Rejection-free algorithm for the Cellular Potts Model

THESIS

submitted in partial fulfillment of the
requirements for the degree of

BACHELOR OF SCIENCE
in
MATHEMATICS

Author :	J. van Dam
Student ID :	s2203669
Supervisor :	M. de Jong
Second corrector :	R. Merks

Leiden, The Netherlands, July 1, 2023

Rejection-free algorithm for the Cellular Potts Model

J. van Dam

Department of Mathematics
Niels Bohrweg 1, 2333 CA Leiden

July 1, 2023

Abstract

The Cellular Potts Model(CPM) has been extensively used to model cell ordering processes such as vasculogenesis, morphogenesis and cancer development. Implementations of the CPM generally discretise space into lattice points and time into Monte Carlo steps, which is an indication of the number of successful updates of the configuration with respect to the lattice size. In this study, the goal is to obtain a rejection-free implementation of the CPM with continuous time, so that a more natural coupling can be made between time of the model and of real world cell processes. Additionally, if this rejection free model functions similar to the traditional CPM, a link could possibly be established between Monte Carlo steps and real time. In this thesis we achieve a rejection-free model. We find that this rejection-free model differs in behaviour from the standard CPM, when we allow propensities for updates to be larger than 1. If we instead limit our propensities to a maximum of 1, we find that the behaviour of the rejection-free model should match that of the standard CPM.

Contents

Chapter 1

Introduction

1.1 Cell sorting

Cell sorting is a naturally occurring phenomenon where different types of cells are seperated from eachother. This process happens based on properties of the cells in addition to the current condition of the cells. Some examples of processes in which cell sorting plays a role are: vasculogenesis (the forming of new blood vessels from endothelial cells), angiogenesis (the forming of new bloodvellels from already existing ones) and morphogenesis (the formation and conservation of the shape of tissue). For the modelling of these processes, different types of underlying models exist.

One of these is the Cellular Potts model (CPM). The CPM is based on the differential adhesion hypothesis(DAH), an hypothesis which explains cellular movement with differences in adhesion strength. This hypothesis originates from a study by P. Townes and J. Holtfreter from 1955 [1]. Later studies have also confirmed the validity of the DAH [2]. The CPM was developed based on the DAH by Graner and Glazier in order to model cell sorting [3, 4], but adaptations are now used for a wide range of other applications. For example a CPM model has been developed by basing is on a multi-agent system [5]. This allowed it to also integrate energy exchange between cells and improved the scalability of the model. Another existing adaptation is to simulate the updating process in a parallel manner [6]. We will take a version similar to the original version of the Cellular Potts model as our basis. In section 1.2, we will explain in depth how the CPM functions exactly.

1.2 The Cellular Potts Model

The Cellular Potts Model (CPM) is a lattice based model, meaning that living cells occupy one or more lattice points. The CPM was originally developed by Graner and Glazier [3, 4] in order to model cell sorting as a result of the differential adhesion hypothesis. The original model only takes into account cell adhesion and cell area. These factors were sufficient to model cell sorting. However the CPM can easily be adapted to take other factors for cell movement into account. Examples of such factors are: cell border length [7] and chemotaxis [8] (cell movement based on a chemical gradient). Although CPM based models do also exist for 3 dimensions, such as the CompuCell3D modelling environment [9], in this thesis only a 2 dimensional variant will be discussed. The basis that we will be working form specifically, will be the Tissue Simulation Toolkit [10, 11].Therefore, living cells occupy one or more lattice points, $(x, y) \in \mathbb{Z}^2$. To each cell, we ascribe a unique spin $\in \mathbb{Z}_{\geq 0}$, which we use to distinguish cells from each other. The function $\sigma((x, y))$, returns the spin of the cell that is currently occupying lattice point (x, y). We use $\sigma(x, y) = 0$ for the case that lattice point (x, y) belongs to the medium. So two lattice points (x, y) and (x', y') belonging to the same cell, means that:

$$\sigma(x, y) = \sigma(x', y'). \tag{1.1}$$

The subset of lattice points that make up a cell i, $\{(x, y) \in \mathbb{Z}^2 | \sigma((x, y)) = i\}$, is not necessarily connected. The occurrence of disconnectivity is usually penalised the reduce the number of occurrences, as is also the case in the model we will work with. Alternatively, disconnectivity could be disallowed altogether, by modifying the CPM.

Cells belong to a certain cell type, with each cell type having potentially different characteristics. In order to keep track of these cell types, we assign an integer value to each individual cell using the function tau, $\tau : \sigma \rightarrow \mathbb{Z}_{\geq 0}$. We again use the value $\tau(\sigma(x, y)) = 0$ to denote the lattice point (x, y) being occupied by the medium. Two different lattice points belonging to the same cell type, means that:

$$\tau(\sigma(x, y)) = \tau(\sigma(x', y')). \tag{1.2}$$

In figure 1.1 is an illustration of how a configuration of the CPM could look.

Lattice points have a certain defined neighbourhood, a surrounding area

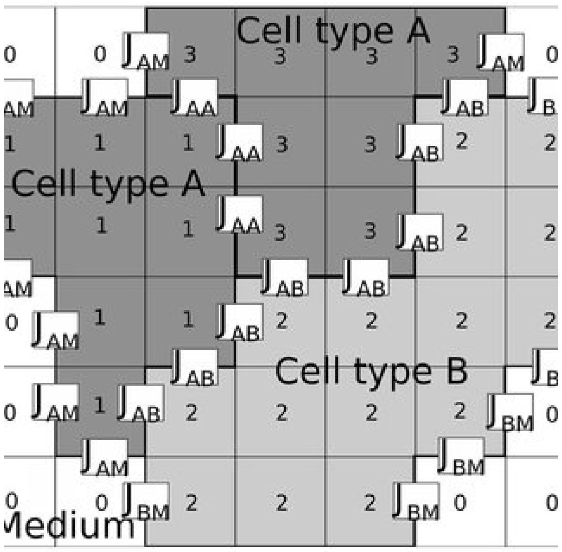

Figure 1.1: *An illustration of a possible part of a configuration in the Cellular Potts model. In this case, there are 3 cells, $\sigma = 1, \sigma = 2$ and $\sigma = 3$. Where cells 1 and 3 have cell type A and cell 2 has cell type B. From: "Multi-Scale Modeling in Morphogenesis: A Critical Analysis of the Cellular Potts Model". [12]*

of other points that impact the condition of the lattice point. We define the ordered pair of a lattice point and one of its neighbouring lattice points as an edge. Two commonly used neighbourhoods are the Moore neighbourhood and the Von Neumann neighbourhood. These can be seen in Figure 1.2.

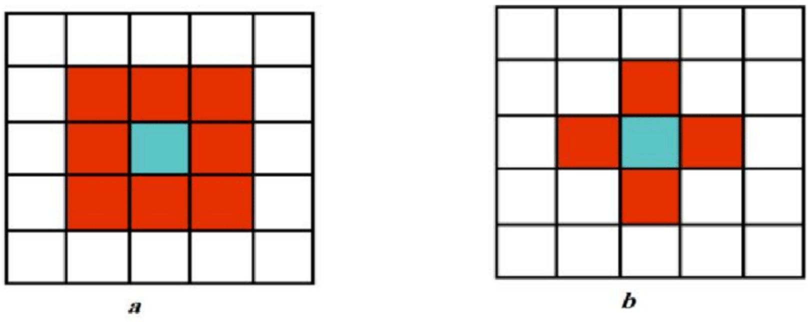

Figure 1.2: *a: The Moore neighbourhood, b: The Von Neumann neighbourhood from: Spatio-Temporal Forest Fire Spread Modeling Using Cellular Automata [13]*

Let L be the lattice of our simulation. Every copy attempt of the model, a random lattice point $(x, y) \in L$ is selected, using a uniform distribution over all lattice points. Then a random neighbour of the point (x, y) is selected: $(x', y') \in L$, again using a uniform distribution. This means that we essentially select a random edge from the set of all edges using a uniform distribution

This edge is now considered for an update, i.e., we try to copy $\sigma(x', y')$ to (x, y). This considered update does not happen immediately. First two conditions need to be checked. The first one being whether or not the update would even change the state of the current configuration. Secondly, the update will be carried out with a certain probability. If a randomly generated number is smaller than this probability, the event will occur.

For the first condition, after an edge has been selected, we check whether or not the lattice points belong to the same cell. If they do indeed belong to the same cell, i.e. $\sigma(x, y) = \sigma(x', y')$, the update would not change the configuration. Hence, no further computation is carried out for this selected update. If the considered edge is a pair of lattice points belonging to different cells we continue with potentially carrying out this update. The probability of the update taking place is dependent on the Hamiltonian function.

The Hamiltonian function describes the total energy of the configuration of the model. It takes the current configuration as an input and computes the total energy as a real number. Using the Hamiltonian function, cell characteristics such as cell adhesion, cell length [7], cell area and chemotaxis [8] can be taken into account by including these aspects in the calculation of the Hamiltonian. In its most basic form, the Hamiltonian is given by:

$$H = \sum_{\vec{x}, \vec{x}'} J_{\tau(\sigma(\vec{x})), \tau(\sigma(\vec{x}'))} \cdot (1 - \delta_{\sigma(\vec{x}), \sigma(\vec{x}')}) + \lambda \sum_{\tau} (\alpha_\tau - A_\tau)^2. \tag{1.3}$$

The first summation is for the energy in the system generated by cell adhesion. It sums over all pairs of neighbouring lattice points $\vec{x}, \vec{x}' = (x, y), (x', y') \in L$. The function $\delta_{\sigma(\vec{x}), \sigma(\vec{x}')}$ simply indicates whether or not the two lattice points belong to the same cell. It returns 1 if $\vec{x}$ and $\vec{x}'$ belong to the same cell and return 0 if they do not. This means that the first summation effectively only sums over edges where $\sigma(x, y) \neq \sigma(x', y')$. The expression $J_{\tau(\sigma(\vec{x})), \tau(\sigma(\vec{x}'))}$ refers to a matrix J, that contains cell adhesion energies, bases on cell types. It returns a value representing the energy of adhesion between two cells, when given the cell types of two cells: $\tau(\sigma(\vec{x}))$ and $\tau(\sigma(\vec{x}'))$. An interpretation of how the values in matrix J will impact cell movement is easier with the measure γ, which is described below. The second summation in equation 1.3, sums together all the energy corresponding to cell areas. It sums over all cells and then calculates how far the cell the cell currently is from its preferred size otherwise known as *target area*. This is then multiplied by some constant λ.

The 'goal' of the Cellular Potts Model is to minimise the value of the Hamiltonian over time. This can be envisioned as a number of forces doing work on the cells that occupy the lattice, causing the value of the Hamiltonian to decrease over time. Ideally, given some starting configuration and a set of parameters, we would like to find some resulting configuration that gives us a global minimum of the Hamiltonian function. However, the Hamiltonian can get into a local minimum. We want to avoid getting stuck in some local minima, preventing the system from reaching the global minimum. Therefore, In order to try to prevent getting stuck in a local minimum, we sometimes accept a change that increases the value of the Hamiltonian. Random cell motility will be possible, so long as it will not increase the Hamiltonian too much. Otherwise, the movement will only occur with some small probability.

