Automata-Theoretic LTL Model Checking

Graph Algorithms for Software Model Checking

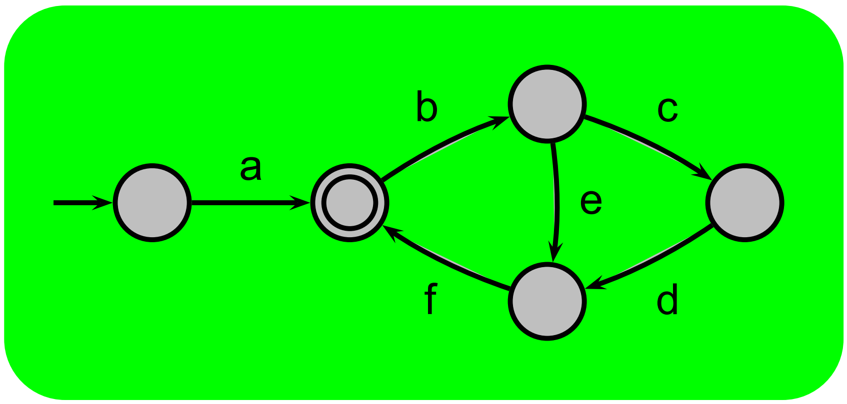
(based on Arie Gurfinkel's csc2108 project)

Emptiness of Büchi Automata

- An automation is non-empty iff
  - there exists a path to an accepting state,
  - such that there exists a cycle containing it

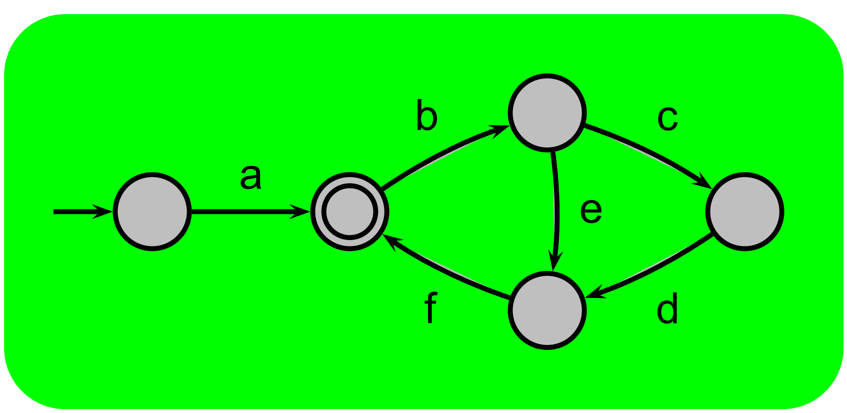
Emptiness of Büchi Automata

- An automation is non-empty iff
  - there exists a path to an accepting state,
  - such that there exists a cycle containing it
- Is this automaton empty?



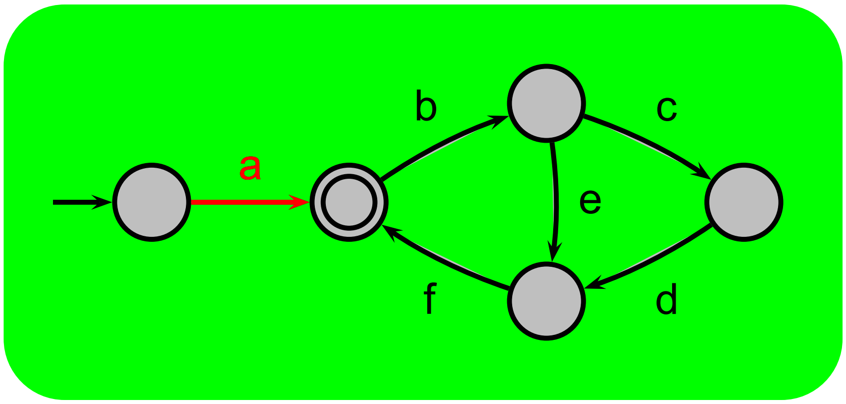
Emptiness of Büchi Automata

- An automation is non-empty iff
  - there exists a path to an accepting state,
  - such that there exists a cycle containing it
- Is this automaton empty?
  - No – it accepts $a(bef)^\omega$



