF
2-ndfs	TSE'07	$\mathcal{O}(n+m)$	1–2	average	2
map	FMCAD'04	$\mathcal{O}(a^2 \cdot m)$	1–N	excellent	1
owcty	SPIN'03	$\mathcal{O}(h \cdot m)$	1–N	excellent	0
owcty-otf	ICFEM'09	$\mathcal{O}(h \cdot (m+n))$	1–N	excelent	1
negc	FSTTCS'01	$\mathcal{O}(n \cdot m)$	1–N	excellent	0
bledge	ASE'03	$\mathcal{O}(m \cdot (n+m))$	1–N	excellent	0
bledge-otf	ENTCS'05	$\mathcal{O}(m \cdot (n+m))$	1–N	excellent	2
mc-ndfs	this talk	$\mathcal{O}(p \cdot (n+m))$	1–N	average-good	2

- n, m = states and edges in the graph
- a = accepting states
- h = height of the graph
- p = working processes
- acceleration obtained through experimentations
- OTF = on-the-flyness

Overview

Principle of NDFS

- based on two DFSs
- a **blue** DFS is used to find all accepting states
- when it backtracks from an accepting state a the **red** DFS is initiated
- the red DFS tries to find a way back to a
  a is called a seed state
- each state is explored at most twice (by the blue or red DFS)
- requires two bits per state to remember explored states

Pseudo-code of the algorithm

Main procedure

```
for each state s
   s.blue := false
   s.red  := false

dfsBlue (initial state)
```

Pseudo-code of the algorithm

Main procedure

```
for each state s
   s.blue := false
   s.red  := false

dfsBlue (initial state)
```

The blue DFS

```
dfsBlue (s)
   s.blue := true
   for (s' in succ (s))
      if (not s'.blue)
         dfsBlue (s')
   if (s is accepting)
      seed := s
      dfsRed (s)
```

Pseudo-code of the algorithm

Main procedure

```
for each state s
   s.blue := false
   s.red  := false

dfsBlue (initial state)
```

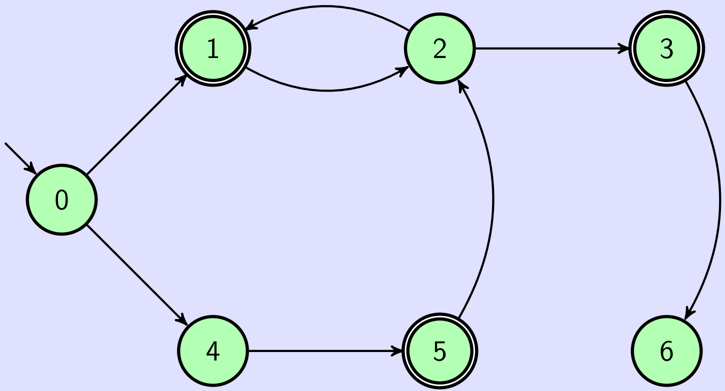
The blue DFS

```
dfsBlue (s)
   s.blue := true
   for (s’ in succ (s))
      if (not s’.blue)
         dfsBlue (s’)
   if (s is accepting)
      seed := s
      dfsRed (s)
```

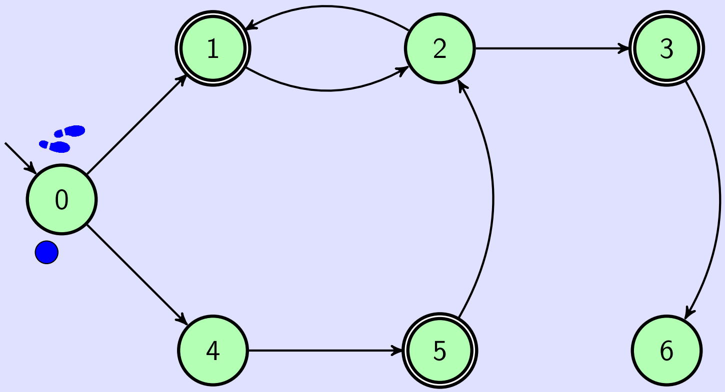
The red DFS

```
dfsRed (s)
   s.red := true
   for (s’ in succ (s))
      if (s’ = seed)
         print "Cycle Found"
      else if (not s’.red)
         dfsRed (s’)
```