This process of trying to minimise the Hamiltonian, while allowing increase according to a probability function, can be seen as a form of simulated annealing. This is a technique to approximate a global optimum for a given function.

For this purpose of preventing getting stuck in a local minimum, we accept updates according to a Boltzman distribution:

$$P(\Delta H) = \begin{cases} 1, & \text{if } \Delta H \leq 0 \\ e^{-\frac{\Delta H}{T}}, & \text{if } \Delta H > 0 \end{cases} \tag{1.4}$$

If a selected update decreases the value of the Hamiltonian, we always accept it. If it increases the value of the Hamiltonian, we accept it based on potential increase in the Hamiltonian ΔH and the cellular temperature T. The higher the temperature, the more chaotic the system behaves. The typical unit of time in which Monte Carlo simulations are measured, is the Monte Carlo step (MCS). For one MCS, $|L|$ update attempts are considered, where $|L|$ is the number of nodes on the lattice. The reason for this way of measuring time, is that the number of updates per time unit will be scale free. A problem with this way of measuring time, however, is that it is hard to make a connection with respect to time between cell simulations and real-world measurements. Because of this, an alternative time measurement is desirable, with the ultimate objective of making a connection between time in the simulation and passed time during cell measurements.

Gamma variable
One measure that is regularly used in the CPM is γ, originally introduced in the paper from Glazier and Graner from 1993 [4]. The measure γ indicates the surface tension between two different cell types, τ and τ'. For now, we assume there are two cell types yellow: $\tau = y$ and red: $\tau = r$. The surface tensions are defined in the following way:

$$\gamma_{ry} = J_{ry} - \frac{J_{rr} + J_{yy}}{2}, \tag{1.5}$$

$$\gamma_{rM} = J_{rM} - \frac{J_{rr}}{2}, \tag{1.6}$$

$$\gamma_{yM} = J_{yM} - \frac{J_{yy}}{2}. \tag{1.7}$$

These γ values represent the energy difference between a homotypic (same cell type) and a heterotypic (different cell type) bond. A different set of values J can still give the same resulting values for γ. It has been shown that the values of gamma determine the outcome of cell sorting [4]. For example, we could have $J_{rr} - J_{yy} \approx J_{yy} - J_{ry}$, which results in a negative γ_{ry}, which corresponds to a chequerboard pattern of the red and yellow cells[14].

1.3 Gillespie algorithm

The Gillespie algorithm is a stochastic algorithm that determines the evolution of a system where reaction rates are known, which had its original application in the area of chemical reaction kinetics. It was popularised by a paper from D. Gillespie from 1977 [15]. The paper also showed the Gillespie algorithm to be mathematically sound at calculating the evolution of a system. It is closely related to the dynamic Monte Carlo method.

Suppose we want to model a particular chemical reaction. One option would be to model this using a set of differential equations. This means we only look at the quantities of all reactants and also assume that everything is well mixed. Possibly more important, we assume that the entire process happens deterministically. These aspects come with a few drawbacks. In practice, it is regularly the case that we can not make the assumption that everything is well mixed. The fact that we only look at the overall quantities means that we completely lack insight into any local processes that might happen in the real world. The process might also not happen deterministically. If the scale is very small for example, the law of large numbers does not apply and therefore we can not expect the system to behave as it should on average. An alternative approach is to use a stochastic discrete-event model, where we include every individual reactant in the model and its possibility to react with other reactants. In order to do this, we make use of the 'master equation' which describes the transition rates between states using a matrix $A(t)$:

$$\frac{d\vec{P}}{dt} = A(t)\vec{P}. \tag{1.8}$$

Where P is a vector that represents the probability distribution of states. In simple cases with few possible states and few reactions, this system can be solved analytically. However, even in simple cases, we might not want to solve the problem analytically, as it ignores the possible stochasticity of the problem. In more complicated cases, it is better modeled as a Markov process. The Gillespie algorithm is one way to model such a Markov process. The Gillespie algorithm to stochastically simulate a system is given in the following way:

1. **Initialisation** Initialise the number of reactants and reaction rates.
2. **Monte Carlo** Generate random numbers r_1 and r_2 to determine what reaction occurs and what amount of time passes for this reaction to happen.
3. **Update** Change the state of the system according to what was generated in step 2.
4. **Iterate** Repeat steps 2 and 3 until some ending condition is reached.

After an update has happened, τ amount of time has passed, with τ being:

$$\tau = \frac{1}{\sum_j a_j(x)} \log\left(\frac{1}{r_1}\right), \tag{1.9}$$

The event that happens is the event j, for which:

$$\sum_{i=1}^{j} a_i(x) > r_2 \sum_j a_j(x) > \sum_{i=1}^{j+1} a_i(x) \tag{1.10}$$

Essentially, the probability of an event happening is equal to its propensity divided by the sum of all propensities.

The Gillespie algorithm is interesting for us since it enables the usage of a continuous time variable, which is our goal for the Cellular Potts model. Therefore we could try to implement a similar algorithm for the CPM. But before we do so, we first discuss a relevant implementation of the CPM in section 1.4 that could aid us in constructing a rejection-free model.

1.4 Alternative Implementations

Edgelist Algorithm

An idea to help us find an efficient KMC implementation for the CPM is to draw inspiration from the edgelist algorithm, discussed in "A combination of convergent extension and differential adhesion explains the shapes of elongating gastruloids" from 'de Jong et al.' [**Jong2023AGastruloids**]. The edgelist algorith is a method to keep track of all the 'useful' edges in an efficient manner. It uses two arrays: `edgelist[ ]` and `orderededgelist[ ]`, for tracking which edges would induce a change in the current configuration. Both arrays are size n, with n being equal to the total number of edges of the entire lattice.

There are essentially two types of entries in `edgelist[ ]`. The first one of these being the value -1. When the entry of `edgelist[`i`]` is -1, that means the possible update on the edge corresponding to entry i would not induce any change, i.e., the two lattice points belonging to this edge are already part of the same cell. The other possible entry is some integer number $a \in \{0, \ldots, m\}$, where m is the number of useful edges. Every possible value of a appears exactly once in `edgelist[ ]`. This way all eventful edges are numbered from 0 to m.

The array `orderededgelist[ ]` is then connected to `edgelist[ ]` with a bijection on the subset of eventful edges in the two arrays. This is done in the following way:
If `edgelist[`a_i`]`$= a$, with $a \neq -1$ then `orderededgelist[`a`]`$= a_i$. So in other words, if an entry of `edgelist[ ]` is not -1, then the array `orderededgelist` stores the position inside `edgelist` of this 'useful' edge. Having this bijection on the set of useful edges inside the two arrays enables us to do the following:

- When we need to randomly select a useful edge, we can use the first m entries of the array `orderedgelist` to find all useful edges. This prevents sampling from all n entries of `edgelist` to find all useful edges.
- When an update occurs in the CPM occurs, edges might go from useful to useless or from useless to useful. Since every position of the array `edgelist` corresponds to one particular edge, we can easily carry out the necessary changes on this array. Since we know to which edge every entry in `edgelist` belongs, we can efficiently update all relevant values in both `edgelist` and `orderedgelist`.

The two arrays are initialised in the following way: First we fill both arrays with the value '-1' and set $m = 0$. Then We move over the entire lattice. We link every edge on the lattice to its own position inside `edgelist`. If we find a useful edge i, then we place the value m inside `edgelist`$[i]$ and increase m by 1. We then set `orderedgelist`$[m] = i$. If we encounter a useless edge, we do nothing and move on to the next edge.

The edgelist algorithm does not take into account any different propensities of the useful edges. It simply uniform randomly samples one of the useful edges. For a KMC implementation, the differing propensities will need to be added to the algorithm. Namely, the selection of an event would need to take into account the propensities of all events. The propensities would need to be stored in some way that allows us to: (1) efficiently update them after a change and (2) efficiently select an update to occur based on all these propensities.

Chapter 2

Method

In Chapter 2 we will first discuss a naive implementation where we tried to use an adaptation of the edgelist algorithm to achieve a Kinetic Monte Carlo implementation for the CPM. This Implementation uses two extra arrays to keep track of the propensities and also sums of propensities, in an attempt to create a system where generating events and updating values in the data structure can happen efficiently. We will then discuss how this implementation has a problem with handling the case of an edge going from useful to useless, i.e., its propensity becoming zero. Then we will end this chapter with a well function implementation, where we use a binary tree as data structure instead of a set of arrays.

2.1 First idea

The initial idea for a rejection free KMC implementation was to store these propensities in a way similar to the edgelist algorithm we discussed in section 1.4. Two arrays would be added to the edgelist structure, meaning we would have a total of four arrays. One of these new array would be coupled to `edgelist` and the other would be linked to the `orderededgelist`. The array linked to the `edgelist` is called the `energylist`, with `energylist`[i] storing the propensity of the copy attempt corresponding to edge i. The array linked to `orderedgelist` is called `sumenergylist`. Values in `sumenergylist` are stored in the following way:

$$\texttt{sumenergylist}[k] = \sum_{i=0}^{n} \begin{cases} \texttt{energylist}[i], & \text{if } 0 \leq \texttt{edgelist}[i] \leq k \\ 0, & \text{else} \end{cases} \tag{2.1}$$

An example of the resulting storage structure is visible in table 2.1.

Position	0	1	2	3	4	5	6
Edgelist	-1	2	-1	0	1	-1	3
Energylist	0	0.1	0	0.7	1.1	0	0.2
Orderedgelist	3	4	1	6	-1	-1	-1
Sumenergylist	0.7	1.8	1.9	2.1	-1	-1	-1

Table 2.1: *Four arrays store propensity data of all edges. The entries* `edgelist` *[i] and* `energylist` *[i] correspond to the same edge, as do* `orderedgelist` *[j] and* `sumenergylist` *[j].*

As discussed in 1.4, the non-negative entries in the array `edgelist` are not necessarily in ascending order, as is also the case in table 2.1.

The purpose of the `energylist` is to easily adjust propensities after an event has occurred. When a lattice update happens, we can easily find the propensities that change since a specific edge always corresponds to a certain unchanging position in `edgelist` and `energylist`. So updating values in the four arrays for a single edge after an update consists of:

1. Calculating the propensity (the value in `energylist`).
2. What we do next depends on the value inside `edgelist`$[i]$:
   (a) if `edgelist`$[i]$=-1, we go to the first negative entry inside `orderedgelist` and `sumenergylist`.
   (b) else we take the value of `edgelist`$[i]$, and go to that position inside `orderedgelist` and `sumenergylist`.
3. Updating the corresponding value in `sumenergylist` and positive values that come after it in the array.
4. In the case of (2a) we now change the value of `edgelist`$[i]$to the position of the previous first negative entry in `orderedgelist`.