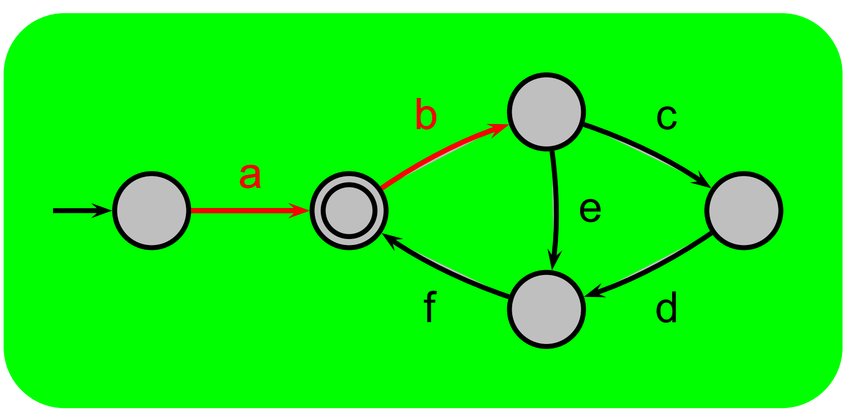
Emptiness of Büchi Automata

- An automation is non-empty iff
  - there exists a path to an accepting state,
  - such that there exists a cycle containing it
- Is this automaton empty?
  - No – it accepts $a(bef)^\omega$



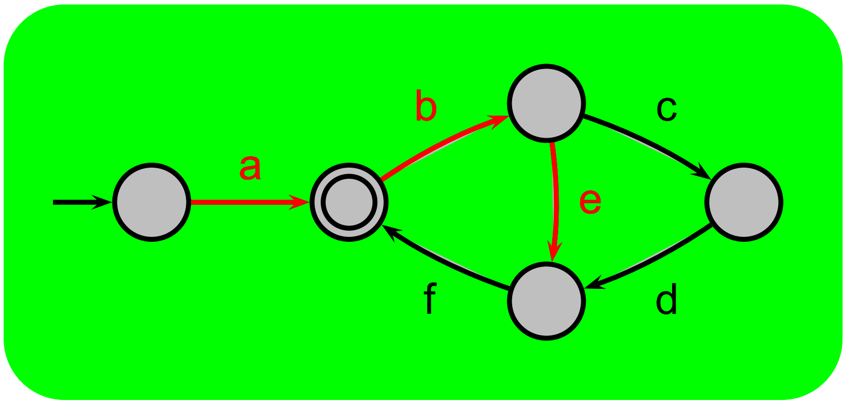
Emptiness of Büchi Automata

- An automation is non-empty iff
  - there exists a path to an accepting state,
  - such that there exists a cycle containing it
- Is this automaton empty?
  - No – it accepts $a(bef)^\omega$



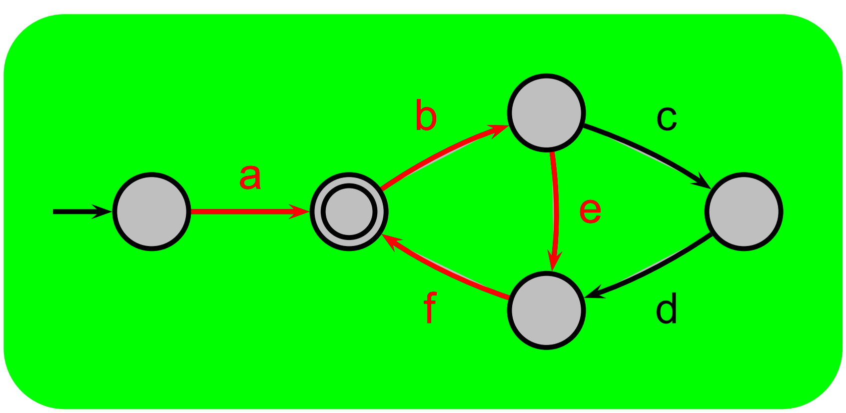
Emptiness of Büchi Automata

- An automation is non-empty iff
  - there exists a path to an accepting state,
  - such that there exists a cycle containing it
- Is this automaton empty?
  - No – it accepts $a(bef)^\omega$



