On the fixation probability of superstars
Details of Moran Process simulator

Josep Díaz* Leslie Ann Goldberg†‡ George B. Mertzios§
David Richerby†‡ Maria Serna* Paul G. Spirakis†¶

In this document, we describe the operation of our simulator for the Moran process on superstars and prove that it simulates the process correctly. A brief introduction to the salient features of the C programming language appears in Section 1. Section 2 gives some mathematical background and a high-level overview of the code. Sections 3 and 4 describe the code in more detail and Section 5 proves that it is correct. The C source code appears in Section 6.

To describe the operation of the code, we say that the *successors* of a vertex v in a graph are all vertices w for which there is an edge (v, w). v's *predecessors* are those vertices u for which there is an edge (u, v).

1 The C programming language

The simulator is written in C. We briefly describe the features of the language so that a reader who is familiar with programming in other languages should be able to understand the code.

Operators. `=` is assignment and `==` tests equality; `!=` tests disequality. For positive integers, division rounds down and `%` is the remainder operator. `x++` and `x--` increase and decrease `x` by one, respectively, known as increment and decrement. `x += y` is equivalent to `x = x + y`; similarly `x -= y`. Where a Boolean value is required, zero means false and any nonzero value means true. `||` is Boolean or; `&&` is Boolean and.

*Departament de Llenguatges i Sistemes Informátics, Universitat Politécnica de Catalunya, Spain. Email: {diaz, mjserna}@lsi.upc.edu.

†Department of Computer Science, University of Liverpool, UK. Email: {L.A.Goldberg, David.Richerby}@liverpool.ac.uk. Supported by EPSRC grant EP/I011528/1 *Computational Counting*.

‡Department of Computer Science, University of Oxford, UK.

§School of Engineering and Computing Sciences, Durham University, UK. Email: george.mertzios@durham.ac.uk.

¶Department of Computer Engineering and Informatics, University of Patras, Greece. Email: spirakis@cti.gr. Partially supported by the EU FET IP Project MULTIPLEX Contract No. 317532.

Arrays. The declaration `int x[10]` declares `x` to be an array of 10 integers. Arrays in C are indexed from zero. The function `memset (x, 0, n*sizeof (int))` (ll. 301–4) zeroes the first `n` elements of the integer array `x`.

Pseudorandom numbers are normally generated by the library function `random()`, with the generator seeded by `srandom()`. `random()%n` gives a result from $\{0, \ldots, n-1\}$ that may be treated as having been drawn uniformly at random (u.a.r.). We have provided an implementation of the pseudorandom number generator used by the GNU C library, to ensure that the program gives identical results, regardless of which compiler is used. Our implementation renames the functions `localrandom()` and `localsrandom()`.

Preprocessor macros. `#define A B` essentially declares `A` to be a constant with value `B`. Before compiling the code, the compiler will substitute `B` for each instance of `A`. `#define A(x) B` (crucially, with no space before `(x)`) is, as it is used in this code, essentially equivalent to defining a function `A` that takes argument `x` and has body `B`, except that any call `A(y)` is implemented by textual substitution rather than standard function call techniques. Traditionally, macro names are written in upper case to distinguish them from ordinary variables and functions. The declaration `#include` is used to include library routines.

2 Overview

The code is designed to simulate the Moran process on large superstars (tens of thousands of vertices) as efficiently as possible. A significant speed-up is obtained by applying the following observation, which also appears in [1].

If the vertex chosen to reproduce at time t is a mutant that has no non-mutant successors (or a non-mutant with no mutant successors), then we must have $X_{t+1} = X_t$. This will happen the majority of the time when either most vertices are mutants or most are non-mutants and very frequently throughout the process. To speed up the simulation, we would like to only choose the vertex to reproduce from among those that have a successor in the opposite state, since the state cannot change if any other vertex reproduces. We describe a vertex as *truly active* if it has a successor the opposite state.

The centre will be truly active for the overwhelming majority of the time as it nearly always has both mutants and non-mutants among its successors. Because of this and because it is somewhat expensive to determine whether the centre is truly active, we did not limit the simulator to reproduce from truly active vertices. Instead, we describe a vertex as *active* if it is the centre or is truly active (or both). We say that all other vertices are *dormant*. We

refer to the variant of the process where only active vertices are selected for reproduction as the *accelerated process*.

As we will see below, the accelerated process has the same fixation probability as the generalized Moran process. Only selecting active vertices to reproduce is particularly effective in superstars because every vertex apart from the centre has a unique successor. This makes it very easy to determine which of these vertices are active and it also guarantees that the state will change if an active vertex, other than the centre, is selected. This makes simulating the accelerated process significantly faster than the ordinary process.

Proposition 1. *The accelerated process has the same fixation probability on any graph as the generalized Moran process.*

Proof. We may consider the states of the generalized and accelerated Moran processes to be triples (X_i, u_i, v_i), where X_i is the set of mutants and u_i and v_i are, respectively, the vertices chosen for reproduction and replacement in the transition from X_{i-1} to X_i. Fixation is now the event of reaching any state (V, u, v) and extinction is reaching any $(\emptyset, u, v)$.

Let $(X_i, u_i, v_i)_{i\geq 0}$ be a generalized Moran process on G and write $A(X_i)$ for the set of active vertices in state X_i. Define the sequence

$$\begin{aligned} t_0 &= 0 \\ t_{j+1} &= \inf\{i > t_j \mid u_i \in A(X_i) \text{ or } X_i = \emptyset \text{ or } X_i = V\}. \end{aligned}$$

The t_j are stopping times of the chain and are almost surely finite. It follows that the sequence $(X_{t_j}, u_{t_j}, v_{t_j})_{j\geq 0}$ is a Markov chain, which corresponds exactly to the accelerated process. Since it is a subsequence of the generalized Moran process, it must have the same fixation probability. □

Thus, each simulation of the process proceeds as follows. First, we choose a vertex v uniformly at random to be the initial mutant. We set v, all its predecessors and the centre to be active. Then, we repeat the following steps until either every vertex is a mutant or none is:

- select an active vertex v to reproduce;
- select a successor to be replaced by v's offspring;
- update the state of the successor;
- update the set of active vertices.

