



A new dichotomic algorithm for the uniform random generation of words in regular languages

Johan Oudinet* Alain Denise Marie-Claude Gaudel
{oudinet,denise,mcg}@lri.fr

Laboratoire de Recherche en Informatique (LRI)
<http://www.lri.fr>
Université de Paris-Sud & CNRS

September 2, 2010



UNIVERSITÉ
PARIS-SUD 11





Outline

- 1 Motivation and Prior Algorithms
- 2 Our Algorithm: Dichopile
- 3 Related work: Boltzmann, Bernardi and Giménez
- 4 Experimental results
- 5 Conclusion

- 1 Motivation and Prior Algorithms
- 2 Our Algorithm: Dichopile
- 3 Related work: Boltzmann, Bernardi and Giménez
- 4 Experimental results
- 5 Conclusion



Motivation

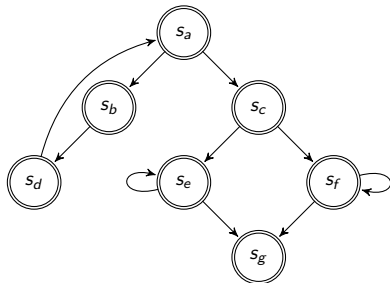
- Goal: Exploration of very large models (10^{20} states or more) for model-based testing and model-checking purposes.
- Subgoal: Exploration of their components (smaller but still large: 10^6 states).
 - too many paths \rightarrow random exploration;
 - Need to assess a good coverage \rightarrow uniform distribution on paths.

Problem

Uniform generation of words in regular languages, which should be **efficient** both in regards to the **length** of words and the **size** of automata.



Hickey and Cohen's algorithm (1983)

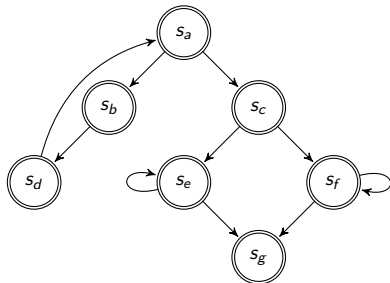


$$\begin{cases} l_s(0) = 1 \\ l_s(i) = \sum_{s \rightarrow s'} l_{s'}(i-1) \quad \forall i > 0 \end{cases}$$

$$Pr(s' | s, i) = \frac{l_{s'}(i-1)}{l_s(i)}$$



Hickey and Cohen's algorithm (1983)



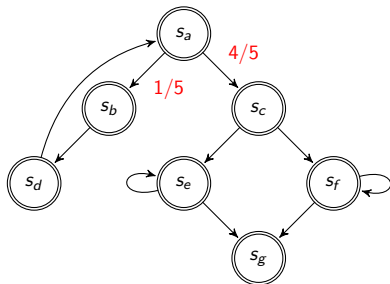
$$\begin{cases} l_s(0) = 1 \\ l_s(i) = \sum_{s \rightarrow s'} l_{s'}(i-1) \quad \forall i > 0 \end{cases}$$

$$Pr(s' | s, i) = \frac{l_{s'}(i-1)}{l_s(i)}$$

#	s_a	s_b	s_c	s_d	s_e	s_f	s_g
0	1	1	1	1	1	1	1
1	2	1	2	1	2	2	0
2	3	1	4	2	2	2	0
3	5	2	4	3	2	2	0



Hickey and Cohen's algorithm (1983)



