Reaction Factoring and Bipartite Update Graphs Accelerate the Gillespie Algorithm for Large-Scale Biochemical Systems

Sagar Indurkhya[1], **Jacob Beal**[2]*

1 Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, United States of America, **2** BBN Technologies, Cambridge, Massachusetts, United States of America

Abstract

ODE simulations of chemical systems perform poorly when some of the species have extremely low concentrations. Stochastic simulation methods, which can handle this case, have been impractical for large systems due to computational complexity. We observe, however, that when modeling complex biological systems: (1) a small number of reactions tend to occur a disproportionately large percentage of the time, and (2) a small number of species tend to participate in a disproportionately large percentage of reactions. We exploit these properties in LOLCAT Method, a new implementation of the Gillespie Algorithm. First, factoring reaction propensities allows many propensities dependent on a single species to be updated in a single operation. Second, representing dependencies between reactions with a bipartite graph of reactions and species requires only $O(n)$ storage for n reactions, rather than the $O(n^2)$ required for a graph that includes only reactions. Together, these improvements allow our implementation of LOLCAT Method to execute orders of magnitude faster than currently existing Gillespie Algorithm variants when simulating several yeast MAPK cascade models.

Citation: Indurkhya S, Beal J (2010) Reaction Factoring and Bipartite Update Graphs Accelerate the Gillespie Algorithm for Large-Scale Biochemical Systems. PLoS ONE 5(1): e8125. doi:10.1371/journal.pone.0008125

Editor: Mark Isalan, Center for Genomic Regulation, Spain

Received May 21, 2009; **Accepted** September 4, 2009; **Published** January 6, 2010

Funding: This work was supported by NSF Grant 6898853. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Competing Interests: BBN Technologies is an independent research laboratory, and for this work, the only financial interest of BBN is the same as that of a university: publication will improve the chances of obtaining grants relating to the material. There are no patents relating to this manuscript. Jacob Beal is affiliated to BBN Technologies; this affiliation does not impede the data sharing policies of PLoS ONE.

* E-mail: jakebeal@bbn.com

Introduction

Dynamic Monte Carlo methods are a common means of simulating the time-evolution of chemical systems. The Gillespie Algorithm (SSA) [1] is the standard algorithm for this process, and has inspired a variety of derivative methods that speed up computation, including the Optimized Direct Method (ODM) [2] and the Next Reaction Method (NRM) [3]. These methods, however, are still computationally costly. Speeding up the Gillespie Algorithm and related hybrid methods will likely play an important role in advancing the productivity of computational systems biology.

In this paper, we develop a new algorithm, LOLCAT Method, that can speed up the exact stochastic simulation of a large class of well-mixed chemical systems by orders of magnitude. The ability to perform simulations orders of magnitude faster will allow scientists to revisit problems that were previously computationally intractable, such as whole-cell simulation. Other applications include the simulation of large genetic regulatory networks and metabolic pathways. Further, we demonstrate that this algorithm can operate on a typical desktop personal computer, bringing the simulation of extremely large chemical systems into the reach of the general scientific community. Our implementation of LOLCAT Method is publicly available at [4]

Simulating Chemical Systems

There are two main approaches to simulating the time-evolution of chemical systems: deterministic and stochastic. Deterministic methods express chemical concentrations with real numbers, and evolve concentrations forward in time via differential equations. Stochastic methods, on the other hand, express concentrations as non-negative integers, and evolve them in discrete steps: at each step, a reaction is chosen and executed, transforming a set of reactants into a set of products and advancing the simulation time by a small amount. The reaction and time step are chosen randomly according to a distribution that produces a statistically correct simulation of the chemical system.

Stochastic methods are generally much slower because they simulate every chemical interaction explicitly, but produce guaranteed valid results for systems where concentrations are very small. Deterministic methods are much faster, but do not handle small concentrations well. This is particularly problematic when dealing with models of biological systems, since these often contain important chemical species with dozens of molecules or less.

Hybrid methods attempt to resolve this dilemma by mixing deterministic and stochastic methods, and are sped up significantly by techniques such as tau-leaping at the expense of not being exact (although they are becoming increasingly accurate and are good enough for many problems) [5,6]. These hybrid methods are often based on some exact method [7], so LOLCAT Method can likely increase the speed of even these hybrid methods. For a thorough treatment of stochastic versus deterministic simulations, see [3].

Stochastic Simulation Algorithm

The Stochastic Simulation Algorithm (SSA), also known as the Gillespie Algorithm, is a standard, well established approach for computing statistically correct trajectories of the time evolution of spatially homogeneous chemical systems. Unfortunately the SSA is extremely computationally expensive. We review the SSA briefly to build a foundation for deriving LOLCAT Method.

