The Algorithmic Verification of Reactive Systems

Lecture 5:

State-Space Exploration and Basic Verification Algorithms

The Verification Approach

- The central part of the verification process is to compute the transition system corresponding to the Formal Concurrent System to be analyzed.

- In practice, one needs to compute the *reachable* states of this transition system.

- One can then check properties by examing this transition systems, e.g. the reachability of a given state, the absence of deadlocks, other properties of all reachable states . . .

- The main problem is that the transition system can be extremely large. One thus needs techniques to optimize the computation of its reachable states.

Computing the Reachable States of a System: Depth-First Search

The following algorithm computes the transition system $ST = \{\Sigma, S, s_0, T\}$ corresponding to a formal concurrent system $FCS = \{\mathcal{P}, \mathcal{M}, \mathcal{T}\}$.

This algorithm uses a stack ($Stack$) and a data structure H used to memorize the set of visited states. This structure is usually a hash table.

The function enabled(s) returns the transitions of $\mathcal{T}$ that are enabled in a state $s \in S$. The function succ(s, t) where $s \in S$ and $t \in \mathcal{T}$ returns the state s' such that $(s, n(t), s') \in T$

```
procedure DFS()
      begin
            s := top(Stack);
            Tr := enabled(s)
            for all t ∈ Tr
                  begin
                  s' := succ(s, t)
                  if s' ∉ H then
                     begin
                     insert s' in H;
                     push s' onto Stack;
                     DFS();
                     end
                  end
            pop(Stack);
      end
Stack := [s_0]; H := {s_0};
DFS();
```

Using Depth-First Search

- The depth-first algorithm makes it possible to check properties of the reachable states: invariants, reachability of a state with a given property, absence of deadlock, ...
- If an erroneous state is reached, the stack contains an execution trace that makes this state reachable. This is very helpful for understanding why such a state can be reached.
- By adding an *observer process*, more elaborate properties can be checked and in particular properties expressed in linear-time temporal logic.
- When performing the depth-first search, the transition relation need not be stored. One refers to this by saying the verification is done **on-the-fly** (while exploring the transition system)

Checking Linear-Time Temporal Logic Properties

- To check that a transition system satisfies a linear-time temporal logic property, one needs to check that *all* its infinite behaviors satisfy this property.
- One checks that all these behaviors are *models* of the property. This procedure is thus called *model-checking*.
- To obtain a model checking procedure, the idea is to use the automaton on infinite words for the property as an observer for the system.
- However, this automaton is nondeterministic and it accepts a behavior is *there exists* some computation for which it accepts.
- Checking that *for all* behaviors of the transition system, *there exists* some accepting computation of the observer is problematic.
- The trick is to first complement the property to be checked.

A Model-Checking Algorithm for LTL

- One is given a formal concurrent system FCS and a LTL property f.
- The first step is to build an automaton on infinite words $A_{\neg f}$ that accepts exactly all infinite sequences that *do not* satisfy f.
- $A_{\neg f}$ is then added as an observer to FCS. If needed, observers for the necessary fairness conditions are added.
- The corresponding transition system is then computed by a depth-first search. It is a transition system with a generalized Büchi acceptance condition.

- One then checks whether this transition system is empty of not (accepts at least one infinite word). Any accepted infinite word is a behavior of the system that does not satisfy the property to be checked.

- Remaining problem: how does one check emptiness of a Generalized Büchi automaton by a depth-first search, i.e. without applying the more complicated strongly connected components algorithm?

Complexity of LTL Model Checking:
Linear in the size of the transition system (NLOGSPACE),
Exponential in the size of the FCS (PSPACE),
Exponential in the size of the LTL property (PSPACE).

A Simple Case for Model-Checking : Safety Properties

- Imagine a property f is such that the corresponding automaton has an empty acceptance condition. This should make model checking simpler.

- No acceptance condition, means that the only way a word cannot be accepted is by reaching a situation in which no transition is possible.

- Furthermore, since there is no acceptance condition, the Rabin-Scott subset construction can be used to determinize and complement such an automaton on infinite words.