Note that whether a vertex is active or dormant depends only on whether it and its successors are mutants or non-mutants, not on whether any neighbours are active or dormant. Therefore, if an active vertex becomes dormant,

or vice-versa, there is no "cascade" of updates to the active/dormant status of its neighbours and on to their neighbours and so on.

The simulator maintains data structures to allow the efficient selection of vertices and so on.

3 Implementation notes

For performance, the simulator was written in C. The simulation parameters (principally k, ℓ, m and the mutant fitness r) are compile-time constants, which allows the compiler to precompute expressions involving them and avoids memory accesses to retrieve their values. The simulator uses only integer arithmetic, again for performance. To allow rational values for the mutant fitness, we give the mutants fitness r and the non-mutants fitness q; this is clearly equivalent to having mutants with fitness r/q and non-mutants with fitness 1.

The simulator is single-threaded. Multi-threading is not advantageous as it adds communication overhead and the program can be trivially parallelized by running multiple copies on different machines and combining the results.

To make the results verifiable, the simulator outputs the value used to seed the pseudorandom number generator. When executed, a value for the seed can be specified as a command-line argument; if no value is given, the program uses the system time, as is standard practice. This, coupled with the provision of a pseudorandom number generator in the code (ll. 90–121) means that a simulation run can be exactly reproduced.

4 Data structures

4.1 Vertex state

Each vertex in the graph has one of four states: it is either a mutant or non-mutant and is either active or dormant. The symbolic constants `MUTANT`, `NONMUT`, `ACTIVE` and `DORMANT` are defined for these states, along with `ACT_MUT`, `ACT_NON`, `DOR_MUT` and `DOR_NON` for the obvious combinations.

The number of mutants in the reservoir of leaf $0, \ldots, L-1$ of the superstar is stored in the array `resmut` and the total number of mutants in all the reservoirs in the integer `resmuts`. In each reservoir, either the mutants or the non-mutants will be active, according to the state of the first chain vertex in the same superstar leaf. The array `resstate` stores the active vertex type for each superstar leaf, which is either `MUTANT` or `NONMUT`. The mutants (respectively, non-mutants) in a particular superstar leaf are not distinguishable so it suffices to store only the number of mutants.

The state of the centre vertex and the chain vertices is stored vertex-by-vertex in the array `state`. The chain vertex $c_{i,j}$ (i.e., the jth chain vertex of the ith superstar leaf) is at position $i + jL$.

4.2 Vertex selection

The program maintains two data structures to allow it efficiently to select the vertex for reproduction. The basic problem is this. We have a sequence of active vertices $v_1, \ldots, v_s$ of which m are mutants. Hence, the total fitness is $W = mr + (s - m)q$ and we wish to select one of the vertices such that each mutant is chosen with probability r/W and each non-mutant with probability q/W. But we are only able to generate a random number `rnd` uniformly from the set $\{0, \ldots, W - 1\}$.

Chain and centre vertices. For the chain and centre vertices, we maintain separate lists of the active mutants and non-mutants in the two-dimensional array `vlist`. The first co-ordinate is the state (`ACT_MUT` or `ACT_NON`) and the second is the index into the list. Now, if $\texttt{rnd} < mr$, we select the $\lfloor \texttt{rnd}/r \rfloor$th mutant; otherwise, we select the $\lfloor (\texttt{rnd} - mr)/q \rfloor$th non-mutant.

We also store the number of vertices in each of the four states (including dormant vertices) in the array `numinstate` and, to allow efficient updating of the lists, we maintain an inverse mapping in the array `posn`. If the vertex v is active, then $\texttt{posn}[v]$ is the index of v in $\texttt{vlist}[\texttt{ACT\_MUT}]$ or $\texttt{vlist}[\texttt{ACT\_NON}]$, according to its state. Thus, for active vertices, we maintain the identity that

$$\texttt{vlist}[\texttt{state}[v]][\texttt{posn}[v]] = v\,.$$

If vertex v is dormant, the value of $\texttt{posn}[v]$ is meaningless.

Reservoir vertices. The above data structure is inadequate for reservoir vertices. Even though the simulator just tracks the number of mutants and non-mutants in each reservoir, we could, in principle, maintain a `vlist`-style list of reservoir vertices to aid selection. However, whenever an active reservoir vertex is chosen to reproduce, it changes the state of the first chain vertex, which means that each of the M vertices in the reservoir changes from active to dormant or vice-versa. Thus, maintaining the lists would be very expensive.

Instead, we use a binary search tree. The nodes in the search tree correspond to leaves of the superstar. The total fitness of the active nodes in each leaf's reservoir is stored in the array `resactfit`. We consider this to be structured as a tree, with node i having its left child at location $2i + 1$ and its right child at $2i + 2$. To enable rapid searching in this tree, we store in the array `restree` the total fitness of active reservoir vertices for all superstar leaves in the left subtree rooted at each node. The macros `LCHILD`,

RCHILD and PARENT compute the left and right children and the parent of a given node, respectively. The macro ISLCHILD computes whether a given node is its parent's left child (i.e., whether its index is odd). No attempt is made to balance the tree since the symmetry of the superstar should keep it reasonably balanced in practice.

The total fitness of all active reservoir vertices (i.e., the sum of the entries in resactfit) is stored in totresactfit; the total fitness of all active non-reservoir vertices is stored in totnonresactfit.