The situation is somewhat different, however, if the propensity of edge i changes from $a_i > 0$ to $a_i = 0$ however. The latter proved to be problematic to implement. We will discuss this problem in depth in 2.2, where we show how the situation of a_i changing to 0 creates a problem for our implementation of `sumenergylist`.

The purpose of the `sumenergylist` is to easily select an update to occur. We first generate a random number r_2, with $r_2 \in (0, a_0)$ uniformly distributed. Next, we start comparing the entries of the array `sumenergylist` to r_2, starting at position 0 in the array. If `sumenergylist`$[a] < r_1$ then the event corresponding to position a in the `orderedgelist` occurs, otherwise we move to position $a+1$ in the array `sumenergylist` and repeat the comparison. Hence, at most m comparisons are needed, where m is the number of useful edges. This could potentially be improved to order $\log(m)$ using binary search.

Aside from keeping track of these two additional arrays, we also keep track of an extra variable namely time t, which indicates the time that has passed so far. So after an event occurs, we update with $t = t + \tau$, with:

$$\tau = \frac{1}{a_0} ln \frac{1}{r_1}, \tag{2.2}$$

which is identical to the Gillespie algorithm as we discussed in section 1.3.

Before we can start running a simulation with our algorithm, we first need to initialise the values inside the four arrays. The `edgelist` and `orderedgelist` arrays are initialised exactly the same as discussed in 1.4. The array `energylist}` is initialised with the propensities that belong to the corresponding edges. The variable '`Sum`' is used to keep track of the sum of all added propensities so far. Whenever an edge is found to be useful, its propensity is added to Sum and its current value is used as an entry in the array `sumenergylist`.

So, in the stepprocess, we first calculate which event happens, given μ which is calculated using the random number $r_1 \in [0, 1]$. The search for the event that will occur, can then be done using binary search, in order work in time $\mathcal{O}(\log(M))$, instead of $\mathcal{O}(m)$. After we find the event that occurs, we recalculate the propensities in the neighbourhood of the carried out update. We then have to recalculate all the M non-negative entries of `sumenergylist`. As a last step, we then update the time according to the function described in equation 2.2 This initialisation and step process can be seen as pseudocode in algorithm 1.

Algorithm 1 First idea

```
Initialise edgelist, energylist, orderededgelist and    ▷ beginning of initialisation
sumenergylist as arrays of length $k \cdot N$ and fill them with $-1$
Set: $m = 0$, Sum$= 0$, $a_0 = 0$, Time$= 0$, Event$= -1$
for sites $s$ in lattice $L$ do
    for neighbours $k$ of site $s$ do
        if neighbour $k$ of $s$ belongs to a different cell then
            edgelist$[s + ki] = m$
            energylist$[s + ki] = P(\Delta H|k$ goes to the state of $s)$
            orderededgelist$[m] = s + ki$
            Sum=Sum $+P(\Delta H|k$ goes to the state of $s)$
            sumenergylist$[m] = Sum$
        end if
    end for
end for                                                  ▷ end of initialisation

Generate $r_1$ and $r_2$ on (0,1)                        ▷ Beginning of stepproces
Calculate $\tau$ according to $r_1$
Set: $i$ =Floor$[\frac{1}{2}m + 1]$
Set: Right= $m$
while Event=-1 do
    if sumenergylist$[i] < \mu$ then
        $i$ =Floor$[\frac{1}{2}(i + 1$+Right$)]$
    else if sumenergylist$[i - 1] > \mu$ && $i! = 0$ then
        $i$ =Floor$[\frac{1}{2}(i + 1)]$
        Right$= i - 1$
    else
        Event $= i$
    end if
end while
Time=Time$+\tau$
Update a lattice point according to event i
Recalculate edgelist for direct neighbours of lattice point i
for all neighbours $j \in [0, k]$ of lattice point $i$ do
    Recalculate energylist values corresponding to point $j$
end for
Recalculate first $m$ entries of sumenergylist         ▷ Ending stepproces
```

2.2 Problem of first idea

So far we have not discussed any issues of the idea for a KMC implementation that we showed in 2.1 for the CPM. However, as we briefly mentioned earlier, a problem arises when a propensity changes to 0. One problem with this implementation is the update of `sumenergylist` after such an event. The `sumenergylist` needs to contain increasing values for the first m positions. Otherwise the process of finding an event to occur does not work properly. After an event occurs, useless edges(`edgelist`$[i] = -1$) might become useful edges(`edgelist`$[i] \neq -1$) or useful edges might become useless edges. This means that non -1 entries will be added to `sumenerergy` list or non -1 entries might be set to -1.

Adding entries to the `sumenergylist` is not a problem, we can simply put them at the position of the first -1 entry and keep track of the value m, to immediately find this position.

An edge i becoming 'useless' (`edgelist`$[i] \geq 0$ changing to `edgelist`$[i] = -1$), raises a problem however. The entry in the `sumenergylist` corresponding to this previously useful edge could be somewhere in the middle of the first m entries of `sumenergylist`. This means that removing it, i.e. setting `sumenergylist` to -1 would create a gap. This creates a problem, since over time the number of gaps can become decently large and we only want the m total of non -1 entries to be at the start. One of the simplest (though not necessarily the fastest) fixes for this, is to move all non -1 entries after the 'removed edge' one position to the left. We would also need to subtract the previous value of `energylist`$[i]$ from all the entries of `sumenergylist`. This will significantly slow down the programme, as every edge that becomes useless will require up to $m - 1$ entries in the `sumenergylist` to be moved, and subsequently in `orderedgelist` to be moved.

A second problem in this implementation is the required updating of the first M entries of `sumenergylist` after an event has occurred. This is necessary because:

$$\sum_{j=0}^{p} a_j \text{ changes for } p \geq i, \quad \text{if } a_i \text{ changes} \tag{2.3}$$

It does help that a big portion of the changes in the `sumenergylist` will probably be repeatedly adding the same value to the current entry, some-thing GPUs are very good at. Still, having to update up to M values in the `sumenergylist` array is an additional load we would ideally avoid. This begs the question of whether there is a more appropriate data structure for our needs. In the next section we will discuss how using a binary tree solves our problems.

2.3 Using Binary Tree

One way to fix the problems that we discussed in 2.2, is to use a different data structure. We can view the problem of finding out what event occurs in the following manner. We have an interval $[0, a_0]$, with a_0 being the sum of all propensities. We can divide this interval in m subintervals, which each correspond to an event. Before we define these intervals, we first introduce the following notation:

$$S_j := \sum_{i=1}^{j} a_j. \tag{2.4}$$

Using these terms S_j, we can define the m partitions of the interval $(0, a_0]$as

$$\text{Interval corresponding to event } j := \begin{cases} (0, S_1], & \text{if } j = 1 \\ [S_{j-1}, S_j], & \text{if } 1 < j < m \\ [S_{m_1}, S_0], & \text{if } j = m \end{cases} \tag{2.5}$$

We need a data structure that allows us to efficiently find what sub-interval, corresponding to some event j, a randomly generated value belongs to. Since finding the correct sub-interval could be done using binary search, an obvious option as data structure is a binary tree. An intuitive way to construct this binary tree would be to assign a propensity corresponding to an edge to every leaf. The other nodes that make up the binary tree will contain values as well. Namely, each parent node will store the sum of the two values of its child nodes. The resulting structure of the binary tree is visible in figure 2.1a.

When a random value r is generated on the interval $[0, a_0]$, we carry out the following process to calculate which partition a_j this generated value r belongs to:

1. We start at the root node of the binary tree.

2. If $r >$[value of left child], then $r = r-$[value left child] and we go to the right node, otherwise we go to the left node.
3. We repeat step 2 until we reach a leaf of the binary tree.
4. The leaf we have reached corresponds to some partition $[S_{j-1}, S_j]$, which consequently corresponds to event j.

A more in-depth version of the event-finding process, written in pseudocode, is given in algorithm 3.

After an event has occurred, the values stored in the binary tree have to be changed appropriately. This happens in the following way:

1. We loop over all neighbours (xn, yn) of the lattice point (x, y) that changed its state in the most recent update.
2. For each neighbour (xn, yn), we have to adjust the values of both edge $(x, y) \rightarrow (xn, yn)$ and edge $(xn, yn) \rightarrow (x, n)$. Since every edge corresponds to one static position in the binary tree, we can easily traverse the tree to the correct leaf. There we change the value stored at the leaf-node, by recalculating the value ΔH and the resulting propensity.
3. After we have changed a value at a leaf node for a certain edge, we then recalculate all traversed parent nodes, starting at the one directly above the leaf node.

A more in-depth version updating process, written in pseudocode, is given in algorithm 2.

We encountered a problem related to the updates of parent nodes. In an earlier version, the change that occurred in a node would be calculated. Then the value of the parent node would be increased by this value. A problem then arises from the possibility of subtracting, almost equal, large values from each other. When a value in the scope of 10^{25} is subtracted from a different value of order 10^{25}, the result is no longer accurate because of floating point precision. One attempt to solve this problem, was to change relevant variables to long doubles, to try to improve the precision. This did not solve the problem however. When nodes no longer contain values equal to the sum the values of their children there will be errors. For example, in the process of calculating what event occurs, we might end up at the incorrect leaf node. This leaf might not even correspond to any edge, since the number of leaves is higher than the number of edges in most cases.

Fortunately, there is an easy solution to just recalculate the value of a parent node by summing the values of its two child nodes after one of the child nodes changed in value. This small and easy adjustment avoids subtraction altogether.