Emptiness of Büchi Automata

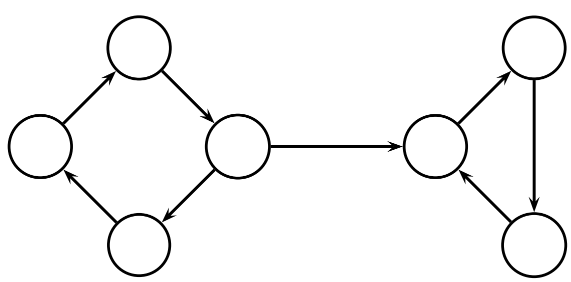
- An automation is non-empty iff
  - there exists a path to an accepting state,
  - such that there exists a cycle containing it
- Is this automaton empty?
  - No – it accepts $a(bef)^\omega$



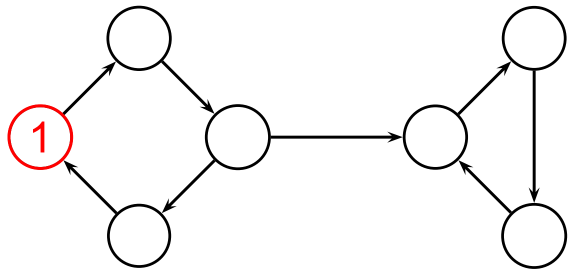
LTL Model-Checking

- LTL Model-Checking = Emptiness of Büchi automata
  - a tiny bit of automata theory +
  - trivial graph-theoretic problem
    - typical solution – use depth-first search (DFS)
- Problem: state-explosion
  - the graph is *HUGE*
- The result
  - LTL model-checking is just a very elaborate DFS