- The complement automaton would then have a single accepting state from which all behaviors are accepted. Once this state is reached, the word is accepted whatever happens next. The automaton for $\neg f$ is thus really an automaton on *finite-words*.

- This has two consequences on the algorithmic aspects of checking safety properties:

  – Fairness conditions are not necessary.

  – The emptiness test is a simple reachability test.

Safety Properties:
An Abstract Definition

- We work in the context in which a property is a set of infinite words over an alphabet Σ.
- A property P is thus a set $P \subseteq \Sigma^\omega$.
- Given two words $w_1, w_2 \in \Sigma^\omega$, let $\mathsf{lcp}(w_1, w_2)$ denote the length of their longest common prefix. Define then the *distance* between two words $w_1, w_2 \in \Sigma^\omega$ as

  $$d(w_1, w_2) = \frac{1}{\mathsf{lcp}(w_1, w_2) + 1}$$

  This distance induces a topology on Σ^ω.
- An *open set* O is one in which each element is surrounded by a neighborhood that is in the set:

  if $w \in O$, then $\exists \varepsilon \forall w' [d(w, w') \leq \varepsilon \supset w' \in O]$
- A *closed set* is the complement of an open set. Note that a closed set contains the limit of its converging sequences of words.
- A **safety property** is a **closed** subset of Σ^ω.

Another Particular Class of Properties: Liveness Properties

- An automaton without an accepting condition defines a safety property. What class of properties is defined by automata that "only have an accepting condition"?

- Formally, a language L is in this class if it satisfies

$$(\forall w \in \Sigma^*)(\exists w' \in \Sigma^\omega) ww' \in L$$

- Topologically, a property $L \subseteq \Sigma^\omega$ is a liveness property if it is a **dense** set:

$$(\forall w \in \Sigma^\omega)\forall \varepsilon(\exists w' \in L) d(w, w') \leq \varepsilon$$

- It is possible to prove that every property is the intersection of a liveness and of a safety property.

Checking the Nonemptiness of (Generalized) Büchi Automata With a Depth-First Search

- A Büchi automaton accepts if some state in F is reachable from the initial state and from itself by a path of length ≥ 1 (For generalized Büchi automata, convert to Büchi).
- To check that it is nonempty, proceed as follows.

1. Do a depth-first search of the reachable states.
2. When doing this search, build a postorder list of reachable accepting states. Let this list be
$$Q = f_1, \ldots, f_k$$
where f_1 is the first postorder reachable accepting state and f_k is the last.
3. Do a second depth-first search from the elements in Q.

The Second Search

1. Start the search from f_1.

2. Once the search from f_i is finished (f_i reached or no more reachable nodes) :

   (a) Restart the search from f_{i+1}, but

   (b) do not reconsider nodes that have been visited during searches from f_j, $j \leq i$.

3. Each node in the graph is only visited once.

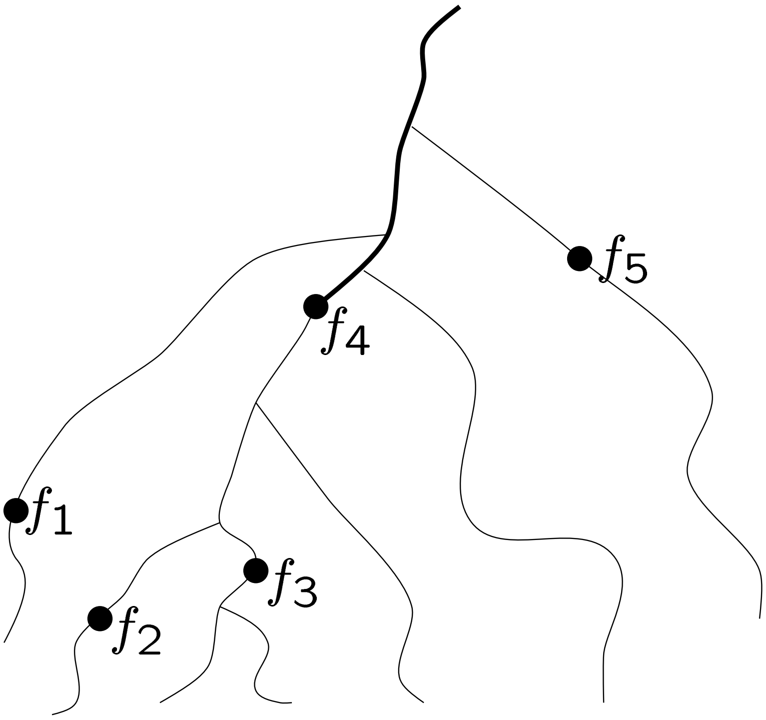
Note that the first and second search can be interleaved.