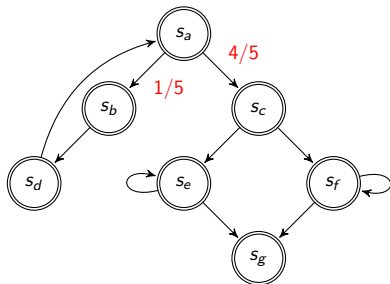
$$\begin{cases} l_s(0) = 1 \\ l_s(i) = \sum_{s \rightarrow s'} l_{s'}(i-1) \quad \forall i > 0 \end{cases}$$

$$Pr(s' | s, i) = \frac{l_{s'}(i-1)}{l_s(i)}$$

#	s_a	s_b	s_c	s_d	s_e	s_f	s_g
0	1	1	1	1	1	1	1
1	2	1	2	1	2	2	0
2	3	1	4	2	2	2	0
3	5	2	4	3	2	2	0



Hickey and Cohen's algorithm (1983)



$$\begin{cases} l_s(0) = 1 \\ l_s(i) = \sum_{s \rightarrow s'} l_{s'}(i-1) \quad \forall i > 0 \end{cases}$$

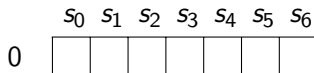
$$Pr(s' | s, i) = \frac{l_{s'}(i-1)}{l_s(i)}$$

#	s_a	s_b	s_c	s_d	s_e	s_f	s_g
0	1	1	1	1	1	1	1
1	2	1	2	1	2	2	0
2	3	1	4	2	2	2	0
3	5	2	4	3	2	2	0

- Building the counting table requires $\Theta(n \times q)$ operations.
- Then, drawing a path needs $\Theta(n)$ arithmetic operations.

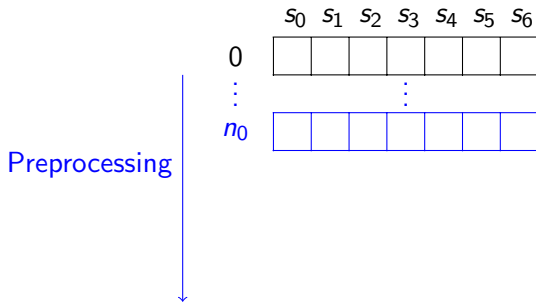


Goldwurm's idea (1995)



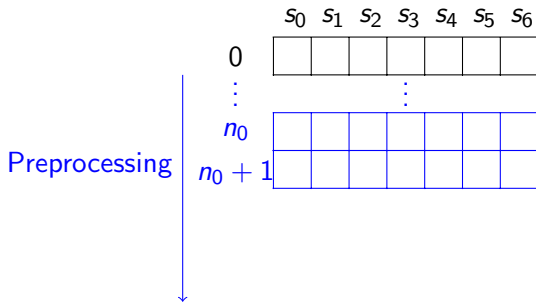


Goldwurm's idea (1995)



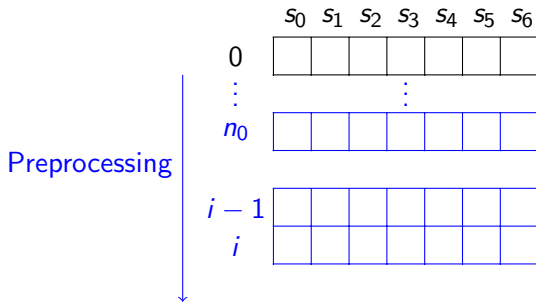


Goldwurm's idea (1995)



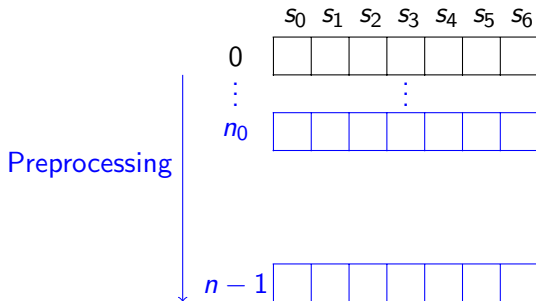


Goldwurm's idea (1995)