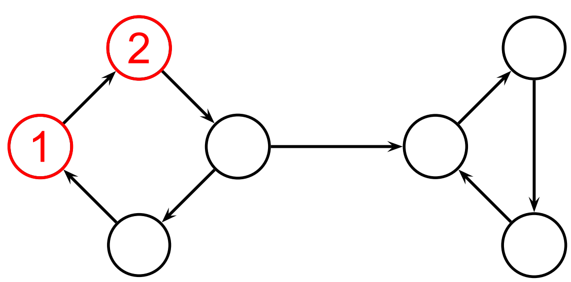
Depth-First Search – Refresher



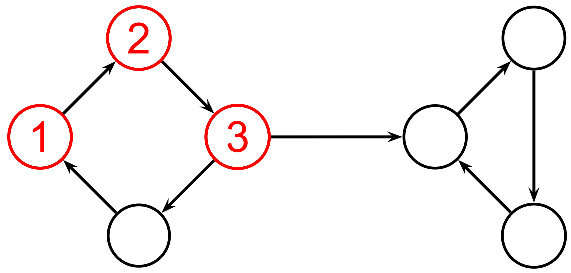
Depth-First Search – Refresher



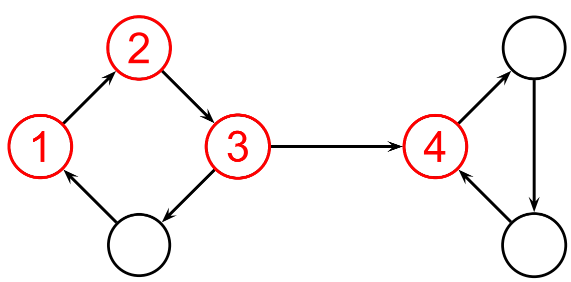
Depth-First Search – Refresher



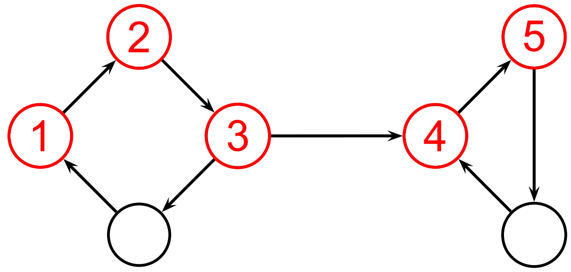
Depth-First Search – Refresher



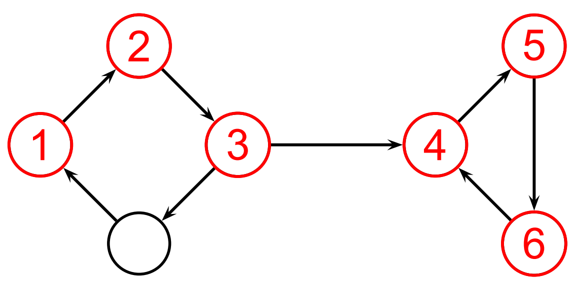
Depth-First Search – Refresher



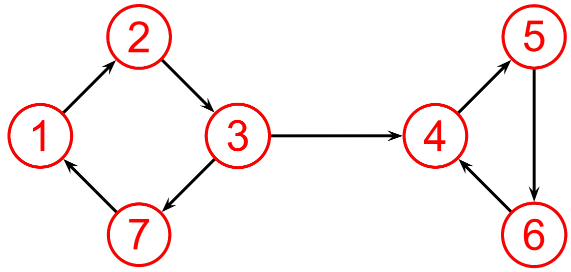
Depth-First Search – Refresher



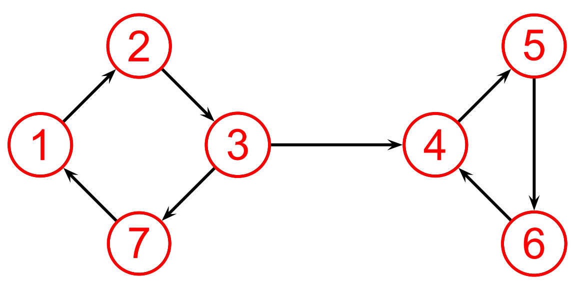
Depth-First Search – Refresher



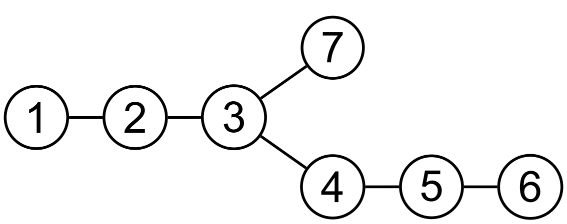
Depth-First Search – Refresher



Depth-First Search – Refresher



- depth-first tree



DFS – The Algorithm

```
proc DFS(v)
    add v to Visited
    d[v] := time
    time := time + 1
    for all w ∈ succ(v) do
        if w ∉ Visited then
            DFS(w)
        end if
    end for
    f[v] := time
    time := time + 1
end proc
```


DFS – Data Structures

- implicit STACK
  - stores the current path through the graph
- *Visited* table
  - stores visited nodes
  - used to avoid cycles
- for each node
  - *discovery time* – array d
  - *finishing time* – array f

What we want

- Running time
  - at most linear — anything else is not feasible
- Memory requirements
  - sequentially accessed – like STACK
    - disk storage is good enough
    - assume unlimited supply – so can ignore
  - randomly accessed – like hash tables
    - must use RAM
    - limited resource – minimize
    - why cannot use virtual memory?

What else we want

- Counterexamples
  - an automaton is non-empty iff exists an accepting run
  - this is the counterexample – we want it
- Approximate solutions
  - partial result is better than nothing!

DFS – Complexity

- Running time
  - each node is visited once
  - linear in the size of the graph
- Memory
  - the STACK
    - accessed sequentially
    - can store on disk – ignore
  - *Visited* table
    - randomly accessed – important
    - $|\textit{Visited}| = S \times n$
    - n – number of nodes in the graph
    - S – number of bits needed to represent each node