Consider a chemical system of constant volume (for systems with variable volume, see the Supporting Information S1) in which there is a set of species S and a set of reactions R governing the interactions of these species. Assume that all reactions have at most two reactants and two products (any reaction with more than two reactants or products can be factored into reactions with at most two reactants and products), as well as a constant reaction rate. The SSA is then as follows:

1. Initialize simulation time $t_{sim}=0$, and the initial concentration of each species.
2. For each reaction $r \in R$, compute the reaction propensity a_r; if reaction r is $A+B \rightarrow C+D, k$ (where k is the reaction rate) we have $a_r = k[A][B]$, where $[A]$ and $[B]$ are the number of molecules present in the system for species A and B respectively. Note that if A and B are the same species, then we instead have $a = k[A]([A]-1)/2$, and for reactions with less then two reactants and products, we use the the chemical $NULL$, which is omnipresent with constant concentration $[NULL]=1$. Compute the sum of the propensities: $a_R = \sum_{r \in R} a_r$.
3. Choose a reaction, r^*, at random from the weighted distribution of reaction propensities. Each reaction $r \in R$ has probability a_r/a_R of being chosen.
4. Execute reaction r^*: subtract 1 from the concentration of each reactant, and add 1 to the concentration of each product.
5. Update the simulation time: $t'_{sim} = t_{sim} - log(z)/a_R$, where z is chosen uniformly randomly from (0,1]
6. Record species concentrations of interest as desired for the experiment. Go to step 2 until the simulation is completed (usually when some desired amount of time has been simulated, or when a desired number of iterations have been completed).

Each iteration of the SSA may also be viewed as two phases: a read phase, in which the algorithm chooses a random number and maps it to the reaction to execute (step 3), and a write phase, in which the reaction is executed, propensities are adjusted, and simulation time is advanced (steps 2, 4, and 5). We use this two-phase view in describing LOLCAT Method, as focuses attention on the interaction of the algorithm with its supporting data structures.

Review of Prior Methods

The SSA has inspired a variety of derivative methods that speed up computation. The two most significant are the Optimized Direct Method (ODM) and the Next Reaction Method (NRM).

The Optimized Direct Method (ODM) [2] stores the reactions in an array. First, a random number q is chosen uniformly from the interval $(0,a_R]$. The algorithm then steps through the array (linearly) subtracting a_{r_k}, (where r_k is the k^{th} reaction) from q until $q \leq 0$, and at that point takes $r^* = r_k$. Several simulations are run ahead of time to determine an average propensity for each reaction and the reactions are sorted by average propensity from greatest to least. In most biochemical systems, a relatively small number of reactions occur a disproportionately high percentage of the time, so ODM's linear search allows for these reactions to be quickly detected at the front of the array. The time for choosing a reaction is $O(|R|)$, but the sorting causes it to be much faster on average.

To recompute the propensities, ODM uses an update dependency graph (UDG) that maps each reaction r to a list of u_r reactions whose propensities should be updated upon execution of r. As a result, recomputing propensities takes $O(u_r)$ time. For large systems, u_r is often high for the most frequently occurring reactions, dramatically slowing down ODM.

The Next Reaction Method (NRM) [3] takes a different approach than ODM. NRM computes for each reaction the amount of time before it will next occur. It then stores the reactions in a binary min-heap structure, so that the next reaction to occur is always at the top. This reaction is executed, and then the necessary reactions have their propensity (and wait-times) updated (using a UDG as ODM does). However, as each reaction propensity is updated, it must be shifted up or down in the heap to maintain the min-heap property. While choosing a reaction takes $O(1)$ time, execution and update requires $O(u_r \log R)$ time. Thus, for large systems the size of u_r tends to slow NRM dramatically, just as it does ODM.

Other methods exist, but their performance and algorithmic structure are similar to ODM or NRM. For example, the Sorting Direct Method [8] is like ODM with dynamic reordering of reactions as propensities change, and Logarithmic Direct Method [9] is like ODM with propensities are stored in a binary structure for faster lookup of r^*. Although these methods sometimes perform better, the measured speedup is only a small constant, and we thus compare only against ODM and NRM.

Methods

We make two observations that appear to apply in many models of large scale biochemical systems:

- A small number of reactions tend to occur a disproportionately large percentage of the time.
- A relatively small proportion of the species take part in a relatively large proportion of the reactions (e.g. ATP and water in some cellular systems). One consequence is that these change concentration with a disproportionately high frequency. We will refer to these species as super-species.

