CELLULAR COOPERATION WITH SHIFT UPDATING AND REPULSION: SUPPLEMENTARY MATERIAL

ANDREAS PAVLOGIANNIS[1], KRISHNENDU CHATTERJEE[1],
BEN ADLAM[2] & MARTIN A. NOWAK[2]

(1) IST Austria, Klosterneuburg, A-3400, Austria
(2) Program for Evolutionary Dynamics, Department of Organismic and Evolutionary Biology, Department of Mathematics, Harvard University, Cambridge, MA 02138, USA

1. Computational aspects of the generic shift-update rule

In this section we describe the computational aspects of the update rules we consider. We outline the methods used in our simulations for obtaining a shift path $P : v_2 \rightsquigarrow v_1$ given a birth and a death event in positions v_2 and v_1, respectively, of an $n \times n$ grid. We have already described in the main article the fact that obtaining robust estimations of the fixation probabilities for populations of non-trivial size is a computationally expensive process that requires many iterations until convergence. Since the computation of such paths P lies at the core of every iteration, it is important that this step is efficient: for example a quadratic time procedure on grid size 16×16 would require around 65×10^3 computational steps, which would make the whole computational procedure very expensive, and hence by efficient we do not mean quadratic or cubic, but something which is almost-linear (such as $N \cdot \log N$, where N is the population size). We describe the details below.

We denote by $[n] = \{0, \dots n-1\}$ the set of natural numbers less than n. The simulation is executed on a grid where every cell occupies a position from the set $[n]^2$. Each cell at position $(i, j) \in [n]^2$ is of one type, either A or B, and has four neighbors at positions (i', j') such that $|i-i'| \pmod{n-2} + |j-j'| \pmod{n-2} = 1$,

(i.e., the grid wraps around and forms a torus). Given a position $v \in [n]^2$, we denote by $\mathsf{Nh}(v) \subseteq [n]^2$ the set of such neighbors of v. Given the current instance of the grid, a function $\mathsf{wt}_v : \mathsf{Nh}(v) \to \mathbb{R}_{\geq 1}$ maps every neighboring position v' of v to the repulsion force between v and v' according to the type of the cells in positions v and v', i.e.

$$\mathsf{wt}_v(v') = \begin{cases} 1 & \text{if } v \text{ and } v' \text{ are occupied by same-type cells} \\ \alpha & \text{otherwise} \end{cases}$$

where $\alpha \geq 1$ is the strength of the repulsion between different-type neighbors. Recall that the population size is $N = n^2$.

Consider a death event at some position v_1 of the grid. A crucial step in the generic shift-update rule is computing the resistance value $d(v)$ of every position v of the grid, defined as $d(v) = \min_{P: v \rightsquigarrow v_1} \alpha(P)$, where $\alpha(P)$ is the sum of the repulsion forces along the path P. This is performed as a standard shortest-path computation using the well-known Dijkstra's algorithm on the underlying graph, where the task is to compute the resistance value of every position in $[n]^2$ to v_1, using wt_v as the weight functions for every position $v \in [n]^2$ (Supplementary Algorithm 1). We describe Dijkstra's algorithm below.

Supplementary Algorithm 1: ResistanceValueComputation : Dijkstra'sAlgorithm

Input: Target grid coordinate v_1 and repulsion force function $\mathsf{wt}_v : \mathsf{Nh}(v) \to \mathbb{R}_{\geq 1}$ for all positions v

Output: The resistance value $d(v)$ of every position v

```
Initialize a priority queue PriorityQueue
PriorityQueue.Push(v_1, 0)
foreach v ∈ [n]^2 with v ≠ v_1 do
    PriorityQueue.Push(v, ∞)
end
while PriorityQueue is not empty do
    Assign (v, x) ← PriorityQueue.Pop()
    Assign d(v) ← x
    foreach v' ∈ Nh(v) in PriorityQueue do
        PriorityQueue.DecreaseKey(v', d(v) + wt_v(v'))
    end
end
return d
```

1.1. **Dijkstra's shortest path algorithm.** Dijkstra's shortest path algorithm considers graphs with non-negative edge weights as input, and a target position v_1, and computes the resistance values (or distance values) from all vertices to the target. The algorithm computes the distances in a greedy fashion. Initially, it sets the resistance value $d(v_1)$ of v_1 as 0. Let X be the set of positions v whose resistance values $d(v)$ to v_1 has been correctly computed.The algorithm stores pairs (v, x) (which is a pair of a grid position and an overapproximation of its resistance value) in a *priority queue*. Given the already computed resistance values for X, in each step the algorithm extends the computation by adding a position greedily that has the least resistance value among the ones whose resistance value has not yet been computed (i.e., the ones that are still in the priority queue). In other words, it considers all neighbors of positions in X, and given the already computed resistance values in X, it finds the position with the minimum resistance value and includes the position in X. Supplementary Algorithm 1 gives a formal description of the algorithm. Informally, the correctness of Supplementary Algorithm 1 lies on two invariants, which are maintained by the use of PriorityQueue. In particular, at every iteration the following hold:

(1) For the element (v, x) extracted from PriorityQueue we have $x = d(v)$.
(2) For every element (v', x') in PriorityQueue we have $x' = \min_{P:v' \rightsquigarrow v_1} \alpha(P)$, where P ranges over paths that traverse only grid positions which have already been extracted from PriorityQueue (except v' itself).