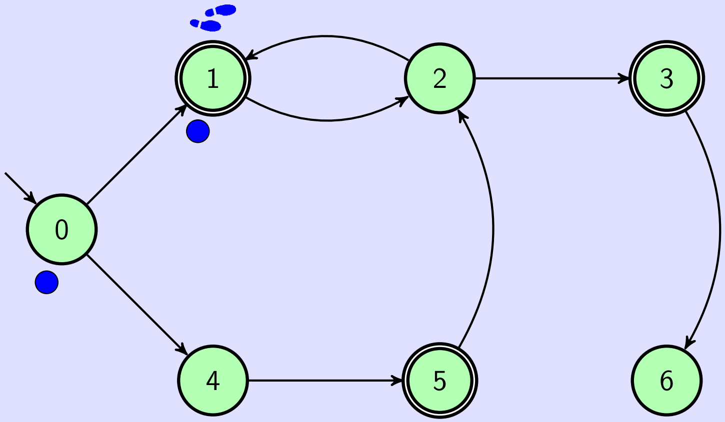
NDFS — Example 1



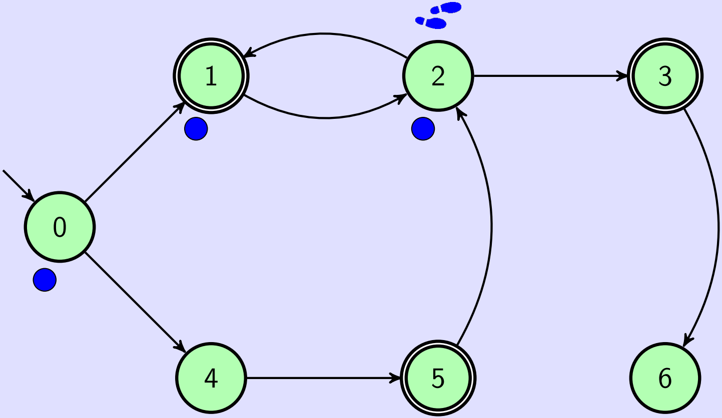
NDFS — Example 1



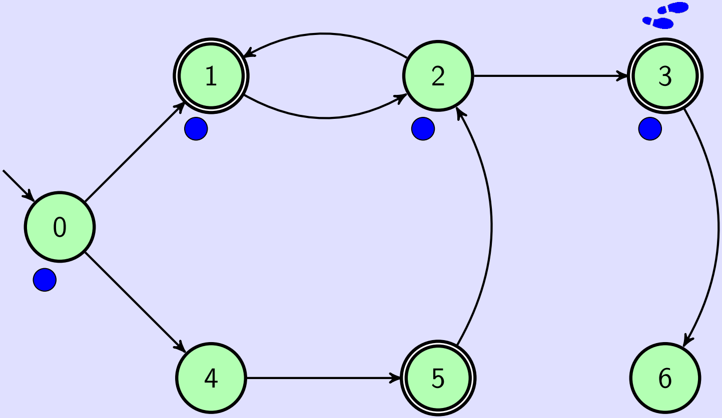
NDFS — Example 1



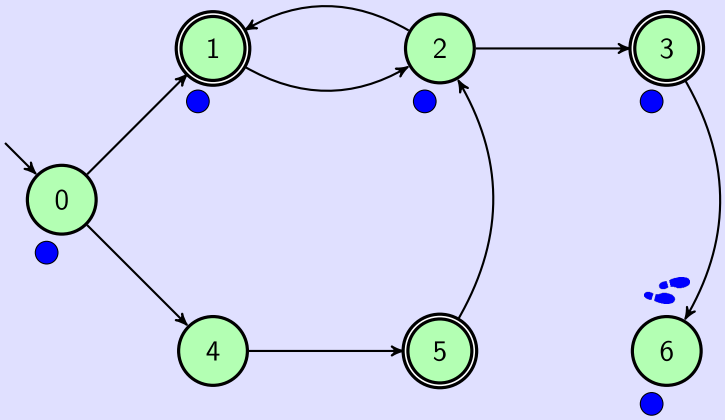
NDFS — Example 1



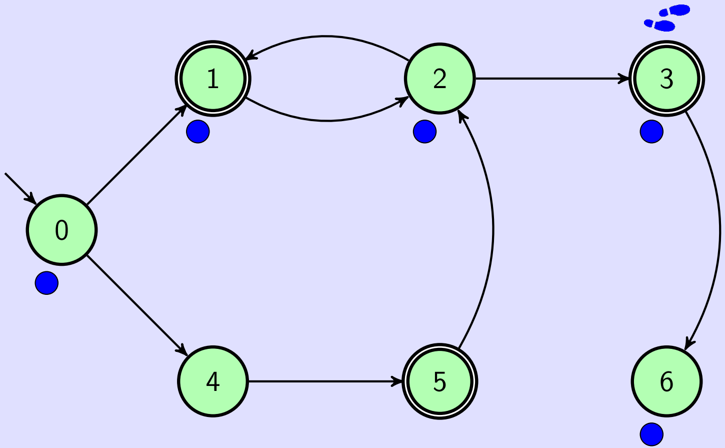
NDFS — Example 1



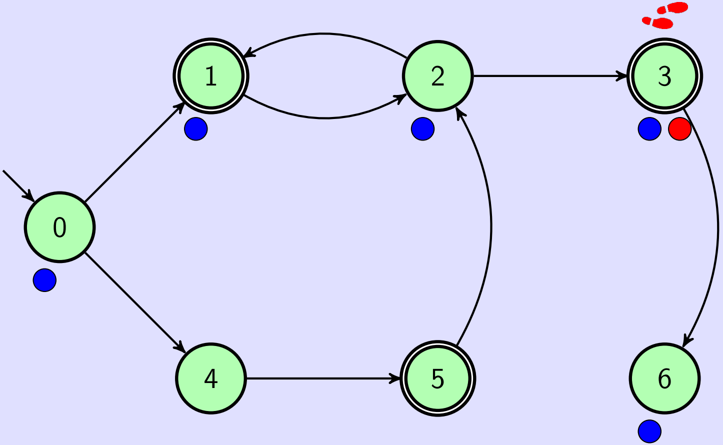
NDFS — Example 1



NDFS — Example 1

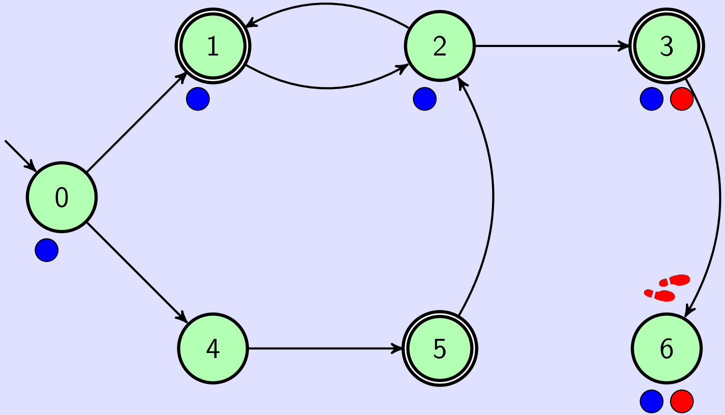


NDFS — Example 1

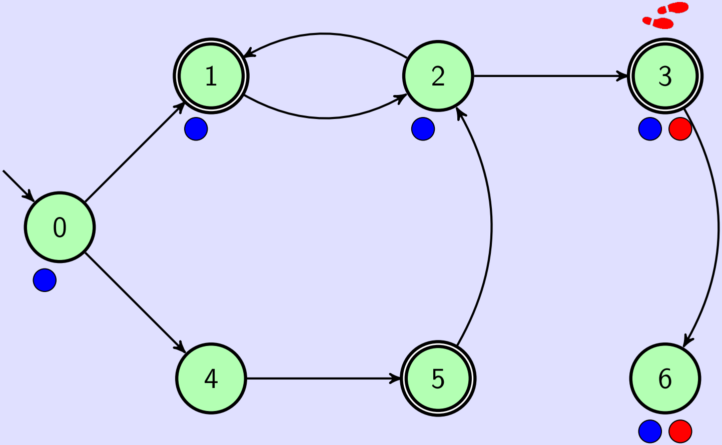


start a red DFS with 3 as seed

NDFS — Example 1

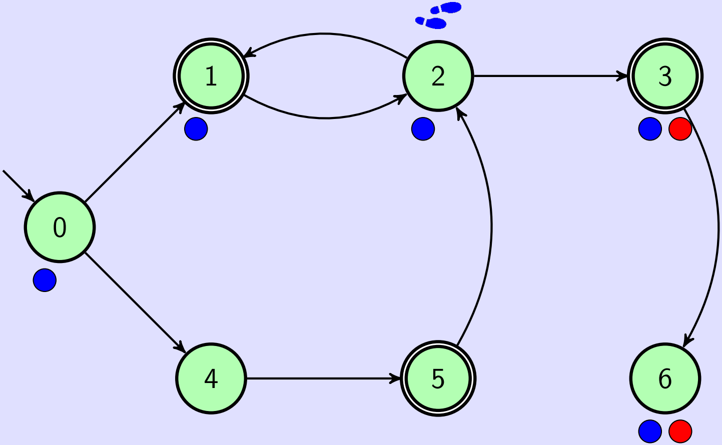


NDFS — Example 1

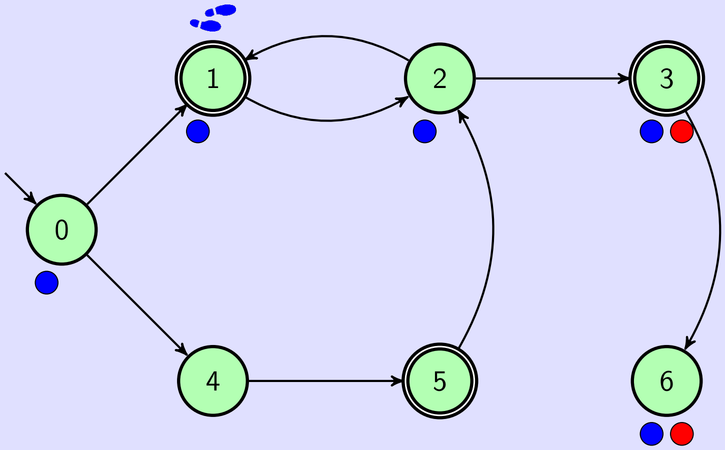


the red DFS terminates ⇒
no accepting cycle around 3

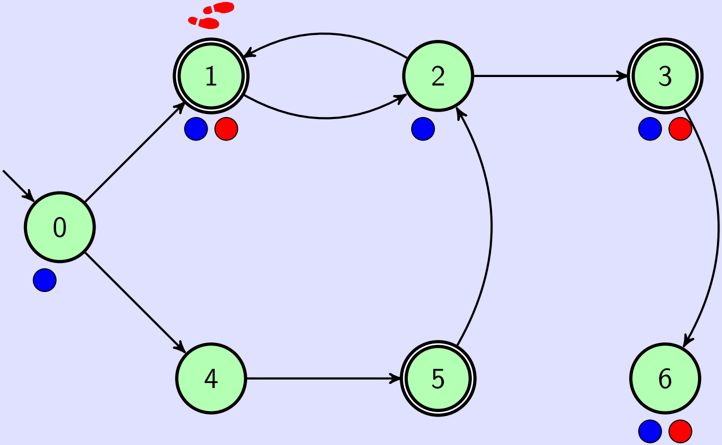
NDFS — Example 1



NDFS — Example 1

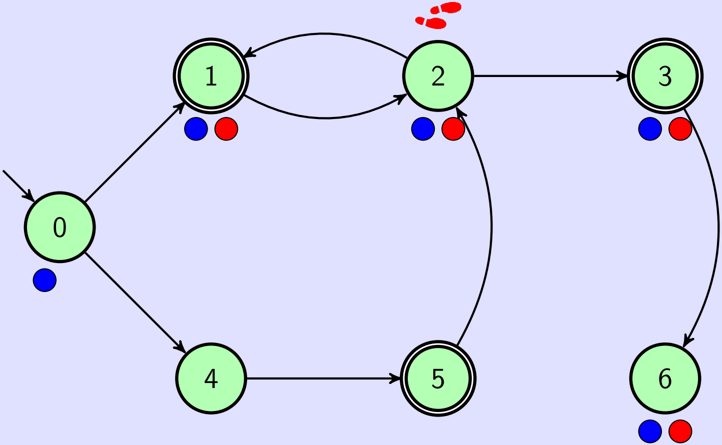


NDFS — Example 1

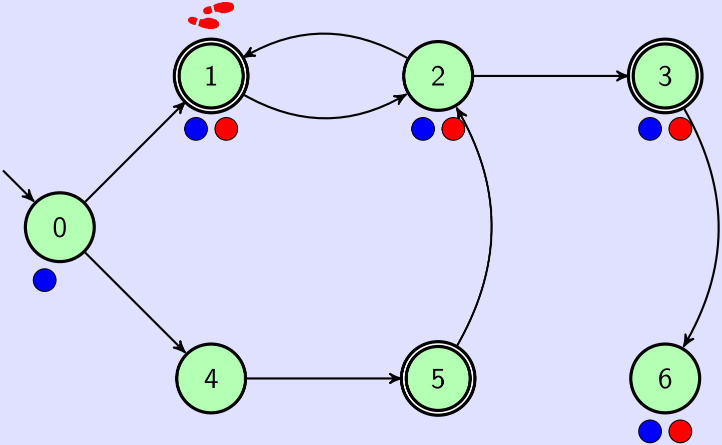


start a red DFS with 1 as seed

NDFS — Example 1

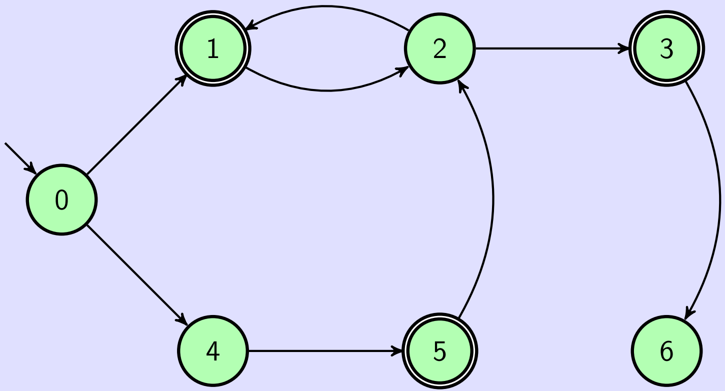


NDFS — Example 1

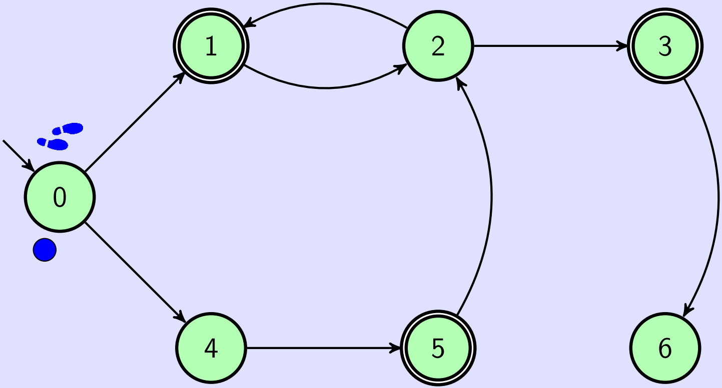


we reach the seed ⇒
accepting cycle found

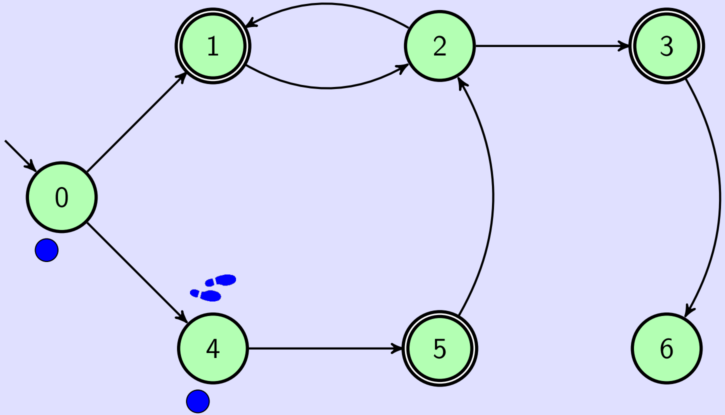
NDFS — Example 2



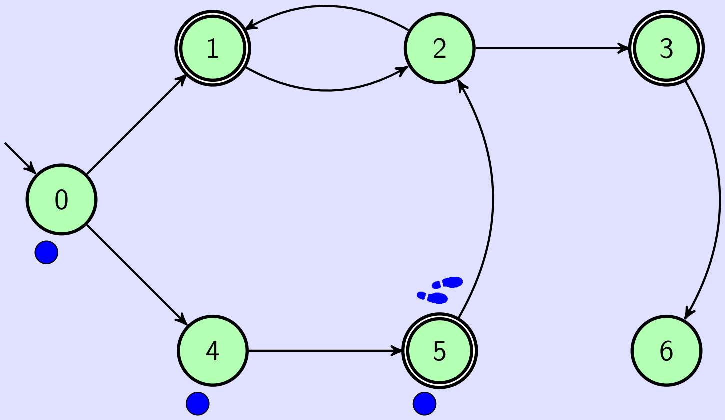
NDFS — Example 2



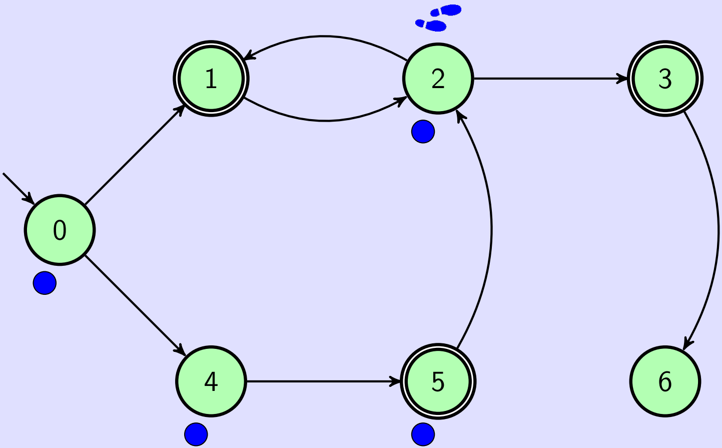
NDFS — Example 2



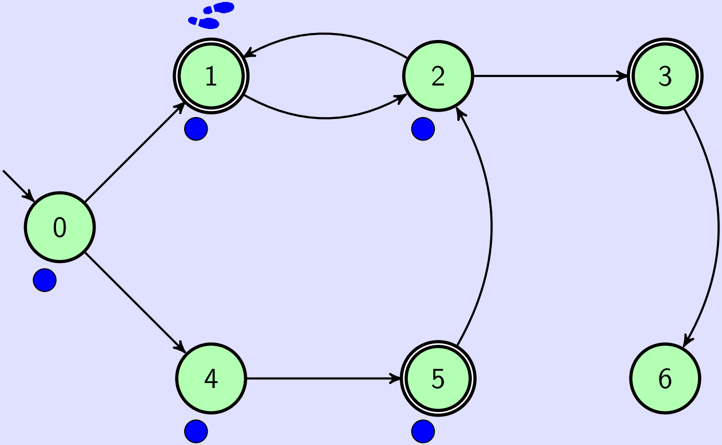
NDFS — Example 2



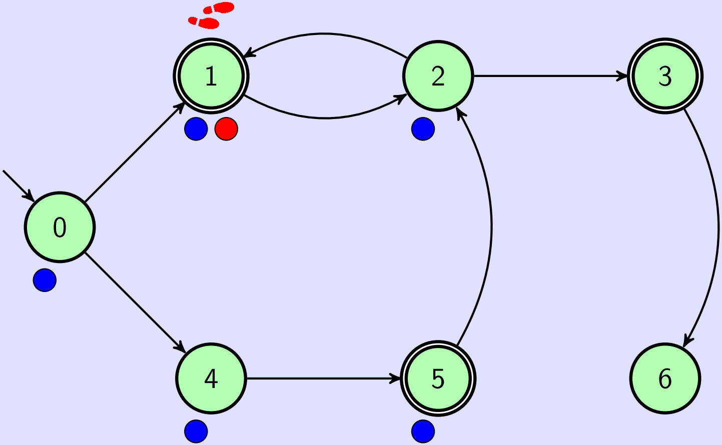
NDFS — Example 2



NDFS — Example 2

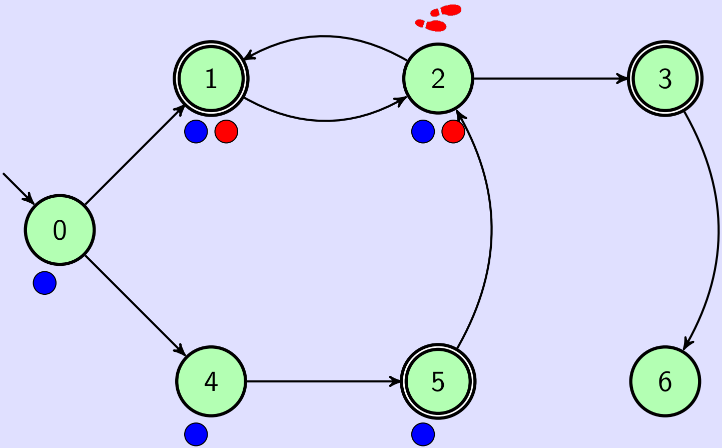


NDFS — Example 2

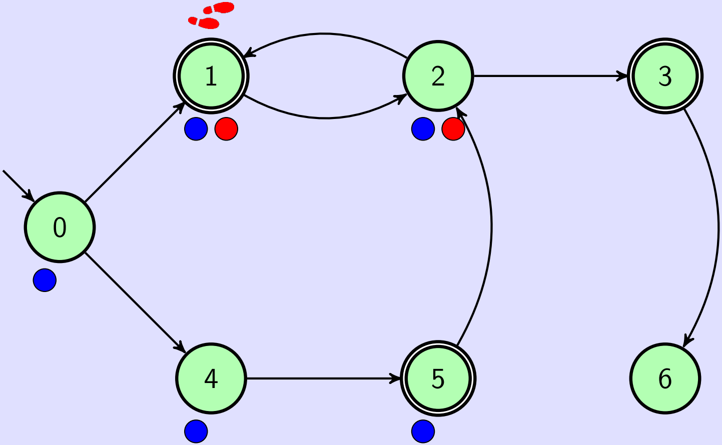