Why Does it Work?

1. If f_i is reachable from f_j, where f_j appears before f_i in postorder, then f_j is reachable from itself.

   - In postorder, the ancestors of a node separate the nodes preceding that node from the nodes following it.
   - Thus a path from a node A to a node following A in postorder must go through an ancestor of A and A is reachable form itself.

2. So, for the first f_i reachable from itself, the path p from f_i to itself cannot contain any node reachable from a previous f_j (if it did, there would be a path from f_j to f_i and f_j would be reachable from itself)


f_1
f_2
f_3
f_4
f_5

Minimizing Memory Requirements: Hashing without Collision Detection (G. Holzmann)

Store visited states in a table of bits H, using a hash function h

- Initially $H[i] = 0, \ \ \forall i$
- To store a state s set $H[h(s)] = 1$
- A state is assumed to be present if $H[h(s)] = 1$

Advantages :	one bit per state,
	can be used with previous Büchi
	nonemptiness algorithm,
Problem :	states might be missed.

Analysis
Probability of No Collision

- t : size of table
- n : number of states to be stored

$$p_{\text{nc}} = \frac{t!}{t^n (t-n)!}$$

$$p_{\text{nc}} \approx e^{-\frac{n^2}{t}}$$

Thus if we wants $1 - p_{\text{nc}} = 10^{-3}$ for $n = 10^6$, t must be 10^{15}

Unrealistic !

Improvement
(Multihash)

Store several bits (k) for each state (Holzmann uses 2)

- One hash function h_i per bit stored
- To store a state s set $H[h_i(s)] = 1$ for all $1 \leq i \leq k$
- A state is assumed to be present if $H[h_i(s)] = 1$ for all $1 \leq i \leq k$

Analysis
Probability of No Collision

- t^k ways to insert first element

- $t^k - k^k$ ways to insert second element without collision (approximately)

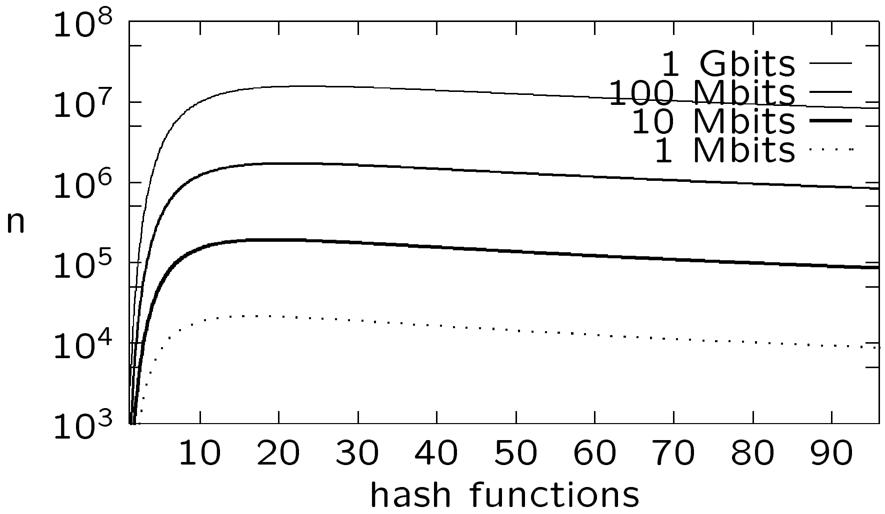
- $t^k - (ik)^k$ ways to insert element $i + 1$ without collision (approximately)