5 Correctness

To demonstrate correctness, we show that the variables state, numinstate, vlist, posn, resmut, resmuts, resstate, resactfit, totresactfit, totnonresactfit and restree are correctly initialized and maintained as a consistent description of the current simulation state, and that transitions between states occur with the correct probabilities.

5.1 Macros

It is immediate that the macros N and NONRES correctly define the number of vertices and number of non-reservoir vertices in terms of K, L and M. CENTRE denotes the position of the centre vertex in VLIST, immediately after the chain vertices, which are in positions 0 to $L(K-2)-1$.

ISMUTANT determines whether a vertex in vlist is a mutant: this is the case if, and only if, its state is even, i.e., ACT_MUT or DOR_MUT.

It is clear from the discussion in Section 4.2 that the macros LCHILD, RCHILD, PARENT and ISLCHILD behave correctly. The macro FIRSTSTLEAF evaluates to the first leaf node of the search tree. There are L nodes, numbered 0 to $L-1$ so node $\lfloor L/2 \rfloor$ is the first one with no child.

5.2 State updates

The code contains a number of functions to update the variables describing the state. We show that each one maps consistent states to consistent states.

Lemma 2. setvstate() *(ll. 138–172) correctly changes the state of a chain or centre vertex, maintaining the lists of active chain and centre vertices and their total fitness.*

Note that changing the mutant/non-mutant state of a vertex may require changes to the active/dormant state of its neighbours. These changes are the responsibility of the caller.

Proof. We wish to move the chain or centre vertex with index v into state s from its old state s'. The necessary steps are as follows. We must set v's

state, increase the number of vertices in state s and decrease the number in s'. If the old state was active, we must delete v from the relevant list of active vertices and update `totnonresactfit`; if the new state is active, we must add v to the relevant list and, again, update `totnonresactfit`.

To delete a vertex from the list, we overwrite it with the last element of the list, update that item's position and decrease the length of the list (ll. 145–56). Because C arrays are indexed from zero, we actually decrease the length first as it requires fewer instructions to decrease `len` by one and refer to `array[len]` than it does to refer to `array[len-1]` and then decrement `len`. To add a vertex to a list, we write it one place beyond the current last element, store its new position in `posn` and increase the length by one (ll. 162–71). □

In the following lemma, `delta` may be negative, in which case the function decreases the specified fitness.

Lemma 3. `incresfit (leaf, delta)` *(ll. 187–97) correctly increases the total fitness of active vertices in a superstar leaf* `leaf` *by* `delta` *and correctly updates the search tree.*

Proof. The function increases the recorded fitness of reservoir vertices in the leaf (l. 195) and the total over all leaves (l. 196). Starting at the leaf's node in the search tree, it performs the following at each node on the path back to the root (excluding the root itself): if the current node is its parent's left child, the total fitness of active vertices in the parent's left subtree is increased by `delta`. □

Suppose the first vertex of a superstar leaf's chain is a mutant. It follows that the mutants in the leaf's reservoir are dormant and the non-mutants are active. If one of these non-mutants is selected for reproduction, the chain vertex becomes a non-mutant and now we must make the reservoir's mutants active and its non-mutants dormant. The converse changes must be made in the case that the chain vertex was originally a non-mutant. We refer to this as *flipping* the active vertices in the reservoir.

Lemma 4. `flipresstate (leaf)` *(ll. 205–16) correctly flips the active vertices in the reservoir of the leaf* `leaf` *and updates the recorded total fitness of active reservoir vertices.*

Proof. Suppose there are m mutants in the leaf and that mutants are active. The current active fitness is, therefore, mr and, after the flip, the active fitness will be $(M - m)q$. The change is $(M - m)q - mr = Mq - (r + q)m$. If non-mutants are currently active, the change is $(r + q)m - Mq$.

The function flips the active vertices of the given leaf (l. 213) using the identity `ACT_NON` $= 1 -$ `ACT_MUT`, computes the amount by which the fitness changes (ll. 201–11) and uses `incresfit` to effect this change (l. 211). □

5.3 Global initialization

The random number generator is seeded, either using the system time or a value supplied on the command line and the number of simulation runs that have reached fixation and extinction are set to zero.

5.4 Per-run initialization

The initial state of the Moran process is that all vertices are non-mutants, except for one mutant placed uniformly at random. The centre is active by our convention; the mutant is active (all its successors are non-mutants) and so is every predecessor of the mutant. Every other vertex is dormant.

We achieve this in two steps. First (ll. 289–308), we set the centre vertex to be an active non-mutant and every other vertex to be a dormant non-mutant. Then (ll. 313–353), we choose a vertex uniformly at random, make it a mutant, make it active and make all its predecessors active.

Lemma 5. *Lines 289–308 correctly set the variables to the state where no vertex is a mutant and every vertex but the centre is dormant.*

Proof. The state of the centre is set to `ACT_NON` (l. 299) and the state of all chain vertices to `DOR_NON` (ll. 294–5). Thus, there is one active non-mutant and every other non-reservoir vertex is a dormant non-mutant (ll. 289–92); the list of active non-mutants is just the centre (l. 297), at the head of the list (l. 298).

No leaf has any mutants in its reservoir (l. 301, l. 306), there are no active reservoir vertices, so no active fitness there (l. 302, l. 307) and the total active fitness in any branch of the search tree is zero (l. 304). Since all chain vertices are non-mutants, the mutants would be active in each leaf, if there were any (l. 303). The total fitness of active vertices is q (l. 308), since the only active vertex is the non-mutant centre. □

Lemma 6. *Lines 313–352 choose an initial mutant uniformly at random and correctly set the variables for this state.*

Proof. A value `rnd` is chosen u.a.r. from $\{0, \ldots, N-1\}$ (l. 313). If $\texttt{rnd} < L(K-2)$, a chain vertex is chosen (l. 334); if $\texttt{rnd} = L(K-2)$ (l. 347), the centre is chosen; otherwise, a reservoir vertex is chosen (l. 322). Hence the initial mutant is chosen with the correct probability. We now show that the state is modified correctly.