The most standard PriorityQueue implementations support the DecreaseKey operation in logarithmic time in the size of the queue, whereas Push and Pop require constant time. Since every position v has a constant number of neighboring positions (i.e., $|\mathsf{Nh}(v)| = 4$), the DecreaseKey operation will be executed at most $4 \cdot n^2$ times. We thus arrive to the following known proposition about the almost-linear running time of Supplementary Algorithm 1.

Proposition 1. *Supplementary Algorithm 1 requires $O(N \cdot \log N)$ time, where N is the number of positions.*

Consider now the case that a birth event has taken place in some position v_2 of the grid. The following two paragraphs describe how the path $P : v_2 \rightsquigarrow v_1$ is constructed algorithmically, for each specific instance of the shift-update rule (i.e., LR and LNR).

1.2. **The least-resistance rule:LR.** The least-resistance rule chooses a path $P : v_2 \rightsquigarrow v_1$ uniformly at random among all least resistance paths, i.e. $\alpha(P) = d(v_2)$. A description of the process is given in Algorithm 2. In words, the process starts from v_2, and the current path $P_{\mathsf{cur}} : v_2 \rightsquigarrow \mathsf{cur}$ is extended by a neighbor $v \in \mathsf{Nh}(\mathsf{cur})$ such that $d(\mathsf{cur}) = d(v) + \mathsf{wt}_{\mathsf{cur}}(v)$, until v_1 is reached.

Supplementary Algorithm 2: LR

Input: Grid coordinates v_1 and v_2 of the cell death and birth events respectively
Output: A least resistance path $P : v_2 \rightsquigarrow v_1$ according to the LR rule

1 Compute the resistance value function $d : [n]^2 \to \mathbb{R}$ using Dijkstra's algorithm (Supplementary Algorithm 1)
2 Assign $\mathsf{cur} \leftarrow v_2$
3 Assign $\mathsf{P} \leftarrow (\mathsf{cur})$
4 **while** $\mathsf{cur} \neq v_1$ **do**
5 $\mathsf{LeastResistanceNeighbors} \leftarrow \{v \in \mathsf{Nh}(\mathsf{cur}) : \ d(\mathsf{cur}) = d(v) + \mathsf{wt}_{\mathsf{cur}}(v)\}$
6 Assign $\mathsf{cur} \leftarrow$ an element of LeastResistanceNeighbors uniformly at random
7 Extend $\mathsf{P} \leftarrow \mathsf{P} \circ (\mathsf{cur})$
8 **end**
9 **return** P

Since the weight functions wt_v map to positive integers, the least resistance path $P : v_2 \rightsquigarrow v_1$ returned by Supplementary Algorithm 2 is acyclic, i.e., it traverses every position of the grid at most once. Hence, given the resistance values of line 1, Supplementary Algorithm 2 requires otherwise linear time. As the population size is $N = n^2$, we obtain the following proposition, which states that the running time of Supplementary Algorithm 2 is almost linear, as desired.

Proposition 2. *The time required by Supplementary Algorithm 2 is* $O(N \cdot \log N)$.

1.3. **The least neighbor-resistance rule:LNR.** The least neighbor-resistance rule chooses a path $P : v_2 \rightsquigarrow v_1$ by a random process, which maintains a current path $P_{\mathsf{cur}} : v_2 \rightsquigarrow \mathsf{cur}$, and extends it with a position from the set $\mathsf{NhR} =$

$\{v \in \mathsf{Nh}(\mathsf{cur}) : \ d(v) < d(\mathsf{cur})\}$ (i.e., a neighboring position that has smaller resistance value than the current). Every element of NhR is chosen with probability proportional to $\frac{1}{d(v)}$, i.e., neighbors that see less resistance to the vacancy of the grid are chosen with higher probability. A description of the process is given in Supplementary Algorithm 3.

Supplementary Algorithm 3: LNR

Input: Grid coordinates v_1 and v_2 of the cell death and birth events respectively
Output: A least resistance path $P : v_2 \rightsquigarrow v_1$ according to the LNR rule

Compute the resistance value function $d : [n]^2 \to \mathbb{R}$ using Dijkstra's algorithm (Supplementary Algorithm 1)
Assign $\mathsf{cur} \leftarrow v_2$
Assign $\mathsf{P} \leftarrow (\mathsf{cur})$
while $\mathsf{cur} \neq v_1$ **do**
    $\mathsf{LessResistanceNeighbors} \leftarrow \left\{ \left(v, \frac{1}{\mathrm{d}(v)} \right) : \ v \in \mathsf{Nh}(\mathsf{cur}) \text{ and } d(\mathsf{cur}) > d(v) \right\}$
    Assign $(\mathsf{cur}, p) \leftarrow$ an element of $\mathsf{LessResistanceNeighbors}$ at random, proportionally to the second component
    Extend $\mathsf{P} \leftarrow \mathsf{P} \circ (\mathsf{cur})$
end
return P

Similarly as in the case of the LR update rule, we obtain the following proposition, which states that the running time of Supplementary Algorithm 3 is almost linear, as desired.

Proposition 3. *The time required by Supplementary Algorithm 3 is* $O(N \cdot \log N)$.