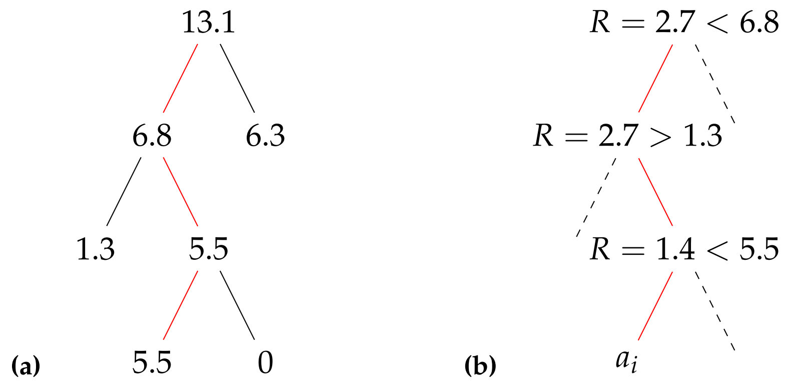

***Figure 2.1: An illustration of traversing a binary tree for the CPM**, the red lines indicate the path taken troughout the tree. (a): The values stored at the nodes inside the tree. (b): All the comparisons made to traverse the tree and the value of R over time, that is used to generate an event*

Algorithm 2 Update process after event

```
Given an update occurred on lattice point $(x, y)$
Use $v(l_1, l_2)$ to denote the value of the leaf corresponding to edge $l_1 \to l_2$
for neighbours $(x', y')$ of $(x, y)$ do
    if $\sigma(x, y) == \sigma(x', y')$ then
        $v(((x, y), (x', y'))) = 0$
        $v(((x', y'), (x, y)) = 0$
    else
        $v(((x, y), (x', y')) = \texttt{DeltaH}((x, y), (x', y'))$
        $v(((x', y'), (x, y)) = \texttt{DeltaH}((x', y'), (x, y))$
    end if
    set values of all nodes from the path of the leaf node
    towards the root node equal to sum of their 2 children
end for
```

Algorithm 3 Update finding an event

```
locator = 0, depth = 0
Node = root
Uniform randomly generate $r_1 \in [0, a_0]$
**while** Node→left ≠ NULL **do**
    **if** remainder $>$ Node →left→value **then**
        remainder = remainder - Node→left→value
        locator $=$ locator $+2^{depth}$
        Node = Node→right
    **else**
        Node = Node→left
    **end if**
**end while**
We are now at some leaf with propensity $a_i > 0$
The corresponding event is ‘locator’ from which we can calculate a
unique edge of the lattice
```

Chapter 3

Results

Now that we have an alternative CPM implementation with continuous time, we want to compare the simulations of this new model, with simulation done in the classic implementation. Specifically, we want to compare the timescale of the two models, and see whether the timescales of the two models can be linked by a conversion from one timescale to another. Recall that the timescale of the RJF- model is continuous while that of the standard model is discrete. An example of such a possible conversion would be that the two timescales have some linear ratio with respect to each other. This means, roughly, that for some event μ that indicates a lattice point copying a state, we have that $t_1(\mu) \approx C \cdot t_2(\mu)$, for some constant $C \in \mathbb{R}$. Another example of this would be a logarithmic relationship. In this case we would roughly get $t_1(\mu) = C \cdot \log(t_2(\mu))$, for some constant C. Alternatively, there might be no function to link the two timescales.

3.1 Simulations

To get insight into differences or similarities between the standard CPM and the rejection-free CPM, we ran simulations with two different matrices J of cell adhesions. One of these is a matrix J such that cell sorting would occur in the standard CPM, and the other is a matrix J such that cell engulfment would occur.

The measure that we use to track the cell sorting over time is the border-ratio, for each possible cell bond (red-red, red-yellow, and yellow-yellow).

For example, the red-red border ratio is calculated in the following way:

$$\text{red-red borderratio} = \frac{\text{number of red-red edges}}{\text{number of cel-cel edges}} \tag{3.1}$$

Using this data, we get an indication of how sorted the cells are for each point in time. We will first run simulation in subsection 3.1.1 using parameters that should cause cell to sort to groups of their own cell type. Then in subsection 3.1.2 we will run simulations, with parameters such that we expect the yellow cells to engulf the red cells.

3.1.1 Sorting parameters

We first ran simulations using cell sorting parameters. In all these simulations we used parameters: $J_{rM} = J_{yM} = 20$, $J_{rr} = J_{yy} = 10$ and $J_{ry} = 40$. Under these parameters we expect cells to eventually grouped together with cells of their own cell type. Some images of these simulations are visible in figure 3.2. One noteworthy thing in these images is that it appears that cells separating from other cells entirely seems more likely in the rejection-free model. We also tracked the border ratio for these simulations. The result is visible in figure 3.1, where we use logarithmic time scales for both models. We can see that the same border lengths for the two different models converge to different values.

3.1.2 Engulf parameters

Then we ran simulations using cell engulfment parameters. In these simulations we used the parameters: $J_{rM} = 40$, $J_{yM} = 10$, $J_{rr} = J_{yy} = 15$ and $J_{ry} = 20$. Under these parameters we expect the yellow cells to engulf the red cells. Some images of these simulations are visible in figure 3.4. The graph of the borderlength ratios over time is visible in figure 3.3. Although the two models still seem to converge to somewhat different values for the border lengths, the differences are much smaller than in figure 3.1.

3.2 Stochastic Equilibrium

Definition 3.2.1 (Stochastic equilibrium) *A stochastic equilibrium is a state of the configuration, where the expected change in the Hamiltonian is equal to zero, so $E(\Delta H) = 0$.*

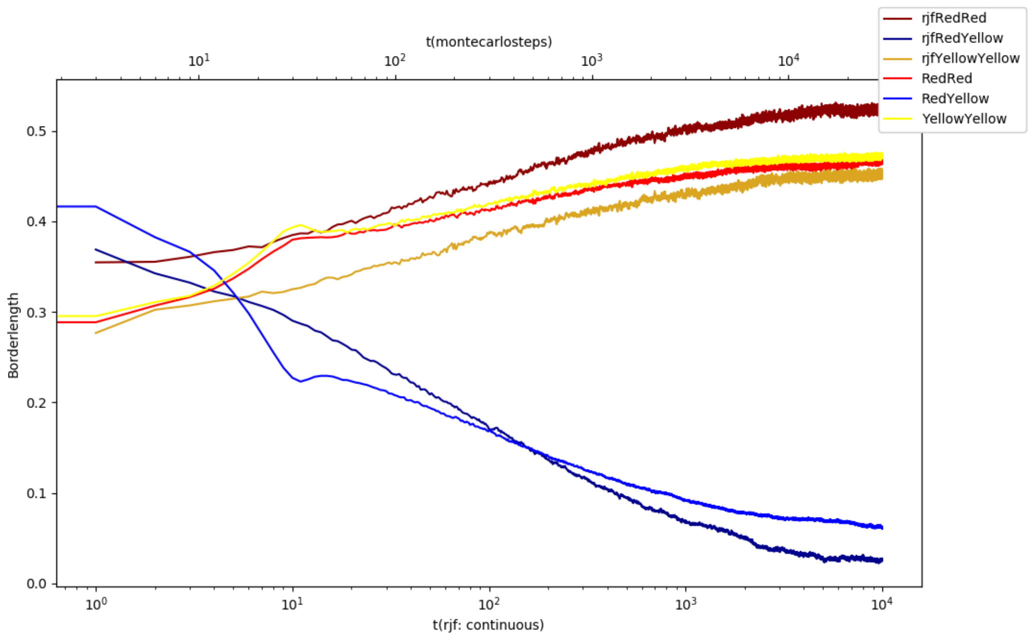

Figure 3.1: *The cell border ratios of the two models over time, plotted with a logarithmic timescale for both models. We used parameters for cell sorting:* $J_{rM} = J_{yM} = 20$*,* $J_{rr} = J_{yy} = 10$ *and* $J_{ry} = 40$*.*

Using this definition for a stochastic equilibrium, we obtain the following condition for a configuration of the CPM to be a stochastic equilibrium. We use Ω to denote the sample space, the set of possible changes in the Hamiltonian.

$$E(\Delta H) = \sum_{\Delta H \in \Omega} p(\Delta H)\Delta H = \sum_{\Delta H \in \Omega, \Delta H > 0} p(\Delta H)\Delta H + \sum_{\Delta H \in \Omega, \Delta H \leq 0} p(\Delta H)\Delta H = 0 \tag{3.2}$$

We now work towards specific expressions for the two models. For the rejection-free model, we then find:

$$E(\Delta H)_{\text{rjf}} = \sum_{\Delta H \in \Omega, \Delta H > 0} \frac{1}{a_0} e^{\frac{-\Delta H}{T}} \Delta H + \sum_{\Delta H \in \Omega, \Delta H \leq 0} \frac{1}{a_0} e^{\frac{-\Delta H}{T}} \Delta H \quad = 0 \tag{3.3}$$

$$\frac{1}{a_0} \sum_{\Delta H \in \Omega, \Delta H > 0} e^{\frac{-\Delta H}{T}} \Delta H = \frac{1}{a_0} \sum_{\Delta H \in \Omega, \Delta H \leq 0} -e^{\frac{-\Delta H}{T}} \Delta H \tag{3.4}$$

$$\sum_{\Delta H \in \Omega, \Delta H > 0} e^{\frac{-\Delta H}{T}} \Delta H = \sum_{\Delta H \in \Omega, \Delta H \leq 0} -e^{\frac{-\Delta H}{T}} \Delta H \tag{3.5}$$

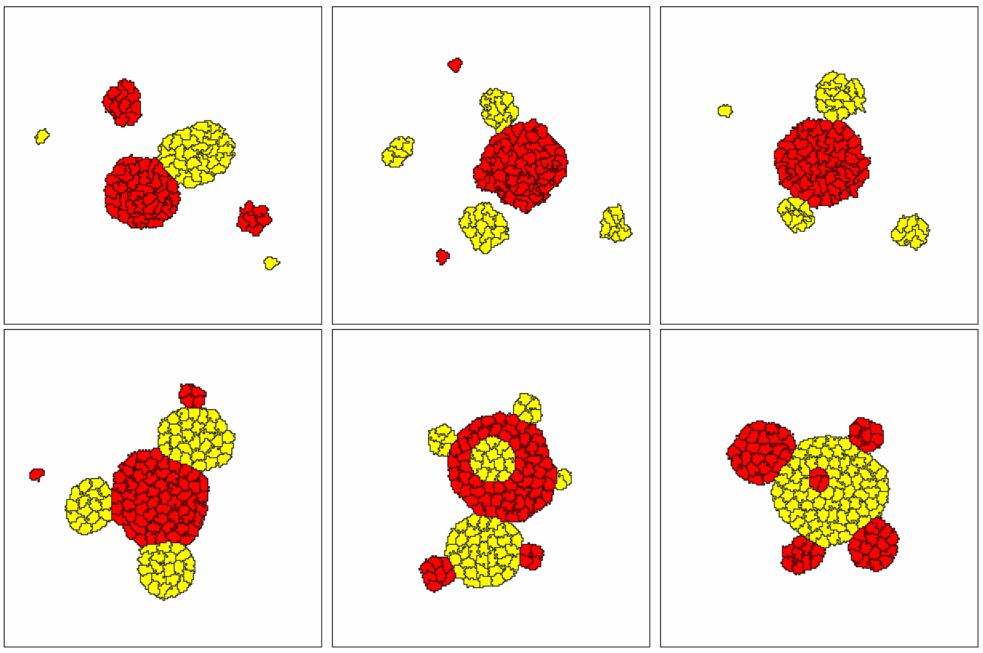

Figure 3.2: *first row: 3 different simulations of the rejection-free model at t=10000, second row: 3 different simulations of the standard model at t=30000 MCS. In all simulations we have parameters:* $J_{rM} = J_{yM} = 20$*,* $J_{rr} = J_{yy} = 10$ *and* $J_{ry} = 40$

However, for the standard model we will find a different expression. This is because, for the standard model, updates with $\Delta H \leq 0$ have the same probability to occur, since the standard model has $P(\Delta H) = 1$, if $\Delta H \leq 0$. We also need to take into account the fact that updates in the standard model are calculated by first randomly selecting an edge with a uniform distribution and then calculating whether the event corresponding to the edge will happen. For the standard model consider this expectation for the next useful event, so useless edges do not need to be accounted for. The probability for considering any given update is $\frac{1}{m}$, where m is the number of useful edges.