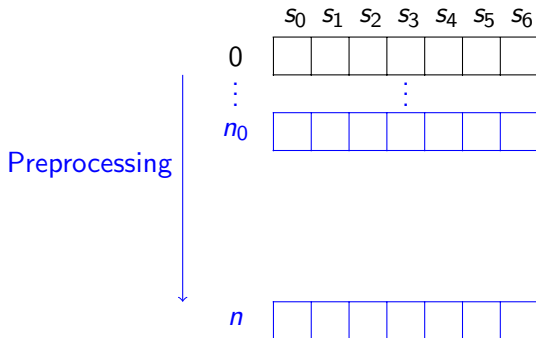


Goldwurm's idea (1995)



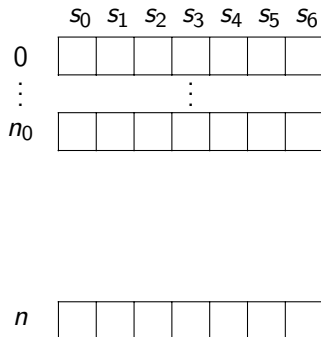


Goldwurm's idea (1995)



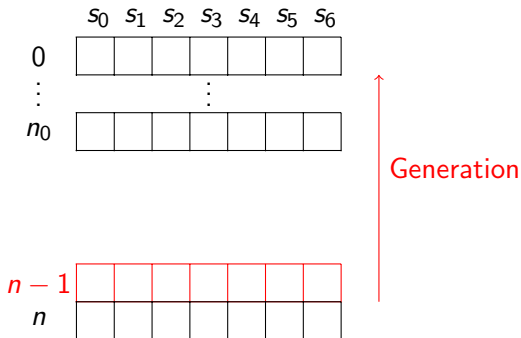


Goldwurm's idea (1995)



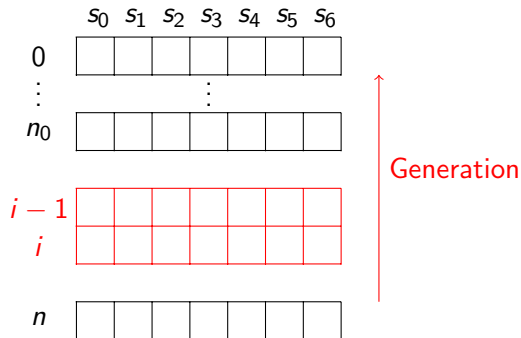


Goldwurm's idea (1995)



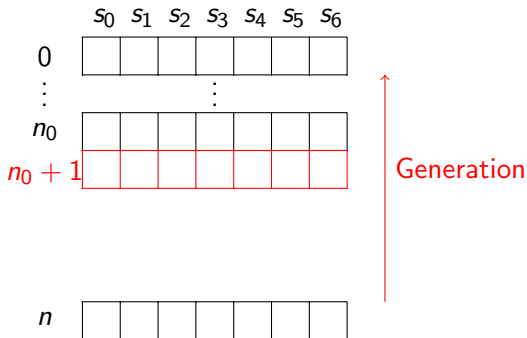


Goldwurm's idea (1995)



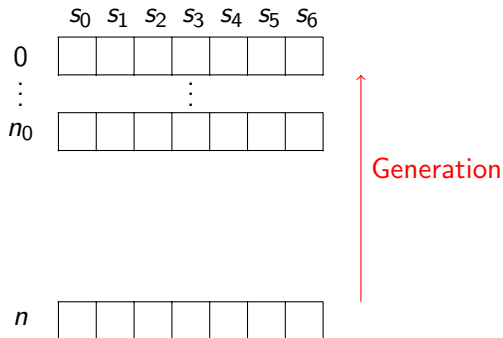


Goldwurm's idea (1995)





