A $m \log m$ algorithm to compute the most probable configurations of a system with multi-mode independent components

A. Rauzy*

1 Summary & Conclusions

In this note, we propose a $m \log m$ algorithm to find the k most probable configurations of a system of n multi-mode independent components, with at most d modes each. m denotes the size of the problem, i.e. $max(nd, k)$. This problem originates in network performance analyses, in which focusing on the most probable configurations is a mean to reduce computational costs. Up to this note, the best known algorithm to extract the most probable configurations was in $\mathcal{O}(n^2d^2 + k \log k)$. Our algorithm achieves thus a substantial improvement.

*IML, CNRS, 163, avenue de Luminy 13288 Marseille cedex 9, France, arauzy@iml.univ-mrs.fr

2 Introduction

The problem we study in this note can be stated as follows.

Let $c_1, \ldots, c_n$ be n components. Each component c_i can be in state $s_{i,j}$, $j = 1, \ldots, d_i$ with a probability $p_{i,j}$ ($\sum_{j=1}^{d_1} p_{i,j} = 1$). Components are assumed to be mutually independent. Given an integer k, find the k most probable configurations of the system formed by these n components. Throughout the article, we keep the same meaning for n, c_i, d_i, $s_{i,j}$, $p_{i,j}$ and k. Moreover, we denote by d the maximum over n of the d_i's.

The above problem originates in network performance analyses, in which focusing on the most probable configurations is a mean to reduce computational costs [Mey92, LS84]. Several authors proposed various algorithms to select those configurations (e.g. [LL86, CL86, YK90]). Recently, T. Gomes, J. Craveirinha and L. Martins reduced the problem to a shortest paths problem [GCM01]. Applying an algorithm proposed in reference [MPS99], they achieved a complexity in $\mathcal{O}(n^2d^2 + k \log k)$. The factor n^2d^2 comes from the reduction (the graph construction), while the factor $k \log k$ comes from the path enumeration.

Our algorithm works also in two steps with a preprocessing phase followed with the enumeration of the most probable configurations. The latter is similar to the one proposed by T. Gomes & al (and has the same complexity, namely $k \log k$). The former exploits better the structure of the problem than their reduction does. It just consists in sorting components and states inside components. It works in $\mathcal{O}(nd \log d + n \log n)$.

The remainder of this note is organized as follows. Data structures to encode configurations and the preprocessing phase are described section 3. The enumeration phase is

presented section 4. An illustrative example as well as some experimental results are given section 5.

3 Preprocessing

We can assume without a loss of generality that, for each i, $p_{i,1} \geq \ldots \geq p_{i,d_i}$. The most probable configuration is therefore $S_1 = \langle s_{1,1}, \ldots, s_{n,1} \rangle$. Its probability is $P_1 = \Pi_{i=1}^{n} p_{i,1}$.

Let $q_{i,j}$ $(1 \leq i \leq n,\ 2 \leq j \leq d_i)$ be defined as follows.

$$q_{i,j} \quad = \quad \frac{p_{i,j}}{p_{i,j-1}} \tag{1}$$

By definition, we have $p_{i,j} = p_{i,1} \times \Pi_{k=2}^{j}\ q_{i,k}$. Moreover, the probability P of any configuration $S = \langle s_{1,h_1}, \ldots, s_{n,h_n} \rangle$ can be written in terms of the $q_{i,j}$'s and P_1 as follows.

$$P = \Pi_{i=1}^{n}\ p_{i,h_i} \quad = \quad P_1 \times \Pi_{i=1}^{n} \Pi_{j=2}^{h_i}\ q_{i,j} \tag{2}$$

Equation 2 gives a mean to represent configurations by successive differences with the root configuration S_1. Let $S = \langle s_{1,h_1}, \ldots, s_{i,h_i}, \ldots, s_{n,h_n} \rangle$ be a configuration of probability P. The probability of the configuration $S' = \langle s_{1,h_1}, \ldots, s_{i,h_i+1}, \ldots, s_{n,h_n} \rangle$, if any, is $Q = P \times q_{i,h_i+1}$. By definition, $Q \leq P$. The idea is therefore to encode the K most probable configurations as a tree, the root of which is the configuration S_1. In other words, each configuration S' (but S_1) is encoded by a pair the pair $S' = \langle S, s_{i,h_i+1} \rangle$. This idea was proposed first in reference [GCM01].