Reservoir vertex. By symmetry, we may place the initial vertex in leaf 0. Thus, this reservoir contains one mutant (l. 324), there is one reservoir mutant in total (l. 325) and the fitness of active vertices in leaf 0's reservoir has increased by r. (Recall that mutants were initially defined to be active in each reservoir, so no further changes are needed.)

Chain vertex. The chosen vertex, which we may take to be the one indexed by `rnd` is set to be an active mutant (l. 336). Recall that the jth chain vertex in the ith leaf has index $i + jL$. Thus, if `rnd` $< L$, the initial mutant is the first vertex in its leaf's chain so non-mutants are now active in that leaf's reservoir (ll. 338–9); otherwise, the previous vertex in the chain becomes active (ll. 340–1).

Centre vertex. The centre is set to be an active mutant (l. 349) and the last vertex of every chain also becomes active (ll. 350–1). □

5.5 Inner simulation loop

It is clear that the inner simulation loop (ll. 358–501) runs until either the total number of mutants (mutants in the reservoirs and active and dormant mutants in the chains and the centre) is either zero or N, the total number of vertices. If there are no mutants left, the run has gone extinct (ll. 506–7); otherwise, it has reached fixation (ll. 508–9). Every ten trials and after all trials have been performed, the number of trials run so far, the numbers of fixations and extinctions seen and the fixation probability are printed (ll. 514-26).[1]

Let W_{res}, W_{chain} and W_{cent} be the total fitness of active vertices in the reservoirs, chains and the centre, respectively, and let $W = W_{\text{res}} + W_{\text{chain}} + W_{\text{cent}}$.

Lemma 7. *The inner simulation loop selects a reservoir (respectively, chain or centre) vertex for reproduction with probability W_{res}/W (respectively, W_{chain}/W or W_{cent}/W)*

Proof. An integer `rnd` is chosen u.a.r. from $\{0, \ldots, W-1\}$ (l. 364). If `rnd` $< W_{\text{res}}$, a reservoir vertex is chosen (l. 369).

Otherwise, a vertex is chosen from `vlist`, i.e., from the active chain vertices and the centre. Each mutant within this set is chosen with probability

$$\frac{r}{W - W_{\text{res}}} \cdot \frac{W - W_{\text{res}}}{W} = \frac{r}{W},$$

and each non-mutant, similarly, with probability q/W (ll. 415–20). Since $W_{\text{cent}} = r$ if the centre is a mutant and q if it is a non-mutant, the centre is chosen with probability W_{cent}/W, so a chain vertex is chosen with probability $(W - W_{\text{res}} - W_{\text{cent}})/W = W_{\text{chain}}/W$. □

Lemma 8. *If a reservoir vertex is chosen, the state is correctly updated.*

[1] The cast to double at line 521 is to force floating-point arithmetic. Without it, the probability would be rounded down to the next integer, which would be zero unless every trial had reached fixation.

Proof. The active reservoir vertices within a superstar leaf are indistinguishable since they all receive an edge from the centre and send an edge to the leaf's first chain vertex, and are either all mutants or all non-mutants. Therefore, given that a reservoir vertex has been chosen, choosing an active vertex with probability proportional to its fitness is equivalent to choosing a leaf of the superstar with probability proportional to the total weight of its active reservoir vertices.

Having chosen the leaf, we must update the state of the leaf's first chain vertex. Conveniently, its index in `state` is the same as the leaf's index in the arrays holding reservoir state. Suppose the active vertices in the reservoir are mutants. Then the first chain vertex will also become an active mutant (l. 392), unless the second chain vertex is also a mutant (ll. 400–3) or $K = 3$ and the centre is a mutant (ll. 394–8). In these two cases, the first chain will become a dormant mutant. Finally, because the first chain vertex is now a mutant, the non-mutants in the reservoir become active (l. 411). The case where the active reservoir vertices were non-mutants is analogous.

It remains to check that we select the leaf with the right probability. For $\ell \in \{0, \ldots, L\}$, let S_ℓ be the set of all nodes in the subtree of the search tree rooted at node ℓ. Let $W_\ell =$ `resactfit[`ℓ`]` be the total fitness of active vertices in the reservoir of leaf ℓ and let $T_\ell = \sum_{i \in S_\ell} W_i$ be the total fitness of all active reservoir vertices in the subtree rooted at node ℓ. Note that $T_0 =$ `totresactfit` is the total fitness of all active reservoir vertices. We claim that, if `rnd` is chosen u.a.r. from $\{0, \ldots, T_\ell - 1\}$, then the procedure at lines 371–88 selects the jth leaf with probability W_j/T_ℓ for each j in the subtree rooted at ℓ.

Proof of the claim is by induction on the initial value of `leaf`, which we call ℓ. If $\ell \geq$ `FIRSTSTLEAF` $= \lfloor L/2 \rfloor$ then $T_\ell = W_\ell$. The loop test fails at the first iteration so, with probability $W_\ell/T_\ell = 1$, we choose leaf ℓ, as required.

Now, suppose that $\ell <$ `FIRSTSTLEAF`. Let $x = 2\ell + 1$ and $y = 2\ell + 2$ be the left and right children of ℓ. There are three cases.

- With probability T_x/T_ℓ, we have $0 \leq$ `rnd` $< T_x =$ `restree[`ℓ`]` (l. 378). We recurse on the left subtree (l. 379) and, by the inductive hypothesis, any leaf $j \in S_x$ is then chosen with probability W_j/T_x.