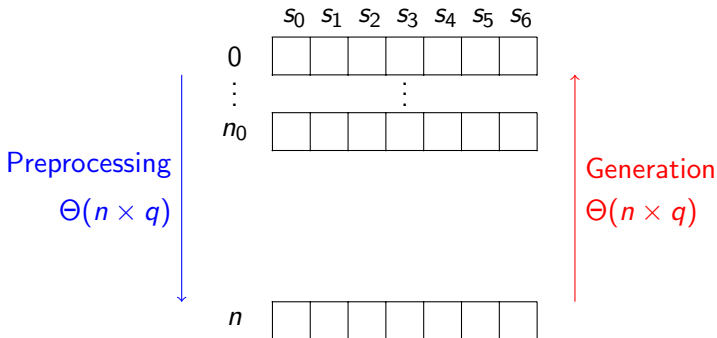
Goldwurm's idea (1995)





Goldwurm's idea (1995)

Space: $\Theta((n_0 + 4) \times q)$ numbers





Complexity of the recursive method and the Goldwurm's variant

q: size of the automaton

n: path length

- Hickey and Cohen (1983), algorithm based on recursive method [Wilf'77, Flajolet et al.'94]:
space: $\mathcal{O}(qn)$ time: $\mathcal{O}(qn) + \mathcal{O}(n)$



Complexity of the recursive method and the Goldwurm's variant

q : size of the automaton

n : path length

- Hickey and Cohen (1983), algorithm based on recursive method [Wilf'77, Flajolet et al.'94]:
space: $\mathcal{O}(qn)$ time: $\mathcal{O}(qn) + \mathcal{O}(n)$
- Goldwurm (1995), keep in memory only a few coefficients:
space: $\mathcal{O}(q)$ time: $\mathcal{O}(qn) + \mathcal{O}(qn)$



Complexity of the recursive method and the Goldwurm's variant

q: size of the automaton

n: path length

- Hickey and Cohen (1983), algorithm based on recursive method [Wilf'77, Flajolet et al.'94]:
space: $\mathcal{O}(qn^2)$ time: $\mathcal{O}(qn^2) + \mathcal{O}(n^2)$
- Goldwurm (1995), keep in memory only a few coefficients:
space: $\mathcal{O}(qn)$ time: $\mathcal{O}(qn^2) + \mathcal{O}(qn^2)$
- The above formulas must be multiplied by $\mathcal{O}(n)$ for the bit complexity, due to the exponential growth of the coefficients.



Using floating-point arithmetic to improve bit complexity

Can *floating-point arithmetic* be used to improve bit complexity?

- ✓ Denise and Zimmermann (1999), using floating-point arithmetic for the Hickey and Cohen's algorithm:

space: $\mathcal{O}(qn \log n)$

time: $\mathcal{O}(qn \log n) + \mathcal{O}(n \log n)$

- ✗ However, involved operations in Goldwurm's algorithm are numerically unstable.



Using floating-point arithmetic to improve bit complexity

Can *floating-point arithmetic* be used to improve bit complexity?

- ✓ Denise and Zimmermann (1999), using floating-point arithmetic for the Hickey and Cohen's algorithm:

space: $\mathcal{O}(qn \log n)$

time: $\mathcal{O}(qn \log n) + \mathcal{O}(n \log n)$

- ✗ However, involved operations in Goldwurm's algorithm are numerically unstable.

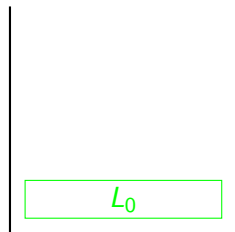
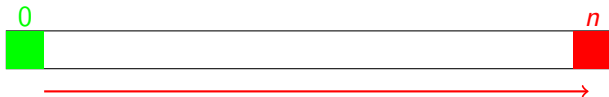
Can we design an algorithm that keeps in memory only a few coefficients and uses floating-point arithmetic?