LOLCAT Method exploits these observations in two ways. First, when possible, reactions are grouped by a common reactant and the common reactant's concentration factored out, allowing the simultaneous update of the propensities of reactions in that group. Second, the update dependency graph is stored in a bipartite representation to reduce the amount of computer memory required. We discuss these ideas in detail one at a time, introducing necessary data structures along the way, and then formally describe the algorithm in its whole.

Factoring Out Common Reactants

The propensities of reactions with one or more common reactants can be grouped together:

1. The sum of the propensities for a set of n reactions Q with *a single common reactant* X_0 can be factored as: $a_Q = \sum_{i=1}^{n} k_i[X_0][X_i] = [X_0]\sum_{i=1}^{n} k_i[X_i]$, where X_i is the other reactant in the ith reaction in Q and k_i is its reaction rate.
2. The sum of the propensities for a set of reactions Q with *both reactants in common* can be factored as: $a_Q = \sum_{i=1}^{n} k_i[X_0][X_1] = [X_0][X_1]\sum_{i=1}^{n} k_i$.

From this observation, we derive a brief mathematical foundation which we will employ in later developing the data structures for LOLCAT Method. Consider a set of reactions T in which all reactions have a common reactant X_α. We may then further partition this set of reactions into sets G and $H_1,H_2,...,H_\psi$, where each set of reactions H_i, $i \in \{1,2,...,\psi\}$ has a unique secondary common reactant X_{β,H_i}, and G contains all other reactions in T (those without shared secondary common reactants, or where the number of reactions sharing a secondary common reactant is too small for a second factoring to be beneficial).

The sum of the propensities of the reactions in G is

$$S_{\alpha G} = \sum_{r \in G} [X_\alpha][X_r]k_r = [X_\alpha] \sum_{r \in G} [X_r]k_r \tag{1}$$

$$= [X_\alpha] S_G \tag{2}$$

defining $S_G = \sum_{r \in G} [X_r]k_r$. Notice that we factored out the $[X_\alpha]$ common to all reactions in T.

The sum of the propensities of the reactions in H_i, $i \in \{1,2,...,\psi\}$ is

$$S_{\alpha H_i} = \sum_{r \in H_i} [X_\alpha][X_{\beta,H_i}]k_r = [X_\alpha]([X_{\beta,H_i}] \sum_{r \in H_i} k_r) \tag{3}$$

$$= [X_\alpha] S_{H_i} \tag{4}$$

defining $S_{H_i} = [X_{\beta,H_i}] \sum_{r \in H_i} k_r$. Note that we factored out the common reactant $[X_\alpha]$ from all H_i, $i \in \{1,2,...,\psi\}$, and factored out each secondary common reactant $[X_{\beta,H_i}]$ as well from each H_i. Also note that $\sum_{r \in H_i} k_r$ is a constant.

We let $S_T = S_G + \sum_{i=1}^{\psi} S_{H_i}$, and $S_{\alpha T} = [X_\alpha] S_T$. $S_{\alpha T}$ is the sum of the reaction propensities for all reactions in T. Now we show some operations we can perform.

1. If we increment the concentration $[X_\alpha]$ by some amount Δx, then we can produce updated values as follows:

$$[X_\alpha]' = [X_\alpha] + \Delta x \tag{5}$$

$$S'_{\alpha T} = [X_\alpha]' S_T \tag{6}$$

   Without the use of factoring, this would have incurred $O(|T|)$ operations rather than $O(1)$. This provides one of the main speedups of LOLCAT Method.

2. If we increment $[X_r], r \in G$ by Δx, then we have:

$$[X_r]' = [X_r] + \Delta x \tag{7}$$

$$S'_G = S_G + k_r \Delta x \tag{8}$$

$$S'_T = S_T + (S'_G - S_G) = S_T + k_r \Delta x \tag{9}$$

$$S'_{\alpha T} = [X_\alpha] S'_T \tag{10}$$

3. If we increment $[X_{\beta,H_i}], i \in \{1,2,...,\psi\}$ by Δx, then we have (noting again that $\sum_{r \in H_i} k_r$ is a constant):

$$[X_{\beta,H_i}]' = [X_{\beta,H_i}] + \Delta x \tag{11}$$

$$S'_{H_i} = S_{H_i} + \Delta x \sum_{r \in H_i} k_r \tag{12}$$

$$S'_T = S_T + (S'_{H_i} - S_{H_i}) = S_T + \Delta x \sum_{r \in H_i} k_r \tag{13}$$

$$S'_{\alpha T} = [X_\alpha] S'_T \tag{14}$$

The latter two provide less speed-up, but factoring still provides some benefit.