- With probability W_ℓ/T_ℓ, we have $T_x \leq$ `rnd` $< T_x + W_\ell$ (ll. 382–3). We break out of the loop (l. 384) and choose ℓ.

- With probability T_y/T_ℓ, we have $T_x + W_\ell \leq$ `rnd` $< T_x + W_\ell + T_y = T_\ell$. (l. 378). We set `rnd` to `rnd` $- T_x - W_\ell$ (l. 385) and recurse on the left subtree (l. 386). By the inductive hypothesis, any leaf $j \in S_y$ is then chosen with probability W_j/T_y.

By the first bullet, any leaf $j \in S_x$ is chosen with probability $(T_x/T_\ell)(W_j/T_x) = W_j/T_\ell$ and, similarly, by the third bullet, any leaf $j \in S_y$ is chosen with

probability W_j/T_ℓ. By the second bullet, ℓ is chosen with probability W_ℓ/T_ℓ. Since $S_\ell = S_x \cup \{\ell\} \cup S_y$, the claim is proven.

Since we initially set `leaf=0` (l. 371), it follows that we choose a leaf ℓ from the whole search tree (S_0) such that the jth leaf is chosen with probability proportional to W_j, as required. □

Lemma 9. *If a chain vertex is chosen, the state is correctly updated.*

Proof. The chosen vertex v has index `src` and, since the jth vertex of the ith leaf has index $i + jL$, the chosen vertex has position $\lfloor \texttt{src}/L \rfloor$ in its chain (l. 427), where position 0 is adjacent to the reservoir and position $K - 3$ is adjacent to the centre.

If v is adjacent to the centre (ll. 432–42), we must change the state of the centre to be that of v, which was necessarily active (l. 434). Since the centre vertex has changed its mutant/non-mutant state, the last vertex of every leaf's chain must change its active/dormant state. These vertices have index $L(K-3), \ldots, L(K-2) - 1$ (ll. 440–1).

If the chosen vertex v is not the centre (ll. 443–63) then its successor w is also a chain vertex and is at position $\texttt{src} + L$ (l. 448). w will receive the same mutant/non-mutant status as v but we must determine whether w will be active or dormant. Specifically, w will be active unless it has the same mutant/non-mutant status as its own successor, which may be the centre or the next vertex in the chain, depending on w's position (ll. 451–60).[2]

Finally, since v now has the same mutant/non-mutant status as its successor, it must be made dormant (l. 461) and w must receive its new status (l. 463). □

Lemma 10. *If the centre vertex is chosen, the state is correctly updated.*

Proof. If the centre is chosen to reproduce, we must choose a reservoir vertex uniformly at random to receive the offspring. This vertex will become either a mutant or non-mutant, according to the state of the centre and we must update appropriately the total weight of active vertices in the relevant leaf and in the search tree. This is implemented in lines 472–99. Note that there is no change to which types of vertex are active: the centre is defined to be active always and whether mutants or non-mutants are active in a given leaf's reservoir depends only on whether that leaf's first chain vertex is a non-mutant or mutant, which does not change.

Since all reservoirs have the same size, choosing a reservoir vertex u.a.r. is equivalent to choosing a leaf u.a.r. (l. 475) and then choosing a vertex in that leaf's reservoir u.a.r. (l. 477). Suppose there are m mutants in the

[2]The test `K > 3 && chainpos == K-4` appears redundant since `chainpos` is necessarily non-negative so can only take the value $K-4$ if $K > 3$. We include the test $K > 3$ because this can be determined at compile-time and, if $K = 3$, the subsequent test on `chainpos` will be omitted as the conjunction must be false.

chosen leaf. There is no fixed ordering of the vertices in the reservoir so we may assume that the mutants are indexed $0, \ldots, m-1$ and the non-mutants are indexed $m, \ldots, M-1$.

If the centre is a mutant and the chosen reservoir vertex is a non-mutant (l. 479), the number of mutants in the reservoir and in total increases by one (ll. 481–2). If mutants are active in that leaf, the total fitness of active vertices will increase by r (there is one more mutant; ll. 483–4); otherwise, non-mutants are active and the total fitness decreases by q (there is one fewer non-mutant; ll. 485–6). The case where the centre is a non-mutant and the reservoir vertex is a mutant is analogous (ll. 488-96). If the centre and reservoir vertex are both mutants or both non-mutants, nothing happens. Finally, if there is a change to be made to the reservoir's fitness and, hence, to the search tree, this is done at lines 497–8. □

5.6 Correctness

Theorem 11. *The program correctly simulates the generalized Moran process on superstars.*

Proof. The program simulates the accelerated Moran process which, by Proposition 1 has the same fixation probability as the generalized Moran process.

By Lemmas 5 and 6, each run is correctly initialized with one mutant placed uniformly at random. At each stage of the simulation, the vertex that will reproduce is chosen from either among the reservoir vertices, the chain vertices or the centre vertex with probability proportional to its weight (Lemma 7) and the state updated accordingly (Lemmas 8, 9 and 10, respectively). Once each run reaches fixation or extinction, the statistics are updated as appropriate.