- $p_{\text{nc}} = \frac{\prod_{i=0}^{n-1}(t^k - i^k k^k)}{t^{kn}}$

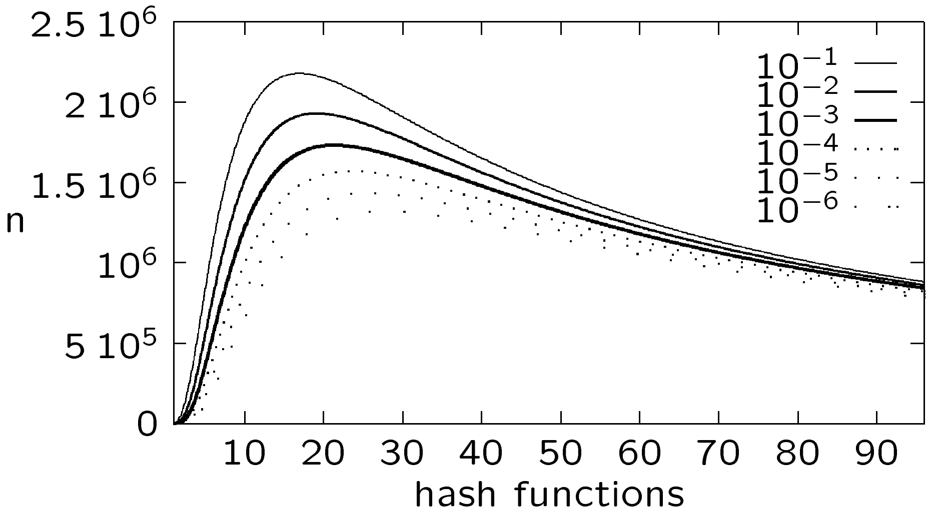
- for $nk \ll t$, $\quad p_{\text{nc}} \approx e^{-\frac{k^k n^{k+1}}{t^k}}$

For example, with $k = 2$ and $n = 10^6$, t still must be $6 \cdot 10^{10}$ for $1 - p_{\text{nc}}$ to be 10^{-3}

Optimal Value of k



States stored with $p_{\mathrm{nc}} = 1 - 10^{-3}$



States stored with M = 100 Mbits

Discussion

- Optimum : 20 hash functions
- Expensive to compute
- No advantages in splitting the table

Alternative Scheme (Hashcompact)

- Compute an address that is long enough (k bits) for the probability of collision to be low (e.g. 64 bits)
- Store this address in a table with collision detection

Analysis

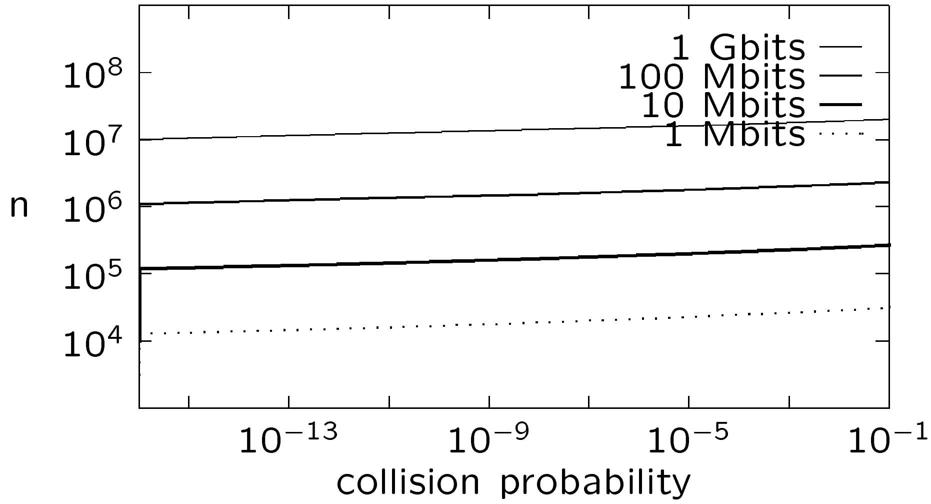
- t : size of table
- k : number of bits stored
- constraint : $tk = M$ (available memory)