Additionally, we need to take into account the fact that an update could be selected, only to then be rejected.

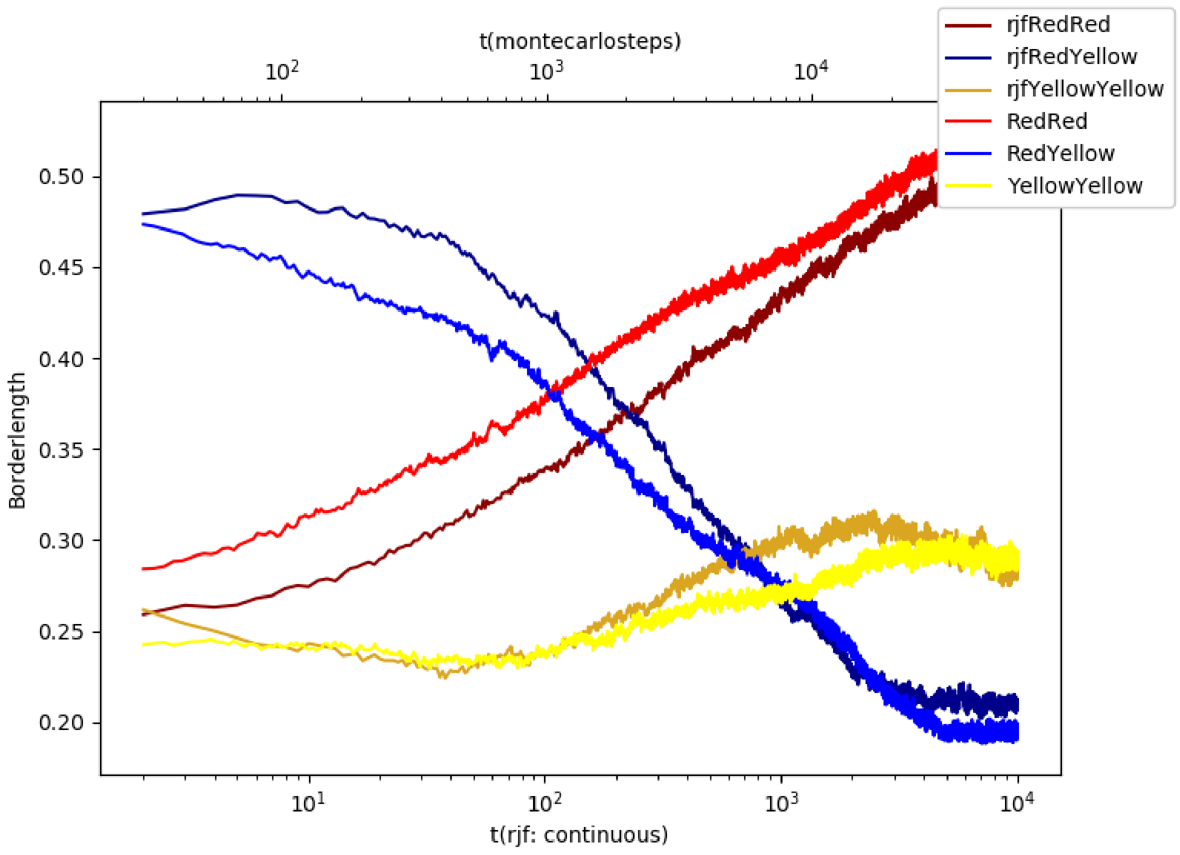

Figure 3.3: *The cell border ratios of the two models over time, plotted with a logarithmic timescale for both models. We used cell engulfment parameters:* $J_{rM} = 40$, $J_{yM} = 10$, $J_{rr} = J_{yy} = 15$ *and* $J_{ry} = 20$.

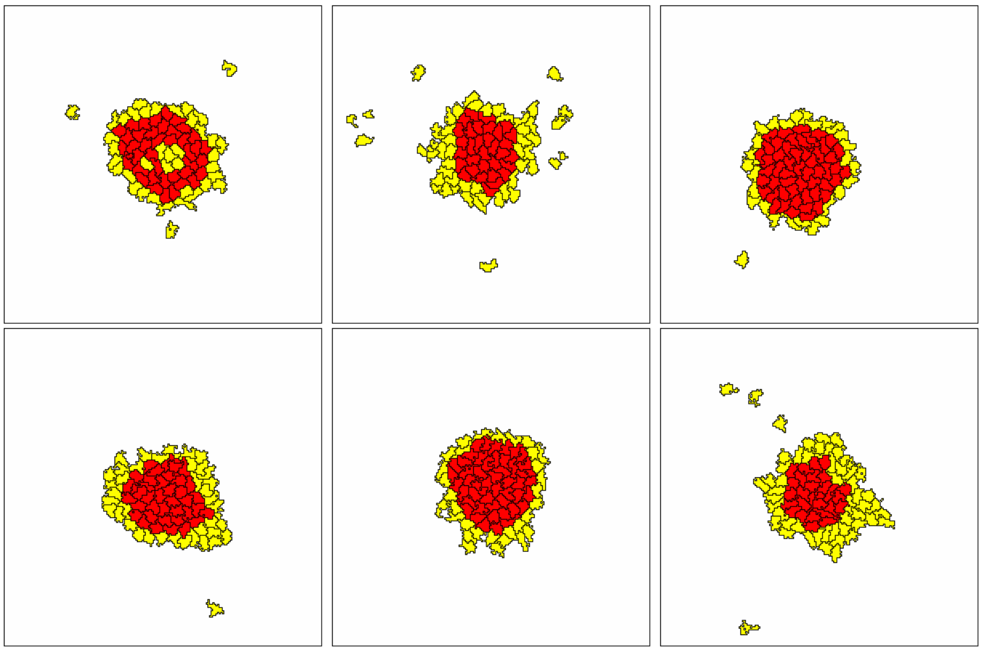

Figure 3.4: *First row: 3 different simulations of the rejection-free model at t=10000, second row: 3 different simulations of the standard model at t=59000 MCS. We ran these simulations longer for the standard model than we did in figure 3.2, to see if cells might separate, similarly to in the rejection-free model, if given some more time. In all simulations we have parameters:* $J_r M = 40$, $J_{yM} = 10$, $J_{rr} = J_{yy} = 15$ and $J_{ry} = 20$.

$$E(\Delta H)_{\text{std}} = 0 \tag{3.6}$$

$$\sum_{\Delta H \in \Omega, \Delta H > 0} \frac{1}{m} \left(e^{\frac{-\Delta H}{T}} \Delta H + (1 - e^{\frac{-\Delta H}{T}}) E(\Delta H)_{\text{std}} \right) + \sum_{\Delta H \in \Omega, \Delta H \leq 0} \frac{1}{m} \cdot 1 \cdot \Delta H = 0 \tag{3.7}$$

$$= \sum_{\Delta H \in \Omega, \Delta H > 0} \frac{1}{m} e^{\frac{-\Delta H}{T}} \Delta H + \sum_{\Delta H \in \Omega, \Delta H \leq 0} \frac{1}{m} \cdot \Delta H = 0 \tag{3.8}$$

$$= \frac{1}{m} \left(\sum_{\Delta H \in \Omega, \Delta H > 0} e^{\frac{-\Delta H}{T}} \Delta H + \sum_{\Delta H \in \Omega, \Delta H \leq 0} 1 \cdot \Delta H \right) = 0 \tag{3.9}$$

$$\sum_{\Delta H \in \Omega, \Delta H > 0} e^{\frac{-\Delta H}{T}} \Delta H = \sum_{\Delta H \in \Omega, \Delta H \leq 0} -\Delta H \tag{3.10}$$

There is a remarkable difference between the right-hand sides of the expressions on lines 3.5 and 3.10. The left-hand sides, which contain all positive changes in ΔH, are exactly the same for a given configuration of the lattice. The right-hand side of line 3.5 also contains the term $e^{\frac{-\Delta H}{T}}$ for the propensities. As we discussed before, there is a difference between the right-hand sides of equations 3.5 and 3.10. Since the differing right-hand sides are dependent on the temperature, it is to be expected that the differing convergent values for the border ratios are also dependent on the temperature.

Why the two models are expected to have different stochastic equilibria
Now suppose we have a configuration of the lattice such that $E(\Delta H) = 0$, for the standard model. This would mean, as per equation 3.10 that

$$\sum_{\Delta H \in \Omega, \Delta H > 0} e^{\frac{-\Delta H}{T}} \Delta H = \sum_{\Delta H \in \Omega, \Delta H \leq 0} -\Delta H \tag{3.11}$$

We know that ΔH for any specific update is the same for both models. If we also assume that the value of the temperature parameter T is the same, then we can substitute our expression for the standard model into equation 3.5 for the rejection-free model since both left-hand sides of equations

3.5 and 3.10 would be exactly the same.

$$\sum_{\Delta H \in \Omega, \Delta H \leq 0} e^{\frac{-\Delta H}{T}} \Delta H = \sum_{\Delta H \in \Omega, \Delta H \leq 0} \Delta H, \quad \text{if both models are at stochastic equilibrium.} \tag{3.12}$$

Since both summations have the condition that $\Delta H \leq 0$ and we also have $T \geq 1$ per standard, we obtain $e^{\frac{-\Delta H}{T}} = e^{\frac{|\Delta H|}{|T|}}$. Since we generally have that $|\Delta H| > T$, it is the case that $e^{\frac{|\Delta H|}{|T|}} > 1$. Equality 3.12 does generally not hold, but under the condition $|\Delta H| > T$, we even find the strict inequality 3.13, 3.14.

So under the assumptions $|\Delta H| > T$, and $E(\Delta H)_{\text{std}} = 0$, the following expression is obtained for the rejection-free model:

$$E(\Delta H)_{\text{rjf}} = \sum_{\Delta H \in \Omega, \Delta H > 0} e^{\frac{-\Delta H}{T}} \Delta H + \sum_{\Delta H \in \Omega, \Delta H \leq 0} e^{\frac{-\Delta H}{T}} \Delta H \tag{3.13}$$

$$< \sum_{\Delta H \in \Omega, \Delta H > 0} e^{\frac{-\Delta H}{T}} \Delta H + \sum_{\Delta H \in \Omega, \Delta H \leq 0} \Delta H = E(\Delta H)_{\text{std}} = 0 \tag{3.14}$$

apart from very specific cases, depending on the parameter T, we expect the two models to converge to different stochastic equilibria. But under the condition $|\Delta H| > T$, we even expect the value of the Hamiltonian of the rejection-free model to decrease, given a configuration for which the standard model would be at a stochastic equilibrium.

Conversely, under the condition that $|\Delta H| < T$, we expect the value of the Hamiltonian of the rejection-free model to increase, given a configuration for which the standard model is at a stochastic equilibrium.