Take 1 – Tarjan's SCC algorithm

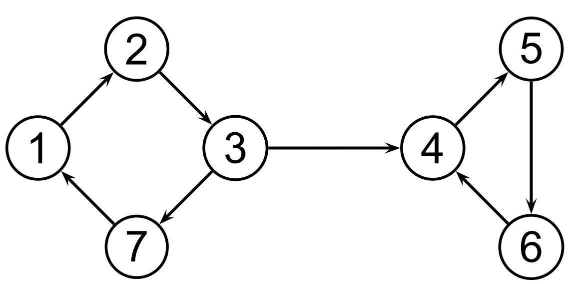
- Idea: find all maximal SCCs: SCC$_1$, SCC$_2$, etc.
  - an automaton is non-empty iff exists SCC$_i$ containing an accepting state

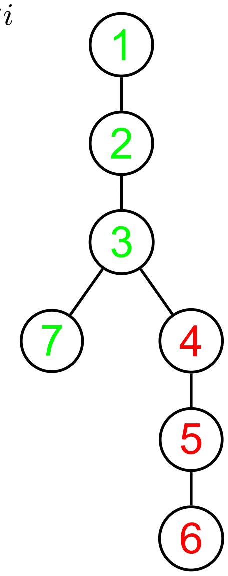
Take 1 – Tarjan's SCC algorithm

- Idea: find all maximal SCCs: SCC_1, SCC_2, etc.
  - an automaton is non-empty iff exists SCC_i containing an accepting state
- Fact: each SCC is a sub-tree of DFS-tree
  - need to find roots of these sub-trees

Take 1 – Tarjan's SCC algorithm

- Idea: find all maximal SCCs: SCC_1, SCC_2, etc.
  - an automaton is non-empty iff exists SCC_i containing an accepting state
- Fact: each SCC is a sub-tree of DFS-tree
  - need to find roots of these sub-trees





Finding a Root of an SCC

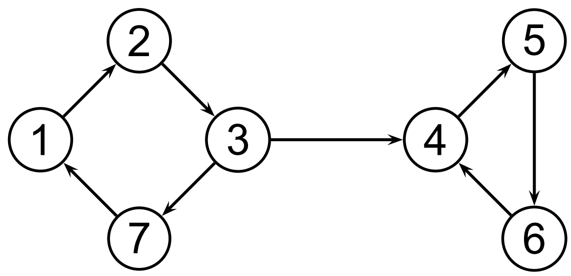
- For each node v, compute $lowlink[v]$
- $lowlink[v]$ is the minimum of
  - discovery time of v
  - discovery time of w, where
    - w belongs to the same SCC as v
    - the length of a path from v to w is at least 1
- Fact: v is a root of an SCC iff
  - $d[v] = lowlink[v]$

Finally: the algorithm

```
proc SCC_SEARCH(v)
  add v to Visited
  d[v] := time
  time := time + 1
  lowlink[v] := d[v]
  push v on STACK
  for all w ∈ succ(v) do
    if w ∉ Visited then
      SCC_SEARCH(w)
      lowlink[v] := min(lowlink[v], lowlink[w])
    else if d[w] < d[v] and w is on STACK then
      lowlink[v] := min(d[w], lowlink[v])
    end if
  end for
  if lowlink[v] = d[v] then
    repeat
      pop x from top of STACK
      if x ∈ F then
        terminate with "Yes"
      end if
    until x = v
  end if
end proc
```

Finally: the algorithm

```
proc SCC_SEARCH(v)
  add v to Visited
  d[v] := time
  time := time + 1
  lowlink[v] := d[v]
  push v on STACK
  for all w ∈ succ(v) do
    if w ∉ Visited then
      SCC_SEARCH(w)
      lowlink[v] := min(lowlink[v], lowlink[w])
    else if d[w] < d[v] and w is on STACK then
      lowlink[v] := min(d[w], lowlink[v])
    end if
  end for
  if lowlink[v] = d[v] then
    repeat
      pop x from top of STACK
      if x ∈ F then
        terminate with "Yes"
      end if
    until x = v
  end if
end proc
```



Tarjan's SCC algorithm – Analysis

- Running time
  - linear in the size of the graph
- Memory
  - STACK – sequential, ignore
  - $Visited$ – $O(S \times n)$
  - $lowlink$ – $\log n \times n$ (wasted space?)
  - n is not known a priori
    - assume n is at least $\geq 2^{32}$
- Counterexamples
  - can be extracted from the STACK
  - even more – get multiple counterexamples
- If we sacrifice some of generality, can we do better?