Reaction Group Data Structures

LOLCAT Method embodies these mathematical insights in a specialized data structure that we call a *cloud*. Each cloud stores the factored propensities for a group of reactions sharing a common species. Continuing with our sets $G,H_1,H_2,...,H_\psi$ and T used previously, we define the cloud's *factor-species* as their common reactant X_α. The cloud then consists of a *primary tree*, a set of *sub-trees*, and a slot which holds S_T.

The *primary tree* is a balanced d-ary tree with each leaf node having a one-to-one mapping to a reaction in G. The value of the leaf node tied to reaction $r \in G$ is $[X_{\beta,r}]k_r$ (where $X_{\beta,r}$ is the reactant in r that is not the *factor-species* X_α), and the value of any non-leaf node holds the sum of its child nodes. Thus the root of the primary tree holds the value S_G.

Note that we choose d-ary trees rather than binary trees. The branching factor of a tree controls its height: shorter trees are faster to update, but more branches require more tests to find a node. On modern processors, the optimal branching factor may be much greater than the 2 dictated by binary trees. See the Supporting Information S1 for details.

The cloud also has ψ different *sub-trees*, which are also d-ary trees. Each leaf node of the i^{th} sub-tree has a one-to-one mapping to a reaction in H_i: the value of the leaf node tied to reaction $r \in H_i$ is k_r. Each non-leaf node holds the sum of its child nodes. Thus the root node of the i^{th} sub-tree holds the value $\frac{S_{H_i}}{X_{\beta,H_i}}$.

In our implementation of LOLCAT Method, clouds are constructed by a preprocessing program that takes in the input system to be simulated, along with samples of the average propensities of each reaction over trial simulation runs. Reactions are assigned to clouds using a greedy approach:

1. A potential cloud is created for each species in the system, and each reaction is put into any cloud that could hypothetically contain it.
2. The sum of propensities of reactions in each cloud (based on a trial run) is computed. This heuristic score estimates how often the cloud's grouping will be taken advantage of.
3. The cloud with the best score is fixed, and all reactions in the newly fixed cloud are removed from any other clouds they are in.
4. Rescore each cloud (discarding empty clouds) and continue the process until every reaction is part of a fixed cloud.

Within a cloud, reactions with two common reactants are assigned to the primary tree unless there are enough of them (three or more in our implementation) that creating a sub-tree for them is deemed worthwhile.

Clouds are stored in a balanced d-ary tree, which we call the *main tree*, for fast access. Each leaf node has a unique one-to-one mapping with a cloud and holds the value $S_{\alpha T}$ for that cloud. Each non-leaf node holds the sum of its child nodes, such that the root node of the tree holds the sum of the propensities of all reactions that reside in a cloud. When selecting a reaction using a random number q, we traverse down this tree using our random number to determine which child node to proceed to until we reach a cloud. If there are C clouds, then this operation takes $O(\log C)$ time. Once we reach a cloud we rescale the random number to aid in selecting a reaction from the cloud. See the formal description of the LOLCAT Method below for more details.

Finally, to take advantage of the observation that a few particular reactions will occur disproportionately frequently, we segregate these reactions out and store them in a small static array, which we call the *super-cache*. This holds the reactions with the largest average propensity (and therefore the reactions most likely to be executed during the simulation). We also maintain a sum, S_{SC}, of the propensities in the super-cache. The super-cache is always searched first, and its size is chosen to balance the cost of linear search against the advantages of local access.

Figure 1 shows an example of reactions organized by LOLCAT Method into a super-cache, main tree, and clouds.

Using a Bipartite Update Dependency Graph

After a reaction executes, the reaction propensity must be updated for all reactions whose reactants' concentrations have changed. ODM and NRM store the dependencies between reactions in an update dependency graph (UDG) where each node is a reaction and directed edges connect it to all reactions whose propensities must be updated when it executes. When the same species appears in many reactions as a reactant but not a product, those reactions form a clique, with every reaction pointing to every other reaction. For a system of n reactions, the number of edges is $O(n^2)$. Because it is often the case in large scale biochemical systems that some species are involved in a relatively