3.3 Hamiltonian in Simulations

It is interesting to check if the rejection-free model does converge to a certain value of the Hamiltonian dependent of the parameter T. This could be verified by simulating a number of random seeds over temperatures while keeping all the other parameters the same. Due to lack of time, this verification falls outside of the scope of this thesis, however. We did track the value of the Hamiltonian for the value $T = 20$ over time, this graph is visible in figure 3.5. We used the same parameters as we used in section 3.1.2.

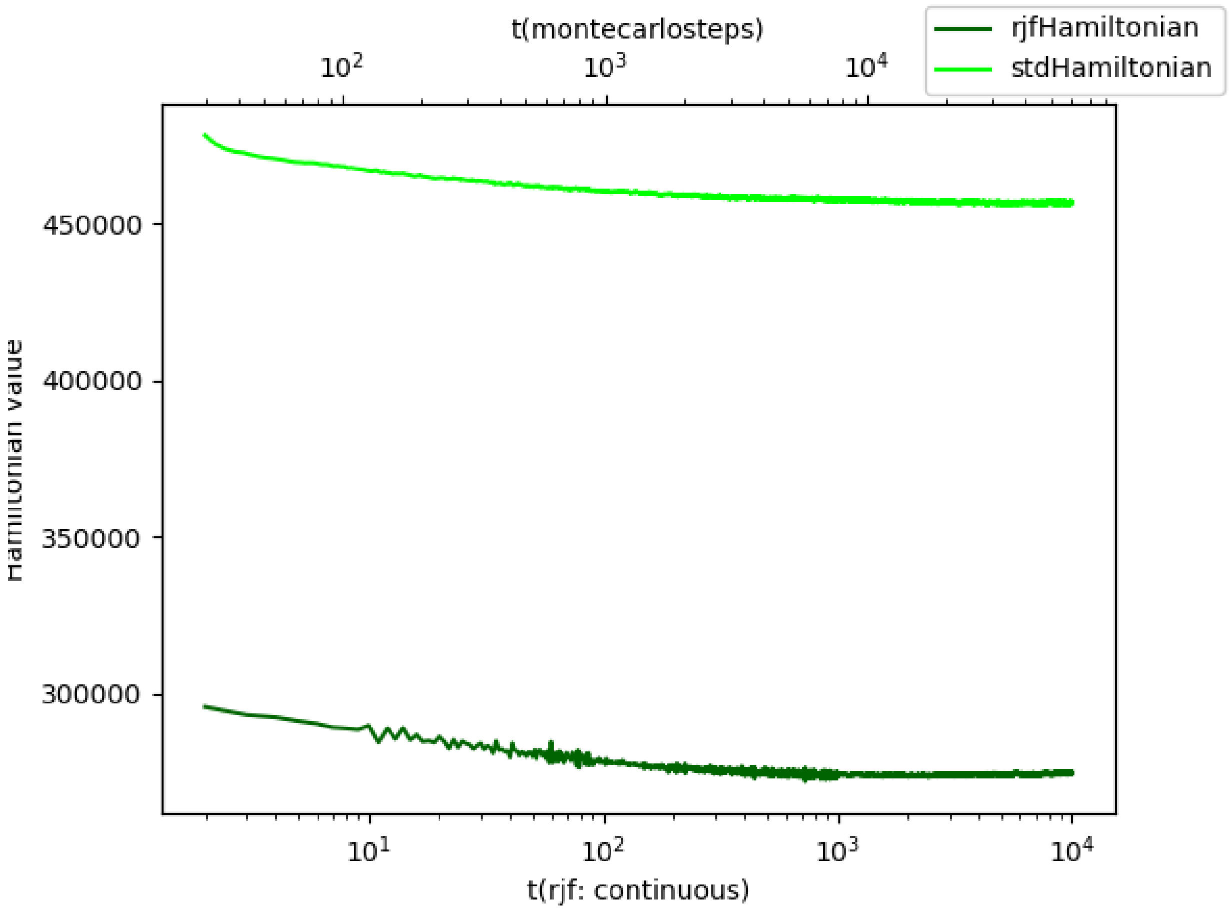

Figure 3.5: *The value of the Hamiltonian over time in the standard Cellular Potts model and the rejection- free version of the Cellular Potts model. The first 30 MCS of the edgelist CPM and the first measurement of the rejection-free CPM have been omitted, because their much higher values caused by initial cell size would warp the graph too much.*

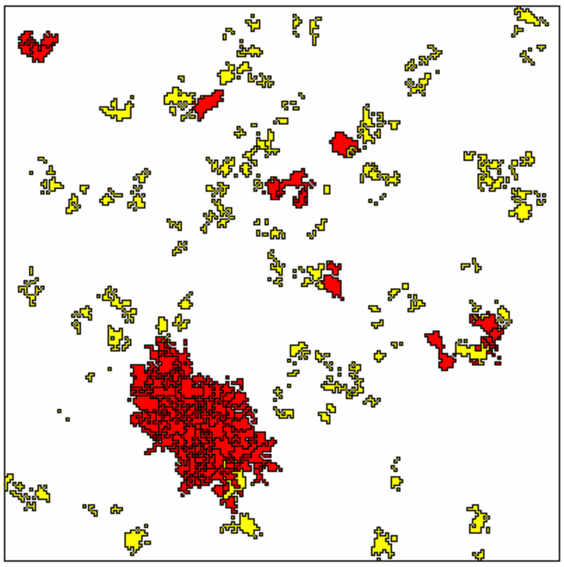

Figure 3.6: *The rejection-free CPM, where propensities $a_i \in [0,1]$. Similar to the earlier unbound rejection-free CPM simulations the disconnectivity parameter has been disabled.*

3.4 Bound propensities

Since one difference between the rejection-free model and the standard model is the fact that propensities can become larger than 1 in the rejection-free model and not in the standard model, we now test a certain adaptation for the rjf-model. Namely, we limit the propensities to $[0,1]$, to see whether this causes the rejection-free model to be similar to the standard model. The result of one such simulation is visible in 3.6. It is clear that this simulation is extremely chaotic, as cells are dispersed over the entire lattice in small fragments. Other simulations looked similar to this figure. We do not know why this chaotic behaviour occurred, as we will discuss in section 3.5 and in the discussion.

3.5 Probability comparison for bounded propensities

In this section we will investigate how the adapted rejection-free model, with bound propensities, as we described in 3.4, compares transition probabilities to the standard CPM. In order to make this comparison we first need to introduce some notation for the states and transitions of the up-

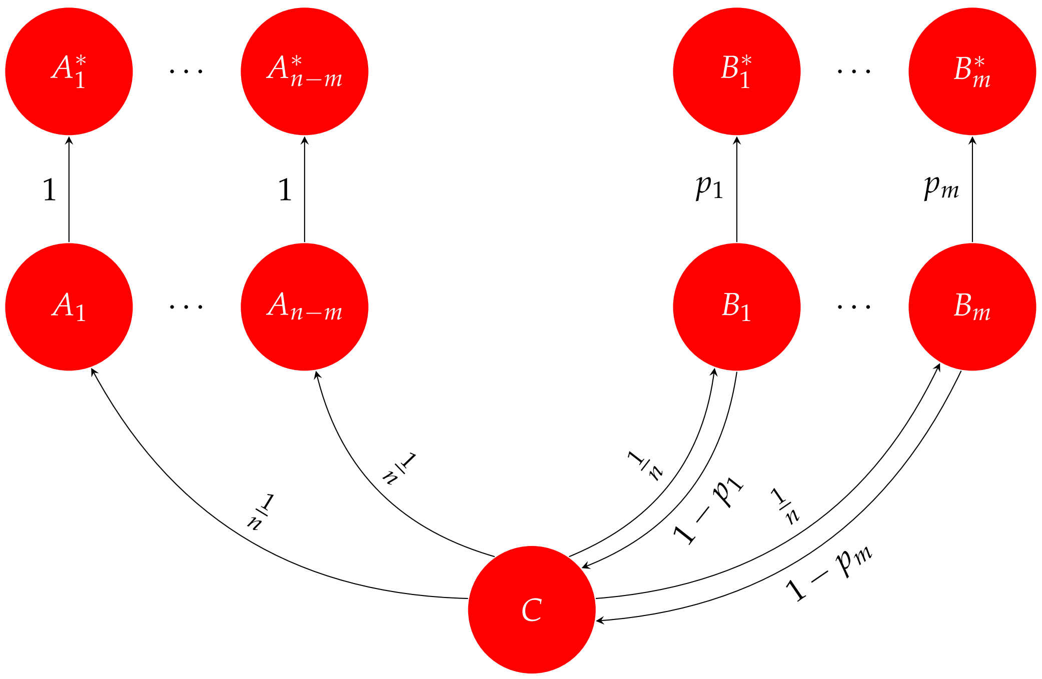

Figure 3.7: An illustration of the resulting Markov chain from updating in the standard CPM. *In state C a random possible update has yet to be generated. We use A_i and B_j to denote states where the specific update is considered, but not yet carried out. We use the states A_i^* and B_j^* to denote the specific update having been accepted.*

dating process.

Let C denote the current state of the lattice where no possible update has been selected yet. Let A_i, B_j denote the states in which the updates i or j are considered, where A_i is used for all updates for which $\Delta H_i < 0$, and B_j for all updates which have $\Delta H_i > 0$. As before, we use n to denote the total number of possible updates and use m to denote the number of updates that increase the value of the Hamiltonian. So for the indices of A_i and B_j we have: $1 \leq i \leq n - m$, and $1 \leq j \leq m$. Now write A_i^*, B_j^*, for the states in which the update i or j has successfully taken place.

The transition probability from A_i to A_i^* is 1, since $\Delta H_i < 0$. The transition probability from state B_j to B_j^* is $p_j < 1$. The state B_j also has a transition probability to C, which is the case where the selected update gets rejected. This probability is $(1 - p_j)$. The resulting Markov process is depicted in figure 3.7.