Take 2 – Two Sweeps

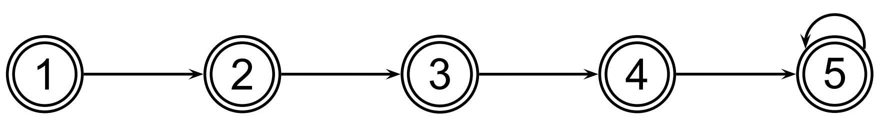
- Don't look for maximal SCCs
- Find a reachable accepting state that is on a cycle
- Idea: use two sweeps
  - sweep one: find all accepting states
  - sweep two: look for cycles *from* accepting states

Take 2 – Two Sweeps

- Don't look for maximal SCCs
- Find a reachable accepting state that is on a cycle
- Idea: use two sweeps
  - sweep one: find all accepting states
  - sweep two: look for cycles *from* accepting states
- Problem?
  - no longer a linear algorithm (revisit the states multiple times)

Take 2 – Two Sweeps

- Don't look for maximal SCCs
- Find a reachable accepting state that is on a cycle
- Idea: use two sweeps
  - sweep one: find all accepting states
  - sweep two: look for cycles *from* accepting states
- Problem?
  - no longer a linear algorithm (revisit the states multiple times)

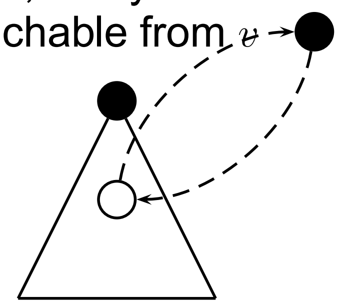


Fixing non-linearity

- Graph Theoretic Result: let v and u be two nodes, such that
  - $f[v] < f[u]$
  - v is not on a cycle
  - then, no cycle containing u contains nodes reachable from v

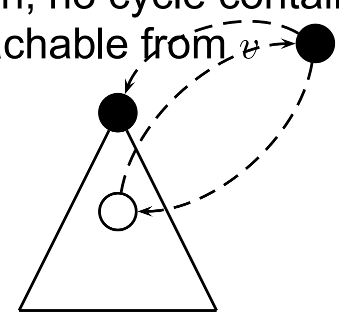
Fixing non-linearity

- Graph Theoretic Result: let v and u be two nodes, such that
  - $f[v] < f[u]$
  - v is not on a cycle
  - then, no cycle containing u contains nodes reachable from v



Fixing non-linearity

- Graph Theoretic Result: let v and u be two nodes, such that
  - $f[v] < f[u]$
  - v is not on a cycle
  - then, no cycle containing u contains nodes reachable from v



Take 3 – Double DFS

```
proc DFS1(v)
  add v to Visited
  for all w ∈ succ(v) do
    if w ∉ Visited then
      DFS1(w)
    end if
  end for
  if v ∈ F then
    add v to Q
  end if
end proc
```

```
proc DFS2(v, f)
  add v to Visited
  for all w ∈ succ(v) do
    if v = f then
      terminate with "Yes"
    else if w ∉ Visited then
      DFS2(w, f)
    end if
  end for
end proc
```

```
proc SWEEP2(Q)
  while Q ≠ [] do
    f := dequeue(Q)
    DFS2(f, f)
  end while
  terminate with "No"
end proc
```

```
proc DDFS(v)
  Q = ∅
  Visited = ∅
  DFS1(v)
  Visited = ∅
  SWEEP2(Q)
end proc
```


Double DFS – Analysis

- Running time
  - linear! (single $Visited$ table for different final states, so no state is processed twice)
- Memory requirements
  - $O(n \times S)$
- Problem
  - where is the counterexample?!