start a red DFS with 1 as seed

NDFS — Example 2



NDFS — Example 2



we reach the seed ⇒
accepting cycle found

Overview

Motivations

our goal: design an LTL model checking algorithm

- for multi-core architectures with shared memory
- that works on-the-fly

Motivations

our goal: design an LTL model checking algorithm

- for multi-core architectures with shared memory
- that works on-the-fly

why?

- such architectures are now widely available
- and with
  - all reduction techniques (partial order, symmetry, state compression)
  - and the amount of RAM available

  we can also face a time explosion problem

$\Rightarrow$ a multi-threaded algorithm can help us with that

Motivations

our goal: design an LTL model checking algorithm

- for multi-core architectures with shared memory
- that works on-the-fly

why?

- such architectures are now widely available
- and with
  - all reduction techniques (partial order, symmetry, state compression)
  - and the amount of RAM available

  we can also face a time explosion problem

$\Rightarrow$ a multi-threaded algorithm can help us with that

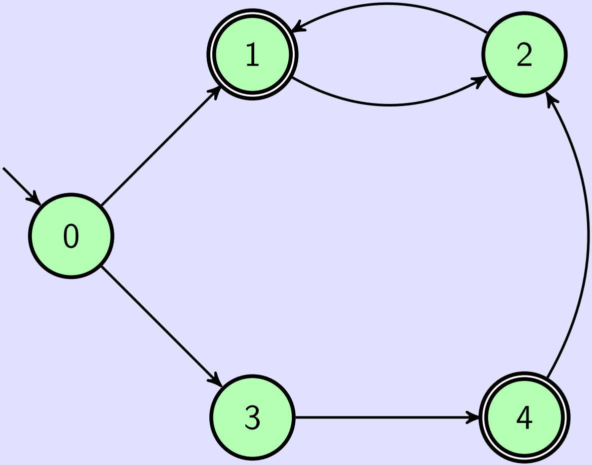
our starting point: ndfs

Why is it difficult to parallelize ndfs?

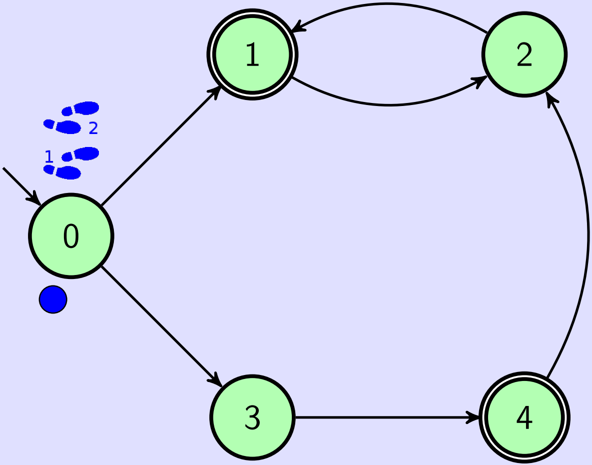
a naive multi-threaded version of ndfs:

- threads are launched concurrently
- each thread performs the ndfs algorithm
- threads share all blue and red bits of ndfs

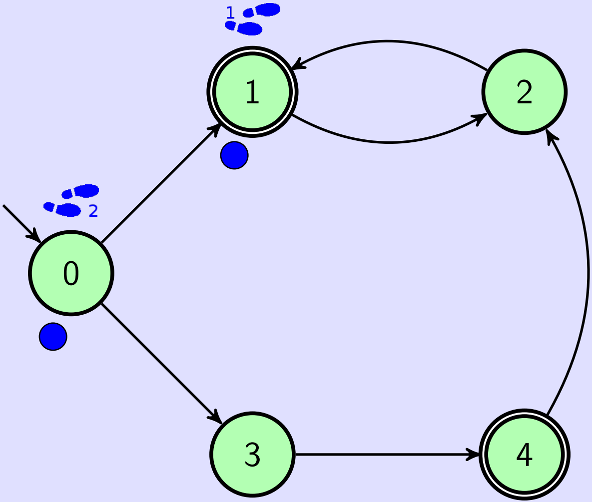
Why is it difficult to parallelize ndfs? — An example