It is clear that the program performs `TRIALS` runs of the process and correctly computes the fixation and extinction probabilities as the proportion of results that reach those states. □

6 Simulator source code

The following is the C source code of the simulator program. The line numbers are included for ease of reference and do not form part of the code. Two very long lines (248 and 263) have been wrapped in this presentation; the continuation lines have not been numbered. The code is also available as an ancillary file.

```
1 /*
2  *  superstar.c -- David Richerby, 2011-12.
3  *  Superstar simulator that keeps track of which vertices are in a
4  *  different state than their out-neighbours so can cause a state
```

```
 5  *  change if selected for reproduction.
 6  *
 7  *  WARNING!  This code will not compile as-is.  For efficiency, the
 8  *  simulation parameters are set as compile-time constants.  It is
 9  *  recommended that the program be compiled with the runsuperstar
10  *  script.  If you are not using that script, replace the values
11  *  __RVAL, __QVAL, __KVAL, __LVAL, __MVAL and __TVAL with the
12  *  appropriate numerical values and compile with as many
13  *  optimizations enabled as possible.  (With gcc, use -O3 .)
14  *
15  *
16  *  $Id: superstar.c,v 1.13 2012/01/18 18:56:20 davidr Exp $
17  */
18
19 #include <stdlib.h>
20 #include <stdio.h>
21 #include <errno.h>
22 #include <string.h>
23 #include <time.h>
24 #include <unistd.h>
25
26 /*
27  *  Simulation parameters.
28  */
29 #define r __RVAL
30 #define q __QVAL
31 #define K __KVAL
32 #define L __LVAL
33 #define M __MVAL
34 #define TRIALS __TVAL
35
36
37 /*
38  *  Useful constants.
39  */
40 #define N  ((L)*((M)+((K)-2))+1)  /* Total number of vertices.        */
41 #define NONRES  ((L)*((K)-2)+1)   /* Number of non-reservoir vertices.*/
42 #define CENTRE  ((L)*((K)-2))     /* Index of centre vertex in vlist. */
43
44 #define FIRSTSTLEAF ((L)/2)   /* First leaf vertex of the reservoir   */
45                               /*     fitness search tree.             */
46
47 /*
48  *  Vertex states
49  */
50 #define MUTANT     0
51 #define NONMUT     1
52 #define ACTIVE     0
53 #define DORMANT    2
```

```
54 #define ACT_MUT    0  /* Active mutant.                                  */
55 #define ACT_NON    1  /* Active non-mutant.                              */
56 #define DOR_MUT    2  /* Dormant mutant.                                 */
57 #define DOR_NON    3  /* Dormant non-mutant.                             */
58 #define NUMSTATES  4
59
60
61 /*********************************************************************/
62 /* Globals                                                           */
63 /*********************************************************************/
64
65 int state[NONRES];          /* State of each vertex.                   */
66 int numinstate[NUMSTATES];  /* Number of vertices in each state.       */
67 int vlist[2][NONRES];       /* List of vertices in each state.         */
68 int posn[NONRES];  /* Position of each vertex in the one list it's in.*/
69 int totresactfit;  /* Total fitness of active vertices in reservoirs. */
70 int totnonresactfit;        /* Total fitness of other active vertices.*/
71 int resmut[L];     /* Number of mutants in each reservoir.            */
72 int resmuts;       /* Total number of mutants in reservoirs.          */
73 int resstate[L];   /* Are mutants or non-mutants active in each res.? */
74 int resactfit[L];  /* Fitness of active vertices in each leaf.        */
75 int restree[L];    /* Search tree for reservoir fitness.              */
76
77
78 /*********************************************************************/
79 /* Random number generation                                          */
80 /*********************************************************************/
81
82 /*
83  *  A re-implementation of the algorithm used by glibc 2.12.2 (and,
84  *  presumably other versions), based on Peter Selinger's description
85  *  at http://www.mscs.dal.ca/~selinger/random/
86  *  Gives identical results to glibc's random() but included here for
87  *  reproducability of results on other systems.
88  */
89
90 #define POOLSIZE 34
91 unsigned int rndpool[POOLSIZE];
92 int rndidx = 0;
93
94 static unsigned int localrandom ()
95 {
96     rndidx = (rndidx + 1) % POOLSIZE;
97     rndpool[rndidx%POOLSIZE] = rndpool[(rndidx+31)%POOLSIZE]
98                                + rndpool[(rndidx+3)%POOLSIZE];
99     return ((unsigned)rndpool[rndidx%POOLSIZE]) >> 1;
100 }
101
102 static void localsrandom (unsigned int seed)
```

```
103 {
104     long long x;
105     int i;
106 
107     rndpool[0] = seed;
108     for (i = 1; i < 31; i++)
109     {
110         rndpool[i] = (16807LL * rndpool[i-1]) % 2147483647;
111         if (rndpool[i] < 0)
112             rndpool[i] += 2147483647;
113     }
114     rndpool[31] = rndpool[0];
115     rndpool[32] = rndpool[1];
116     rndpool[33] = rndpool[2];
117 
118     rndidx = -1;
119     for (i = 34; i < 344; i++)
120         (void)localrandom();
121 }
122 
123 /*********************************************************************/
124 /* Vertex state manipulation                                         */
125 /*********************************************************************/
126 
127 /*
128  *  True if the chain/centre vertex with index v is a mutant (active or
129  *  dormant).
130 */
131 #define ISMUTANT(v) ((state[(v)] % 2) == 0)
132 
133 /*
134  *  void setvstate (int v, int s)
135  *  Change the state of vertex v to s, adjusting total fitness of
136  *  active vertices as appropriate.
137  */
138 void setvstate (int v, int s)
139 {
140     int olds = state[v];
141 
142     /*
143      *  Remove from old state.
144      */
145     numinstate[olds]--;
146     if (olds <= ACT_NON)
147     {
148         int movedvtx = vlist[olds][numinstate[olds]];
149 
150         vlist[olds][posn[v]] = movedvtx;
151         posn[movedvtx] = posn[v];
```

```
152         if (olds == ACT_NON)
153             totnonresactfit -= q;
154         else
155             totnonresactfit -= r;
156     }
157 
158     /*
159      *  Place in new state.
160      */
161     state[v] = s;
162     if (s <= ACT_NON)
163     {
164         vlist[s][numinstate[s]] = v;
165         posn[v] = numinstate[s];
166         if (s == ACT_NON)
167             totnonresactfit += q;
168         else
169             totnonresactfit += r;
170     }
171     numinstate[s]++;
172 }
173 
174 /*
175  *  Macros for manipulating reservoir fitness search tree.