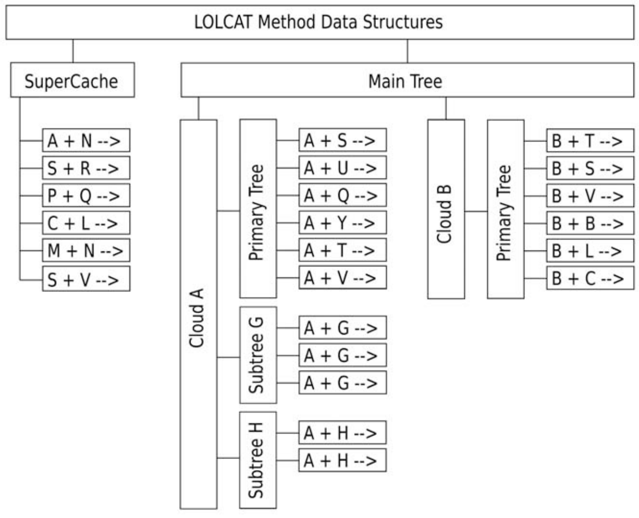

Figure 1. Example of LOLCAT Method's data structure. In practice, our implementation only creates subtrees for sets of reactions significantly larger than those shown in Cloud A's subtrees, for efficiency reasons. As a result, most clouds are like Cloud B and have no subtrees.
doi:10.1371/journal.pone.0008125.g001

large proportion of the reactions, such systems may often approach this bound.

LOLCAT Method uses a bipartite UDG instead, where each reaction points to the species whose concentrations it changes, and each species points to the reactions in which it is a reactant. Each reaction has two reactants and can affect the concentration of no more than 4 species, so the number of edges is bounded above by $6n$.

The difference between $O(n)$ and $O(n^2)$ is not noticeable for low n, but as n grows the amount of memory required to store the graph grows. A reaction-only UDG bounded by $O(n^2)$ rapidly overwhelms the cache and eventually even the main memory (RAM), making it far slower or even impossible to execute. Thus for large systems where some species participate in many reactions there is no question that we should use a bipartite UDG. To plainly demonstrate this point, we have created and benchmarked (see below) a Modified ODM (MODM) that is identical to ODM except that it uses a bipartite UDG.

Note that every time a reaction is executed, the set of update computations that must be performed are identical. An additional minor speed increase may thus be obtained by compiling this dependency graph together with the cloud structures into a cache of per-reaction and per-species update instructions rather than using a generic function that references the graph. See Supporting Information S1 on the Optimized Interpreter and Figure S1 for details.

Formal Description of the LOLCAT Method

The following algorithm is repeated until a termination criteria is satisfied, such as completing a desired number of iterations or t_{sim} reaching some desired value:

1. **Phase 1: Choose a reaction to execute**

(a) Generate a random number q uniformly in the interval $(0,a_R]$.

(b) If $q \le S_{SC}$, then q maps to the super-cache. If it does, step linearly through the super-cache, subtracting the propensity of each reaction until q is less than or equal to the propensity of next reaction. Let this reaction be r^*, and go to **Phase 2**.

(c) Otherwise, subtract S_{SC} from q and descend down the main tree. At each level, subtract the propensity of left branches from q until the propensity of a branch is greater than q: this is the branch that is selected. When this is a leaf, it contains c^*, the cloud that q maps to. Let $q' = \frac{q}{[X_\alpha]_{c^*}}$.

(d) If $q' \le S_G$ then q' maps to c^*'s primary tree. If it does, then descend down that primary tree as above, subtracting the cumulative propensity of untaken left branches from q', to find which reaction in the primary tree q' maps to. Let this reaction be r^*, and go to **Phase 2**.

(e) Otherwise, step through the list of sub-trees of c^*, subtracting the S_{H_i} of each non-selected sub-tree from q'. The sub-tree for which $q' < S_{H_i}$ is the sub-tree that q' maps to. Let $q'' = \frac{q'}{[X_{\beta,H_i}]_{c^*}}$.

(f) Descend down the sub-tree s_i as above, subtracting the cumulative propensity of untaken left branches from q'' to find the reaction that q'' maps to. Let this reaction be r^*, and go to **Phase 2**.

2. **Phase 2: Execute the chosen reaction**

(a) Update the simulation time: $t_{sim}' = t_{sim} - log(z)/a_R$, where z is a random number in $(0,1]$.

(b) Use the (compiled) update graph to adjust species concentrations and reaction propensities for an execution of r^*, as per the mathematical rules given above. For details on the efficient implementation of updates and recording, see the Supporting Information S1 on the Optimized Interpreter and Figure S1.

Note that the actual propensity of each individual reaction (and thus its probability of selection) is precisely identical to that used by the Gillespie Algorithm—LOLCAT Method only organizes and records this information differently. Because the probability of selecting any given reaction and the effect on propensities following its selection is identical for LOLCAT Method and the Gillespie Algorithm, we are guaranteed that LOLCAT Method is a correct implementation of the Gillespie Algorithm, assuming that a sufficient number of bits are used in the floating point representations of propensities.