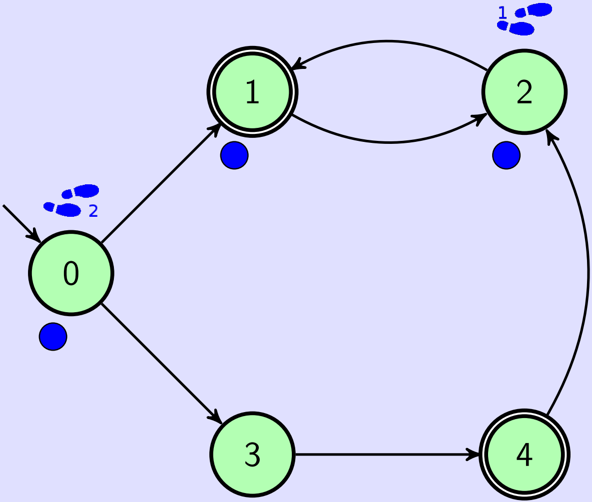
Why is it difficult to parallelize ndfs? — An example



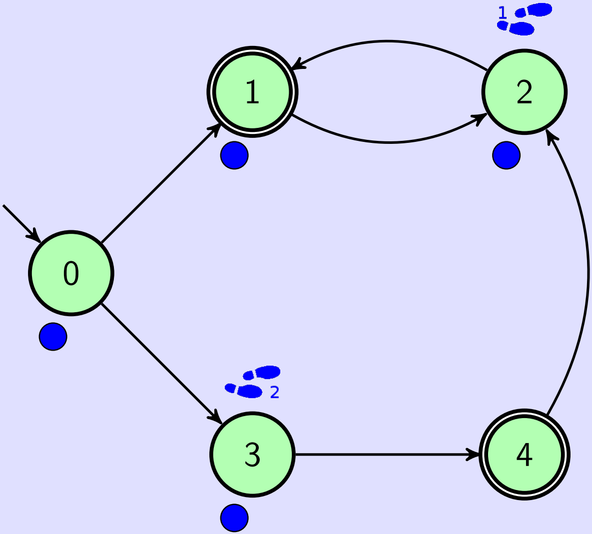
Why is it difficult to parallelize ndfs? — An example



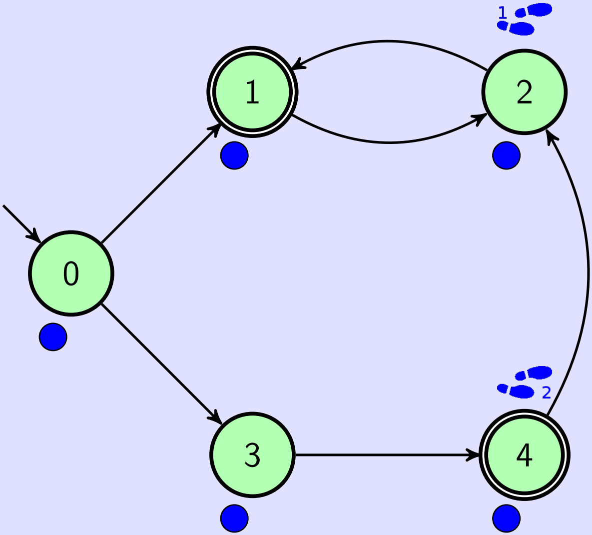
Why is it difficult to parallelize ndfs? — An example



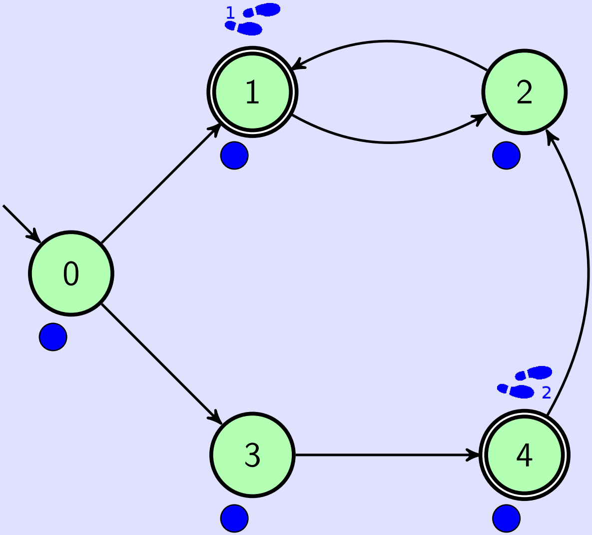
Why is it difficult to parallelize ndfs? — An example



Why is it difficult to parallelize ndfs? — An example

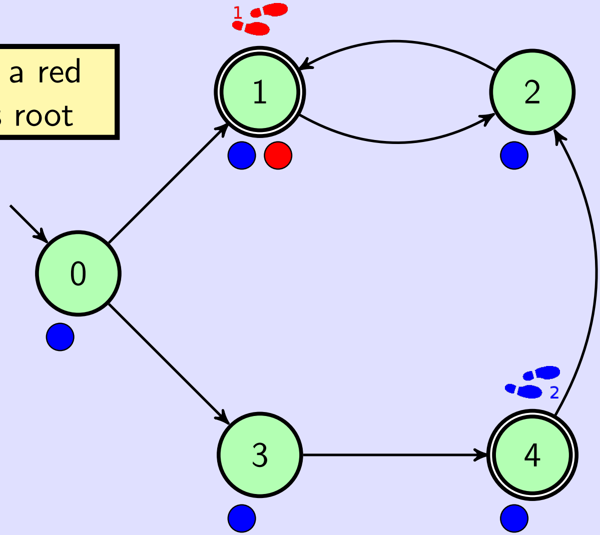


Why is it difficult to parallelize ndfs? — An example



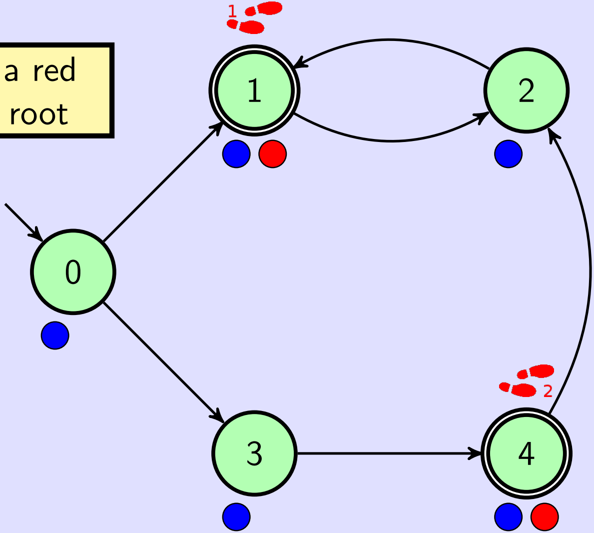
Why is it difficult to parallelize ndfs? — An example

thread 1 starts a red DFS with 1 as root

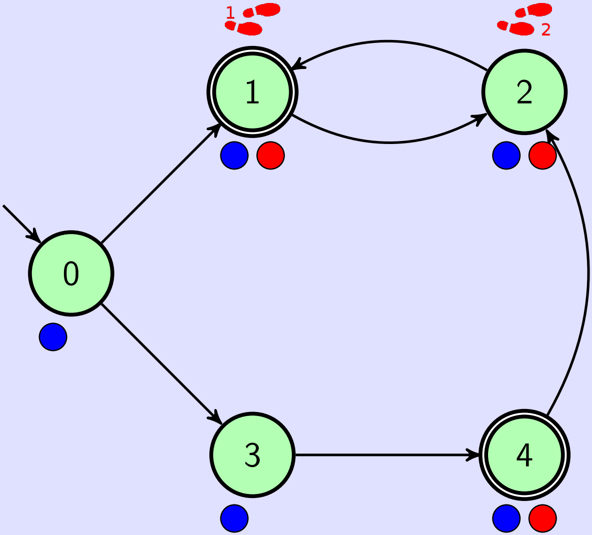


Why is it difficult to parallelize ndfs? — An example

thread 2 starts a red DFS with 4 as root

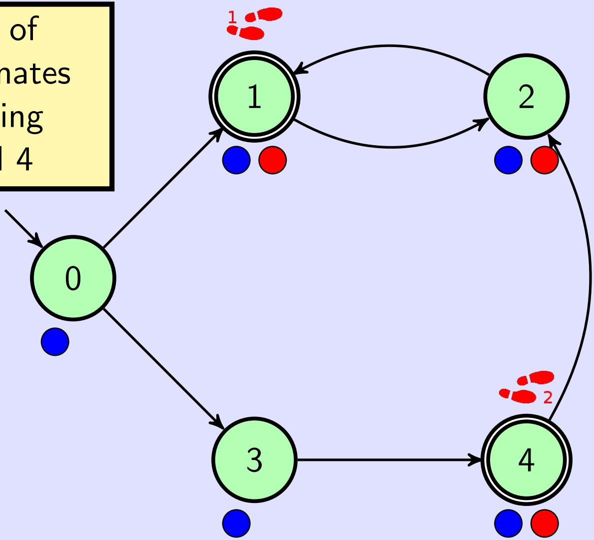


Why is it difficult to parallelize ndfs? — An example



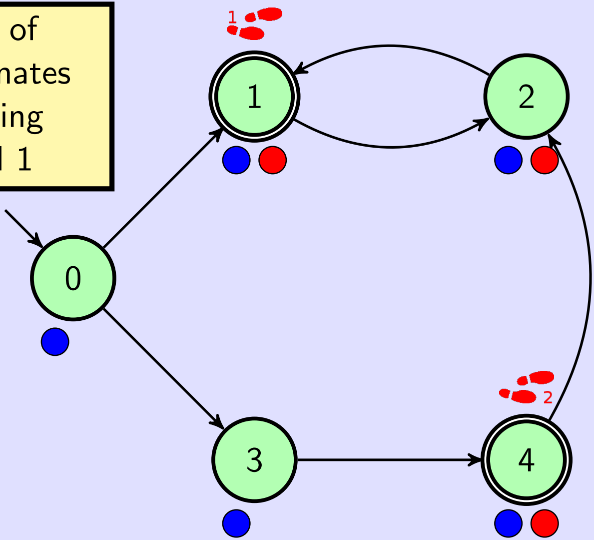
Why is it difficult to parallelize ndfs? — An example

the red DFS of
thread 2 terminates
$\Rightarrow$ no accepting
cycle around 4



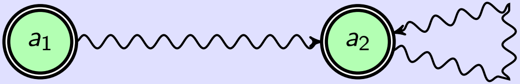
Why is it difficult to parallelize ndfs? — An example

the red DFS of
thread 1 terminates
⇒ no accepting
cycle around 1



Why is it difficult to parallelize ndfs?