176  */
177 #define LCHILD(x) (2*(x)+1)       /* Left child of node x.           */
178 #define RCHILD(x) (2*(x)+2)       /* Right child of node x.          */
179 #define PARENT(x) (((x)-1)/2)     /* Parent of node x.               */
180 #define ISLCHILD(x) ((x)%2 == 1)  /* True iff node x is a left child.*/
181 
182 /*
183  *  void incresfit (int leaf, int delta)
184  *  Increase the fitness of the leaf-th reservoir by delta and update
185  *  the reservoir fitness search tree accordingly.
186  */
187 void incresfit (int leaf, int delta)
188 {
189     int i;
190 
191     for (i=leaf; i>0; i = PARENT(i))
192         if (ISLCHILD(i))
193             restree[PARENT(i)] += delta;
194 
195     resactfit[leaf] += delta;
196     totresactfit += delta;
197 }
198 
199 /*
200  *  void flipresstate (int leaf)
```

```
 *  Flip the state of the leaf-th reservoir from mutants being active
 *  to non-mutants, or vice versa, making the necessary changes to the
 *  leaf's fitness and the search tree.
 */
void flipresstate (int leaf)
{
    int delta;

    delta = M * q - (r + q) * resmut[leaf];
    if (resstate[leaf] == ACT_NON)
        delta *= -1;

    resstate[leaf] = 1 - resstate[leaf];

    incresfit (leaf, delta);
}


/*********************************************************************/
/* Main program                                                      */
/*********************************************************************/

/*
 *  A vertex is "active" if it sends an edge to a vertex in the
 *  opposite state; otherwise, it is "dormant".
 */

int main(int argc, char **argv)
{
    int i;            /* General loop counter.                       */
    int t;            /* Loop counter for trials.                    */
    int rnd;          /* Output of random().                         */

    int src, dest;    /* Source and target vertices of reproduction.  */
    int newstate;     /* The new state for the target vertex.        */

    int extinctions,  /* The number of extinctions and fixations in a */
        fixations;    /*     batch of simulations.                   */

    unsigned int seed;  /* Random number generator seed value.       */

    /*
     *  Print simulation parameters.
     */
    setbuf (stdout, NULL); /* Turn off buffering on stdout so        */
                           /*     progress reports appear properly.  */
    printf ("$Id: superstar.c,v 1.13 2012/01/18 18:56:20 davidr Exp $\n");
    printf ("K=%d, L=%d, M=%d, N=%d, r=%d/%d, %d trials\n", K, L, M, N,
            r, q, TRIALS);
```

```
249
250     /*
251      *  Initialize random number generator.  If there is one, use the
252      *  first command-line parameter as the seed; otherwise, use the
253      *  system time.
254      */
255     seed = time (NULL);
256     if (argc > 1)
257     {
258         char *p;
259
260         long int val=strtol (argv[1], &p, 10);
261         if (p == argv[1] || *p != '\0' || val < 0)
262         {
263             fprintf (stderr, "Error.  If an argument is supplied, it
                                  must be a positive integer seed the\n
                                  random number generator.  Call with no
                                  argument to seed with the system time.\n");
264             return 1;
265         }
266         seed = (unsigned)val;
267     }
268
269     printf ("Random seed = %u\n\n", seed);
270
271     localsrandom (seed);
272
273     /*
274      *  Initialize statistics collection.
275      */
276     extinctions = fixations = 0;
277
278     /*
279      *  Outer simulation loop (simulate TRIALS times to fixation).
280      */
281     for (t = 1; t <= TRIALS; t++)
282     {
283         /*
284          *  Initialize mutant lists.  The initial state is fully
285          *  consistent: everything is a dormant non-mutant, except the
286          *  centre, which is an active non-mutant.  In a moment, we'll
287          *  pick an initial mutant and change its state.
288          */
289         numinstate[ACT_MUT] = 0;
290         numinstate[ACT_NON] = 1;
291         numinstate[DOR_MUT] = 0;
292         numinstate[DOR_NON] = NONRES-1;
293
294         for (i = 0; i < CENTRE; i++)
```

```
295             state[i] = DOR_NON;
296
297         vlist[ACT_NON][0] = CENTRE;
298         posn[CENTRE] = 0;
299         state[CENTRE] = ACT_NON;
300
301         memset (resmut, 0, L * sizeof (int));
302         memset (resactfit, 0, L * sizeof (int));
303         memset (resstate, 0, L * sizeof (int));
304         memset (restree, 0, L * sizeof (int));
305
306         resmuts = 0;
307         totresactfit = 0;
308         totnonresactfit = q;
309
310         /*
311          *  Choose initial mutant.
312          */
313         rnd = localrandom() % N;
314
315         /*
316          *  Initial mutant is a reservoir vertex.  WLOG, we may place
317          *  it in the first leaf.  And the active vertices are already
318          *  correct: the centre is active and any leaf whose first
319          *  chain vertex is a non-mutant has the mutants active, even
320          *  if there aren't actually any.
321          */
322         if (rnd > CENTRE)
323         {
324             resmut[0] = 1;
325             resmuts = 1;
326             incresfit (0, r);
327         }
328         /*
329          *  Initial mutant is in a chain.  If it's the first vertex in
330          *  its chain, the non-mutants in the leaf's reservoir are
331          *  active; otherwise, the mutant's predecessor in the chain
332          *  is.
333          */
334         else if (rnd < CENTRE)
335         {
336             setvstate (rnd, ACT_MUT);
337
338             if (rnd < L) /* Head of the chain.  */
339                 flipresstate (rnd);
340             else
341                 setvstate (rnd-L, ACT_NON);
342         }
343         /*
```

```
344          *  Initial mutant is in the centre.  The tail of each chain
345          *  becomes active.
346          */
347         else
348         {
349             setvstate (rnd, ACT_MUT);
350             for (i = L*(K-3); i < CENTRE; i++)
351                 setvstate (i, ACT_NON);
352         }
353
354         /*
355          *  Inner simulation loop (simulate until the mutants take
356          *  over or die out).
357          */
358         while (numinstate[ACT_MUT] + numinstate[DOR_MUT] + resmuts > 0
359                && numinstate[ACT_MUT] + numinstate[DOR_MUT] + resmuts < N)
360         {
361             /*
362              *  Choose a source from the set of active vertices.
363              */