- 1 Motivation and Prior Algorithms
- 2 Our Algorithm: Dichopile**
- 3 Related work: Boltzmann, Bernardi and Giménez
- 4 Experimental results
- 5 Conclusion



General principle [1/3]

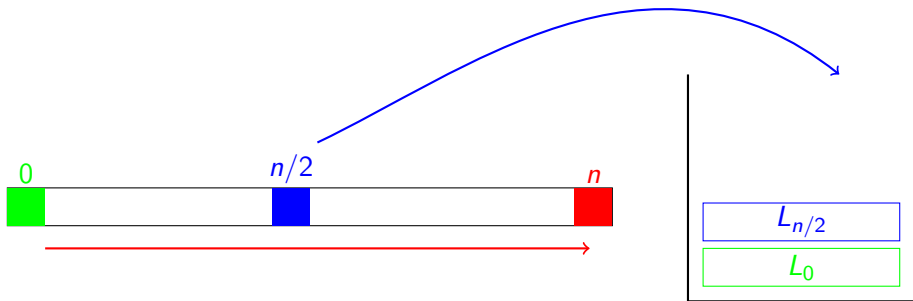
- Step 0: compute L_n from L_0 as in Hickey and Cohen's algorithm, but save only some lines L_i (i.e., the vector of the $l_s(i)$'s values for every state s).





General principle [1/3]

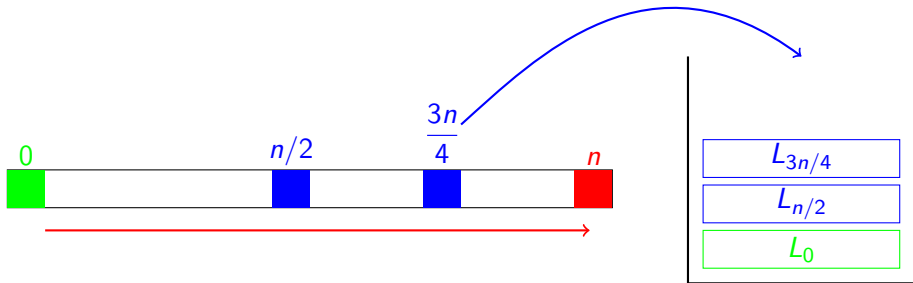
- Step 0: compute L_n from L_0 as in Hickey and Cohen's algorithm, but save only some lines L_i (i.e., the vector of the $l_s(i)$'s values for every state s).





General principle [1/3]

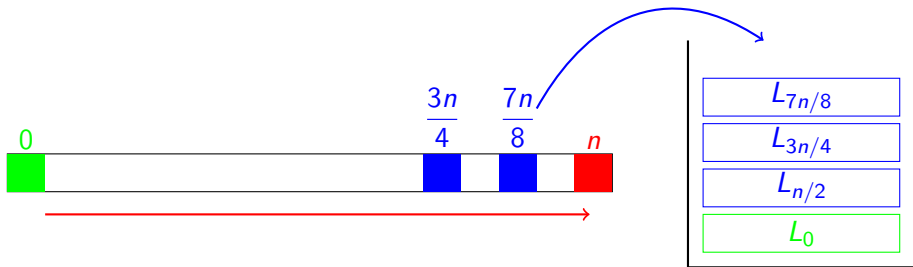
- Step 0: compute L_n from L_0 as in Hickey and Cohen's algorithm, but save only some lines L_i (i.e., the vector of the $l_s(i)$'s values for every state s).





General principle [1/3]

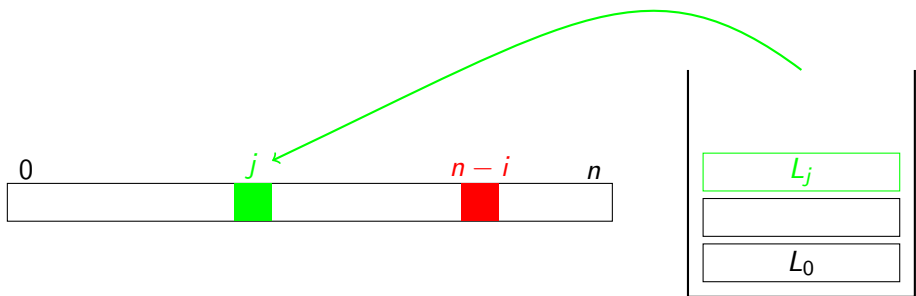
- Step 0: compute L_n from L_0 as in Hickey and Cohen's algorithm, but save only some lines L_i (i.e., the vector of the $l_s(i)$'s values for every state s).