- because the invokation order of the red DFS is important!
- basically if we have two accepting states a_1 and a_2 with
  - $a_1 \rightsquigarrow a_2 \wedge \neg a_2 \rightsquigarrow a_1$
  - $a_1 \notin$ an accepting cycle and $a_2 \in$ an accepting cycle:



- then $dfsRed(a_1)$ must be invoked **after** $dfsRed(a_2)$
  - otherwise $dfsRed(a_1)$ will mark states around a_2 as red
  - and $dfsRed(a_2)$ will not discover the accepting cycle around a_2
- we call this situation a **conflict**
- why does ndfs work? because the red DFS is nested in the blue DFS

$\Rightarrow$ $dfsRed(a_2)$ will be invoked **before** $dfsRed(a_1)$

- but a naive multi-core ndfs does not preserve this order

Principle of MC-NDFS

- mc-ndfs = multi-core ndfs
- mc-ndfs spawns multiple threads that all execute (a modified) ndfs
- exploration based on 2 principles: **randomisation** and **synchronisation**
- randomisation: threads explore the graph in a random way so that they (hopefully) engage in different parts of the graph
- synchronisation: shared memory is used to avoid as much as possible redundant revisits by different threads

Resolution of conflicts

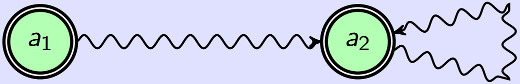
- mc-ndfs follows an **optimistic** approach to resolve conflicts:
  - we let threads explore the graph without taking care of conflicts
  - there is a way to detect when these conflicts occur
  - in that case, a thread relaunchs a red DFS by only using local data
  - why?
    - because shared attributes modified by other threads have corrupted the result of a red DFS launched
    - using only local data we simulate ndfs and are thus on the safe side
- thus we have two layers algorithm
  - a **multi-core** layer with inter-process synchronisation
  - an **emergency** (without synchronisation) layer triggered in case of conflict

✔ we avoid all synchronisations/waitings due to the prevention of conflicts

✘ in case of conflicts, states will be revisited multiple times

How do we detect and fix conflicts?

- a conflict occurs when a red DFS initiated on a_1 reaches a_2 that is
  - accepting
  - but not red ($\Rightarrow$ the red DFS on a_2 has necessarily not terminated)
- this situation possibly corresponds to a conflict:

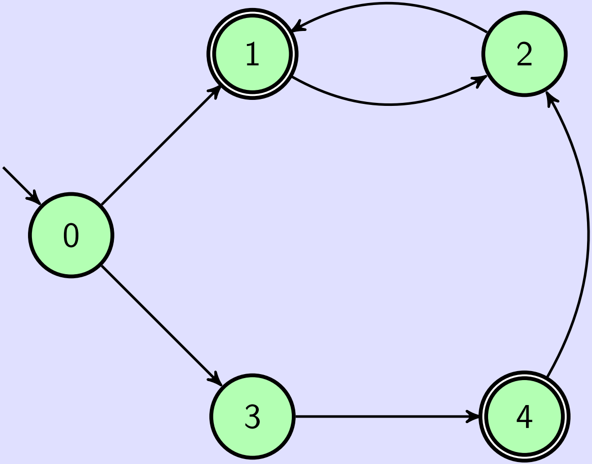


- what do we do then? we mark a_2 as **dangerous**

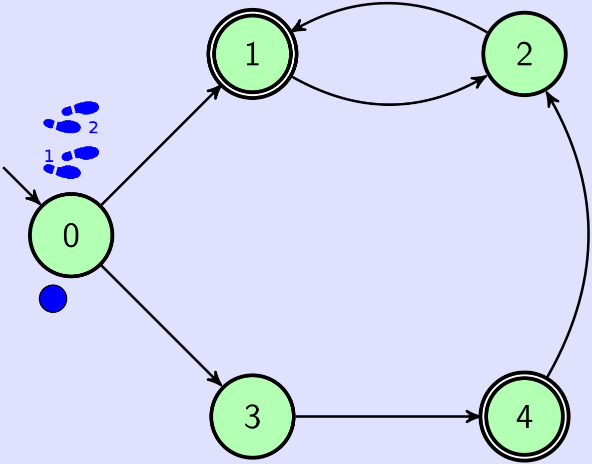
$\Rightarrow$ this means that the emergency level must be triggered for a_2

- the red DFS that will be initiated on a_2 does not report an accepting cycle $\Rightarrow$ relaunch a red DFS on a_2 and ignore global red flags