We are now interested in the probability of ending up in state A_i^* or B_j^*, given that we start in C. In other words, we want to know the explicit probability of a certain update taking place. We first consider the case of ending up at state A_i^*, and afterwards the probability of ending in B_j. A useful observation is the fact that in order to end up in A_j^*, all possible paths go back and forth between states C and any of the B_j some amount of times and then transition to state A_j^*. We know all of these transition rates, namely $p(C \to B_j) = \frac{1}{n}$, $p(B_j \to C) = 1 - p_j$, $p(C \to A_i) = \frac{1}{n}$ and $p(A_i \to A_i^*) = 1$. We can now simply sum over all the possible times of bouncing between states C and B_j, this amount we denote with k. This gives us the following expression for the probability of ending up in A_i^*.

$$p(C \to A_i*) = \frac{1}{n} + \sum_{k=1}^{\infty} \frac{1}{n} \left(\sum_{j=0}^{m} \frac{1}{n}(1-p_j) \right)^k \tag{3.15}$$

$$= \frac{1}{n} + \frac{1}{n} \sum_{k=1}^{\infty} (Q)^k \qquad \text{, with } Q = \sum_{j=0}^{m} \frac{1}{n}(1-p_j) \tag{3.16}$$

We recognise the geometric series in the second summation, so we can write this as:

$$p(C \to A_i^*)_{\text{std}} = \frac{1}{n} + \frac{1}{n} \cdot \frac{Q}{1-Q} = \frac{1}{n} + \frac{1}{n} \cdot \frac{\sum_{j=0}^{m} \frac{1}{n}(1-p_j)}{1 - \sum_{j=0}^{m} \frac{1}{n}(1-p_j)} \tag{3.17}$$

$$= \frac{1}{n} + \frac{1}{n} \cdot \frac{\frac{m}{n} - \frac{1}{n}\sum_{j=0}^{m} p_j}{1 - \frac{m}{n} + \frac{1}{n}\sum_{j=0}^{m}} \tag{3.18}$$

$$= \frac{1}{n} + \frac{1}{n} \cdot \frac{m - \sum_{j=0}^{m} p_j}{n - m + \sum_{j=0}^{m}} \tag{3.19}$$

$$= \frac{1}{n} + \frac{m - a_0 + n}{n(n - m + a_0 - n)} \tag{3.20}$$

$$= \frac{n - m + a_0 - n + m - a_0 + n}{n(n - m + a_0 - n)} = \frac{n}{n(n - n + a_0)} \tag{3.21}$$

$$= \frac{1}{a_0}. \tag{3.22}$$

For the rejection-free model it immediately follows that we have the probability:

$$p(C \to A_i^*)_{\text{rjf}} = \frac{1}{n + \sum_{j=0}^{m} p_j} = \frac{1}{a_0}. \tag{3.23}$$

So the transition probability towards a state where an update is accepted which decreases the Hamiltonian is the same for the two cases. We also find:

$$p(C \to B_i^*)_{\text{std}} = \frac{1}{n}p_i + \frac{1}{n}p_i \cdot \frac{Q}{1-Q} \tag{3.24}$$

$$= p_i\left(\frac{1}{n} + \frac{1}{n} \cdot \frac{Q}{1-Q}\right) = \frac{p_i}{a_0}, \tag{3.25}$$

And

$$p(C \to B_i^*)_{\text{rjf}} = \frac{p_i}{n + \sum_{j=0}^{m} p_j} = \frac{p_i}{a_0}. \tag{3.26}$$

So all transition probabilities are the same for the two models. Therefore we expect the rejection-free model with bound propensities and the edge-list version of the CPM that we use, to behave the same. This conclusion is in conflict with the findings of our simulations however, as can be seen in figure 3.6. We will discuss this conflict in section 4.1.

Chapter 4

Discussion

4.1 Discussion

We constructed a Kinetic Monte Carlo implementation for the Cellular Potts Model, by drawing inspiration from the Gillespie algorithm. We then ran simulations to compare this rejection-free model with the edge list algorithm. First, we ran these simulations with unbound propensities. This resulted in the two model types converging to different values for the border ratios, even though we used the same sets of parameters. It was then proven in section 3.2 that under the condition $|\Delta H| > T$, we expect the Hamiltonian of the rejection-free model to decrease, given a configuration where the standard model has a stochastic equilibrium. The converse also holds. We expect the Hamiltonian in the rejection-free model to increase for a configuration of where the standard model is at stochastic equilibrium, if we generally have $|\Delta H| < T$.

This strict difference in stochastic equilibria was then verified by tracking the value of the Hamiltonian over time for the simulations. In figure 3.5, we see how the value of the Hamiltonian is significantly higher for the standard model, compared to that of the rejection-free model. This is what we expect under the condition $|\Delta H| > T$. This condition should hold for our simulations since we used $T = 20$, whereas for ΔH values of up to 3500 are fairly common in our simulations. The question then arose if we could mimic the behaviour of our simulations using the edgelist model, by bounding our propensities to $[0, 1]$. Simulations we ran with bound propensities behaved extremely chaotic, with cells dispersing into many small fragments. When we compared the update probability functions of the rjf-model with bound propensities to those of the standard model, we found that these were the same. This is in conflict with our simulations with the bound propensities. Therefore, there is probably some error in the code causing the strange behaviour in the simulations. We discuss this more in-depth further on in the discussion.

However, the question arises as to whether we actually want to match the probability function for events of the standard CPM. When we allow $a_i \in [0, \infty)$, we possibly obtain a more natural difference in event probabilities. This is because, in the standard CPM, two events that both decrease the Hamiltonian have an equal probability of occurring at the next update. Whereas in the rejection-free CPM, where we have unbound propensities, any difference between decrements of the Hamiltonian will impact the probability these events will occur. In short, the more an event would decrease the Hamiltonian, the more likely this event will happen next.

Now that we have a Kinetic Monte Carlo implementation for the CPM, we are closer to being able to link the timescale of simulation using the Cellular Potts model with real-time. Since we found that the behaviour of the KMC implementation with bound propensities should be the same as that of the standard model, it might even be possible to equate one MCS to some amount of time in the continuous time model with bound propensities. So the question arises, when given a T_1 for the standard model if it is possible to find a T_2 for the rejection-free model with bound propensities, such that both models have stochastic equilibria for the same configurations of the CPM? Preferably we would want to find some explicit function $f(T)$ such that $f(T_1) = T_2$. If we could find such a function $f(T)$, it would most likely depend on parameters that affect the Hamiltonian. This is because the difference between stochastic equilibria of the two models is dependent on the relation between T and the overall values of ΔH that occur.

However, the question is whether we want to use a model with bound propensities, as we discussed earlier in the discussion. We do not expect there to be some simple conversion of timescales between the standard model and the rejection free model with unbound propensities, as the behaviour is not the same as we illustrated with the differing stochastic equilibria in section 3.2.

This rejection-free model should also be easily adaptable to 3-dimensional variants of the CPM. For such variants, all propensities can be stored in a binary tree, similar to how we do for our 2-dimensional model.

As has been shown in 3.5, we facilitated the rejection-free CPM to match the standard CPM by having our propensities bound to $[0, 1]$. However, we were not able to obtain data from simulations to support this claim. This is because the simulations we ran behaved extremely chaotic when we bound the propensities to $[0, 1]$, even though they should not according to our calculations in 3.5. We currently lack insight as to where this possible error in the code might be.

Unfortunately, after running all the simulations, an error was discovered in the code of the rejection-free CPM. The code did not implement the disconnectivity parameter correctly. This parameter essentially penalises updates that cause two parts of a cell to become separated. The disconnectivity parameter should be added to ΔH if separation would happen. This did not happen in the simulations of the rejection-free model present here, however. This resulted in higher propensities for updates that caused cell splitting than what the propensities should have been. Because of this mistake, the simulations of the rejection-free model should be redone with the correct implementation of the disconnectivity parameter, in order to match the implementation of the edgelist CPM. However, since cells of the rejection-free CPM in figures 3.2 and 3.4 seem to mostly be connected, we expect the results of the new simulations to not differ too much from those presented in here.

This mistake in using the disconnectivity parameter was discovered while trying to simulate the rejection-free model with propensities bound at 1. The simulation for that implementation actually did have the correct disconnectivity parameter. Unfortunately, the simulations we ran with this setup still do not seem to give expected results, both with and without the usage of the disconnectivity parameter. To be more precise, as we have shown in paragraph 3.5, the probability function for the next update of the rejection- free CPM with bounded propensities matches the probability function of the standard CPM. This means that we should also expect very similar outcomes of simulations of the two implementations. However, as we can clearly see, image 3.6 looks wildly different from what is shown in figures 3.2 and 3.4 for the edgelist model and the rejection-free models with unbound propensities. This seems to indicate that some piece of code does not quite function properly.

Although this potential error only becomes apparent in the model with bound propensities. It is most probably also affecting the model with unbound propensities in some capacity since the same code was used. This means that the results of simulations of the rejection-free model presented here are further put into question, as both the absence of the disconnectivity parameter and the potential unknown error could impact the results we found from the simulations.

Further research can be done into actually trying to link up the timescale of the rejection-free model with bound propensities to real-world time in experiments. With the establishment of such a link, the real-world time equivalent of earlier studies of the CPM could also be provided. Alternatively, research could be done into whether models with bound or unbound propensities match better in behaviour with in vitro experiments.

Additionally, now that we are closer to equating time in the simulations to real-time, an effort could be made to also link the parameters of the model to empirical data. For example, parameters for cell adhesion energies can be measured in ways described in a publication from D. Zhou and A. Garcia [16]. An example of such a method is to slowly increase the force with which two cells are being pulled apart from each other. This pulling is done using a micropipette (2-50 μm diameter depending on the requirements). A similar setup in forces could then be generated in the model. Since we would already have a matching time scale, we can estimate the cell adhesion energy by running the simulation for entries in the cell adhesion matrix, J.

4.2 Conclusion

Our goal was to obtain a KMC implementation for the Cellular Potts model. We managed to achieve this in an efficient manner by utilising a binary tree as a data structure. This implementation has the option of having either unbound propensities or of propensities being bound to the interval $[0, 1]$. We have shown how the latter should match in behaviour to the standard version of the Cellular Potts model. This did not happen in the simulations, however. Therefore, more work is needed in order to verify this finding. When we instead do not limit the values that propensities can take, we find that the behaviour of the rejection-free model and the standard model will differ for most normally used temperatures. We showed this by using stochastic equilibria as a tool. With this research we are one step closer to relating time in simulations of the Cellular Potts model to time in real-world experiments. Consequently, we are also closer to being able to match parameters in the model to experimentally gathered data of energies related to cell movement. Which in turn would further support the potency of the Cellular Potts model as a research tool.

Bibliography

[1] PL Townes and J Holtfreter. “Directed movements and selective adhesion of embryonic amphibian cells”. In: *Journal of Experimental Zoology* 128.1 (Feb. 1955), pp. 53–120. ISSN: 0022-104X. DOI: 10.1002/jez.1401280105.

[2] RA Foty and MS Steinberg. “The differential adhesion hypothesis: a direct evaluation”. In: *Developmental Biology* 278.1 (Feb. 2005), pp. 255–263. ISSN: 00121606. DOI: 10.1016/j.ydbio.2004.11.012.

[3] F Graner and JA Glazier. “Simulation of biological cell sorting using a two-dimensional extended Potts model”. In: *Physical Review Letters* 69.13 (1992), pp. 2013–2016. ISSN: 00319007. DOI: 10.1103/PhysRevLett.69.2013.

[4] JA Glazier and F Graner. *Simulation of the differential adhesion driven rearrangement of biological cells*. Tech. rep. 1993.

[5] S Tripodi, P Ballet, and V Rodin. “Computational Energetic Model of Morphogenesis Based on Multi-agent Cellular Potts Model”. In: 2010, pp. 685–692. DOI: 10.1007/978-1-4419-5913-3{_}76.