364             rnd = localrandom() % (totnonresactfit + totresactfit);
365
366             /*
367              *  Source is a reservoir vertex.
368              */
369             if (rnd < totresactfit)
370             {
371                 int leaf = 0;
372
373                 /*
374                  *  Find which leaf's reservoir we've hit.
375                  */
376                 while (leaf < FIRSTSTLEAF)
377                 {
378                     if (rnd < restree[leaf])
379                         leaf = LCHILD(leaf);
380                     else
381                     {
382                         rnd -= restree[leaf];
383                         if (rnd < resactfit[leaf])
384                             break;
385                         rnd -= resactfit[leaf];
386                         leaf = RCHILD(leaf);
387                     }
388                 }
389
390                 /* Don't need to set dest, as the the index of the
391                  * first vertex of the leaf-th leaf is just leaf. */
392                 newstate = resstate[leaf];
```

```
393
394                 if (K == 3)
395                 {
396                     if (resstate[leaf] == state[CENTRE]%2)
397                         newstate += DORMANT;
398                 }
399                 else
400                 {
401                     if (resstate[leaf] == state[leaf+L]%2)
402                         newstate += DORMANT;
403                 }
404                 setvstate(leaf, newstate);
405
406                 /*
407                  *  The head of the chain has flipped state so all
408                  *  active vertices in the leaf's reservoir become
409                  *  dormant and vice-versa.
410                  */
411                 flipresstate(leaf);
412             }
413             else
414             {
415                 rnd -= totresactfit;
416
417                 if (rnd < r * numinstate[ACT_MUT])
418                     src = vlist[ACT_MUT][rnd/r];
419                 else
420                     src = vlist[ACT_NON][(rnd - r * numinstate[ACT_MUT])/q];
421
422                 /*
423                  *  If the source is a chain vertex...
424                  */
425                 if (src < CENTRE)
426                 {
427                     int chainpos = src / L;
428
429                     /*
430                      *  Last vertex in its chain.
431                      */
432                     if (chainpos == K-3)
433                     {
434                         setvstate (CENTRE, state[src]);
435
436                         /*
437                          *  Centre is changing state so need to flip
438                          *  active/dormant for chain tails.
439                          */
440                         for (i = L*(K-3); i < CENTRE; i++)
441                             setvstate (i, (state[i] + DORMANT) % NUMSTATES);
```

```
442                     }
443                     else
444                     /*
445                      *  Not the last vertex in its chain.
446                      */
447                     {
448                         dest = src+L;
449                         newstate = state[src];
450 
451                         if (K > 3 && chainpos == K-4)
452                         {
453                             if ((ISMUTANT(src)) == (ISMUTANT(CENTRE)))
454                                 newstate += DORMANT;
455                         }
456                         else if (K > 4)
457                         {
458                             if ((ISMUTANT(src)) == (ISMUTANT(dest+L)))
459                                 newstate += DORMANT;
460                         }
461                         setvstate (src, state[src] + DORMANT);
462                         setvstate (dest, newstate);
463                     }
464                 }
465                 /*
466                  *  If the source is the centre...  Since the centre is
467                  *  the only vertex with more than one out-edge, this is
468                  *  the only case where an active vertex might not update
469                  *  anything.
470                  */
471                 else
472                 {
473                     int delta = 0;
474                     int dest = 0;
475                     int leaf = localrandom() % L;
476 
477                     dest = localrandom() % M;
478 
479                     if ((ISMUTANT(CENTRE)) && dest >= resmut[leaf])
480                     {
481                         resmut[leaf]++;
482                         resmuts++;
483                         if (resstate[leaf] == ACT_MUT)
484                             delta = r;
485                         else
486                             delta = -q;
487                     }
488                     else if (!ISMUTANT(CENTRE) && dest < resmut[leaf])
489                     {
490                         resmut[leaf]--;
```

```
491                         resmuts--;
492                         if (resstate[leaf] == ACT_MUT)
493                             delta = -r;
494                         else
495                             delta = q;
496                     }
497                     if (delta != 0)
498                         incresfit (leaf, delta);
499                 }
500             }
501         } /* End of inner simulation loop. */
502 
503         /*
504          *  Statistics collection.
505          */
506         if (numinstate[ACT_MUT] + numinstate[DOR_MUT] + resmuts == 0)
507             extinctions++;
508         else
509             fixations++;
510 
511         /*
512          *  Report progress every 10 trials and at the end.
513          */
514         if (t % 10 == 0 || t == TRIALS) {
515             if (isatty (1)) {
516                 printf ("\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b"
517                         "\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b"
518                         "\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b");
519             }
520             printf ("t=%-10d  Nfix=%-10d  Next=%-10d  Pfix=%-1.4f ",
521                     t, fixations, extinctions, fixations/(double)t);
522 
523             if (!isatty (1)) {
524                 printf ("\n");
525             }
526         }
527     }
528 
529     printf ("\n");
530 
531     return 0;
532 }
```

References

[1] V. C. Barbosa, R. Donangelo, and S. R. Souza. Early appraisal of the fixation probability in directed networks. *Physical Review Series E*, 82(4):046114, 2010.