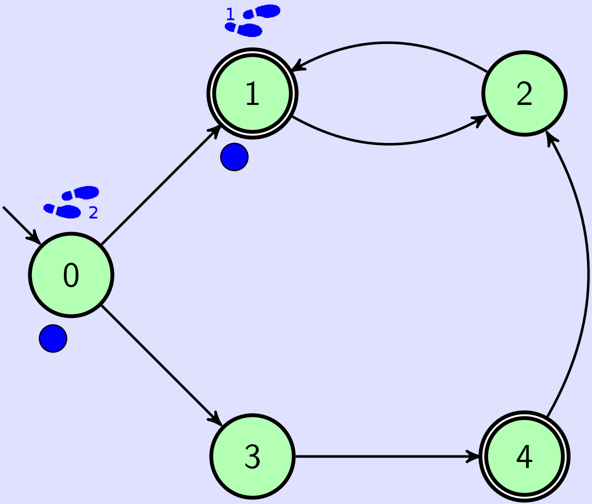
Example



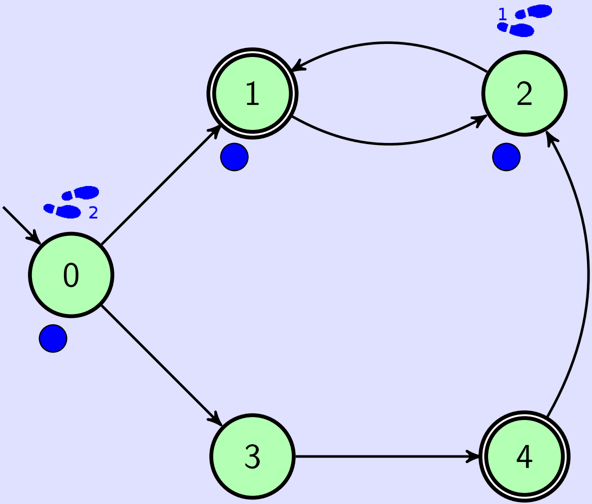
Example



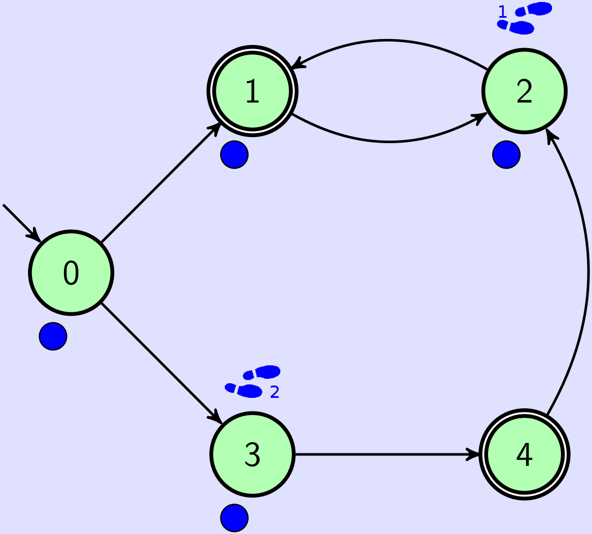
Example



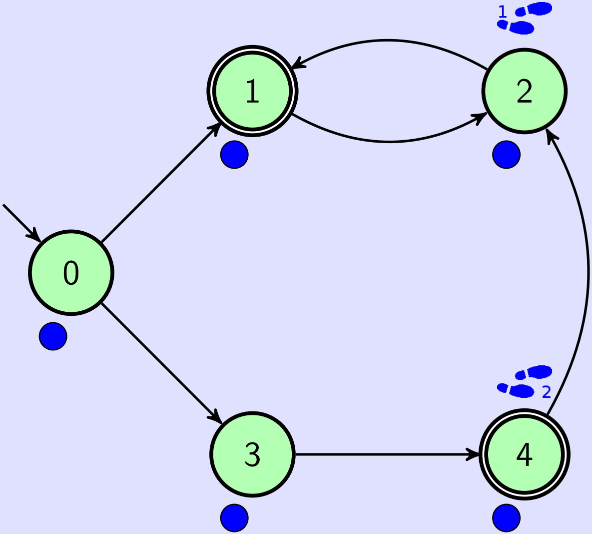
Example



Example

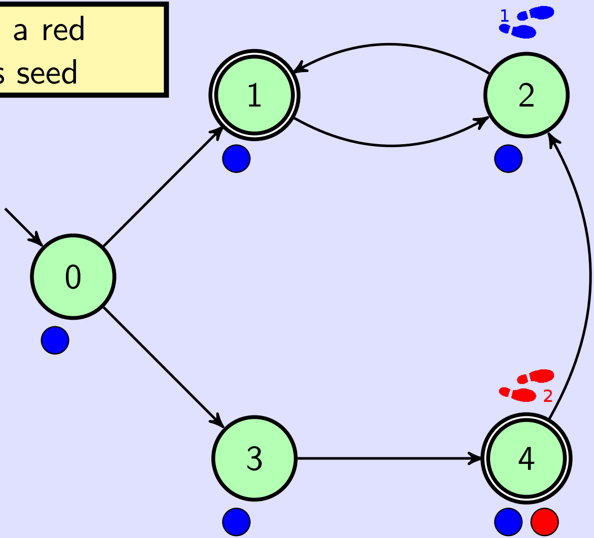


Example

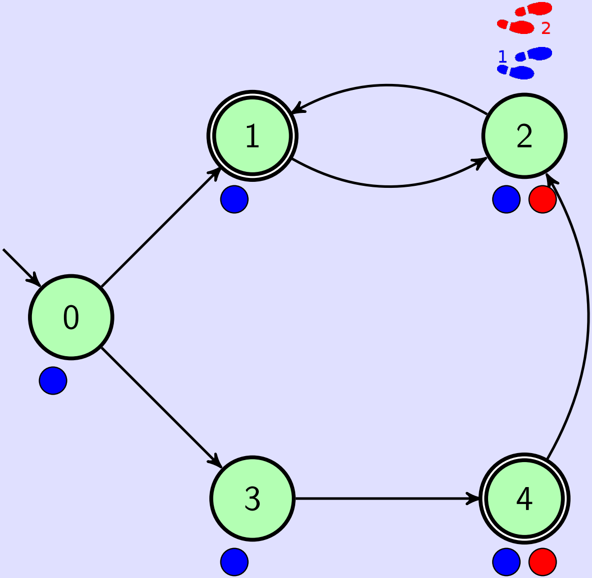


Example

thread 2 starts a red DFS with 4 as seed

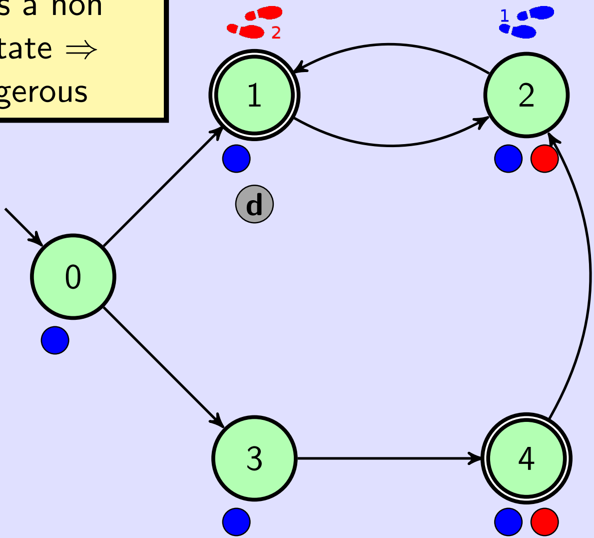


Example

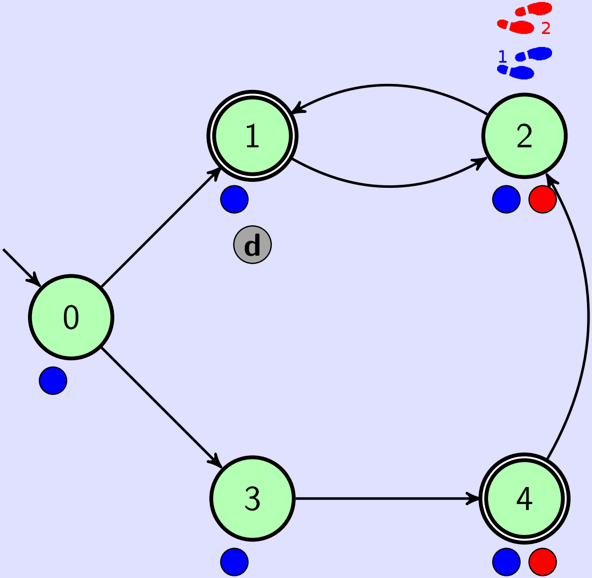


Example

thread 2 reaches a non red accepting state ⇒ mark 1 as dangerous

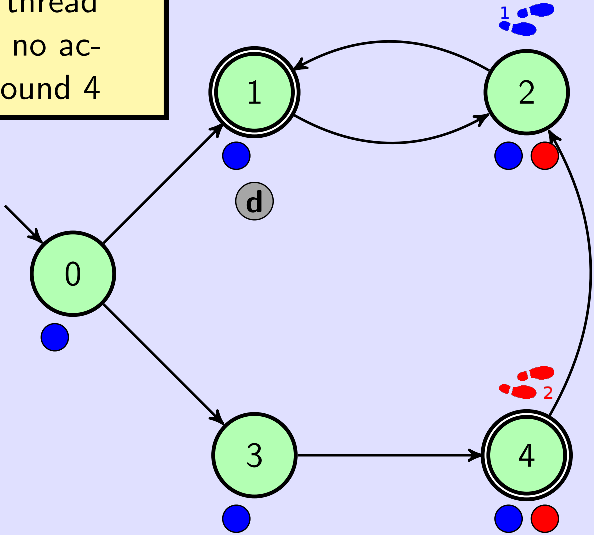


Example

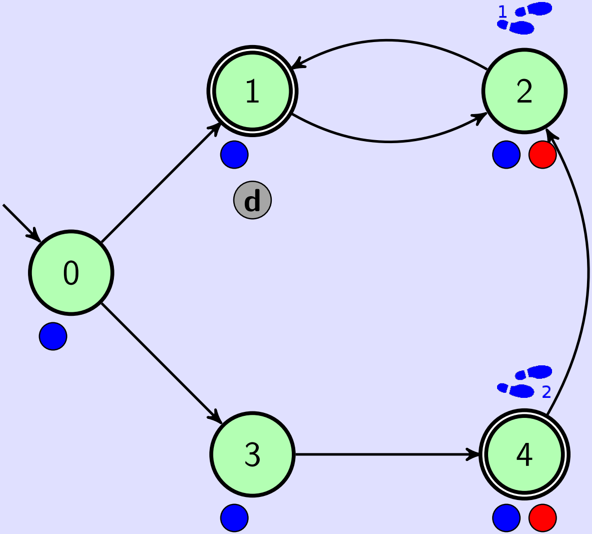


Example

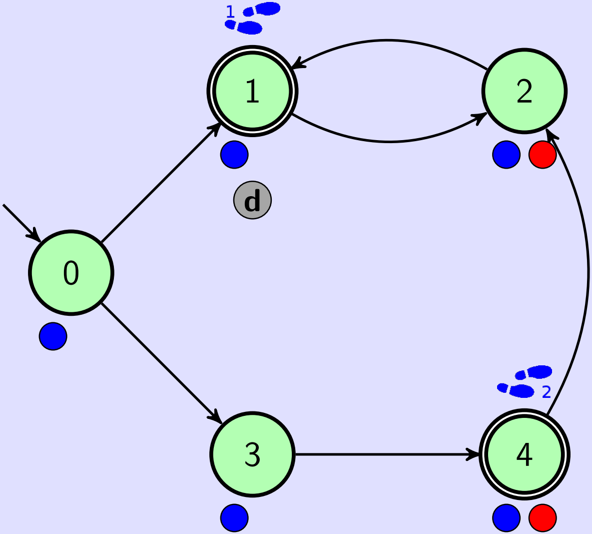
the red DFS of thread 2 terminates ⇒ no accepting cycle around 4



Example

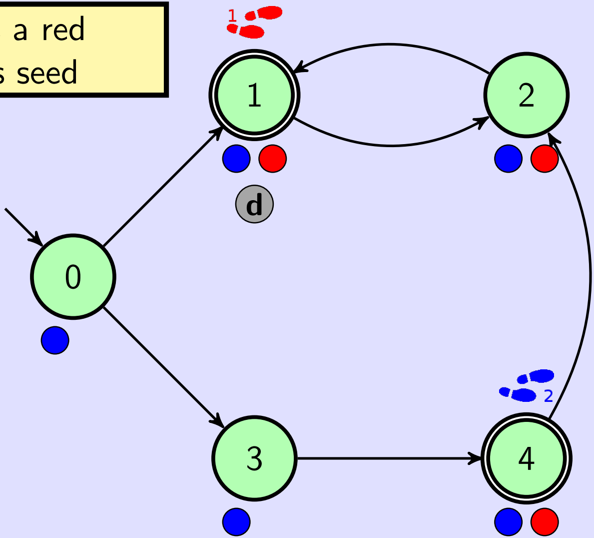


Example



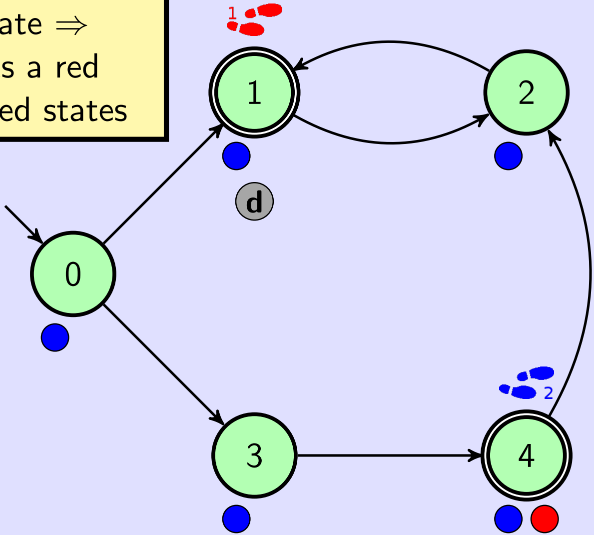
Example

thread 1 starts a red
DFS with 1 as seed

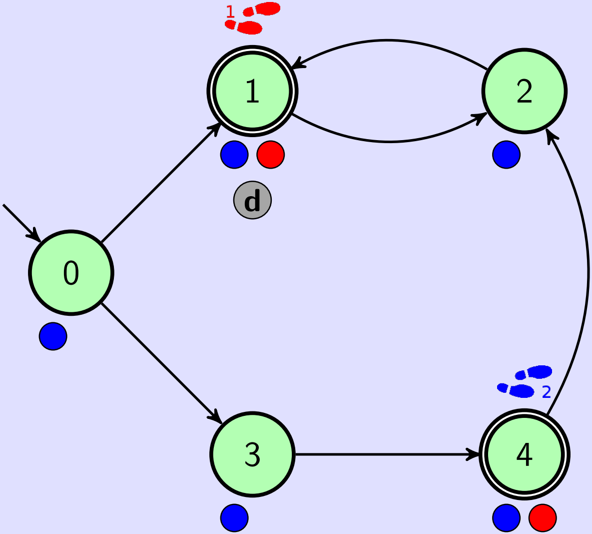


Example

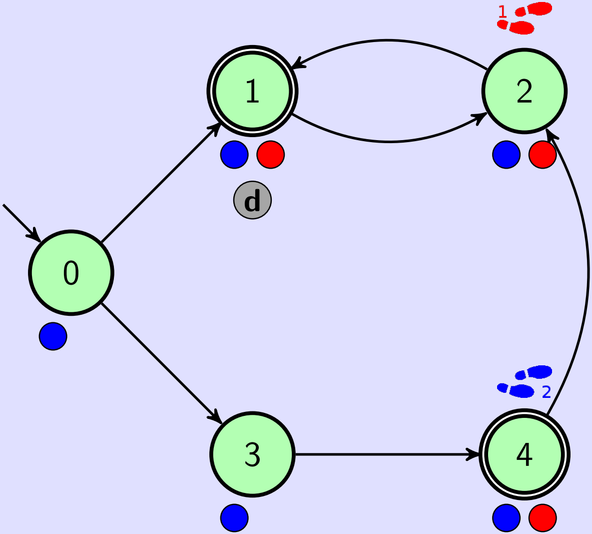
red DFS terminated on a dangerous state ⇒ thread 1 restarts a red DFS and ignore red states