General principle [2/3]

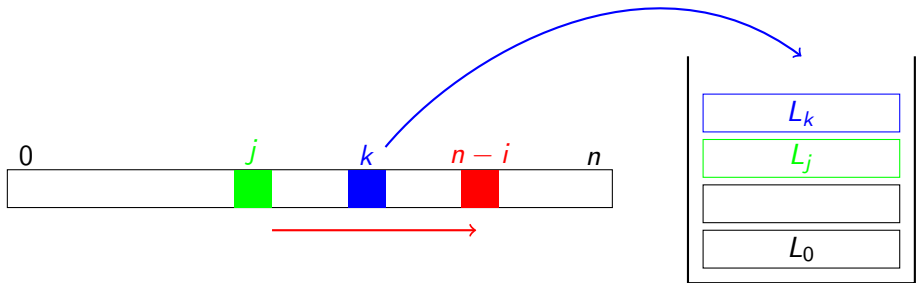
- Step 0: compute L_n from L_0
- Step i :
 - pop L_j from the stack





General principle [3/3]

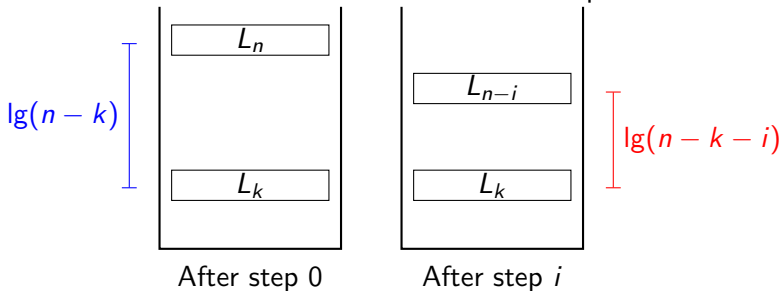
- Step 0: compute L_n from L_0
- Step i :
 - pop L_j from the stack
 - compute L_{n-i} from L_j





Complexity analysis: Space \sim stack size

- After Step 0, there are $\lg n$ elements on the stack;
- And there will never be more elements in the sequel:

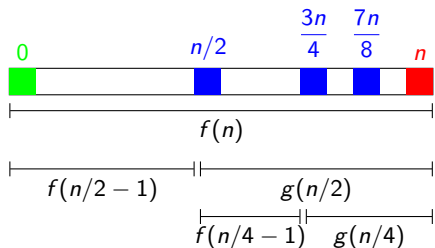


- As each L_i occupies $\mathcal{O}(q \log n)$ bits, space complexity is in $\mathcal{O}(q \log^2 n)$ bits.



Complexity analysis: Time \sim number of L_i calculations

- No preprocessing step.
- Complexity of computing L_i from L_{i-1} is in $\mathcal{O}(q \log n)$
- The number of these calculations is equal to $f(n)$, defined as:



$$\begin{cases} f(n) &= n + f(\lfloor n/2 \rfloor - 1) + g(\lceil n/2 \rceil) \\ g(n) &= f(\lfloor n/2 \rfloor - 1) + g(\lceil n/2 \rceil) \\ f(1) &= 1 \\ g(1) &= 0 \end{cases}$$

- $f(n) = \Theta(n \log n) \Rightarrow$ Time complexity is in $\mathcal{O}(qn \log^2 n)$

- 1 Motivation and Prior Algorithms
- 2 Our Algorithm: Dichopile
- 3 Related work: Boltzmann, Bernardi and Giménez**
- 4 Experimental results
- 5 Conclusion



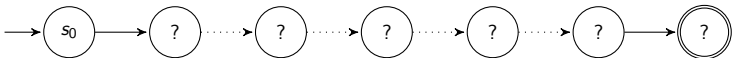
Boltzmann sampling

- Free generator: path length is variable.
- Uniformity among paths of same length.
- Duchon et al. (2004), Boltzmann generation (for exact size):
space: $\mathcal{O}(q)$ time: $\mathcal{O}(q^k \log^{k'} n) + \mathcal{O}(n^2)$
- Is k and k' small enough to generate long paths in large automata?