In what follows, we will denote by $S.s_{i,j+1}$ the configuration S in which the state $s_{i,j+1}$ has been substituted for the state $s_{i,j}$.

The first phase of the algorithm consists in three steps: first, states of each component are sorted by decreasing order of their probabilities. Second, the $q_{i,j}$'s are computed. Third, components are sorted in decreasing order of the $q_{i,2}$'s.

In what follows, we denote by $R[i]$ the ith component in the above order and by $R[i,j]$ the $(j+1)$th state of the ith component.

Sorting the states of a component in decreasing order of their probabilities is done in $\mathcal{O}(d \log d)$ (using, for instance, the quicksort algorithm [CLR90]). Computing the $q_{i,j}$'s of a component is done in $\mathcal{O}(d)$. Sorting the components is done in $\mathcal{O}(n \log n)$. The first step is thus in $\mathcal{O}(nd \log d + n \log n)$.

4 Enumeration of the most probable configurations

The second phase of the algorithm consists of a loop iterating from 1 to k and producing the most probable configurations in order. In addition to the tree encoding of the configurations and their probabilities, and the row R, it uses a list M of the most probable configurations already computed and a set C of candidate configurations. C is sorted in decreasing order of the probabilities of its members. Each configuration $S.s_{i,j}$ stored in C is encoded by means of a triple $\langle S, r, l \rangle$, where (r,l) is the unique pair of indices such that $R[r,l] = s_{i,j}$.

The pseudo-code for the second step is given Fig. 1.

The basic idea of the algorithm is the following. Let $S = \langle s_{1,h_1}, \ldots, s_{n,h_n} \rangle$ be a configuration, let i be the index with the greatest q_{i,h_i+1}, finally let $S' = S.q_{i,h_i+1}$. There are basically two ways to degrade S beyond S'. Either one degrades c_i again or one degrades another component. Let c_j be the component with the second greatest q_{j,h_j+1}. The configuration that degrades the less S (after S') is necessarily either $S.s_{j,h_j+1}$ or $S'.s_{i,h_i+2}$. For the same reason, the configuration that degrades the less S' is necessarily either $S'.s_{j,h_j+1}$ or $S'.s_{i,h_i+2}$. Therefore, the order computed during the preprocessing ensures that, at any step of the algorithm, the set C contains the most probable candidate configuration.

The second step of the algorithm consists of $k-1$ iterations of the loop. Each iteration consists of one insertion in M, at most 3 insertions in C (lines 6, 7 and 8) and at most 4 removings from C (lines 5 and 9-10). Thanks to the representation of configurations by difference, building the new configurations is done in constant time. C contains at most $k-i+1$ configurations at the ith iteration. Insertions and removings in a sorted set with m elements can be done in $\log m$, using a binary heap or an AVL-Tree to encode the set [CLR90]. Therefore, the second step is in $\mathcal{O}(k.\log k)$.

The complexity of the whole algorithm is thus in $\mathcal{O}(nd \log d + n \log n + k.\log k)$. It is reasonnable to assume that in practice $d \ll n$ — components of real networks have in general only very few observable modes. The size m of the problem can be defined as nd (the total number of states). However, this definition has the drawback not to take into account the size of the result. This is the reason why it seems more suitable to define m as $max(nd, k)$. With this latter definition, our algorithm is in $\mathcal{O}(m \log m)$, hence the title of our note. There are good reasons to think that this complexity is near optimal, at least

if reasonable assumptions are made on d, n and k (i.e. $d \ll n$ and $k \ll \Pi_{i=1}^{n} d_i$). To store the problem, one needs at least to store the nd $p_{i,j}$'s. Therefore the improvement, if any, should be on the factor $\log d + \log n/d$, which for practical applications is always very low. Similarly, constructing k configurations is at least in $\mathcal{O}(k)$. Therefore, the improvement, if any, should be on the factor $\log k$, which again should be low as well.

Finally, it is worth noticing that it could be interesting to let k change dynamically in order to produce sufficiently many configurations to cover the whole state space with a reasonnably high probability. It is easy to modify our algorithm to achieve this goal.

5 Experimental results

An illustrative example: As an illustration, we consider the example proposed in reference [GCM01]. The system is made of 3 components whose parameters are described in table 1.