Example

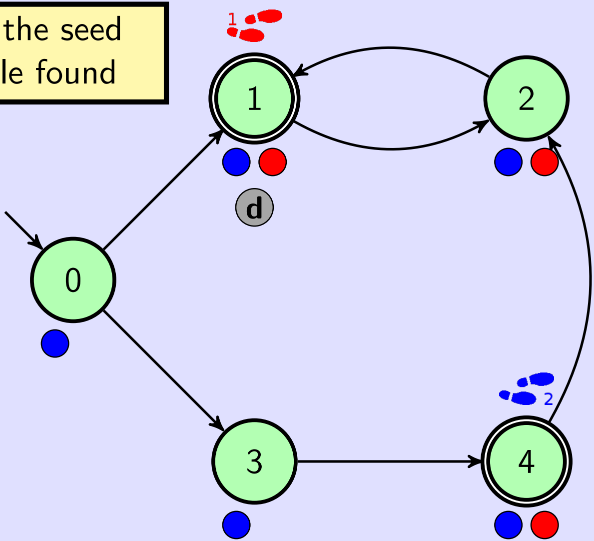


Example



Example

thread 1 reaches the seed
⇒ accepting cycle found



Time complexity

- in the worst case, a thread will explore each state of the graph
- then mc-ndfs is equivalent to spawn p unsynchronised instances of ndfs
- and we do not gain anything through multi-threading
- "good" input graphs: graphs clustered in many small SCCs (or acyclic)
  - with randomization threads visit different SCCs $\Rightarrow$ no/few state revisits
- "bad" input graphs: graphs with a single SCC

  $\Rightarrow$ lot of state revisits

Overview

Experimentation context

- prototype implementation in C on top of the pthread library
- algorithms exeperimented: mc-ndfs and map
- input models from the BEEM database:
  `http://anna.fi.muni.cz/models`
- 163 graphs with more than 10^6 states
  - 44 do not have an accepting cycle
  - 119 do have one
- out of the 119 “positive” graphs the accepting cycle was trivial to found
  - ndfs could report it after the visits of hundred states at most

  $\Rightarrow$ using a multi-core algorithm did not make sense
  - in 6 cases, an accepting cycle was hard to find and mc-ndfs could report it much faster

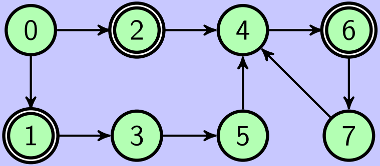
$\Rightarrow$ next we only report experiments on “negative” graphs

A Brief Overview of MAP

- MAP assumes a total order relation $>_S$ on states
- $map(s)$ is the Maximal Accepting Predecessor of s
  - the biggest accepting state (w.r.t. $>_S$) that is backward reachable from s
- $map(s)$ can be computed in $\mathcal{O}(a \cdot (n + m))$ using a modified BFS
- trivially: $map(s) = s \Rightarrow s$ is part of an accepting cycle
- but the converse is not true
- MAP will then relaunch the search after the deletion of some states
- algorithm stops when
  - an accepting cycle is found
  - or all accepting states have been deleted
- BFS is easy to distribute $\Rightarrow$ MAP is suited for parallel architectures

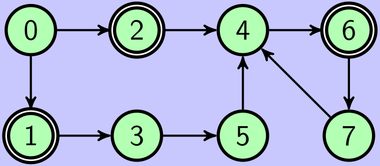
A Brief Overview of MAP — An Example

Our input graph

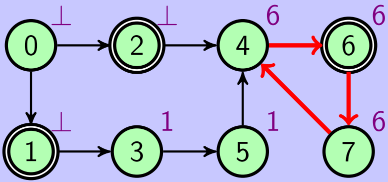


A Brief Overview of MAP — An Example

Our input graph

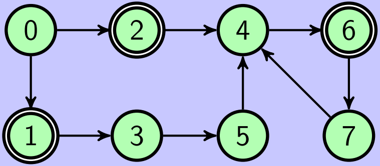


MAP with $6 >_S 2 >_S 1$

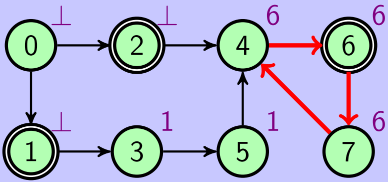


A Brief Overview of MAP — An Example

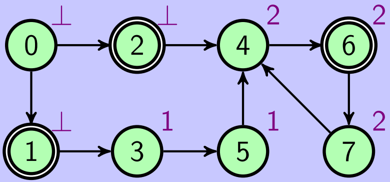
Our input graph



MAP with $6 >_S 2 >_S 1$

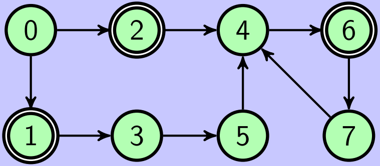


MAP with $2 >_S 1 >_S 6$

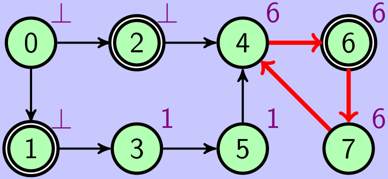


A Brief Overview of MAP — An Example

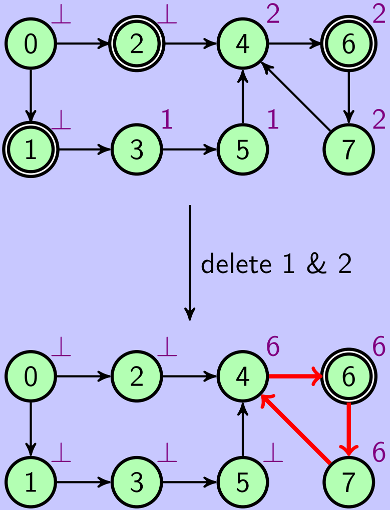
Our input graph



MAP with $6 >_S 2 >_S 1$



MAP with $2 >_S 1 >_S 6$



Evaluation Methodology

- we considered the following performance criterion

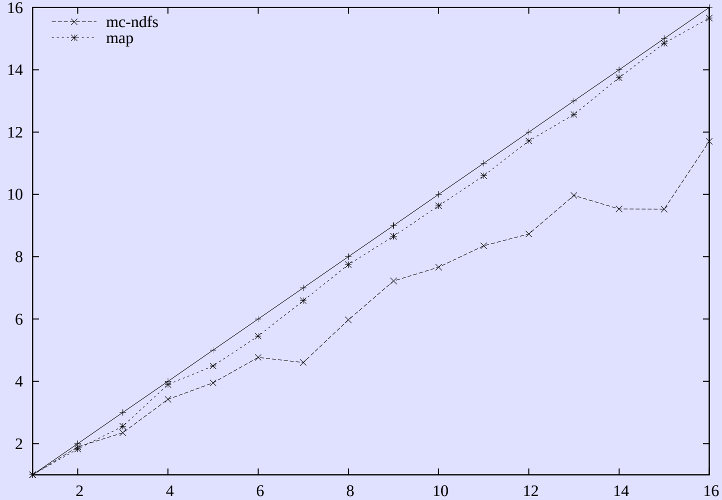
$$\max_{t \in \text{threads}} (\text{number of states explored by } t)$$

- for several reasons:
  - the graph explored was given explicitly (stored on disk)
    - all time consuming operations (computing successors, serialising states) were already done
    - synchronisations dominate the whole exploration time

  $\Rightarrow$ the time performance for both map and mc-ndfs were rather bad
  - it is implementation independent
  - it gives a better idea on the "theoretical" performance than time
- acceleration for N threads is measured as

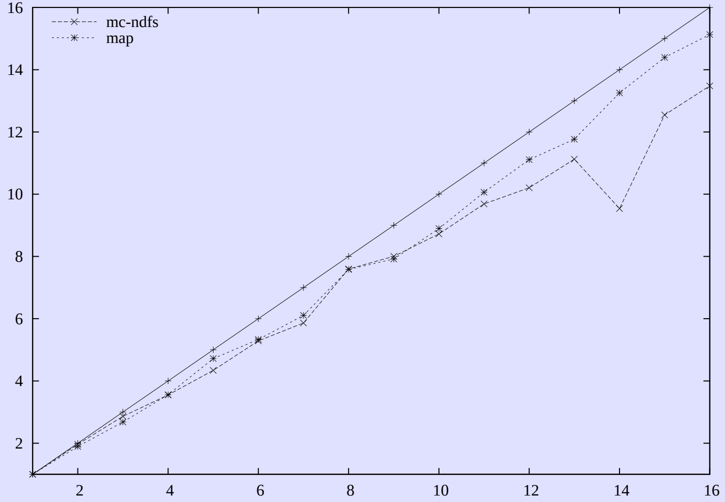
$$\frac{\text{performance for 1 thread}}{\text{performance for } N \text{ threads}}$$