Results

We experimentally verified the speed advantage of LOLCAT Method on a set of yeast MAPK cascade models obtained from the Yeast Pheromone Model repository [10]. Note that we are concerned only with the fact that these are complex biochemical models that a scientist would reasonably wish to simulate, not with the correctness of these particular models. Six different versions of the cascade model were used, each with a different number of reactions and species. Each model was run to steady-state for 100,000 seconds (about 28 hours) of simulation time. We then changed the pheromone concentration from 0 nM to 100 nM for each model, and benchmarked ODM, NRM, MODM and LOLCAT Method.

For the purpose of benchmarking the various algorithms, all simulations were written in ISO compliant C++ and carefully optimized. The Intel C++ Optimizing Compiler v. 10.1 was used to compile the source to machine code. Benchmarks were measured on a machine with an Intel Xeon 5355 Quad-Core 2.66 Ghz 64-bit processor with the SSSE3 instruction set, a 4 MB cache, 8 GBs of RAM, 12 GB swap space and a 250 GB hard drive. Our implementation of LOLCAT Method [4] is publicly available in MIT's DSpace archival storage at http://hdl.handle.net/1721.1/46710.

For each model, we ran 10 trials of 40 million iterations for each of the four different algorithms and recorded the mean and standard deviation of the runtimes. We computed the ratio of the runtimes, normalizing by the mean time taken by LOLCAT Method, and present the resulting speedup factors in Table 1. We do not report the preprocessing time for the various methods, as preprocessing needs to be done only once for a batch of many simulations (often thousands or more per batch) and all of the methods evaluated completed preprocessing in less than one second. As the structure of the reaction graph is critical to the behavior of simulations, we also computed the distributions of the reaction valences (the number of other reactions whose propensities change when the reaction executes) with respect to propensity in the 6 different models after they had reached steady-state and the pheromone was added, and present the cumulative distribution function for all model in Figure 2.

In every case, LOLCAT method greatly outperforms the other methods. As the size of the model increases, the advantage of LOLCAT rises by orders of magnitude. Indeed, ODM and NRM were not even able to run the largest model we benchmarked due to the size of their dependency graph, while Model D consumed more than 4 GB of RAM and Model E consumed more than 12 GB of memory. This means that ODM and NRM require a 64 bit architecture to run models D, E and F, while LOLCAT method consumed less than 4 megabytes and fit into the L3 cache, and can run on a 32 bit architecture if desired.

Table 1. Benchmarking results for ODM, NRM, MODM and LOLCAT Method.

System Parameters			**Runtimes(s)** (μ,σ)			
ID	**# Rxns.**	**# Species**	**ODM**	**NRM**	**MODM**	**LOLCAT**
A	2040	236	8.45, 0.04	25.52, 5.75	25.08, 0.12	3.06, 0.14
B	11492	5092	61.08, 0.17	162.68, 6.15	64.21, 0.22	1.68, 0.04
C	35003	11402	152.18, 0.64	374.73, 4.63	166.06, 0.46	3.73, 0.06
D	84301	15087	374.66, 2.30	720.35, 62.32	390.85, 1.41	4.73, 0.13
E	162150	14766	640.72, 3.01	964.49, 99.97	623.39, 1.58	1.88, 0.04
F	292190	15287	–	–	2527.93, 61.09	9.30, 0.09
System Parameters			**Slowdown Factor**			
ID	**# Rxns.**	**# Species**	**ODM**	**NRM**	**MODM**	**LOLCAT**
A	2040	236	2.77	10.37	8.21	1.00
B	11492	5092	36.35	93.78	38.21	1.00
C	35003	11402	40.78	98.51	44.49	1.00
D	84301	15087	79.21	159.29	82.64	1.00
E	162150	14766	340.03	447.67	330.83	1.00
F	292190	15287	–	–	271.73	1.00

Six versions of the Yeast Pheromone Model [10] were benchmarked. Slowdown factors are mean time normalized against the performance of LOLCAT Method for each model. The "–" in the last entry for ODM and NRM indicates that those simulations could not be run because the dependency graph consumed more than the RAM and swap space, roughly 20 GB, of the host machine. The varying structure of the simulated system may account for the non-uniform scaling of runtimes.