$S_1 = \langle s_{1,1}, s_{2,1}, s_{3,1} \rangle$ and $P_1 = 0.7 \times 0.5 \times 0.8 = 0.28$. Moreover, $q_{2,3} \geq q_{2,2}$, and $q_{2,2} > q_{1,2} > q_{3,2}$. Therefore, $R[1,1] = s_{2,2}$, $R[1,2] = s_{2,3}$, $R[2,1] = s_{1,2}$ and $R[3,1] = s_{3,2}$.

The table 2 gives the contents of M and C at the different steps of the algorithm. The algorithm is applied with no limit on k.

Experiment results: In order to verify our algorithm, we applied it on several instances generated as follows. All the components have the same number of states d. The probability is each state (but the last one) is drawn at random according to a normal law with a

mean $1/d$ and a standard deviation $1/5d$. Experiments were performed on a pentium III, cadenced at 733MHz and running Linux.

First, we applied the algorithm on instances with $n = 1000$, $k = 10^6$ and $d = 3, 4, \ldots, 10$. Running times to parse instances range from 10ms (for $d = 1$) to 40ms (for $d > 7$). Running times for the preprocessing never exceeds 10ms. Running times for the loop are the same for all values of d, namely arround 350ms. As a comparison, the running times reported in reference [GCM01] — that were obtained on a similar machine and system — for $d = 3, 4$ and 5, were respectively of 200ms, 350ms and 500ms for the graph construction and 550ms for the loop.

Then, we applied the algorithm on instances with $d = 10$, $k = 10^6$ and $n = 1000$, 2000,..., 10000. Running times are given table 3. These results show that running times of the loop increase slighly as n increases. This is due to the increasing number of candidate configurations. Our generation model has been designed to make this number increase with n.

Finally, we applied the algorithm on instances with $d = 5$, $n = 1000$ and $k = 0.5\ 10^5$, $1.0\ 10^6$, ..., $5\ 10^6$. Running times are given table 4.

These results show that running times grow almost linearly with k.

All these latter experiments show that our algorithm is very efficient, even on very large instances. The problem of finding the most probable configurations of a system of n multi-mode independent components can be considered as solved, at least for what concerns practical instances.

References

[CL86] S.N. Chiou and V.O.K. Li. Reliability analysis of a communication network with multimode components. *IEEE Journal on Selected Areas in Communications*, 4(7):1156–1161, 1986.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990. ISBN 0-262-03141-8.

[GCM01] T. Gomes, J. Craveirinha, and L. Martins. An efficient algorithm for sequential generation of failure states in a network with multi-mode components. *Reliability Engineering and System Safety*, 2001. submitted.

[LL86] Y.F. Lam and V.O.K Li. An improved algorithm for performance analysis of networks with unreliable components. *IEEE Transactions of Communications*, 34(5):496–497, 1986.

[LS84] V.O.K. Li and J.A. Silvester. Performance analysis of networks with unreliable components. *IEEE Transactions On Communications*, 32(10):1105–1110, 1984.

[Mey92] J.F. Meyer. Performability: a retrospective and some pointers to the future. *Peformance Evaluation*, 14((3,4)):139–156, 1992.

[MPS99] E. Martins, M. Pascoal, and J. Santos. Deviation algorithms for ranking shortest paths. *International Journal of Foundations of Computer Science*, 10(3):247–263, 1999.

[YK90] C.-L. Yang and P. Kubat. An algorithm for network reliability bounds. *ORSA Journal on Computing*, 2(2):336–345, 1990.

Figures

1: $C \leftarrow \{\langle S_1, 1, 1\rangle\}$, $M \leftarrow \{S_1\}$, $i \leftarrow 2$
2: **while** $i \leq k$ **do**
3: Let $T = \langle S, r, l\rangle$ be the first configuration of C, i.e. the one such that
4: $S.R[r, l]$ has the highest probability.
5: $i \leftarrow i + 1$, $C \leftarrow C \setminus \{T\}$, $M \leftarrow M \cup \{S.R[r, l]\}$
6: **if** $l = 1$ and $r < n$ **then** $C \leftarrow C \cup \langle S, r + 1, 1\rangle$
7: **if** $l <$ length$(R[r])$ **then** $C \leftarrow C \cup \langle S.R[r, l], r, l + 1\rangle$
8: **if** $r < n$ **then** $C \leftarrow C \cup \langle S.R[r, l], r + 1, 1\rangle$
9: **if** C contains more than $k - i + 1$ configurations, **then** remove the last,
10: i.e. the less probable, one until C contains exactly $k - i + 1$ configurations.
11: **done**