Acceleration of MC-NDFS and MAP



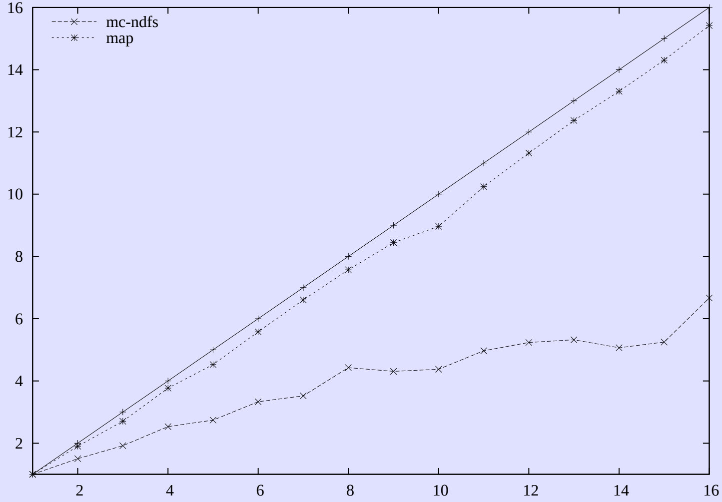
- model: **pgm_protocol** (pragmatic multicast protocol)
- property: every packet loss is followed by a negative ack
- graph size: 7,233,361 nodes

Acceleration of MC-NDFS and MAP



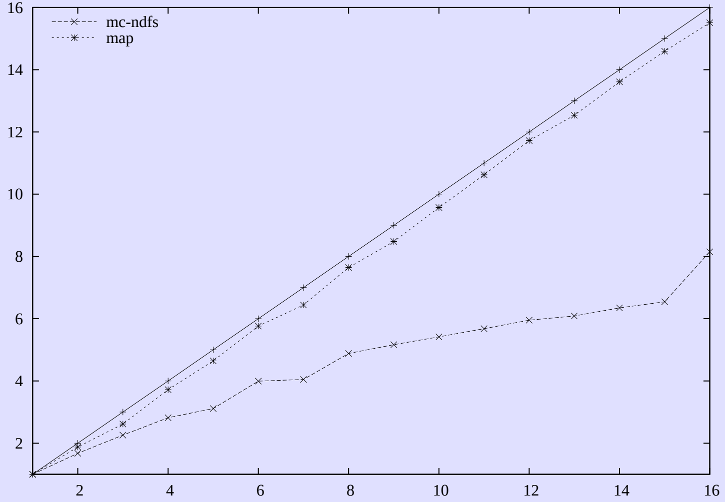
- model: **leader_filters** (election protocol)
- property: a leader is eventually elected
- graph size: 26,302,351 nodes

Acceleration of MC-NDFS and MAP



- model: **lup** (shared memory model)
- property: processor 0 will eventually have access to RAM
- graph size: 34,425,340 nodes

Acceleration of MC-NDFS and MAP



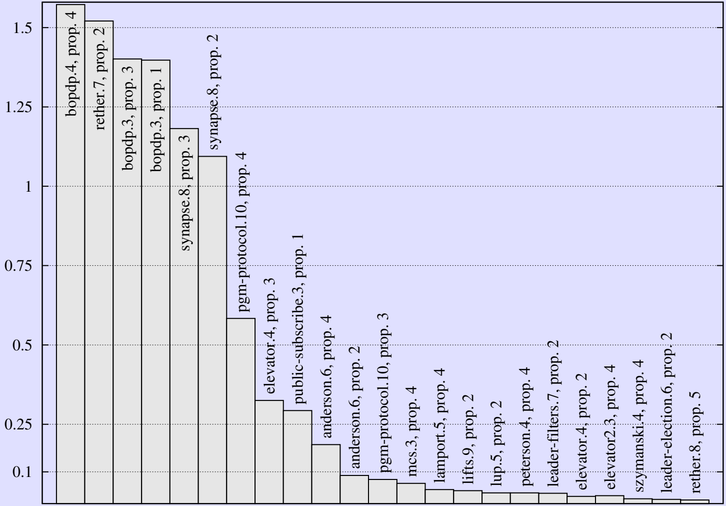
- model: **publish_subscribe** (Publish/subscribe notification protocol)
- property: ???
- graph size: 1,977,587 nodes

Acceleration of MC-NDFS and MAP

Conclusions

- mc-ndfs can clearly not compete with map on that point
- map: excellent accelerations in all situations
- mc-ndfs: the graph structure influences the acceleration
  - **pgm_protocol** and **leader_filters**: many small SCCs
    $\Rightarrow$ good acceleration
  - **lup** and **publish_subscribe**: one large SCC
    $\Rightarrow$ redundant revisits by different threads

Absolute Performances of MC-NDFS and MAP



- data plotted: $\frac{\text{performance of map}}{\text{performance of mc-ndfs}}$ for 16 working threads
- example: for graph **bopdp.4, prop. 4**, map is potentially 1.5 $\times$ faster

Absolute Performance of MC-NDFS and MAP

Conclusion

- we have seen that map clearly wins w.r.t. acceleration
- but it has a polynomial complexity in $a^2 \cdot (n + m)$
- so mc-ndfs is usually more efficient than map
- map is better than mc-ndfs when the graph has few/no accepting states

$\Rightarrow$ map is then equivalent to a parallel BFS

Overview

To sum up

- we have introduced mc-ndfs an LTL model checking algorithm for multi-core computers
- mc-ndfs is an adaptation of the sequential ndfs algorithm
- its principle
  - launch multiple threads executing a modified ndfs
  - each thread visits the graph in a random way
  - conflicts are not prevented but fixed a posteriori
  - (main) modification to ndfs: relaunch a red DFS in a safe mode when a conflict is detected
- perfomances largely depends on the graph structure
- but on some graphs we observed good accelerations

Perspectives

More experimentations

- with other kinds of models (e.g., Petri nets)
- comparison with other multi-core algorithms (e.g., bledge-otf)

Perspectives

More experimentations

- with other kinds of models (e.g., Petri nets)
- comparison with other multi-core algorithms (e.g., bledge-otf)

Better implementation

- we observed good "theoretical" accelerations with our prototype
- but not necessarily reflected in the time performance

Perspectives

More experimentations

- with other kinds of models (e.g., Petri nets)
- comparison with other multi-core algorithms (e.g., bledge-otf)

Better implementation

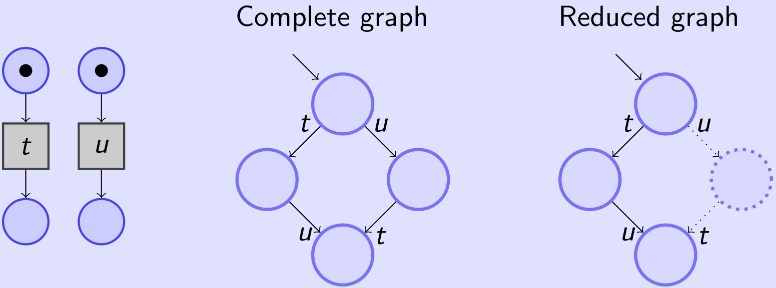
- we observed good "theoretical" accelerations with our prototype
- but not necessarily reflected in the time performance

Combination of mc-ndfs with other reduction techniques

- partial order reduction
- state caching

Partial order reduction

- the representation of interleaving is a major source of state explosion

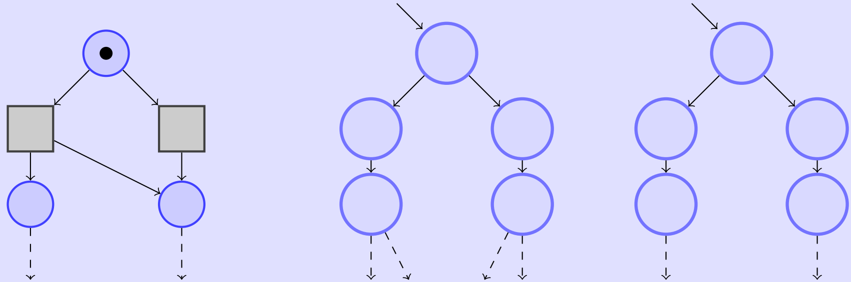


sufficient to execute $t.u$ if we are looking for deadlock states

- idea: perform a **selective** search to build a **reduced** graph
  1. perform a classical search, e.g., depth or breadth first
  2. at each state s, compute a **persistent** set of actions P
  3. only execute the actions of P and delay the execution of the others

The ignoring problem

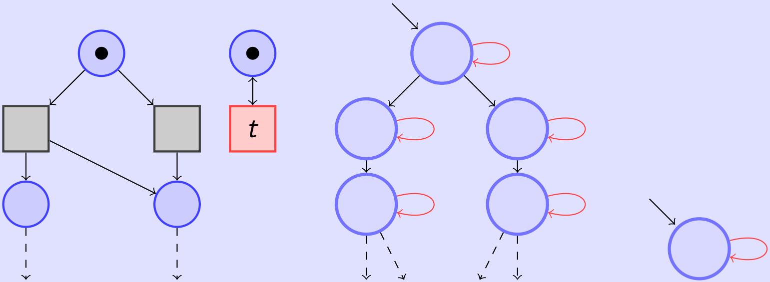
Net	Complete graph	Reduced graph



let's assume both graphs are equivalent with respect to a property Ψ

The ignoring problem

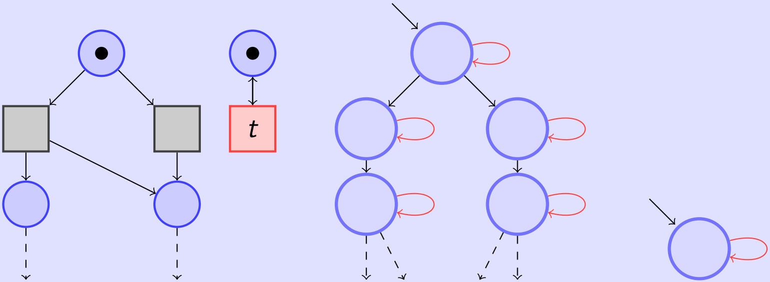
Net | Complete graph | Reduced graph



the reduced graph is useless (only useful if Ψ = deadlock freeness)
all transitions but t are infinitely delayed

The ignoring problem

Net	Complete graph	Reduced graph



the reduced graph is useless (only useful if Ψ = deadlock freeness)
all transitions but t are infinitely delayed

we have to ensure that mc-ndfs does not ignore transitions