doi:10.1371/journal.pone.0008125.t001

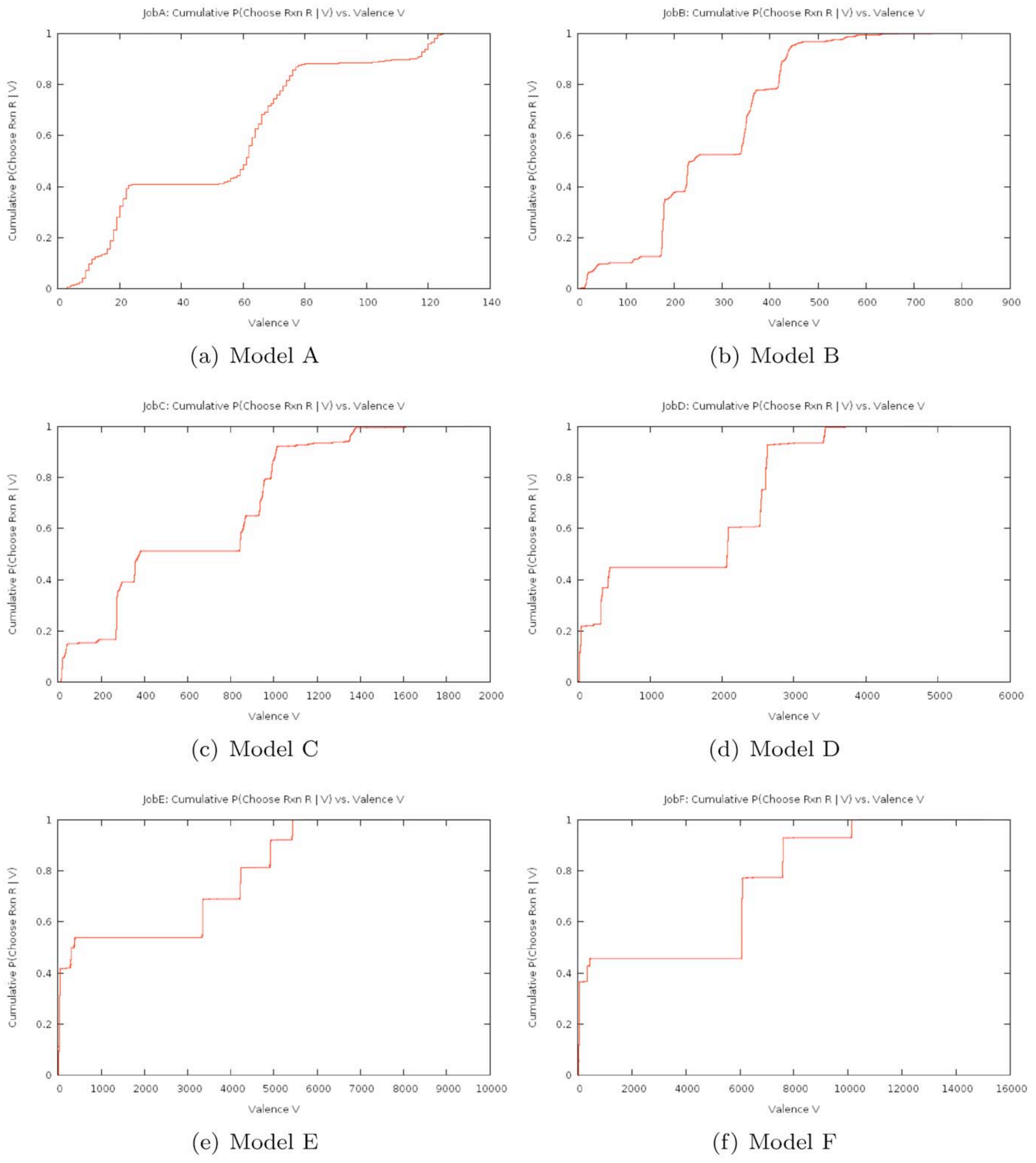

Figure 2. Cumulative distribution function (CDF) plots of reaction valence for all six models. To compute the CDF we first computed the PDF of reaction valence, weighting each reaction valence by the reaction's average propensity over a pre-computed sample trajectory. We then computed the CDF from the PDF to increase visual salience of the sparsely distributed weights. A steep climb near a particular valence means a significant probability of a randomly chosen reaction having that approximate valence. This, in turn, often indicates the presence of a super-species that is involved in many reactions and benefits greatly from factoring.
doi:10.1371/journal.pone.0008125.g002

Discussion

LOLCAT Method uses two key ideas: (1) grouping reactions with common reactants and updating the propensities of many reactions in a single operation, and (2) using a bipartite update dependency graph of species and reactions, resulting in a much more compact form. Note that the factoring of reactions allows for the dependency graph to be further compressed beyond the simple species-based dependency graph used by MODM. These two principles allow LOLCAT Method to outperform other popular methods by orders of magnitude on the chemical systems we benchmarked. Furthermore, the performance advantage of LOLCAT Method is expected to increase as the size of the systems being modeled increases. LOLCAT Method is also able to gracefully handle systems with a large number of interdependent reaction propensities, something that all previous methods are not able to do.