Bernardi and Giménez's algorithm

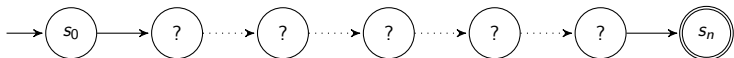
- Another dichotomic approach but very different from the one used in Dichopile:
 - 1 choose final state s_n according to $I_{s_0,s}(n)$;





Bernardi and Giménez's algorithm

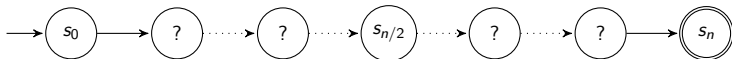
- Another dichotomic approach but very different from the one used in Dichopile:
 - 1 choose final state s_n according to $l_{s_0,s}(n)$;
 - 2 choose middle state $s_{n/2}$ according to $l_{s_0,s}(n/2)$ and $l_{s,s_n}(n/2)$;





Bernardi and Giménez's algorithm

- Another dichotomic approach but very different from the one used in Dichopile:
 - 1 choose final state s_n according to $I_{s_0,s}(n)$;
 - 2 choose middle state $s_{n/2}$ according to $I_{s_0,s}(n/2)$ and $I_{s,s_n}(n/2)$;
 - 3 repeat step 2 to choose states $s_{n/4}$ and $s_{3n/4}$, and so on.

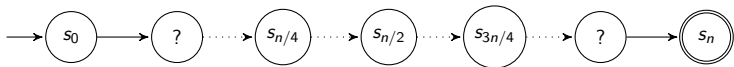




Bernardi and Giménez's algorithm

- Another dichotomic approach but very different from the one used in Dichopile:

- 1 choose final state s_n according to $I_{s_0, s}(n)$;
- 2 choose middle state $s_{n/2}$ according to $I_{s_0, s}(n/2)$ and $I_{s, s_n}(n/2)$;
- 3 repeat step 2 to choose states $s_{n/4}$ and $s_{3n/4}$, and so on.





Bernardi and Giménez's algorithm

- Another dichotomic approach but very different from the one used in Dichopile:
 - 1 choose final state s_n according to $I_{s_0, s}(n)$;
 - 2 choose middle state $s_{n/2}$ according to $I_{s_0, s}(n/2)$ and $I_{s, s_n}(n/2)$;
 - 3 repeat step 2 to choose states $s_{n/4}$ and $s_{3n/4}$, and so on.



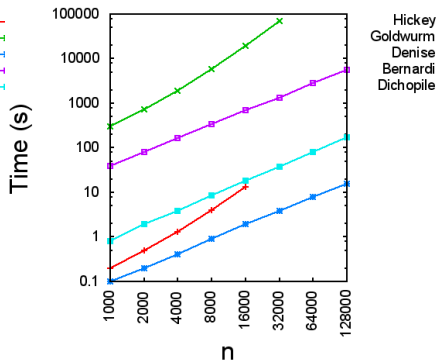
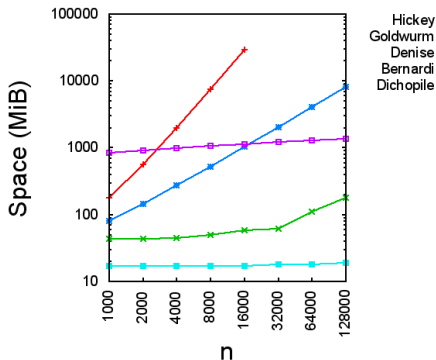
- Bernardi and Giménez (2010), divide and conquer algorithm:
space: $\mathcal{O}(q^2 \log(qn) \log n)$
time: $\mathcal{O}(q^3 \log^2(qn) \log n) + \mathcal{O}(qn \log(qn))$