Take 4 – Nested DFS

- Idea
  - when an accepting state is finished
    - stop first sweep
  - start second sweep
    - if cycle is found, we are done
  - otherwise, restart the first sweep
- As good as double DFS, but
  - does not need to *always* explore the full graph
  - counterexample is readily available
    - a path to an accepting state is on the stack of the first sweep
    - a cycle is on the stack of the second

A Few More Tweaks

- No need for two *Visited* hashtables
  - empty hashtable wastes space
  - merge into one by adding one more bit to each node
    - $(v, 0) \in Visited$ iff v was seen by the first sweep
    - $(v, 1) \in Visited$ iff v was seen by the second sweep
- Early termination condition
  - nested DFS can be terminated as soon as it finds a node that is on the stack of the first DFS

On-the-fly Model-Checking

- Typical problem consists of
  - description of several process $P_1, P_2, \ldots$
  - property φ in LTL
- Before applying DFS algorithm
  - construct graph for $P = \Pi_{i=1}^{n} P_i$
  - construct Büchi automaton $A_{\neg\varphi}$ for $\neg\varphi$
  - construct Büchi automaton for $P \cap A_{\neg\varphi}$

On-the-fly Model-Checking

- Typical problem consists of
  - description of several process $P_1, P_2, \ldots$
  - property φ in LTL
- Before applying DFS algorithm
  - construct graph for $P = \Pi_{i=1}^{n} P_i$
  - construct Büchi automaton $A_{\neg\varphi}$ for $\neg\varphi$
  - construct Büchi automaton for $P \cap A_{\neg\varphi}$
- But,
  - all constructions can be done in DFS order
  - combine everything with the search
  - result: on-the-fly algorithm, only the necessary part of the graph is built

State Explosion Problem

- the size of the graph to explore is huge
- on real programs
  - DFS dies after examining just 1% of the state space
- What can be done?
  - abstraction
    - false negatives
  - partial order reduction. (to be covered)
    - exact – but not applicable to full LTL
  - partial exploration – explore as much as possible
    - false positives
- In practice – combine all 3

Partial exploration techniques

- Explore as much of the graph as possible
- The requirements
  - must be compatible with
    - on-the-fly model-checking
    - nested depth-first search
  - size of the graph not known a priori
    - must perform as good as full exploration when enough memory is available
  - must degrade gracefully
- We will look at two techniques
  - bitstate hashing
  - hashcompact – a type of state compression

Bitstate Hashing

- a hashtable is
  - an array d of k entries
  - a hash function $hash$: States $\rightarrow 0..k-1$
  - a collision resolution protocol
- to insert v into a hashtable
  - compute $hash(v)$
  - if $d[hash(v)]$ is empty, $d[hash(v)] = v$
  - otherwise, apply collision resolution
- to lookup v
  - if $d[hash(v)]$ is empty, v is not in the table
  - else if $d[hash(v)] = v$, v is in the table
  - otherwise, apply collision resolution

Bitstate Hashing

- if there are no collisions, don't need to store v at all!
  - instead, just store one bit – empty or not
- even better, use two hash functions
  - to insert v, set $d[hash_1(v)] = 1$ and $d[hash_2(v)] = 1$
- sound with respect to false answers
  - if a counterexample is found, it is found!
- in practice, up to 99% coverage
- collisions increase gradually when not enough memory
- coverage decreases at the rate collisions increase

Why does this work?

- If nested DFS stops when a successor to v in $DFS2$ is on the stack of $DFS1$, how is soundness guaranteed, i.e., why is the counterexample returned by model-checker real?
- Answer: States are stored on the stack without hashing, since stack space does not need to be saved.

Hashcompact

- Assume a large virtual hashtable, say 2^{64} entries
- For each node v,
  - instead of using v,
  - use $hash(v)$, its hash value in the large table
- Store $hash(v)$ in a normal hashtable,
  - or even the one with bitstate hashing
- When there is enough memory
  - probability of missing a node is $< 10^{-3}$
- Degradation
  - expected coverage decreases rapidly, when not enough memory

Symbolic LTL Model-Checking

- LTL Model-Checking = Finding a reachable cycle
- Represent the graph symbolically
  - and use symbolic techniques to search
- There exists an infinite path from s, iff $||EG \text{ true}||(s)$
  - the graph is finite
    - infinite $\Rightarrow$ cyclic!
  - exists a cycle containing an accepting state a iff a occurs infinitely often
    - use fairness to capture accepting states
- LTL Model-Checking = EG true under fairness!

food for slide eater