Figure 1: The pseudo-code for the second step of the algorithm

Tables

i	$p_{i,1}$	$p_{i,2}$	$p_{i,3}$	$q_{i,2}$	$q_{i,3}$
1	0.7	0.3		$0.3/0.7 \approx 0.49$	
2	0.5	0.3	0.2	$0.3/0.5 = 0.6$	$0.2/0.3 \approx 0.667$
3	0.8	0.2		$0.2/0.8 = 0.25$	

Table 1: A small system and its parameters.

k	configurations added to M	configurations added to C
1	$S_1 = \langle s_{1,1}, s_{2,1}, s_{3,1} \rangle$	*line* 1 $S_2 = \langle S_1, 1, 1 \rangle$ $P_2 = 0.168$
2	$S_2 = S_1.s_{2,2} = \langle s_{1,1}, s_{2,2}, s_{3,1} \rangle$	*line* 6 $S_3 = \langle S_1, 2, 1 \rangle$ $P_3 = 0.120$ *line* 7 $S_4 = \langle S_2, 1, 3 \rangle$ $P_4 = 0.112$ *line* 8 $S_5 = \langle S_2, 2, 1 \rangle$ $P_5 = 0.072$
3	$S_3 = S_1.s_{1,2} = \langle s_{1,2}, s_{2,1}, s_{3,1} \rangle$	*line* 6 $S_6 = \langle S_1, 3, 1 \rangle$ $P_6 = 0.070$ *line* 8 $S_7 = \langle S_3, 3, 1 \rangle$ $P_7 = 0.030$
4	$S_4 = S_2.s_{2,3} = \langle s_{1,1}, s_{2,3}, s_{3,1} \rangle$	*line* 8 $S_8 = \langle S_4, 2, 1 \rangle$ $P_8 = 0.048$
5	$S_5 = S_2.s_{1,2} = \langle s_{1,2}, s_{2,2}, s_{3,1} \rangle$	*line* 6 $S_9 = \langle S_2, 3, 1 \rangle$ $P_9 = 0.042$ *line* 8 $S_{10} = \langle S_5, 3, 1 \rangle$ $P_{10} = 0.018$
6	$S_6 = S_1.s_{3,2} = \langle s_{1,1}, s_{2,1}, s_{3,2} \rangle$	
7	$S_8 = S_4.s_{1,2} = \langle s_{1,2}, s_{2,3}, s_{3,1} \rangle$	*line* 6 $S_{11} = \langle S_4, 3, 1 \rangle$ $P_{11} = 0.028$ *line* 8 $S_{12} = \langle S_8, 3, 1 \rangle$ $P_{12} = 0.012$
8	$S_9 = S_2.s_{3,2} = \langle s_{1,1}, s_{2,2}, s_{3,2} \rangle$	
9	$S_7 = S_3.s_{3,2} = \langle s_{1,2}, s_{2,1}, s_{3,2} \rangle$	
10	$S_{11} = S_4.s_{3,2} = \langle s_{1,1}, s_{2,3}, s_{3,2} \rangle$	
11	$S_{10} = S_5.s_{3,2} = \langle s_{1,2}, s_{2,2}, s_{3,2} \rangle$	
12	$S_{12} = S_8.s_{3,2} = \langle s_{1,2}, s_{2,3}, s_{3,2} \rangle$	

Table 2: Trace of the second step of the algorithm.

d	1	2	3	4	5	6	7	8	9	10
parsing	30	40	60	80	100	110	130	150	170	180
preprocessing	10	10	20	20	30	40	50	50	60	60
loop	340	370	410	410	450	450	460	470	500	510

Table 3: Running times in milliseconds for with $d = 10$ and $k = 10^6$

$k.10^{-6}$	0.5	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	5.0
loop	180	350	520	690	860	1040	1210	1390	1550	1730

Table 4: Running times in milliseconds for with $d = 5$ and $n = 1000$