- 1 Motivation and Prior Algorithms
- 2 Our Algorithm: Dichopile
- 3 Related work: Boltzmann, Bernardi and Giménez
- 4 Experimental results**
- 5 Conclusion



Space and elapsed time to generate 100 paths [1/2]

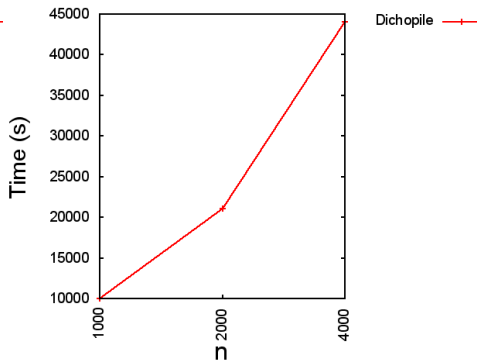
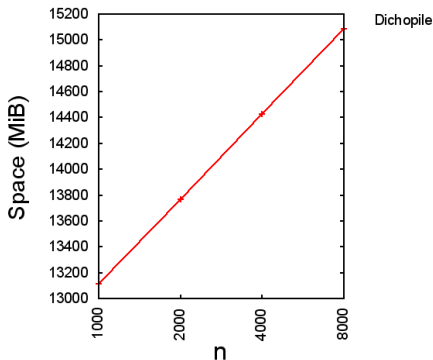
- model vasy_1_4 from the VLTS benchmark suite: 1, 183 states and 4, 464 edges.





Space and elapsed time to generate 100 paths [2/2]

- model vasy_12323_27667 from the VLTS benchmark suite: 12,323,703 states and 27,667,803 edges.



- 1 Motivation and Prior Algorithms
- 2 Our Algorithm: Dichopile
- 3 Related work: Boltzmann, Bernardi and Giménez
- 4 Experimental results
- 5 Conclusion**



Summary of the bit complexities

Method	Space	Time	
		Preprocessing	Generation
Hickey & Cohen	$\mathcal{O}(qn^2)$	$\mathcal{O}(qn^2)$	$\mathcal{O}(n^2)$
Goldwurm	$\mathcal{O}(qn)$	$\mathcal{O}(q^2 + qn^2)$	$\mathcal{O}(qn^2)$
Denise & Zimmermann	$\mathcal{O}(qn \log n)$	$\mathcal{O}(qn \log n)$	$\mathcal{O}(n \log n)$
Boltzmann	$\mathcal{O}(q)$	$\mathcal{O}(q^k \log^{k'} n)$	$\mathcal{O}(n^2)$
Bernardi & Giménez	$\mathcal{O}(q^2 \log(qn) \log n)$	$\mathcal{O}(q^3 \log^2(qn) \log n)$	$\mathcal{O}(qn \log(qn))$
Dichopile	$\mathcal{O}(q \log^2 n)$	$\mathcal{O}(1)$	$\mathcal{O}(qn \log^2 n)$



Conclusion

- Recursive method is fast but unusable for large n and q due to its huge space requirement.
- Boltzmann method needs less memory but is slower according to n .
- Bernardi & Giménez's algorithm and Dichopile are excellent compromises when considering both space and time complexities.
- Dichopile offers a better space complexity and the increase of time complexity according to n is quite low.



Further work

- Boltzmann generation is well fitted for approximate size generation: average time for the generation of words of length between $(1 - \varepsilon)n$ and $(1 + \varepsilon)n$, for a fixed value ε , is in $\mathcal{O}(n)$.
- Application to simulation, model-based testing, and model-checking: <http://rukia.lri.fr>