[6] N Chen, JA Glazier, and MS Alber. “A Parallel Implementation of the Cellular Potts Model for Simulation of Cell-Based Morphogenesis”. In: 2006, pp. 58–67. DOI: 10.1007/11861201{_}10.

[7] RMH Merks et al. “Cell elongation is key to in silico replication of in vitro vasculogenesis and subsequent remodeling”. In: *Developmental Biology* 289.1 (Jan. 2006), pp. 44–54. ISSN: 00121606. DOI: 10.1016/j.ydbio.2005.10.003.

[8] NJ Savill and P Hogeweg. “Modelling Morphogenesis: From Single Cells to Crawling Slugs”. In: *Journal of Theoretical Biology* 184.3 (Feb. 1997), pp. 229–235. ISSN: 00225193. DOI: 10.1006/jtbi.1996.0237.

[9] MH Swat et al. "Multi-Scale Modeling of Tissues Using CompuCell3D". In: 2012, pp. 325–366. DOI: `10.1016/B978-0-12-388403-9.00013-8`.

[10] RMH Merks and JA Glazier. "A cell-centered approach to developmental biology". In: *Physica A: Statistical Mechanics and its Applications* 352.1 (July 2005), pp. 113–130. ISSN: 03784371. DOI: `10.1016/j.physa.2004.12.028`.

[11] JT Daub and RM Merks. "Cell-based computational modeling of vascular morphogenesis using Tissue Simulation Toolkit." In: *Vascular morphogenesis*. Springer, 2015, pp. 67–127.

[12] A Voss-Bohme. "Multi-Scale Modeling in Morphogenesis: A Critical Analysis of the Cellular Potts Model". In: *PLoS ONE* 7.9 (Sept. 2012), e42852. ISSN: 1932-6203. DOI: `10.1371/journal.pone.0042852`.

[13] H Gazmeh et al. "Spatio-Temporal Forest Fire Spread Modeling Using Cellular Automata , Honey Bee Foraging and GIS". In: 3.December (2013), pp. 201–214.

[14] H Honda, H Yamanaka, and G Eguchi. "Transformation of a polygonal cellular pattern during sexual maturation of the avian oviduct epithelium: computer simulation". In: *Development* 98.1 (Nov. 1986), pp. 1–19. ISSN: 0950-1991. DOI: `10.1242/dev.98.1.1`.

[15] DT Gillespie. "Exact stochastic simulation of coupled chemical reactions". In: *Journal of Physical Chemistry* 81.25 (1977), pp. 2340–2361. ISSN: 00223654. DOI: `10.1021/j100540a008`.

[16] DW Zhou and AJ Garcia. "Measurement Systems for Cell Adhesive Forces". In: *Journal of Biomechanical Engineering* 137.2 (Feb. 2015). ISSN: 0148-0731. DOI: `10.1115/1.4029210`.

Appendices

Appendix A

Standard variable usage

x: width of the lattice
y: height of the lattice
n: the number of edges
m: number of useful edges
k: number of neighbours each lattice point has
σ: individual cell of a lattice point
τ: cell type of an individual cell
a_i: propensity of the useful edge i
a_0: the sum of all a_i
H: The Hamiltonian function, energy of entire system
ΔH_i: change in H corresponding to edge i

Appendix B

Acronyms

CPM:	Cellular Potts Model
KMC:	Kinetic Monte Carlo
MCS:	Monte Carlo Step
rjf:	rejection-free
std:	standard

Appendix C

Code

[language=C++]

```
Node::~Node(){
    delete left;
    delete right;
}

void CellularPotts::addbranch(int depthtogo, Node *position){ //+++
    if (depthtogo == 0){return;}
    position->left = new Node(0);
    position->right = new Node(0);
    int newdepthtogo = depthtogo-1;
    addbranch(newdepthtogo, position->left);
    addbranch(newdepthtogo, position->right);
}

void CellularPotts::buildtree(){ //+++
    int edges = (par.sizex-2) * (par.sizey-2)
    * nbh_level[par.neighbours]; //<< store edges on a better place?
    int depth = 0;
    while(pow(2,depth) < edges){//2 to power depth
        depth++; //<<kan ook met log
    }
    addbranch(depth, root);
}

double CellularPotts::returntime(){
  return continuoustime;
```

```
}

void CellularPotts::makechange(int locator, double newDH,
bool useful, Node *position){//+++
    // This function calculates change in a
    //propensity and makes changes upwards towards the root
    long double change;  //return to add this to all
    parent nodes
    long double newpropensity;
    int nextlocator = locator / 2;
    long double leftsum;
    long double rightsum;
    long double data;
    long double temperature = par.T;
    long double helpvar;
    if (position->left != nullptr){//we are not at the
    //leaves of the tree
        if (locator%2==0){//binary 0 -> go left
            makechange(nextlocator, newDH, useful,
            position->left);
        }
        else {//binary 1 -> go right
            makechange(nextlocator, newDH, useful,
            position->right);
        }
        position->data = position->left->data +
        position->right->data;
    }
    else{
        if(useful){
            newpropensity = exp(-newDH/temperature);
        }
        else{
            newpropensity = 0;
        }
        //printtree(root, 0, 0);
        position->data = newpropensity;
    }
}

void CellularPotts::initializeHamiltonian(){//+++
Hamiltonian = 0;
int cellsize[par.n_init_cells];
int xp,yp;
int sxy, neighsite;
double H;

for (int x=1;x<sizex-1;x++){
  for (int y=1;y<sizey-1;y++){
    if (sigma[x][y] > 0){
      cellsize[sigma[x][y]]++;
    }

    sxy = sigma[x][y];
    for(int i=1; i<=n_nb; i++){ //Energy for celladhesion
      xp=x+nx[i]; yp=y+ny[i];

      if (par.periodic_boundaries) {
        // since we are asynchronic, we cannot just copy
        //the borders once every MCS

        if (xp<=0) xp=sizex-2+xp;
        if (yp<=0) yp=sizey-2+yp;
        if (xp>=sizex-1) xp=xp-sizex+2;
        if (yp>=sizey-1) yp=yp-sizey+2;
        neighsite=sigma[xp][yp];

      } else {
        if (xp<=0 || yp<=0 || xp>=sizex-1 || yp>=sizey-1)
          neighsite=-1;
        else neighsite=sigma[xp][yp];
      }

      if (neighsite==-1) { // border
        Hamiltonian += (sxy==0?0:par.border_energy);
      } else {
        Hamiltonian += (*cell)
        [sxy].EnergyDifference((*cell)[neighsite]);
      }
    }
  }
}
```

```
  //cout << "Hamiltonian1:" << Hamiltonian << endl;
  int temp = Hamiltonian;

  for (vector<Cell>::iterator c=cell->begin();
  c!=cell->end();c++) { //Energy for cellsize
    //cout << "Area:" << c->Area() << endl;
    Hamiltonian += par.lambda * pow((c->Area() -
    c->TargetArea()), 2);
  }
  //cout << "Hamiltonian2:" << Hamiltonian-temp << endl;
  //exit(0);
}

void CellularPotts::filltree(PDE *PDEfield){//+++
    int edges = (par.sizex-2) * (par.sizey-2) *
    nbh_level[par.neighbours];
    int targetsite, targetneighbour;
    int x, y, c;
    int xp, yp, cp;
    double DH;
    continuoustime = 0;
    for(int i=0; i<edges; i++){
        targetsite = i/nbh_level[par.neighbours];
        x = targetsite%(par.sizex-2)+1;
        y = targetsite/(par.sizex-2)+1;

        targetneighbour = i%nbh_level[par.neighbours]+1;
        xp = nx[targetneighbour]+x;
        yp = ny[targetneighbour]+y;

        c = sigma[x][y];

        if (par.periodic_boundaries) {

      // since we are asynchronic, we cannot just copy the
      borders once
      // every MCS

                        if (xp<=0)
                                xp=sizex-2+xp;
            if (yp<=0)
```

```
                            yp=sizey -2+yp;
                if (xp>=sizex -1)
                            xp=xp-sizex +2;
                if (yp>=sizey -1)
                            yp=yp-sizey +2;

            cp=sigma[xp][yp]; //cell of neighbour
        }
            else if (xp<=0 || yp<=0 || xp>=sizex -1 ||
     yp>=sizey -1)
                        cp=-1; //cell is part of boundary
        else
                        cp=sigma[xp][yp];
        if (cp != c && cp != -1){
            double DH = DeltaH(x,y, xp, yp, PDEfield); //<<
            makechange(i, DH, true, root);
        }

    }
}

void CellularPotts::printtree(Node* position, int depth,
int location){//debug function to see the tree
    if(position->left != nullptr){
        printtree(position->left, depth+1, location);
        printtree(position->right, depth+1,
        location+pow(2,depth));
    }
    //if(position->left == nullptr){
        //cout << "[" << depth << ", blad:" << location
        << "," << position->data << "]" << endl;
    //}
    if(location >= (par.sizex -2)*(par.sizey -2)*8 &&
    position->data > 0){
            cout << "[" << depth << ", blad:" << location
            << "," << position->data << "]" << endl;
            cout << "crash because location not on
            lattice";
            exit(1);
    }
    if(position->data < -1){
            cout << "[" << depth << ", blad:" << location
            << "," << position->data << "]" << endl;
            cout << "left:" << position->left->data <<
            endl;
            cout << "right" << position->right->data <<
            endl;
            cout << "path to edge:" << endl;
            printedge(location);
            cout << "crash because negative data value";
            exit(1);
    }
}

void CellularPotts::printedge(int locator){
    Node* position = root;
    int depth = 0;
    while(position->left != nullptr){
        cout << "[" << depth << "," <<
        position->left->data << "," << position->data <<
        ","  << position->right->data << "]" << endl;
        if(locator%2 == 1){
            position = position->right;
        }
        else{
            position = position->left;
        }
        locator = locator / 2;
        depth++;
    }
}

int CellularPotts::findevent(long double r2){//+++
    Node *position = root;
    int locator = 0;
    int depth = 0;
    long double leftsum;
    long double rightsum;
    long double remainderr2 = r2;
    while(position->left != nullptr){
        leftsum = position->left->data;
        rightsum = position->right->data;
        if(leftsum > remainderr2){ //we go left
            position = position->left;
        }
        else{ //we go right
            locator = locator+pow(2,depth);//going right
            //equals binary 1
            remainderr2 = remainderr2 - leftsum;
            position = position->right;
        }
        depth++;
    }
    //cout << "propensity:" << position->data << endl;
    return locator;
}



void CellularPotts::BaseInitialisation(vector<Cell>
*cells) {
  CopyProb(par.T);
  cell=cells;
  if (par.neighbours>=1 && par.neighbours<=4)
    n_nb=nbh_level[par.neighbours];
  else
    throw "Panic in CellularPotts: parameter neighbours
    invalid (choose [1-4]).";

}

double CellularPotts::passedtime (){
    double r1 = RANDOM();
    double timeincrement = (1/root->data)*(log(1/r1));
    return timeincrement;
}
```