$$p_{\text{nc}} \approx e^{-\frac{n^2}{2^k}}$$

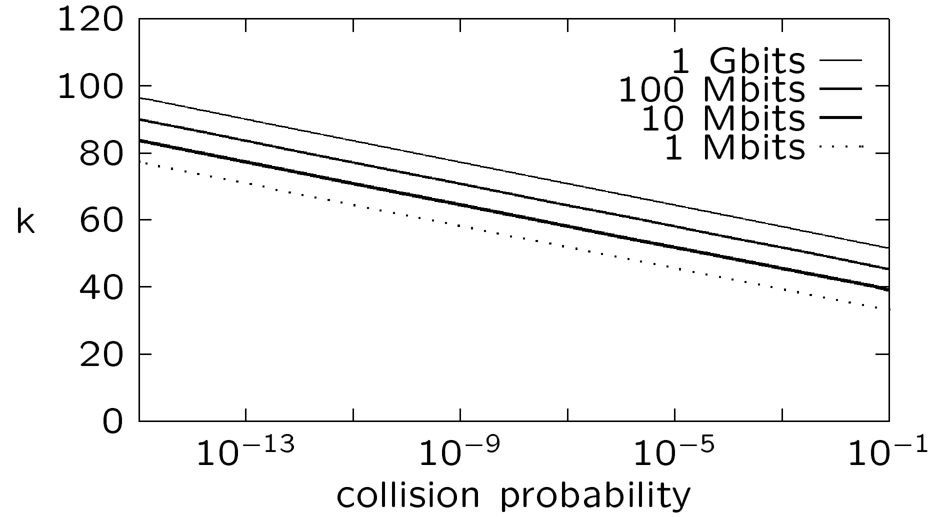
Maximum number of states when table is full, i.e. $n = t$ and thus $k = \frac{M}{n}$

$$p_{\text{nc}} \approx e^{-\frac{n^2}{2^{M/n}}}.$$

Results



Optimal n

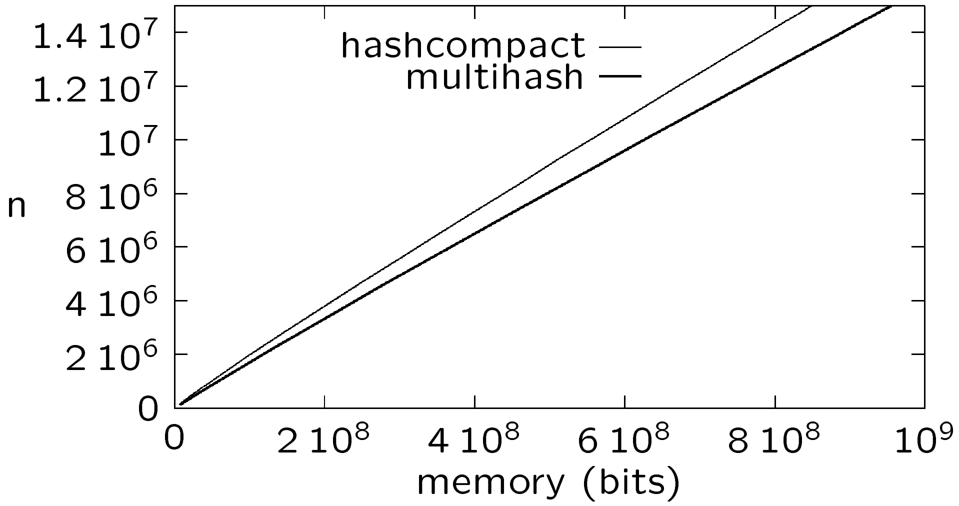


Optimal k

- $k = 64$ generally sufficient

Comparing the Methods

$p_{\text{nc}} = 10^{-3}$, schemes used optimally



Note : The methods degrade differently

- hashcompact stalls
- multihash starts missing states

Experimental results

Protocol	Algorithm	Stored States	Memory use	States missed ?
DTP	exhaustive	427,567	73 Mb	No
	bitstate	427,446	33.8 Mb	Yes
	hashcompact	427,567	3.6 Mb	No
PFTP	exhaustive	409,257	34 Mb	No
	bitstate	405,969	33.9 Mb	Yes
	hashcompact	409,257	3.6 Mb	No