It is important however, to note that the speedup of LOLCAT Method cannot be measured simply as a function of the number of reactions or species or even the average reaction valence. Rather, the speedup factor seems to be best measured by studying the reaction valence CDF of the system being modeled. However, we are optimistic that many large scale biochemical systems will prove to have CDFs that mean they are amenable to orders of magnitude speedup via LOLCAT Method simulation.

Note also that the results presented above do not separate the advantage due to factoring from the advantage due to logarithmic search for a reaction to execute, but we believe the first to be dominant for large systems due to the large size of clouds generated by the greedy search in our application of LOLCAT to these models.

The authors would like to acknowledge the recent publication of another variant of the Gillespie Algorithm, SSA-CR, published in which Phase 1 of the algorithm is reduced to $O(1)$ time via a clever method based on rejection-sampling [11]. Phase 2 of this algorithm, however, still cannot scale well when the average reaction valence is high, and in the case of high reaction valence systems, Phase 2 dominates the cost of the Gillespie Algorithm. Thus, we believe that if we were to benchmark SSA-CR, LOLCAT Method would outperform it by a similar margin.

We would also like to note that there is an aggressive trend in computational biology to tackle computationally expensive problems by throwing hardware at the problem. Sometimes this approach generates interesting methods, such as the use of FPGAs [12]. Some of these gains can be realized on desktop computers simply by paying careful attention to the interaction between software and hardware (see the Supporting Information S1 for details). As the size of chemical systems to be simulated grows steadily larger, however, we argue that it is more important to reduce algorithmic complexity by searching for exploitable hidden structure.

If LOLCAT Method is able to take advantage of grouping reactions with common reactants as well as we believe it can, then LOLCAT Method may have a significant impact on what kinds of systems researchers are able to simulate. We hope the performance increase will help to transform computational biology into a more streamlined, interactive exercise.

Supporting Information

Figure S1

Found at: doi:10.1371/journal.pone.0008125.s001 (0.95 MB EPS)

Supporting Information S1

Found at: doi:10.1371/journal.pone.0008125.s002 (0.31 MB PDF)

Acknowledgments

The authors would like to thank Thomas F. Knight Jr., Gerald J. Sussman, Roger Brent, Steve Andrews and Mark M. Tobenkin for discussion, Christopher J. Terman for providing computational resources, and Ty M. Thomson for giving us the Yeast Pheromone Model.

Author Contributions

Conceived and designed the experiments: SI. Performed the experiments: SI. Analyzed the data: SI JB. Contributed reagents/materials/analysis tools: SI JB. Wrote the paper: SI JB.

References

1. Gillespie D (1977) Exact stochastic simulation of coupled chemical reactions. Journal of Physical Chemistry 81: 2309–2586.
2. Cao Y, Li H, Petzold L (2004) Efficient formulation of the stochastic simulation algorithm for chemically reacting systems. Journal of Physical Chemistry 121: 4059–4067.
3. Gibson M, Bruck J (2000) Efficient exact stochastic simulation of chemical systems with many species and many channels. Journal of Physical Chemistry A 104: 1876–1889.
4. Indurkhya S, Beal J (2009) Code for lolcat method. MIT Dspace: http://hdl.handle.net/1721.1/46710.
5. Gillespie D, Petzold L (2003) Improved leap-size selection for accelerated stochastic simulation. Journal of Chemical Physics 119: 8229–8234.
6. Sali H, Kaznessis Y (2005) Accurate hybrid stochastic simulation of a system of coupled chemical or biochemical reactions. Journal of Chemical Physics 122: 054103–054115.
7. Rao C, Arkin A (2003) Stochastic chemical kinetics and the quasi-steady-state assumption: Application to the gillespie algorithm. Journal of Chemical Physics 118: 4999–5010.
8. Mccollum JM, Peterson GD, Cox CD, Simpson ML, Samatova NF (2006) The sorting direct method for stochastic simulation of biochemical systems with varying reaction execution behavior. Computational Biology and Chemistry 30: 39–49.
9. Li H, Petzold L (2006) Logarithmic direct method for discrete stochastic simulation of chemically reacting systems. Technical Report http://www.engr.ucsb.edu/~cse, Department of Computer Science, University of California, Santa Barbara.
10. Thomson T (2008) Models and Analysis of Yeast Mating Response: Tools for Model Building, From Documentation to Time-Dependent Stimulation. Ph.D. thesis, Massachusetts Institute of Technology.
11. Slepoy A, Thompson A, Plimpton S (2008) A constant-time kinetic monte carlo algorithm for simulation of large biochemical reaction networks. Journal of Chemical Physics 128: 205101–205109.
12. Salwinski L, Eisenberg D (2004) In silico simulation of biological network dynamics. Nature Biotechnology 22: 1017–1019.